Memik Yanık

Visual Studio 2017 Eşliğinde

C# 7.0

Nesneye Yönelik Programcılığın Temelleri

- Namespace, Sınıflar, Struct'lar ve Enum'lar
- Nesneye Yönelik Programcılık İlkeleri
- Interface'ler, İndeksleyiciler
- Delegate ve Olaylar
- Generic Sınıflar
- Koleksiyon Sınıfları
- Stream Sınıfları
- Operatörleri Aşırı Yüklemek
- Pattern Matching
- Hata Yakalama ve Exception Sınıfları
- LINQ Sorguları
- Lambda İfadeleri

Yeni Sürüme Göre Güncellenmiş

7. Baskı

SEÇKİN | **Teknik**

seçkin

Visual Studio 2017 Eşliğinde Microsoft C# 7.0
Memik Yanık

SEÇKİN | **Teknik**
No: 262

ISBN 978 975 02 4885 6

Birinci Bası: Haziran 2018 (Ankara)

Sayfa Tasarımı:
Seçkin Yayıncılık

Kapak Tasarımı:
Musa Gündoğan

680 Sayfa, 16x23,5 cm.

1. Microsoft
2. c#
3. Programlama
4. Bilgisayar
5. İnternet
6. Visual Studio
7. 2017
8. Namespace
9. Nesneye Yönelik Programcılık

© Seçkin Yayıncılık San. ve Tic. A.Ş.
Bu kitabın her türlü yayın hakkı Seçkin Yayıncılık San. ve Tic. A.Ş.'ye aittir. Yayınevinin yazılı izni olmadan, tanıtım amaçlı toplam bir sayfayı geçmeyecek alıntılar hariç olmak üzere, hiçbir şekilde kitabın tümü veya bir kısmı herhangi bir ortamda yayımlanamaz ve çoğaltılamaz.

Satış ve Dağıtım:
Yayınevi Merkezi
Eskişehir Yolu, Mustafa Kemal Mah. 2158. Sokak No: 13 Çankaya/ANKARA
Tel: 0-312-435 30 30 Faks: 0-312-435 24 72 seckin@seckin.com.tr

İstanbul Çağlayan Adliyesi Şubesi	İstanbul Kartal Adliyesi Şubesi	İstanbul Şişli Şubesi	İstanbul Bölge Adliye Mahkemesi Şubesi	Ankara Strazburg Cad. Şubesi
D Blok 2. Bodrum Kat No: 1 Çağlayan	C Blok Zemin Kat No: 29 Kartal	Abide-i Hürriyet Cad. No: 183/A Şişli	C Blok Zemin Kat No:23 Kartal	Strazburg Cad. 23/B Sıhhiye
Tel: (212) 240 00 15	Tel: (216) 303 11 23	Tel: (212) 234 34 77	Tel: (216) 303 20 00 / 5462	Tel: (312) 230 52 62
Faks: (212) 240 00 15	Faks: (216) 303 11 23	Faks: (212) 231 24 69	bamsube@seckin.com.tr	Faks: (312) 230 52 62
caglayan@seckin.com.tr	kartalsube@seckin.com.tr	sislisube@seckin.com.tr		ankarasube@seckin.com.tr

Web Sitesi: www.seckin.com.tr

Seçkin Yayıncılık Sertifika No: 12416

Baskı:
Vadi Grafik Ltd. Şti. – Sertifika No: 33748
2284 Sokak No: 101 İvedik Organize Sanayi - Yenimahalle/ANKARA - Tel: (0-312) 395 85 71

Önsöz

Öncelikle bu kitabı kimler için yazıp yayınladığımızın cevabını 2 cümleyle vermeliyiz: Bu kitabı, C# ile uygulama geliştirmeye kendisini hazırlamak isteyenler için yazdık. Programcılık konusunda henüz yolun başında olan birisinin karşısına ileride çıkması muhtemel kavramları sırayla C# üzerinden temelden anlattık.

C# programlama dilinin ilk sürümü yaklaşık 15 yıl önce kullanıma sunuldu. 2018 yılı itibarıyla en son C# sürümün nosu 7. Her yeni sürümde C# programlama dilinin bazı özelliklerinde değişikliğe gidilirken çok sayıda yeni özellik ve teknoloji programcıların istifadesine sunuldu. Bu kitap, C# programlama dilini öğrenmek isteyenleri veya C# programla dili konusunda eksikliklerini gidermeyi amaçlayanları hedeflediği için önceliği konuların özüne verdik. Örneğin ilk C# sürümüyle gelen koleksiyon sınıfları anlatmadan generic koleksiyonlardan söz etmedik. Elbette bazı okurlar generic olmayan koleksiyonların son yıllarda programcılar tarafından fazla tercih edilmediğini ifade edeceklerdir. Bize sorarsanız, generic olmayan koleksiyonlarla biraz uğraşan birisi konunun temeli hakkında bilgi sahibi olacaktır. Gün gelip bu kitabın generic koleksiyonlar hakkında bilgi içeren bölümüne geçtiğinde yazıları ilk okumada anlayabilecektir.

C# programlama dilinin ilk sürümünü bile bütün ayrıntıları ile anlatmak için binlerce sayfalık kitap yazmak gerekirken her yeni sürümde dile yeni teknolojiler ekleniyorken 7. sürümü anlatmak kaygısıyla kitap yazmak istenildiğinde en önemli sorun içerik seçimini yapmak olmaktadır. Bu kitapta C# hakkında temelden bilgi edinmek isteyen birisinin bütün beklentilerini karşılayacak bir içerik hazırlamaya çalıştık. Yani kitabın okuru, kitabı sonuna kadar okuyup verilen kısa örnekleri kodlayıp test ettiğinde, C# ve nesneye yönelik programcılık hakkında zor öğrenilen konuları kavramış, C# uzmanı olmanın eşiğine gelmiş olacaktır.

Nesneye yönelik programcılık için altyapı oluşturup programcı olmaya çalışırken kavranması kolay olmayan çok sayıda kavramla karşı karşıya kalacaksınız. Keşke bu kavramları, birer alan ve metoda sahip 2 sınıf üzerinden şu şudur, şu işleve sahipti denilince hemen benimsenip kafada netleştirebilip, uygulamaya dökebilmek mümkün olsaydı. Çok sayıda usta programcının kaleminden çıkan ve nesneye yönelik programcılığı anlatmak amacıyla hazırlanmış metinlerde konu hakkında tek cümle etmeden hemen kod üzerinde anlatmaya başlamalarının en önemli nedeni konuyu ezbere öğrenmiş olmalarıdır. Ezbere öğrenmiş olmak tek başına çalışan programcının iyi kod yazmasını engellemez. Asıl sorun ezbere öğreniler. konuları başkalarına aktarırken ve birlikte kod yazarken ortaya çıkar. Bu kaygıyla bu kitapta her seferinde konunun özüne inmeye çalıştık.

Kitabın sayfa sayısını azaltmak adına bağımsız okunabilinecek bazı bölümleri PDF haline getirdik. PDF olarak seckin.com.tr'den indirebileceğiniz bölümlerin hangi sırada okunmasıyla ilgili önerilerimiz PDF'lerin içinde yer almaktadır.

İÇİNDEKİLER

Sayfa No

1. Bölüm

C# İLE PROGRAMCILIĞA BAŞLAMAK ---------- 1

Class Kavramı 5
C# Programında Metot Hazırlamak 8
Namespace'ler Hakkında Önbilgi 16
Write() ve WrileLine() Metotları 18
Statik Olan-Olmayan Metotlar 23

2.Bölüm

C# İLE PROGRAMCILIĞA GİRİŞ ---------- 31

Visual Studio Sayesinde C# Kodu Yazmak 32
Program.cs Dosyası 35
Formlar İçin Hazırlanan Kodlar 40
Olaylar ve Metot Hazırlamak 44
Olaylarla Metotları İlişkilendirmek 47
MessageBox Sınıfı 49
Kod Yazarak Nesnelerin Özelliklerini Değiştirmek 51
Projelerde Birden Fazla Form Kullanmak 52
Başlangıç Formunu Seçmek 55
TextBox Kontrolü 56

3. Bölüm

KONSOL UYGULAMALARI ---------- 59

Konsol Uygulamasında Pencere Hazırlamak 63
Konsol Uygulamaları ve Application Sınıfı 67
Not Defteri İle Visual Studio Benzeri Proje Hazırlamak 71
Main() Metodunu Parametreli Kullanmak 75
Birden Fazla Main() Metodu Hazırlamak 77

4. Bölüm

NAMESPACE HAZIRLAMAK ve KULLANMAK ---------- 79

Using Deyimi İle Namespace'lere Takma Ad Vermek 87
Using Deyimi ve IDisposeable Arayüzü 89

5. Bölüm

DEĞİŞKEN TANIMLAMAK ve KULLANMAK ---------- 91

Değişkenlere İlk Değer Aktarmak 93
Yerel Olan-Olmayan Değişkenler 94
Değişkenleri Puplic Yapmak 96
Statik Değişkenler 99
Sabit Tanımlamak 104
Object Değişkenler 106
GetType() Metodu 108
Var Anahtar Kelimesiyle Değişken Tanımlamak 109
Char Değişkenler 110
String Değişkenler 112
Escape Karakterler 114

Bool Değişkenler ... 115
Integer, Short, Byte ve Long Değişkenler ... 116
Binary Sayılar ... 120
Float ve Double Değişkenler ... 121
DateTime Değişkenler ... 123
Decimal Değişkenler ... 124
Tip Dönüşümleri ... 125
ToString() Metodu İle Dönüştürme Yapmak ... 127
Convert Sınıfı İle Dönüştürme Yapmak ... 129
Parse Metodu ... 130
Nullable Değişkenler ... 133
Dynamic Değişkenler ... 136
Object-Dynamic Karşılaştırması ... 139

6. Bölüm

DİZİ DEĞİŞKENLER ... 141

Dizi Değıskenlerin Varsayılan İlk Değerleri ... 143
SetValue() ve GetValue() Metotları ... 144
CreateIntance() Metodu İle Dizi Değişken Tanımlamak ... 145
Dizi Değişkenlerin Eleman Sayısını Çalışma Anında Belirlemek ... 146
İlk Değer Atayarak Dizi Değişken Tanımlamak ... 147
Çok Boyutlu Dizi Değişkenler ... 148
Var Deyimi İle Dizi Değişken Tanımlamak ... 149
Length Özelliği İle Eleman Sayısını Öğrenmek ... 150
Dizi Değişkenlerin İçeriklerini Silmek ... 152
Dizi Değişkenlerin İçeriklerini Kopyalamak ... 153
Dizi Değişkenlerin Elemanlarını Sıralamak ... 154
Dizi Değişkenlerde Arama Yapmak – BinarySearch Metodu ... 155
IndexOf() Metodu İle Arama Yapmak ... 156
Dizi Değişkenin İçeriğini Birleştirmek – Join() Metodu ... 157
Düzensiz Diziler ... 158

7. Bölüm

TARİH ve ZAMAN İŞLEMLERİ ... 161

Today Özelliği ... 162
String Bilgiye DateTime Tipine Dönüştürmek ... 162
DayOfWeek Özelliği ... 163
DayOfYear Özelliği ... 163
Day, Month ve Year Özellikleri ... 165
DateSerial Metodu ... 165
AddDays(), AddMonths() ve AddYears() Metotları ... 166
DaysInMonth() ve IsLeapAYear() Metotları ... 166
Hour, Minute ve Second Özellikleri ... 167
AddHours(), AddMiutes() ve AddSeconds() Metotları ... 167
Parse Metodu İle String Bilgileri DateTime Tipine Çevirmek ... 168
Ticks Özelliği ... 169
TimeSpan Yapısı ... 170

8. Bölüm

OPERATÖRLER ---- 173
Operatörlerin Öncelik Sıraları ... 178
Arttırma ve Eksiltme Operatörleri ... 179
Atama İşlemleri ... 179
Kaydırma Operatörleri ... 180
As ve Is Operatörü ... 181
Bit Düzeyinde İşlem Yapan Operatörler ... 184
Ternary Operatörü ... 187
[] Operatörü ... 190
Null – Coalesing Operatörü ... 190

9. Bölüm

BLOK KONTROL DEYİMLERİ ---- 193
Switch Case Deyimi ... 195
While Döngüsü ... 198
For Döngüsü ... 200
Foreach Döngüsü ... 201

10. Bölüm

CLASS'lar ve NESNEYE YÖNELİK PROGRAMCILIK ---- 205
Sınıfların Alanlara Sahip Olması ... 210
This Anahtar Kelimesi ... 216
Private Alanlara Dışarıdan Erişim ... 217
Özellikler – Properties ... 220
Automatic Properties ... 226
Object Initializer – Nesne Hazırlayıcılar ... 231
Partial(Parçalı) Class'lar ... 233
Nested Class'lar ... 235
Kalıtım Kavramı ... 240
Abstract Sınıflar ... 245
Interface'ler ve Özet Sınıflar ... 252
Statik Sınıflar ... 253
Yapıcı Metotlar ... 249
Using Static Ayarlaması ... 262
Ad Gizleme – Name Hidding ... 264
Metot Override İşlemleri ... 268
Polimorfizm İlkesi ... 272
Yapıcı Metotlar ... 277
Kalıtım ve Yapıcı Metotlar ... 283
Base Class'ın Yapıcı Metodu ... 287
Statik Yapıcı Metotlar ... 289
Desructor – Yok Edici Metotlar ... 292
ExpandoObject Sınıfı ... 293

11. Bölüm

METOT HAZIRLAMAK ve KULLANMAK ---- 297
Metotlar İçin Parametre Tanımlamak ... 300
Metotların Geriye Bilgi Göndermeleri ... 304

Return Anahtar Kelimesi ... 306
Dizi Değişkenleri Parametre Olarak Kullanmak ... 308
Metotların Geriye Birden Fazla Değer Göndermeleri ... 309
İsteğe Bağlı Parametreler ... 311
Değer ve Referans Parametreleri ... 313
C# 7.0'da Ref ve Out Yenilikleri ... 317
Ref Anahtar Kelimesi ... 322
Tuple Sınıfı ... 324
ValueTuple Yapısı ... 326
Metotları Aşırı Yüklmek ... 328
Metotların Kendi Kendilerini Çağırması ... 332
Metotlara Nesneleri Parametre Olarak Vermek ... 333
Partial Metotlar ... 334
Yerel Metotlar ... 336

12. Bölüm
INTERFACE'ler - ARAYÜZLER ... 339
Interface'lerde Özelliklere Yer Vermek ... 344
Bir Sınıfa Birden Fazla Interface Uygulamak ... 345
GetInterface Metodu ... 349
Interface'lerde Kalıtım ... 349
Interface'ker ve Astract Sınıflar ... 351
Interface Referansları ... 352

13. Bölüm
DELEGATE HAZIRLAMAK ve KULLANMAK ... 357
Bir Delegate ve Biden Fazla Metot ... 367
Multicast Delegate ... 369
İşletilecek Metodu Çalışma Anında Seçmek ... 374

14. Bölüm
OLAY HAZIRLAMAK ve KULLANMAK ... 381
Windows Forms Uygulamasında Olay Hazırlamak ... 385
Kendi Sınıflarımız İçin Olay Hazırlamak ... 387

15. Bölüm
OPERATÖRLERİ AŞIRI YÜKLEMEK ... 395
Implicit ve Explicit Operatörlerini Aşırı Yüklemek ... 406

16. Bölüm
İNDEKSLEYİCİLER ... 416
Indexer'ları Aşırı Yüklemek ... 423

17. Bölüm
OBJECT SINIFI ... 429
ReferanceEqual Metodu ... 434
ToString ve GetType Metotları ... 436

18. Bölüm
STRUCT HAZIRLAMAK ve KULLANMAK ... 441
Struct'lar İçin Yapıcı Metot Hazırlamak ... 445

Struct'larda Basit Tanımlamak 448
Struct'lar ve Interface'ler 451

19. Bölüm

ENUM HAZIRLAMAK ve KULLANMAK 453
Enum Sınıfı 457
GetValues() Metodu 458
Parse ve TryParse Metotları 459
Format Metodu 463
Enum'larda Veya Operatörünü Kullanmak 464

20. Bölüm

HATA YAKALAMAK ve EXCEPTION SINIFLARI 467
Sıfıra Bölme Hatasını Yakalamak 471
Entegre Hata Yakalayıcı ve Exception Sınıfları 473
Finally Bloğu 476
Hata Yakalama Sınıfları 477
FileNotFoundException Sınıfı 480
InvalidCastException Sınıfı 481
Hata Yakalama Filtreleri 481
Throw Metodu İle Exception Hazırlamak 483
Exception Sınıfı Hazırlamak 486
Checked ve Unchecked Deyimleri 489

21. Bölüm

KOLEKSİYONLAR - STACK, QUEUE ve SORTEDLIST SINIFLARI 491
Koleksiyion Sınıfları Özet Bilgi 493
ArrayList Sınıfı İle Koleksiyon Hazırlamak 494
ArrayList'te Arama İşlemleri 499
GetEnumerator Metodu 503
Stack Sınıfı 505
Queue Sınıfı 508
Hashtable Sınıfı 510
SortedList Sınıfı 515
OrderedDictionary Sınıfı 517
Generic Olan-Olmayan Koleksiyonlar 518
IEnumerable ve IEnumerator Arayüzleri 520
ICollection Arayüzü 528
CollectionBase Sınıfı 530
IDictionary Arayüzü 533

22. Bölüm

GENERIC SINIFLAR, METOTLAR, INTERFACE'ler 537
Generic Class Hazırlamak 540
İndeksleyici Desteği Olan Generic Koleksiyon 547
Generic Sınıflarda Metotlar 549
Generic Metotlara Sınıflama Koymak 555
Generic Class'larda Miras 557
Generic Interface 560
Generic Sınıflarda Tipleri Sınırlamak 561
Generic Delegate'ler 565

23. Bölüm

GENERIC KOLEKSİYON SINIFLARI ---- 567
Generic Stack Sınıfı ... 576
Generic Queue Sınıfı ... 578
Generic Dictionaray ve SortedDictionary Sınıfları ... 580
LinkedList Sınıfı ... 582

24. Bölüm

PATTERN MATCHING ---- 585
is Operatörü ve Type Pattern ... 585
Constant Pattern ... 592
Var Pattern ... 593
Switch Pattern ... 596

25. Bölüm

LINQ SORGULARINA HAZIRLIK ---- 601
Var Anahtar Kelimesi ... 601
İsimsiz Tipler ... 604
Extension Metotlar ... 609
İsimsiz Metotlar ... 622
Object Initializer ... 629

26. Bölüm

LINQ İLE SORGULAMA YAPMAK ---- 633
Where Operatörü ... 642
Ertelenmiş Yürütme ... 644
LINQ İle Koleksiyondan Birden Fazla Alanı Çekmek ... 646
LINQ ve İsimsiz Tipler ... 650
LINQ İle Dosyaları Listelemek ... 653

27. Bölüm

LAMBDA İFADELERİ ---- 655
DİZİN ---- 667

1. BÖLÜM

C# İLE PROGRAMCILIĞA BAŞLAMAK

Bu bölümü kaleme alırken karşımızda programcılık nedir az çok bilen, C# ve diğer .NET dillerine henüz ilgi duyan bir okurun olduğunu varsaydık. Bu nedenle daha önce C#, Visual Basic, Java veya C++ gibi bir programlama diliyle az çok uğraşmış birisine bu bölümün anlatım dili basit gelebilir. Bize sorarsanız C# hakkında bilgi düzeyiniz ne olursa olsun bu bölümü atlamadan sonuna kadar okumanız yararınıza olacaktır. Çünkü bu kitapta tercih ettiğimiz anlatım şeklinde duvar örer gibi tuğlalar üst üste konulmakta, her yeni bölüm bir önceki bölümün üzerine inşa edilmektedir.

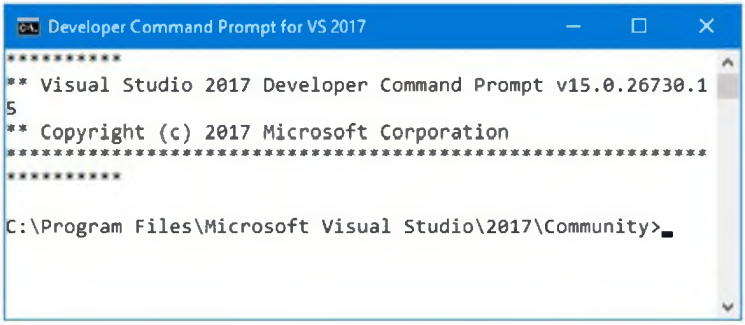
Örnek vermek gerekirse, .NET programcılığında namespace önemli bir kavramdır. Bu kitabın ortasında bir yerde namespace'ler bütün ayrıntılarıyla anlatılsa bile öncesinde sorun yaşanır. Çünkü daha ilk kod satırında namespace'lerle yüz yüze gelinmektedir. Kitabın ilk bölümünde namespace'ler ayrıntılı bir şekilde işlense bu kez anlaşılmada sorun yaşanacaktı. Biz ne mi yaptık? Bu bölümde namespace'ler hakkında yüzeysel bilgi vererek orta yolu bulmaya çalıştık.

Senaryo 1: Şartlarınız gereği C# programlama dilini öğrenmeniz gerekiyor: Okulunuzda verilecek C# derslerine katılmayı düşünüyorsunuz. Arkadaşlarınızdan C# derslerinin zor olduğu şeklinde değerlendirmeler duymuş olabilirsiniz. Bu kaygı ile C# konulu kitaplar, videolar ve makaleler toplamaya başladınız. Tam bu günlerde bir arkadaşınızın elinde bu kitabı gördünüz. Size önerimiz şudur: Arkadaşınızdan birkaç saat süre isteyin ve bu kitabın ilk bölümünü baştan sona dikkatlice okuyun. Bu okumadan sonra şuna emin olabilirsiniz: Hem derslerde anlatılanlar size yabancı gelmeyecek hem de okuduğunuz makalelerde yazılı olanları daha kolay anlayacaksınız.

Senaryo 2: C# programlama dili hakkında temel bilgilere sahipsiniz, kendinizi Nesneye Yönelik Programcılık konusunda geliştirmek ve C# programlama diliyle ilgili eksikliklerinizi tamamlamak istiyorsunuz. Dolayısıyla elinizin altında C# konulu kitaplar, her an erişebileceğiniz videolar, makaleler ve ders notları vardır. Ancak bu kaynaklara bakmak, videoları izlemek, kitapları okumak size çekici gelmiyor. Çünkü okuduklarınızı sindirmekte zorlanıyorsunuz, aradan birkaç gün geçince okuduklarınızı unutuyorsunuz. Dolayısıyla ilerleyemiyorsunuz. Size önerimiz şudur: Bu bölümde anlatılanlar için "ben bunları zaten biliyorum" demeden baştan sona dikkatlice okuyun. Okuma bittikten sonra muhtemelen ilk tepkiniz şu olacaktır: Bu metni keşke C# dili derslerine başlamadan önce okusaydım.

Program Yazmaya Başlıyoruz.

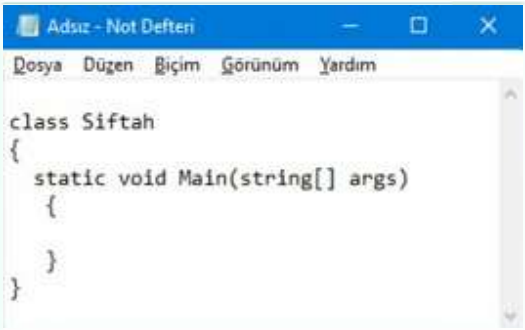
Kullandığınız bilgisayara Visual Studio'nun 2017 sürümünü kurdunuz, hemen başlatıp Windows Forms veya konsol uygulaması hazırlayıp C# üzerinde çalışmaya başlayabilirsiniz. Bu bölümde Visual Studio yerine kodlarımızı Not Defteri ile yazıp konsolda derlemeyi tercih edeceğiz. Bu nedenle Visual Studio 2017 kurulduğunda Windows'un Başlat menüsüne eklenen **Developer Command Prrompt for VS 2007** komutundan haberdar olmalısınız. Başlat menüsünden bu komutu verdiğinizde karşınıza aşağıdaki gibi bir pencere gelecektir.

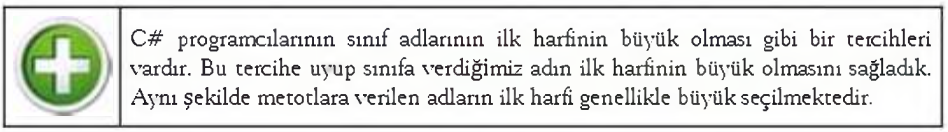


Kod yazmaya başlamadan önce kod dosyalarınızı ve Visual Studio ile hazırlayacağınız projeleri yerleştireceğiniz bir klasör hazırlamanız önerilir. Bizler "C:" sürücüsünün kök klasöründe "\Projeler" adında bir klasör hazırladık ve kod dosyalarımızı yerleştirmek istediğimiz bu klasöre geçtik.

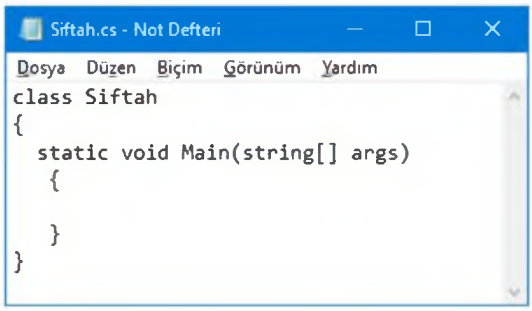


Devamında Not Defteri ile aşağıdaki gibi en kısıtlı C# programını yazdık. C# programları class'lar şeklinde düzenlendiği için henüz işlem yapan herhangi bir satıra sahip olmayan bu kod dosyasında bir class (sınıf) hazırladık. **class** anahtar kelimesi ile hazırlanan class'ların başı "{" ile, sonu ise "}" ile işaret edilmektedir.

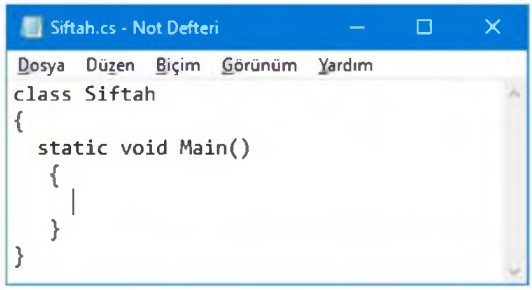



C# programcılarının sınıf adlarının ilk harfinin büyük olması gibi bir tercihleri vardır. Bu tercihe uyup sınıfa verdiğimiz adın ilk harfinin büyük olmasını sağladık. Aynı şekilde metotlara verilen adların ilk harfi genellikle büyük seçilmektedir.

C# programlama dili kod dosyalarına, sınıf adından farklı ad verilmesine izin vermektedir. Buna rağmen konuya dikkatinizi çekmek için kod dosyasına sınıfımızın adını verdik. Aşağıda verilen ekran görüntüsünü Not Defteri ile hazırladığımız C# kodumuzu CS uzantısı ile kaydettikten sonra aldık.

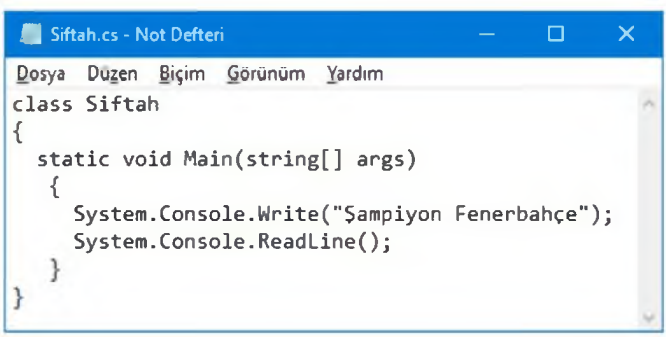


En kısıtlı C# programında bile **Main**() adında bir metodun olması gerektiği için herhangi bir işlem yapmayan bir metot hazırladık. Bu metodun "string" tipte bir parametreye sahip olmasını sağladık. C# programcılarının bu parametreye "args" adını vermek gibi bir alışkanlıkları vardır. Bu C# kodundaki **Main**() metodunun bir parametreye sahip olmasını sağlamış olsak bile aşağıdaki gibi parametresiz olarak düzenleyebilirdik.



Daha sonra sözü edileceği üzere konsolda çalıştırılmak üzere Visual Studio tarafından hazırlanan C# programlarındaki Main() metodu string tipte bir parametreye sahip olduğu için bu bölümde vereceğimiz örneklerde Main() metodunun **args** adında ve string tipte bir parametreye sahip olmasını sağladık.

Bu C# kodunu derlediğinizde hata meydana gelmez. Ancak derleme sonucu hazırlanacak .NET uyumlu EXE dosyanın çalışması ile, çalışmasının sona ermesi bir olur. Çünkü bu programın yaptığı herhangi bir işlem yoktur. Şimdi bu C# programına DOS penceresine yani Başlat menüsünden komut vererek açtığımız **Developer Command Prrompt for VS 2007** konsoluna bilgi yazan bir satır ekleyeceğiz.

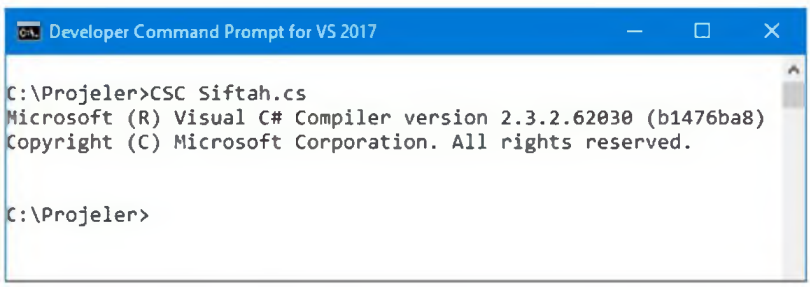


Bu şekilde düzenlenen C# programı konsolda derlenip çalıştırılırsa, **Console** sınıfının **Write**() metodu sayesinde belirtilen bilgi konsola yazılır. Console sınıfından önce yazılan **System**, NET Framework ile gelen bir namespace'tir. Namespace'ler istenen sayıda sınıf içerebilen yapılardır.

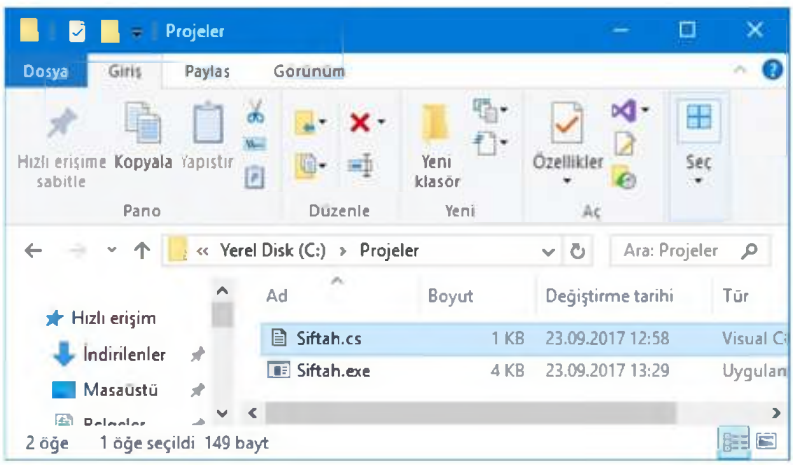
Bir sonraki satırda yer verilen **Console** sınıfının **ReadLine**() metodu sayesinde enter tuşuna basılıncaya kadar program çalışmaya devam eder. Bu programın çalışması Main() metodunun sonuna gelindiğinde sona erer. Bu programda ReadLine() metoduna yer vermemiş olsaydık programın çalışması ile çalışmasının sona ermesi bir olurdu.

Bu örnekte ReadLine() yerine **Read**() metodu kullansaydık değişen bir şey olmazdı. Bu C# kodunda **ReadLine**() yerine Console sınıfının **Readkey**() metodunu kullanmış olsaydık programın çalışması sona erdirilirken Enter yerine herhangi bir tuşa basmak yeterli olurdu.

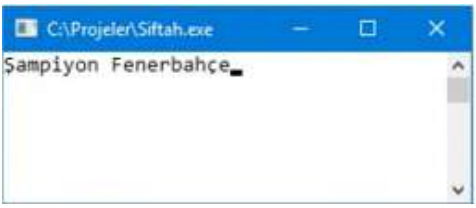
İlk C# programımızı yazıp üzerinde biraz konuştuktan sonra şimdi sıra geldi bu programı derlemeye. Bu amaçla konsola geçip komut satırına "CSC Siftah.cs" yazıp enter tuşuna bastık. C# derleyicisi kodumuzu derleyip EXE dosya hazırladı. Yazdığımız programda herhangi bir hata olmadığı için derleme sorunsuz gerçekleşti. Aşağıda verilen ekran görüntüsünü derleme işlemini tamamladıktan sonra aldık.



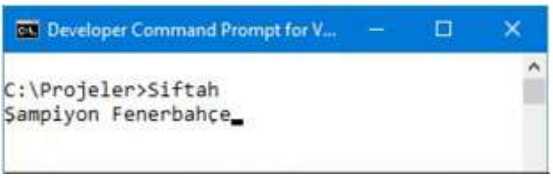
Bu şekilde yapılan derleme işlemi sonucu CS uzantılı C# programı ile aynı ada sahip bir EXE dosya hazırlanır. Başka bir deyişle bu şekilde yapılan derleme işlemi sayesinde "Siftah.exe" adında .NET uyumlu bir EXE dosya hazırlanır. Aşağıda verilen ekran görüntüsünü incelediğinizde bu EXE dosya oldukça küçüktür.



Derleme sonucu hazırlanan bu dosya EXE olsa bile ancak .NET Framework kurulu bilgisayarlarda çalışır. Windows kurulurken .NET Framework otomatik olarak kurulduğu için bunun üzerinde durmaya gerek yoktur. Bu sırada derleme sonucu hazırlanan EXE dosyayı çift tıklarsanız Windows ayrı bir konsol penceresi açar ve bu programı çalıştırır.



Yukarıdaki sayfalarda işaret edildiği gibi Windows ile birlikte .NET Framework zaten geldiği için kullanıcının .NET uyumlu programları çalıştırmak için hazırlık yapması gerekmiyor. Aşağıda verilen ekran görüntüsü bu EXE dosya çalıştırıldıktan sonra alındı.



C# programlarını derlerken kod dosyasından farklı bir isme sahip EXE dosya hazırlansın istiyorsanız **out** parametresi ile istediğiniz dosya adını belirtmeniz gerekir. Aşağıdaki gibi bir derleme yapılırsa "Siftah.cs" dosyasının derlenmiş haline "ilk.exe" adı verilir.



Class Kavramı

Konu .NET uyumlu programlama dilleri olduğunda ilk karşılaşacağınız kavram **Class**'tır. C# programları class'lar şeklinde düzenlendiği için en başından class kavramından biraz söz etmek gerekmektedir. Class'lar hakkında ayrıntılı bilgi sahibi olmadan Visual Basic 6 ve Delphi gibi programlama dilleri ile program yazmak mümkün iken, C++, C# ve Java gibi programlama dillerinde class kavramı hakkında bilgi sahibi olmak şarttır.



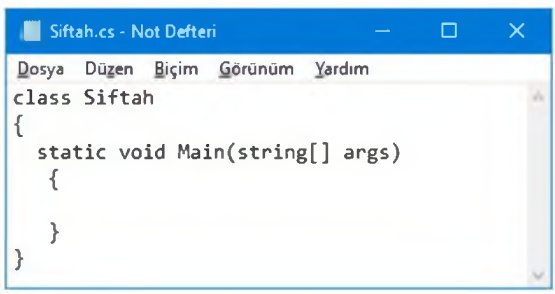
C# ve diğer nesneye yönelik programlama dillerini anlatmak kaygısı ile hazırlanan metinlerin çoğunda Class ile Object'in, yani Sınıf ile Nesne'nin karıştırıldığı belirtilir. Bu tespitlerinde haksız sayılmazlar. Bizce bu karıştırmanın önemli nedenlerinden birisi daha yolun başında olan birisine programcılığın özünün özü 2 kavramı 2 cümle ile ifade etmeye çalışmaktan kaynaklanıyor.

Pascal ve C gibi klasik programlama dillerinde kod dosyaları içinde fonksiyonlar ve procedure'ler bulunuyordu. Hatta C'de fonksiyon-procedure ayrımı yoktur. C programcıları

yapmak istedikleri işlemlere uygun fonksiyonları yazıp kod dosyalarında topluyorlardı. Sonra da gerek duydukça bunları çağırıp işletiyorlardı. Zamanla programlar büyüyüp karmaşıklaştıkça organize edilmeleri ihtiyacı doğdu. Bu ihtiyaç, nesneye yönelik programcılığın kısa sürede benimsenmesine neden oldu.

C# gibi nesneye yönelik programcılık ilkelerine bağlı dillerde programcılar hazırladıkları fonksiyonları (C#'ta fonksiyonlar **metot** olarak anılmaktadır) class olarak paketliyorlar. Günümüzde yaygın olarak kullanılan programlama dillerinde class'lar, metotlardan başka olayları, özellikleri ve delegate'leri içerebilmektedir. Yani C# programcısı yapmak istediği işlemlere uygun metotları hazırlayıp istediği sayıda class'ta toplamaktadır.

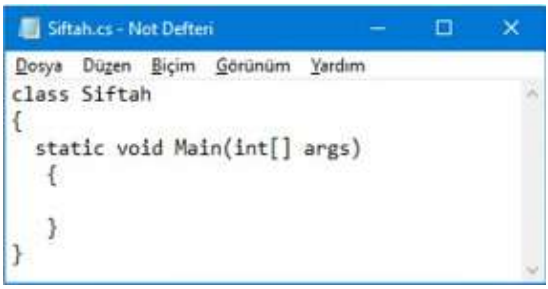
Konuyu adım adım ele almak için yukarıda verilen C# kodunu tekrar dikkatinize sunacağız. Bu program en az satıra sahip C# programıdır. Çünkü C# programı yazmak istiyorsanız bir class bloğu hazırlamanız şarttır. Ayrıca C# programlarının başlangıç noktası **Main**() metodu olduğundan hazırlanan class'ta bu metoda yer vermek gerekmektedir. Tabi uygulamamızda birden fazla class veya birden fazla CS uzantılı kod dosyası olsaydı hepsinde Main() metodunu hazırlamak gerekmeyecekti.



En az satıra sahip bu C# programındaki Main() metodu hem **static** hem de **void**'dir. C# programlarının başlangıç noktası olan Main() metodunun statik olması zorunludur ama **void** olması şart değildir. C#'ta geriye bir değer göndermeyen metotların adından önce **void** deyimine yer verilmektedir. Burada hazırlanan Main() metodu geriye herhangi bir bilgi göndermediği için adından önce void anahtar kelimesine yer verdik. Bu konunun ayrıntılarından daha sonra söz edilecektir. Şimdilik bilmenizi istediğimiz şudur: C# programlarında Main() metodunun olması zorunludur ve metot adında önce **static** kelimesinin yazılması şarttır.

Main() metoduna ait parantezlerin içine yazılan "args" bir parametredir. Başka bir deyişle, burada hazırladığımız Main() metodu "args" adında bir parametreye sahiptir. C# programcılarının Main() metodunun bu parametresine "args" adını vermek gibi alışkanlıkları vardır. Bu alışkanlık Visual Studio'dan gelmektedir. Çünkü Visual Studio ile konsolda çalıştırılmak üzere hazırlanan C# uygulamalarındaki Main() metodunun "args" adında bir parametreye sahip olması sağlanmaktadır.

Parametre adından önce yazılan "string" bu parametrenin tipini işaret etmektedir. C# programlarının başlangıç noktası olan Main() metodu parametresiz olabilir ama parametre kullanıldığında tipinin "string" olması şarttır. Bu nedenle Main() metodu aşağıdaki gibi düzenlenip parametrenin tipi int olarak seçilirse derleyici buna itiraz edip hata rapor eder.



Main() metoduna ait parantezin içinde yer verilen parametrenin tip adından sonra kullanılan "[]" işaretleri bu parametrenin gerçekte string tipte birden fazla bilgiyi dizi şeklinde metoda iletebileceğini göstermektedir. Burada kullanılan köşeli parantezler "[]" silinirse hata meydana gelir ve derleme başarıya ulaşmaz. Bu kitabın ileriki sayfalarında Main() metoduyla ilgili diğer ayrıntılardan söz edilecektir. Şimdilik her C# programında bir Main() metodunun bulunmasının zorunlu olduğunu ve bu metodun başına zorunlu olarak **static**, isteğe bağlı olarak **void** anahtar kelimelerinin yazıldığını bilmeniz yeterlidir.

Şimdi "Siftah.cs" adını verdiğimiz programın başlangıç noktası olan **Main**() metodu içinde bir değişken tanımlayacağız. Çünkü her C# programında değişkenlere gerek duyulur. Değişkenlerin nasıl tanımlandığını aşağıda görebilirsiniz. C# kodu içinde açıklama metnine yer vermek istiyorsanız "//" işaretlerinden yararlanabilirsiniz.

```
class Siftah
{
 static void Main(string[] args)
  {
   int yil; // değişken tanımlama
  }
}
```

Bu şartlarda bu C# kodunu konsolda derlersek derleyici EXE dosyayı hazırlar ama bize uyarıda bulunur. Çünkü tanımlayıp "yil" adını verdiğimiz değişkeni hiç kullanmadık. Derleme sırasında verilecek bu uyarı EXE dosyanın hazırlanmasına engel değildir. Derleyici uyarı yerine hata mesajı vermek zorunda kalsaydı EXE dosya hazırlanmazdı.

Burada "yil" adını verdiğimiz değişkeni Main() metodu içinde tanımladık. Dolayısıyla bu değişkenin yaşam alanı Main() metodunun içiyle sınırlıdır. Burada yaptığımız gibi metotların içinde tanımlanan değişkenlere **yerel değişken** adı verilmektedir.

Bu değişkeni Main() metodu tarafından sınırlanan blok yerine aşağıda yapıldığı gibi **Class** bloğu içinde tanımlayabilirdik. Bu durumda bu değişken "Siftah" adını verdiğimiz class'ın her yerinde yaşardı. Konuyu nesneye yönelik programcılık penceresinden bakıldığında class bloğu içinde tanımlanan değişkenlere **alan (field)** adı verilmektedir.

```
class Siftah
{
 int yil;  // class bloğu içinde tanımlanan değişken
 static void Main(string[] args)
  {
  }
}
```

Burada "yıl" adını verdiğimiz değişkeni gerekirse class bloğu içinde tanımlayabileceğimizi işaret ettikten sonra bu satırı tekrar Main() metodunun içine aldık. C# programcılarının metotların içinde tanımlanan, dolayısıyla yerel değişken olarak anılan değişkenlere ad seçerken ilk harfi küçük seçmek gibi bir alışkanlıkları vardır.

```
class Siftah
{
 static void Main(string[] args)
  {
   int yil;
  }
}
```

C# programlarında değişken tanımlanırken önce değişkenin tipi sonra adı yazılmaktadır. Buradaki **int** bu değişkenin tamsayı tipindeki bilgileri tutabileceğini işaret etmektedir. "yil" adını verdiğimiz bu değişkeni tanımlamakla derleyiciye demiş oluyoruz ki, "Main() metodu içinde iken integer tipindeki (tamsayı) bilgileri bellekte tutma gereğini duyabiliriz".

Bir C# programı olarak görebileceğimiz bu kısacak kodu birkaç cümle ile anlatmak gerekirse şunlar söylenebilir:

- Bu koddaki "Siftah" adlı class'ın **Main**() adında bir metodu bulunmaktadır.
- Main() metodu, C# programlarının başlangıç noktasıdır ve **static** olması şarttır.
- Main() metodunun başına **void** anahtar kelimesi genellikle yazılır. Başına void yazılan metotlar geriye değer göndermeyen metotlardır.
- Main() metodu içinde yaşayabilecek "yil" adında ve integer tipindeki bilgilerin aktarılabileceği bir değişken tanımlanmıştır.
- Bu değişken Main() metodu içinde tanımlandığı için yalnızca Main() metodunun sınırları içinde yaşayabilen yerel bir değişkendir.
- C# programlarında class ve metot başlatılan satırlar hariç her satırın sonuna ";" konulmaktadır.

C# Programında Metot Hazırlamak

Tekrar etmek gerekirse yukarıda hazırladığımız C# programı Main() adında bir metoda sahipti. Bu metot her C# uygulamasında bulunması zorunlu bir metot olduğu için ilk yazdığımız C# programında bu metoda yer vermiştik.

Şimdi yukarıda yazdığımız en kısıtlı C# programı içinde hazırladığımız class'ta değişiklik yapıp 2. bir metoda yer vereceğiz. Herhangi bir satır içermeyen ve **Mesaj()** adını verdiğimiz bu metot bu hali ile herhangi bir işlem yapmaz.

```
class Siftah
{
 static void Main(int[] args)
  {
  }
 static void Mesaj() //statik ve geriye bilgi göndermeyen metot
  {
  }
}
```

Bu metodu hem **static** hem de **void** olarak ayarladığımızı fark etmiş olmalısınız. Başka bir deyişle bu metoda verdiğimiz adın önüne hem static" hem de "void" anahtar kelimelerini yazdık. Anahtar kelimeler, C# programlama dili için özel işlevi olan ayrılmış kelimelerdir.

Dikkatinizi yeni hazırlanan metoda çekmek için Main() metodundaki değişken tanımlama satırını sildik. Metotların adlarına "()" parantezlerin eklendiğini ve metotlara ait kodların "{}" arasına yazıldığını fark etmiş olmalısınız. Metot adının yazıldığı satır ve başlangıç-bitiş yerini gösteren '{}' parantezlerin kullanıldığı satırların sonuna ';' konulmadığını biliyorsunuz.



Verilen koddan fark etmiş olacağınız gibi hazırladığımız her iki metot class bloğu içinde yer almaktadır. C# dilinde metotlar mutlaka bir class'ın üyesi olmalıdır. Yani metotlar class bloğunun dışında bulunamazlar. C# dilinde metotlar struct'ların içinde yer alabilseler bile şimdilik ilgi alanımız sadece class'lardır.

Bu hali ile bu C# programı (CS uzantılı kod dosyası) konsolda derlenip EXE dosya hazırlanıp çalıştırılırsa Main() metodu otomatik olarak işletilir. Çünkü yukarıda söylendiği gibi C# programları Main() metodundan itibaren çalışmaya başlıyorlar. Yani, C# programlarının başlangıç noktası Main() metodudur. Main() metodunun sonuna gelindiğinde ise programın çalışması sona ermektedir.

Şimdi yukarıda verdiğimiz C# kodunda Mesaj() adını verdiğimiz metodu Main() metodunun içinden işleteceğiz. Mesaj() adını verdiğimiz statik metot ile, Main() metodu aynı sınıfın içinde oldukları için herhangi bir hazırlık yapmadan direk adıyla çağırıp işletmek mümkündür. Bu metodu nasıl işlettiğimizi aşağıda görebilirsiniz.

```
class Siftah
{
 static void Main(string[] args)
  {
   Mesaj();  // Mesaj() metodu işletilmektedir.
  }
 static void Mesaj()
  {
  }
}
```

Bu C# programını aşağıdaki gibi düzenlemiş olsaydık, yani önce "Mesaj()" sonra Main() metodunu yazsaydık değişen bir şey olmazdı. Çünkü C# derleyicisi Main() metodunu bulup programın başlangıç noktası yapmaktadır.

```
class Siftah
{
 static void Mesaj()
  {
  }
 static void Main(string[] args)
  {
   Mesaj();
  }
}
```

Şimdi Mesaj() adını verdiğimiz bu metotta değişiklik yapıp işletildiği zaman konsola bilgi yazılmasını sağlayacağız. Konsolda çalışan programlarda ekrana bilgi yazma işlemleri .NET Framework ile gelen **Console** sınıfının **Write**() ve **WriteLine**() metotları ile yapılmaktadır. Console sınıfına ait WriteLine() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
class Siftah
{
 static void Main(string[] args)
  {
   Mesaj();
  }
 static void Mesaj()
  {
   System.Console.WriteLine("Fenerbahçe");
  }
}
```

Bu şekilde düzenlenen C# programı konsolda derlenip hazırlanan EXE dosya çalıştırıldığında konsola ilgili bilgi yazıldıktan sonra programın çalışması sona erer. Çünkü Main() metodunun sonuna gelindiğinde C# programlarının çalışması sona etmektedir. Aşağıda verilen ekran görüntüsünü bu programı derleyip çalıştırdıktan sonra aldık.



Derleyip çalıştırdığımız C# programın işleyişi hakkında biraz konuşmak faydalı olacaktır. Konsolda derleme sonucu hazırladığınız EXE dosyanın adını yazıp Enter tuşuna basmanızla birlikte C# programı çalışmaya başlar. İlk olarak Main() metodu bulunup işletilmeye başlanır. Bizim programda Main() metodundaki tek satırda Mesaj() adını verdiğimiz metot çağrıldığı için programın işletimi Mesaj() metoduna geçer. Mesaj() metodundaki tek satırın gereği yapılıp sabit bilgi konsola yazıldıktan sonra işletim tekrar Main() metoduna geçer. Main() metodunda işletilecek başka satır olmadığından programın çalışması sona erer.



Son yıllarda biraz da akıllı telefonların etkisiyle en kısıtlı C# programına bile uygulama demek gibi bir alışkanlık edindik. Bu nedenle bu kitapta "C# uygulaması" ile "C# programı" aynı anlamda kullanılmaktadır.

Yukarıda Mesaj() metoduna yazdığımız satır hakkında biraz konuşmak gerekirse şunlar söylenebilir: "System.Console.WriteLine("Fenerbahçe");" satırındaki **System**, .NET Framework ile gelen bir namespace'tir. Konsola bilgi yazmak üzere hazırladığımız satırdaki **Console** ise bir sınıftır. Bu koddaki **WriteLine**(), **Console** sınıfının bir metodudur.

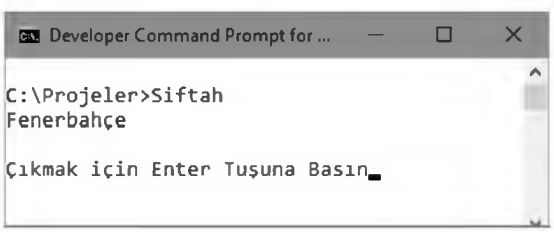
Şimdi yukarıda hazırladığımız bu kodda öyle bir ayarlama yapacağız ki programın çalışması Enter tuşuna basılıncaya kadar devam edecek. Programın çalışmasının Enter tuşuna basılıncaya kadar devam etmesini sağlamak üzere "Siftah" adını verdiğimiz sınıfın Main() metoduna 3 yeni satır yazdık.

```
class Siftah
{
 static void Main(string[] args)
  {
   Mesaj();
   System.Console.WriteLine(); // Boş bir satıra neden olur
   System.Console.Write("Çıkmak için Enter Tuşuna Basın");
   System.Console.Read();
  }
 static void Mesaj()
  {
   System.Console.WriteLine("Fenerbahçe");
  }
}
```

Gördüğünüz gibi **Console** sınıfının WriteLine() metodunu hem **Mesaj**() adını verdiğimiz metodun hem de Main() metodunun içinde kullandık. Bu örnekte Main() metodu içinde önce Mesaj() adını verdiğimiz metot çağrıldığı için ekrana "Fenerbahçe" yazılır. Mesaj() metodunun çağrıldığı satırdan sonra yine **WriteLine**() metodu kullanılmaktadır. Bu kez ekrana yazılmak üzere WriteLine() metoduna parametre olarak bilgi verilmediği için ekranın o satırı boş kalır. Bir sonraki satırda ise Console sınıfının WriteLine() yerine **Write**() metodunu kullandık.

WriteLine() metodu kendisine parametre olarak verilen bilgiyi konsola yazdıktan sonra imleci bir sonraki satırın başına almaktadır. Write() ise WriteLine() metodundan farklı olarak imleci bir sonraki satırın başına almıyor.

Burada programın çalışmasının kendiliğinden sona ermesini engelleyen metot ise **Read**(). Bu şekilde düzenlenen program derlenip hazırlanan EXE dosya çalıştırılırsa aşağıdaki gibi bir sonuç alınır. Bu ekran görüntüsünü aldığımız sırada programın işletimi Main() metodunun son satırında bekliyordu. Bu sırada Enter tuşuna basılırsa programın çalışması sona erer.



Console sınıfının **Read**() metodu kullanıcının konsola yazacağı bilginin konsoldan okunup bir değişkene aktarılması sırasında işlevsel olmaktadır. Bu örnekte kullanıcının konsola bilgi yazmasını istemediğimiz için **Read**() metodundan önce herhangi bir değişken adı yazmadık. Kullanıcının konsola yazdığı bilgilerin nasıl okunup program içinde nasıl kullanıldığından ileriki sayfalarda söz edilecektir.

Madem yukarıda programın çalışması hemen sona ermesin diye Console sınıfının Read() metodunu kullandık, hakkında biraz konuşmakta fayda vardır. Console sınıfının **Read**() metodunun işlevinin anlaşılmasını sağlamak için yukarıda hazırladığımız kodda Main() metodundan başka Mesaj() adında bir metot hazırlayıp bu metodu Main() metodu dahilinde işlettik.

Şimdi yazdığımız programda değişiklik yapıp "Mesaj()" adını verdiğimiz metodu silip daha önce tanımlayıp "yil" adını verdiğimiz değişkene çalışma anında bilgi aktarmaya çalışacağız. Bu amaçla "Siftah.cs" adını verdiğimiz kod dosyasının içeriğini aşağıdaki gibi düzenledik.

```
class Siftah
{
 static void Main(string[] args)
 {
  int yil;
  System.Console.Write("Hangi Yıldayız : ");
  yil = System.Console.Read();
 }
}
```

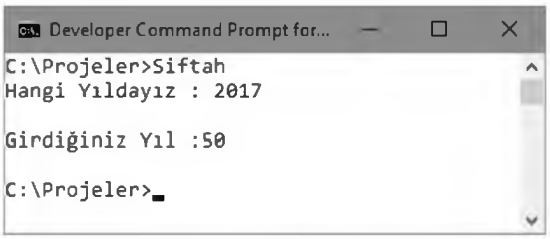
Bu şekilde değiştirilen C# programında önce **Console** sınıfının **Write**() metodu ile kullanıcıya mesaj verdik. Bir sonraki satırda ise kullanıcının konsola yazacağı bilgiyi **Read**() metoduyla okuyup "yil" adını verdiğimiz değişkene aktardık. Aşağıda verilen ekran görüntüsü bu program derlenip çalıştırıldıktan sonra alındı.



Konsola bilgi yazarken Write() yerine **WriteLine**() metodunu kullansaydık imleç bilgi girişi için bir sonraki satırda beklerdi. Şimdi bu programa ekleme yapıp Read() metodu sayesinde konsoldan okunup "yıl" değişkenine aktarılan bilgiyi Console sınıfının WriteLine() metodu ile tekrar konsola yazacağız.

```
class Siftah
{
 static void Main(string[] args)
 {
  int yil;
  System.Console.Write("Hangi Yıldayız : ");
  yil = System.Console.Read();
  System.Console.WriteLine();
  System.Console.Write("Girdiğiniz Yıl :");
  System.Console.WriteLine(yil);
 }
}
```

Bu şekilde düzenlenen C# kodu derlenirken hata meydana gelmediği gibi çalıştırıldığında da hata ile karşılaşılmaz. Ancak klavye kullanılarak konsola yazılan yıl bilgisi ile, sonradan Write() metodu ile konsola yazılan yıl bilgisi birbirinden farklı olur. Aşağıda verilen ekran görüntüsü bu C# programı derlenip çalıştırıldıktan sonra alındı.



Yukarıda verilen koda dikkat ederseniz konsoldan bilgi okuma işlemi **Read**() metodu ile yapılmaktadır. Read() metodu, konsoldan okuma yaparken konsola doğrudan yazılan bilginin ilk karakterini okuyup ASCII kodunu geriye göndermektedir. Daha önce az çok kod yazmış olanlar için bu sonuç ilginç değilken programcılıkta yolun başında olanlar için açıklama yapmamız gerekmektedir:

1. Önce "yil" adında ve int tipinde bir yerel değişken tanımladık.
2. Ardından Console sınıfının Write() metodu ile konsola mesaj yazıp "Hangi Yıldayız" sorusunu yönelttik.
3. Bu soruya karşılık kullanıcının klavyeyi kullanarak konsola yazacağı cevabı Console sınıfının Read() metodu ile okuyup "yil" değişkenine aktardık.
4. Sonrasında ise "yil" değişkenin içeriğini Console sınıfının WriteLine() metodu ile konsola yazdık.

Burada karşılaşılan sorun ise Read() metodunun çalışma şeklinden kaynaklanmaktadır. O sırada konsola ne kadar bilgi yazılırsa yazılsın, Read() metodu sadece bir karakteri okumaktadır. Ayrıca okuduğu bu karakterin ASCII kodunu geriye göndermektedir.

Yukarıda "Hangi Yıldayız" sorusuna 2017 ile cevap verdiğimiz için yazdığımız ilk karakter olan 2'nin ASCII kodu 50 sayısı Read() metodu tarafından "yil" değişkenine aktarıldı. "Hangi Yıldayız" sorusuna 1999 ile cevap vermiş olsaydık 1 rakamının ASCII kodu olan 49 sayısı "yil" değişkenine aktarılırdı.

Console sınıfının **ReadLine**() metodu ise farklı bir şekilde çalışmaktadır. ReadLine() metodu konsoldan okuduğu bilginin String tipte olduğunu varsaymaktadır. Bu nedenle yukarıda verilen kodda değişiklik yapıp okuma işlemini Read() yerine ReadLine() metodu ile yaptık.

```
class Siftah
{
 static void Main(string[] args)
  {
   int yil;
   System.Console.Write("Hangi Yıldayız :");
   yil = System.Console.ReadLine();
   System.Console.Write("Girdiğiniz Yıl :");
   System.Console.WriteLine(yil);
  }
}
```

Bu kodda "yil" adında bir değişken tanımladık.Değişken adı olarak yil'i kendi namı hesabına register etmiş birisi uyanda bulunursa bu kitabın bir sonraki baskısında "yil" adını verdiğimiz değişkene "yillar_yillar" veya "kaybolan_yillar" gibi bir ad verebiliriz.

Bu şekilde düzenlenen C# programı derlenmek istenirse derleyici hata rapor eder. Çünkü Console sınıfının **ReadLine**() metodu tarafından konsoldan okunup geriye gönderilen bilgi

string tipte olmasına rağmen burada **int** tipindeki değişkene aktarmaya çalışıyoruz. Bu hatanın önüne geçmek için int yerine **string** tipte bir değişken tanımlayıp kullandık. Bu şekilde düzenlenen C# programı derlenip çalıştırılırsa beklenen sonucu verir.

```
class Siftah
{
 static void Main()
  {
    string str; // string tipte değişken tanımlama
    System.Console.Write("Hangi Yıldayız : ");
    str = System.Console.ReadLine();
    System.Console.Write("Girdiğiniz Yıl :");
    System.Console.WriteLine(str);
  }
}
```

Kod Yazmaya Devam Ediyoruz

Yukarıdaki sayfalarda Not Defteri ile bir C# programı hazırlayıp "Siftah.cs" adıyla kaydettik. Devamında bu CS uzantılı C# programını konsolda derleyip "Siftah.exe" adında bir .NET uyumlu EXE dosya hazırladık ve çalıştırdık.

Şimdi yukarıda hazırladığımız kodda değişiklik yapıp kullanıcının konsola 2 sayısal bilgi yazmasını isteyip bunları birbirleri ile çarptıktan sonra sonucu yine konsola yazacağız. Bu amaçla 2 **string** değişken tanımlayıp **Console** sınıfının **ReadLine**() metodu ile konsola yazılan bilgileri okuyup bu değişkenlere aktardık.

```
class Siftah
{
 static void Main(string[] args)
  {
    string str1;
    string str2;
    System.Console.Write("İlk sayıyı girin : ");
    str1 = System.Console.ReadLine();
    System.Console.Write("İkinci sayıyı girin:);
    str2 = System.Console.ReadLine();
  }
}
```

Bu program bu hali ile kaydedilip derlenip çalıştırılırsa konsola yazılan bilgiler string değişkenlere aktarılır. Bu örnekteki amacımız kullanıcının sayısal bilgi girmesini ve bu sayıları birbiri ile çarpıp bulunan sonucu yine konsola yazmaktır. Bu nedenle matematiksel işlem öncesi string değişkenlerin içeriklerini **int** tipindeki değişkenlere aktarmak gerekecektir. Bu amaçla bu programa ekleme yapıp **int** tipinde 2 değişken daha tanımladık. Ardından string değişkenlerin içeriklerini dönüştürüp **int** tipindeki bu değişkenlere aktardık.

```
class Siftah
{
 static void Main(string[] args)
 {
  string str1;
  string str2;
```

```
    int sayi1;
    int sayi2;
    System.Console.Write("İlk sayıyı girin  : ");
    str1 = System.Console.ReadLine();
    System.Console.Write("İkinci sayıyı girin:);
    str2 = System.Console.ReadLine();
    sayi1 = System.Int16.Parse(str1); // Dönüştürme işlemi
    sayi2 = System.Int16.Parse(str2);
   }
}
```

Bu kodda sizin için yeni olan .NET Framework ile gelen **System** adlı namespace'de yer alan **Int16** yapısı ve bu yapının **Parse**() metodudur. Bu kitabın daha sonra verilecek bölümlerinde burada yapılan dönüştürme işlemlerinin ayrıntılı bir şekilde ele alınacaktır.

String bilgiyi dönüştürüp **int** tipindeki değişkene aktarırken bu yapının (struct) yani .NET Framework ile gelen Int16 yapısının **Parse**() metodunu kullandık. Bu şekilde konsoldan girilen bilgileri **Parse**() metodu ile dönüştürüp **int** tipindeki değişkenlere aktardıktan sonra matematiksel işlem yapabiliriz. Matematiksel işlem sonucu bulunacak sayısal değeri aktarmak üzere **int** tipinde 3. bir değişken tanımladık.

```
class Siftah
{
 static void Main(string[] args)
  {
   string str1, str2;
   int sonuc, sayi1, sayi2;
   System.Console.Write("İlk sayıyı girin  : ");
   str1 = System.Console.ReadLine();
   System.Console.Write("İkinci sayıyı girin: ");
   str2 = System.Console.ReadLine();
   sayi1 = System.Int16.Parse(str1);
   sayi2 = System.Int16.Parse(str2);
   sonuc = sayi1 * sayi2;
  }
}
```

Bu kodu dikkatice incelerseniz bir satırda aynı tipte birden fazla değişkeni tanımlamış olduğumuzu görürsünüz. Bu program derlenip çalıştırılıp sonra konsola yazılan 1. sayı ile 2. sayının çarpımı sonucu elde edilecek sayı 2.147.483.647'den büyük olursa hata meydana gelir. Çünkü **int** tipindeki değişkenlere 2.147.483.647'den büyük sayı aktarılamaz. Ayrıca bu kodda string bilgileri **Int16** yapısının **Parse**() metodu ile dönüştürdüğümüz için konsoldan girilen sayılar 32767'den büyük olmamalı. Int16 yerine **Int32** yapısının Parse() metodunu kullansaydık 1. veya 2. sayı 2.147.483.647 gibi büyük bir değer olabilirdi.

Çarpma işlemi sonucu bulunan sayısal değeri **Console** sınıfının **WriteLine**() metodu sayesinde direk konsola yazmak mümkündür. Buradaki amacımız bazı konuları anlatmak olduğu için üçüncü bir string değişken tanımlayıp bulunan sayısal değeri dönüştürüp bu string değişkene aktardık. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
class Siftah
{
 static void Main(string[] args)
 {
  string str1, str2, sonuc_str;
  int sonuc, sayi1, sayi2;
  System.Console.Write("İlk sayıyı girin   : ");
  str1 = System.Console.ReadLine();
  System.Console.Write("İkinci sayıyı girin: ");
  str2 = System.Console.ReadLine();
  sayi1 = System.Int32.Parse(str1);
  sayi2 = System.Int32.Parse(str2);
  sonuc = sayi1 * sayi2;
  sonuc_str = System.Convert.ToString(sonuc);
 }
}
```

Bu kodda **int** tipindeki değişkenin içeriğini **string** değişkene aktarabilmek için System'deki **Convert** sınıfının **ToString**() metodundan yararlandık. ToString() metodu ile int bilgiyi string'e dönüştürdükten sonra Console sınıfının WriteLine() metodu ile konsola yazdık.

```
class Siftah
{
 static void Main(string[] args)
  {
   string str1,str2, sonuc_str;
   int sonuc, sayi1, sayi2;
   System.Console.Write("İlk sayıyı girin   : ");
   str1 = System.Console.ReadLine();
   System.Console.Write("İkinci sayıyı girin: ");
   str2 = System.Console.ReadLine();
   sayi1 = System.Int32.Parse(str1);
   sayi2 = System.Int32.Parse(str2);
   sonuc = sayi1 * sayi2;
   sonuc_str = System.Convert.ToString(sonuc);
   System.Console.WriteLine("Sayıların çarpımı  :"+ sonuc_str);
   System.Console.Write("Devam etmek için bir tuşa basın");
   System.Console.Readkey();
  }
}
```

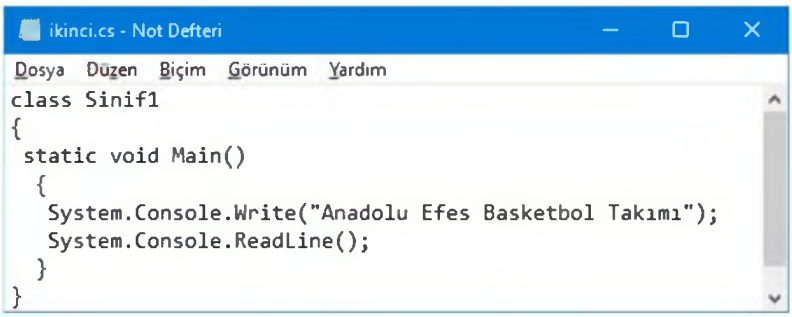
Namespace'ler Hakkında Önbilgi

C# programı yazarken Visual Studio'dan henüz yararlanmadığımız için kodumuzu Not Defteri ile hazırlayıp CS uzantılı dosyaya kaydediyoruz. Sonra bu CS uzantılı dosyayı konsolda derleyip .NET uyumlu EXE dosya hazırlayıp çalıştırıyoruz. Konsolla bilgi alış verişi yapan programlara **konsol uygulaması** adı verilmektedir.



Bu başlıkta namespace'ler hakkında önbilgi verileceği belirtilse bile namespace'lerle ilgili olarak buraya yazılacak bir sayfalık açıklamayı okuduğunuzda bile namespace kavramı kafanızda netleşmemiş olabilir. Bize sorarsanız bunu sorun yapmayın. Konular ilerledikçe, özellikle "Namespace Hazırlamak ve Kullanmak" adlı bölümü okuduktan sonra namespace kavramı sizin için bilinmez olmaktan çıkacaktır.

Namespace'ler hakkında bilgi vermek amacıyla hazırladığımız C# programı aşağıdadır. Ekran görüntüsü olarak verilen koda dikkat ederseniz bu kez class adı ile, kod dosyası adının farklı olduğunu görürsünüz. Bunu yapmamızın nedeni, C# programlama dilinin kod dosyası adı konusunda programcıyı serbest bıraktığına vurgu yapmaktır.



Bu şekilde hazırlanan C# programı derlenip çalıştırılırsa belirtilen bilgi Console sınıfının **Write**() metodu sayesinde konsola yazılır. Bir sonraki satırda kullanılan **ReadLine**() metodu sayesinde enter tuşuna basılıncaya kadar program çalışmaya devam eder. Yani, bu kodda sizin için yeni bir şey yoktur.

.NET Framework adı verilen sınıf kitaplığı ile gelen sınıfların **Namespace** adı verilen yapılarda toplandığını biliyorsunuz. Yazdığınız C# programlarında kullanmak istediğiniz class hangi namespace'in içinde yer alıyorsa o namespace'i **using** deyimi ile işaret etmelisiniz veya yukarıda yapıldığı gibi namespace'in adını class adıyla birlikte her seferinde yazmalısınız.

Konsolla ilgili **Console** sınıfı **System** adlı namespace'te yer aldığı için **Write**() metodundan önce ayrıca bu namespace'in adını yazdık. Söz konusu namespace'teki sınıfları programınızda çok kez kullanıyorsanız her seferinde namespace'in adını sınıf adının önüne yazmak yerine aşağıda yapıldığı gibi bu namespace'i **using** deyimi ile koda dâhil edebilirsiniz.

using deyimi ile olmayan bir namespace'in kaynaklarından yararlanmak istemeniz halinde hata meydana gelir. Aslında using deyimiyle burada koda dahil edilen bir şey yoktur. Birçok kaynakta using deyimi ile yapılan işlemden "koda dahil etme" diye söz edildiğinden dolayı böyle bir cümle kurduk. Şimdilik bu konunun geri planı üzerinde durmuyoruz.

```
using System;
Class Sinif1
{
 public static void Main()
  {
   Console.Write("Anadolu Efes Basketbol Takımı");
   Console.ReadLine();
  }
}
```

C# programlarınızda .NET Framework ile gelen namespace'lerdeki sınıfları kullanırken namespace kavramıyla yüz yüze gelmenin ötesinde kendi hazırladığınız class'ları bir namespace'in içine koymanız gerekiyor. Yukarıda bunu yapmamış olmamıza rağmen hata meydana gelmedi. Çünkü programcı kendi hazırladığı sınıfı bir namespace'e yerleştirmediği

zaman C# derleyicisi bunu geri planda kendiliğinden yapmaktadır. Şimdi yukarıda verilen programdaki "Sinif1" adlı class'ı dahil etmek üzere bir namespace hazırlayacağız. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
using System;
namespace Uzay1
{
 class Sinif1
  {
   public static void Main()
   {
    Console.Write("Anadolu Efes Basketbol Takımı");
    Console.ReadLine();
   }
  }
}
```

Verilen kodda görebileceğiniz gibi namespace hazırlama işlemi **namespace** anahtar kelimesi ile yapılmaktadır. Normalde programcı önce kendi namespace'ini hazırlar. Sonra bu namespace'in içinde istediği class'ları kodlar. Kod yazmaya namespace hazırlayarak başlamış olsaydık ilk yazdıklarımız aşağıdaki gibi olurdu. Devamında namespace'imize ait bloğun içinde class'ımızı yukarıda yaptığımız gibi kodlardık.

```
namespace Uzay1
{
}
```



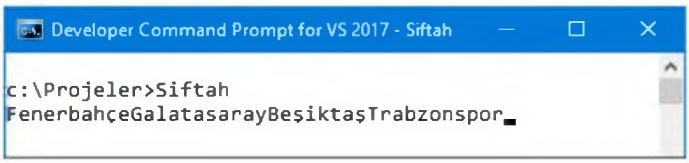
Namespace'lerle ilgili olarak söyleyeceklerimiz şimdilik bu kadardır. Daha sonra hem namespace hem de .NET programcılığıyla ilgili diğer kavramlara sahip olacaksınız. Yeter ki bu kitabı baştan sona kadar okuyup verilen basit örnekleri veya benzerlerini kendiniz kodlayın.

Write() ve WriteLine() Metotları

Yukarıda verilen örneklerde konsola yalnızca bir bilgi yazdık ve konsola yazılanları okuduk. Şimdi konunun anlaşılmasına katkı olması için arka arkaya 4 bilgiyi konsola yazacağız. Bu amaçla aşağıda verilen C# kodunu hazırladık.

```
using System;
namespace Uzay1
{
 class Sinif1
  {
   public static void Main()
   {
     Console.Write("Fenerbahçe");
     Console.Write("Galatasaray");
     Console.Write("Beşiktaş");
     Console.Write("Trabzonspor");
     Console.ReadLine();
   }
  }
}
```

Bu kısa programda **Write**() metodu ile konsola 4 bilgi arka arkaya yazılmakta ve **ReadLine**() metodu ile programın işletimi Enter tuşuna basılıncaya kadar askıya alınmaktadır. Write() metodu ile konsola bilgi yazıldıktan sonra imleç yazma işleminin tamamlandığı yerde bekleyeceği için bu program derlenip çalıştırılırsa aşağıdaki gibi bir sonuç alınır.



Bu bilgilerin ayrı satırlara yazılmasını sağlamak için Write() yerine **WriteLine**() metodunu kullanabilirsiniz. **WriteLine**() metodu kendisine parametre olarak verilen bilgiyi konsola yazdıktan sonra imleci bir sonraki satırın başına almaktadır.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  public static void Main()
  {
   Console.WriteLine("Fenerbahçe");
   Console.WriteLine("Galatasaray");
   Console.WriteLine("Beşiktaş");
   Console.WriteLine("Trabzonspor");
   Console.ReadLine();
  }
 }
}
```

Write() metodu ile konsola yazılan metnin sonuna satır başı karakterini("\n") ekleyerek aynı sonucu alabilirsiniz. Satır başı karakterinin nasıl kullanıldığını aşağıda görebilirsiniz.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  public static void Main()
  {
   Console.Write("Fenerbahçe \n");
   Console.Write("Galatasaray \n");
   Console.Write("Beşiktaş \n");
   Console.Write("Trabzonspor \n");
   Console.ReadLine();
  }
 }
}
```

Şimdiye kadar yazılanlara bakılırsa, Console sınıfının Write() ve WriteLine() metotlarını sıklıkla kullanıyoruz. Gerçekte, bu 2 metot daha çok C# programlama dilini öğrenme sürecinde kullanılmaktadır. Yoksa yazılım firmaları tarafından hazırlanıp kullanıma sunulan C# uygulamalarında Console sınıfının metotları pek kullanılmaz.

Madem, bu kitabı okurken veya C# hakkında araştırma yaparken Console sınıfının metotları sıklıkla karşınıza çıkacak, bu metotlar hakkında bilgi edinmekte fayda vardır. Console sınıfının WriteLine() metodu hakkında bilgi vermek için üzerinde çalıştığımız C# kodunu Not Defteri ile aşağıdaki gibi düzenledik.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
  {
   string takim = "Adana Demirspor";
   int kurulus = 1940;
   string stadyum = "5 Ocak";
   Console.ReadLine();
  }
 }
}
```

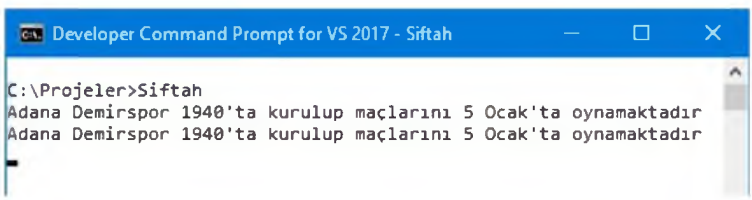
Bu sırada elimizde ikisi string, birisi int tipinde olan 3 değişken var. Amacımız konsola şöyle bir bilgi yazmak: "Adana Demirspor 1940'ta kurulup maçlarını 5 Ocak'ta oynamaktadır." Bu bilgiyi konsola yazdırmak için ilk akla gelen yöntem Console sınıfının WriteLine() metodunu aşağıdaki gibi kullanmaktır.

```
using System;
namespace Uzay1
{
 class Sinif1
  {
   static void Main()
   {
    string takim = "Adana Demirspor";
    int kurulus = 1940;
    string stadyum = "5 Ocak";
    Console.WriteLine(takim + " " + kurulus +
         "'ta kurulup maçlarını " + stadyum +
             "'ta oynamaktadır");
    Console.ReadLine();
   }
 }
}
```

İlk bakışta ne yaptığı anlaşılan bu satır istenen sonucu verir. Ancak çok sayıda programcı bu sonucu almak üzere WriteLine() metodunu aşağıdaki gibi kullanmayı tercih etmektedir. Çünkü bu şekilde kullanımda çok hata yapılmaktadır. Çok sayıda tırnak(") ve + operatörüne yer vermek gerekmektedir. WriteLine() metodunun bu 2. tip kullanımında metoda ilk parametre olarak string bir bilgi verilmektedir. String bilgi içinde bir değişkenin içeriği kullanılmak istendiğinde {} süslü parantezlere yer verilmektedir. Kod satırı fazla uzamasın diye string bilgiyi 2 ayrı parçaya böldük.

```
namespace Uzay1
{
 class Sinif1
  {
   static void Main()
   {
    string takim = "Adana Demirspor";
    int kurulus = 1940;
    string stadyum = "5 Ocak";
    Console.WriteLine(takim + " " + kurulus +
        "'ta kurulup maçlarını " + stadyum +
            "'ta oynamaktadır");
    Console.WriteLine("{0} {1}'ta kurulup maçlarını {2}'ta " +
        "oynamaktadır", takim, kurulus, stadyum);
   }
  }
}
```

String bilgi içinde kullanılan süslü parantezlerin içine 0'dan başlamak üzere içeriği kullanılmak istenen değişkenlere sıra numarası verilmektedir. Konsola yazmak istediğimiz mesajın içinde 3 değişkenin içeriğine yer vermek istediğimiz için string bilginin içinde {} parantezlere 3 kez yer verdik. String bilginin tırnağı kapatıldıktan sonra içeriği kullanılan değişkenlerin adları sıra numaralarına uygun bir şekilde aralarına virgül konularak yazılır. Bu şekilde düzenlenen kod derlenip çalıştırılırsa aşağıdaki gibi bir sonuç alınır.



Write() veya WriteLine() metodunu bu şekilde hatasız kullanabilmek için denemeler yapıp alışkanlık kazanmak gerekiyor. Birçok programcıya göre WriteLine() metodunun 2. kullanım şekli de pratik değildir. Bu nedenle Microsoft firması C# 6.0'da WriteLine() metodu için 3. kullanım şeklini kullanıma sundu. **Interpolated Strings** adı verilen bu yeniliğin nasıl kullanıldığını aşağıda görebilirsiniz.

```
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
   {
    string takim = "Adana Demirspor";
    int kurulus = 1940;
    string stadyum = "5 Ocak";
    Console.WriteLine(
       $"{takim} {kurulus}'ta kurulup maçlarını {stadyum}'ta "
         + "oynamaktadır");
   }
 }
}
```

C# 6.0 ile gelen yenilik sayesinde string bilginin içinde değişkenin sıra numarası yerine adını yazmak mümkün olmaktadır. String bilginin başına $ işareti koymak gerektiğini fark etmiş olmalısınız. String bilginin başına $ yazmakla şunu demiş oluyoruz: Bu bilginin içinde değişkenler bulunmaktadır ve bu değişkenlerin içeriğini kullan. Interpolated Strings yeniliği hakkında biraz daha konuşmak için WriteLine() metodu ile konsola yazdığımız metni önce bir değişkende topladık, sonra WriteLine() metoduna parametre olarak verdik.

```
using System;
class Sinif1
{
 static void Main()
 {
  string takim = "Adana Demirspor";
  int kurulus = 1940;
  string stadyum = "5 Ocak";
  string mesaj =
       $"{takim} {kurulus}'ta kurulup maçlarını {stadyum}'ta" +
              " oynamaktadır";
   Console.WriteLine(mesaj);
  }
}
```

Bu şekilde düzenlenen C# kodu derlenip çalıştırılırsa yine aynı sonuç alınır. Kodun son halini dikkatlice incelerseniz "mesaj" adını verdiğimiz string değişkene aktardığımız bilginin uzun olması nedeniyle kod satırını kısaltmak amacıyla 2 parçaya bölüp "+" operatörü ile birleştirdik. Şimdi bu string bilgiyi farklı bir yerden böleceğiz.

```
using System;
namespace Uzay1
{
 class Sinif1
  {
   static void Main()
   {
    string takim = "Adana Demirspor";
    int kurulus = 1940;
    string stadyum = "5 Ocak";
    string mesaj = $"{takim} {kurulus}'ta kurulup maçlarını "+
              "{stadyum}'ta oynamaktadır";
    Console.WriteLine(mesaj);
  }
 }
}
```

Bu kod hataya neden olur. Bu kodun yukarıdan verilenden farkı, içinde değişken olan {} parantezlerden sonuncusunu bir sonraki satıra almış olmamızdır. Derleyici burada 2 string bilginin olduğunu ve bunların "+" ile birleştirdiğini düşünmektedir. Birinci parçanın başında "$" varken ikincisinde yoktur. Dolayısıyla ikinci parçadaki değişken adı hataya neden olur. Bu hatanın çözümü burada kullandığımız string bilginin 2. parçasının başına da $ koymaktan geçiyor. Bu nedenle aşağıda verilen kod hata üretmez ve istediğimiz sonucu verir.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
   {
    string takim = "Adana Demirspor";
    int kurulus = 1940;
    string stadyum = "5 Ocak";
    string mesaj = $"{takim} {kurulus}'ta kurulup maçlarını "+
              $"{stadyum}'ta oynamaktadır";
    Console.WriteLine(mesaj);
   }
 }
}
```

Statik Olan-Olmayan Metotlar

Yukarıda belirtildiği üzere en küçük C# uygulamasını bile **Class** olarak düzenlemek gerekiyor. Örneğin yukarıda verilen kodda "Sinif1" adında bir class hazırlayıp bu class'ı "Uzay1" adını verdiğimiz namespace'e yerleştirdik. Ardından bu class dahilinde hazırladığımız statik Main() metodu sınırları içinde .NET Framework ile gelen **Console** sınıfının statik Write() ve ReadLine() metotlarını kullandık. Başka bir deyişle şimdiye kadar hep statik metotları kullandık ve hazırladığımız metotlar statikti. Statik olmayan metotlar nasıl kullanılır sorusuna cevap verebilmek için önce aşağıda verilen kodu hazırladık.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
   {
    Console.Write("Karekökü alınacak sayıyı girin :");
    string str = Console.ReadLine();
    int number = Convert.ToInt32(str);
    double karekok = System.Math.Sqrt(number);
    Console.Write("Sayının Karekökü : ");
    Console.Write(karekok);
   }
  }
}
```

Bu kodun üzerinde biraz konuşmamız gerekecek. İlk olarak System'de yer alan **Console** sınıfının statik **Write**() metodu ile kullanıcıya konsol aracılığıyla mesaj verip karekökü alınacak sayıyı konsola yazmasını istedik. Devamında kullanıcının konsola yazdıklarını okuyup "str" adını verdiğimiz string değişkene aktardık.

Kullanıcının konsola yazdıklarını okuyup string değişkene aktarırken Console sınıfının statik olan **ReadLine**() metodundan yararlandık. Devamında aşağıda verilen satır ile kullanıcının konsola yazdığı bilgiyi **Convert** sınıfının yine statik olan **ToInt32**() metodu ile "int" tipine dönüştürüp "number" adlı değişkene aktardık.

```
int number = Convert.ToInt32(str);
```

Tabi kullanıcı burada **int** tipine dönüştürülemeyecek bilgiyi konsola yazarsa dönüştürme sırasında hata meydana gelir ve programın çalışması kırılır. Kullanıcının mesaja uyup karekökü alınabilinecek sayı gireceğini varsayıyoruz.

Kodun devamında .NET Framework ile gelen **Math** sınıfının statik **Sqrt()** metodundan yararlanıp int tipindeki bu değişkenin içeriğinin karekökünü alıp "karekok" adını verdiğimiz **double** tipindeki değişkene aktardık.

Gördüğünüz gibi statik metotları kullanmak için herhangi bir hazırlık yapmak gerekmiyor; önce sınıfın adını sonra işletilmek istenen statik metodun adını yazmak yeterli oluyor. Tabi bazılarınız keşke .NET Framework ile gelip C# kodu içinde kullandığımız sınıfların bütün metotları statik olsa daha iyi olmaz mıydı diye sorabilirler. Hayır, daha iyi olmazdı. Çünkü o zaman nesneye yönelik programcılığın ilkelerinden uzaklaşılmış olunurdu.

Statik olmayan metotları kullanma şekli oldukça farklıdır. Statik olan ve olmayan metotların arasındaki farkı ortaya çıkarmak için aşağıdaki gibi 2 class'a sahip bir uygulama hazırladık. Şimdiye kadar hazırladığımız bütün C# programları bir class'a sahipti. Bu 2 class'ı da aynı namespace'e üye yaptığımızı fark etmiş olmalısınız.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
   {
   }
 }
 class Sinif2
 {
  public static int Kare(int number)
  {
   return number * number;
  }
 }
}
```

Bu kodda dikkatinizi "Sinif2" adını verdiğimiz class'taki **Kare()** metoduna çekmek istiyoruz. Bu metotta ilk fark etmenizi isteğimiz ayrıntı, bu metodun adının önüne **static** yazılmış olmasıdır. İkinci önemli nokta, bu metodun geriye **int** tipinde bilgi göndermesidir. Şimdi bu metodun imzasında "public static int Kare(int number)" yani ilk satırında kullandığımız kelimeler üzerinde biraz konuşacağız.

- **public** : Erişim belirleyicisi olarak adlandırılan bu kelime, burada "Sinif2" adını verdiğimiz sınıftaki bu metodun uygulamadaki ilk sınıfta yani Main() metoduna sahip diğer sınıfta kullanılabilinmesinin önünü açıyor.
- **static:** Bu anahtar kelime bu metodun statik olmasını sağlıyor. Bu kısımda asıl amacımız bu anahtar kelime olduğu için bu maddeye fazla bir şey yazmadık.
- **int**: Şimdiye kadar kullandığımız metotlar geriye bilgi göndermedikleri için **void** metotlardı. Başka bir deyişle geriye değer göndermeyen metotların adının önüne yukarıdaki sayfalarda **void** yazıyorduk. "Kare" adını verdiğimiz metodun geriye int tipinde bilgi göndermesini istediğimiz için bu metodun adından önce **int** yazdık.

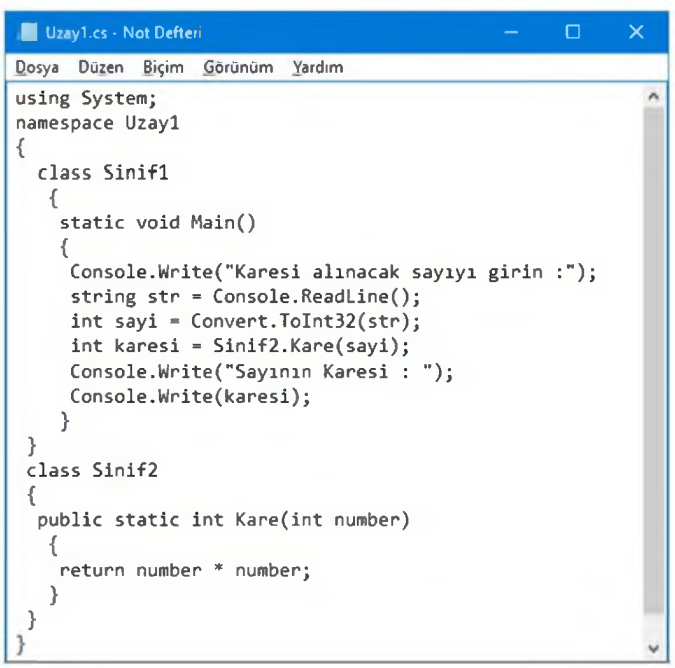
- **Kare**: Metoda verdiğimiz ad. C# programcılarının metotlara ad olarak seçtikleri kelimelerin ilk harfini büyük olarak seçmek gibi bir tercihleri vardır.
- **int number**: Hazırladığınız metot dışarıdan bilgi alacaksa metot adından sonra parantezin içinde değişken tanımlar gibi parametre tanımlamak gerekir. "Kare" adını verdiğimiz bu metot dışarıdan int tipinde bir bilgi aldığı için bu tipte bir parametreye sahip olmasını sağladık. Bu parametreye "number" adını verdik.

C# programlama dilinde metotlarda geriye bilgi gönderime işlemi **return** anahtar kelimesi ile yapılmaktadır. Bu metotla istenen sayının karesini bulmak istediğimiz için parametreyi kendisi ile çarpıp bulunan sonucu **return** anahtar kelimesi ile geriye gönderdik.

Şimdi sırada bu metottan yani burada 'Sinif2' adını verdiğimiz sınıftaki Kare() metodundan yararlanmak var. Bu amaçla konsol uygulamasındaki "Sinif1" adını verdiğimiz sınıftaki Main() metodunu aşağıdaki gibi düzenleyerek **Kare**() metodunu işlettik. Hangi sınıfın hangi statik metodu kullanılmak isteniyorsa önce sınıfın adı, sonra metodun adı yazılmalıdır.

```
static void Main()
{
 Console.Write("Karesi alınacak sayıyı girin : ");
 string str = Console.ReadLine();
 int sayi = Convert.ToInt32(str);
 int karesi = Sinif2.Kare(sayi);
 Console.Write("Sayının Karesi : ");
 Console.Write(karesi);
}
```

Bu kod üzerinde konuşurken bazı ayrıntılardan söz edebilmek için Not Defteri ile hazırladığımız kodun tümünü ekran görüntüsü olarak buraya aldık. CS dosyasına ad seçerken namespace'e verdiğimiz adı kullandık.



Bu uygulamayı 2 sınıf olarak düzenlediğimizi ve bu 2 sınıfı da "Uzay1" adını verdiğimiz namesepace'e üye yaptığımızı biliyorsunuz. Bu uygulamayı Visual Studio ile hazırlasaydık değişen bir şey olmazdı. Bazı ayrıntıları dikkatinize sunmak için Not Defterini tercih ettik.

Bu şartlarda 2 class içeren bu CS dosyası konsolda derlenip çalıştırılırsa beklenen sonuç alınır. Yani derleme sonucu hazırlanan "exe" dosya çalıştırıldığında ilk sınıftaki Main() metodu işletilir. Bu metot dahilinde ikinci sınıftaki Kare() metodu işletilir ve istenen sayının karesi bulunup konsola yazılır. Burada işlettiğimiz Kare() metodu başka bir sınıfın içinde olduğu için metodu işletirken metot adından önce sınıfın adını yazdık.

```
int karesi = Sinif2.Kare(sayi);
```

Buradaki amacımız deneysel; yoksa Kare() adını verdiğimiz metodu ilk sınıfın içinde kodlayabilirdik. Burada Kare() metodunu işlettiğimiz sınıf başka bir namespace'e üye olsaydı sınıf adından önce namespace adını da yazardık.

Statik metotlarla ilgili bazı ayrıntılardan söz edebilmek için bu 2 sınıfı ayrı kod dosyalarına aldık. İlk class'ın yer aldığı kod dosyasını "Sinif1.cs" ve ikinci class'ın bulunduğu kod dosyasının adını "Sinif2.cs" olarak seçtik. Yaptığımız ayarlamayı net olarak görebilmeniz için bu kod dosyalarının içeriğini ekran görüntüsü olarak verdik.

Sinif1.cs - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
  {
   Console.Write("Karesi alınacak sayı:");
   string str = Console.ReadLine();
   int sayi = Convert.ToInt32(str);
   int karesi = Sinif2.Kare(sayi);
   Console.Write("Sayının Karesi : ");
   Console.Write(karesi);
  }
 }
}
```

Sinif2.cs - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
namespace Uzay1
{
 class Sinif2
 {
  public static int Kare(int number)
   {
    return number * number;
   }
 }
}
```

Bu şartlarda konsolda derleme yaparken hem "Sinif1.cs" hem de "Sinif2.cs" dosyasının adını yazarsak derleme hatasız gerçekleşir ve adı ilk yazılan kod dosyası ile aynı ada sahip bir EXE dosya hazırlanır. Hazırlanan EXE dosya çalıştırıldığında program istenen sonucu verir.

Administrator: Developer Command Pr...

```
C:\Projeler>csc Sinif1.cs Sinif2.cs
```

Bu derleme işleminden sonra EXE dosya çalıştırılır çalıştırılmaz Sinif2'deki Kare() metodu çağrılıp işletilmeye hazır haldedir. Çünkü bu metot statik olduğu için işletebilmek için hazırlık yapmak gerekmiyor. Bu uygulamada başka class'lar olsaydı onlar da bu metottan yararlanabilirdi.

Varsayalım ki çok sayıda sınıfa ve her sınıfında çok sayıda metodun bulunduğu bir uygulama yazmanız gerekiyor. Uygulamadaki bütün metotları statik yapıp sonra derleyip çalıştırırsanız bütün metotlar işletilmek üzere bellekte hazır olarak bekler. Bu istenen bir durum değildir. Bu nedenle nesneye yönelik programcılıkta metotların gerekmedikçe statik yapılması istenmez.

Şimdi yukarıda Sinif2'deki "Kare" adını verdiğimiz metodun normal metot olmasını sağlayacağız. Başka bir deyişle metot adının önüne yazdığımız **static** anahtar kelimesini sileceğiz. Metodun gerisini aynı bıraktık.

```
class Sinif2
{
 public int Kare(int number)
  {
   return number * number;
  }
}
```

Bu andan itibaren bu metot yukarıdaki gibi kullanılamaz. Kare() adını verdiğimiz bu metodu kullanabilmek için önce ilgili sınıfın örneğini almak, başka bir deyişle ilgili sınıftan yola çıkıp nesne hazırlamak gerekiyor. "Sinif2" adlı sınıfın örneğini nasıl aldığımızı, yani bu sınıftan yola çıkıp nasıl nesne hazırladığımızı aşağıda görebilirsiniz.

```
class Sinif1
{
 static void Main()
  {
   Sinif2 nesne;
   nesne = new Sinif2();
  }
}
```

Burada Main() metodu içinde önce "Sinif2" tipinde bir değişken tanımladık. Devamında **new** anahtar kelimesi ile bu sınıfın örneğini alıp (yani nesne hazırladık) tanımladığımız değişkene aktardık. Bu andan itibaren Sinif2'den yola çıkarak hazırladığımız nesnenin üzerinden **Kare** metodunu kullanabiliriz. Bu metodu nasıl işlettiğimizi aşağıda görebilirsiniz.

```
static void Main()
{
 Sinif2 nesne;
 nesne = new Sinif2();
 Console.Write("Karesi Alınacak sayı :");
 string str = Console.ReadLine();
 int sayi = Convert.ToInt32(str);
 int karesi = nesne.Kare(sayi);
 Console.Write("Sayının karesi :");
 Console.Write(karesi);
}
```

Bu kodda Kare() adını verdiğimiz metodu işletirken karesini almak istediğimiz sayıyı parantezlerin arasına yazdık. C#'ta metotların sonuna mutlaka "()" parantez eklenir. Yani Kare() metodu herhangi bir parametreye sahip olmasaydı bile metot adına parantez eklemek gerekirdi. Şimdi dikkatinizi sınıftan yola çıkıp nesne hazırladığımız 2 satıra çekmek istiyoruz.

```
Sinif2 nesne;
nesne = new Sinif2();
```

Her ne kadar bu 2 satırı bir tek satırda birleştirmek mümkün olsa bile asıl işaret etmek istediğimiz **new** anahtar kelimesinin kullanıldığı satırda sınıf adına eklenen parantezlerdir. "Sinif2" adında bir sınıfımız olsa bile burada sonuna "()" parantez eklenen bir sınıf olmayıp bir metottur. Programcılar tarafından hazırlanan sınıfların sınıfla aynı ada sahip bir metodu genellikle bulunmakta ve bu metoda sınıfın **yapıcı metodu (constructor)** adı verilmektedir. Bu metotlar sınıftan yola çıkıp nesne hazırlanmak istendiği zaman kullanılmaktadır.

Dikkat ederseniz yukarıda "Sinif2" için böyle bir metot hazırlamadık. Programcı kendi sınıfı içinde sınıfla aynı ada sahip yapıcı metodu hazırlamadığı zaman C# derleyicisi geri planda standart özelliklere sahip bir yapıcı metodu otomatik olarak hazırlamaktadır.

Şimdi yukarıda verilen "Sinif2" adlı class'ın yapıcı metoda sahip olmasını sağlayacağız. Bu şartlarda herhangi işletilebilir bir satıra sahip olmayan bu metodun varlığı ile yokluğu arasında bir fark yoktur.

```
class Sinif2
 {
  public int Kare(int number)
   {
    return number * number;
   }
 public Sinif2() // Yapıcı metot
  {
  }
}
```

Şimdi yukarıda statik olmayan metotla ilgili olarak yaptığımız işlemleri maddeler halinde tekrar anlatacağız:

- Üzerinde çalıştığımız C# programında başlangıçta sadece bir class vardı ve bu class'ı herhangi bir namespace'e üye yapmamıştık. Dolayısıyla derleme sırasında derleyici geri planda bize bir namespace hazırlamıştı.
- Uygulamadaki tek class'ta Main() metoduna yer vermiş ve Main() metodu içinde .NET Framework ile gelen Console sınıfının birkaç statik metodunu kullanmıştık. Derleyici Main() metodunun statik olmasını istediği için metodun adından önce **static** anahtar kelimesine yer vermiştik. Main() metodunu statik olarak ayarlamasaydık derleme sırasında hata meydana gelirdi.
- Devamında uygulamamızın gerek duyduğu namespace'i hazırlama işini derleyiciye bırakmayıp kendimiz bir namespace hazırlamış, "Sinif1" ve "Sinif2" adlı class'larımızı bu namespace'e üye yapmıştık.

- Devamında "Sinif1" ve "Sinif2" adını verdiğimiz sınıfları ayrı kod dosyalarına aldık. Sinif1'i içeren kod dosyamıza Sinif1.cs adını verdik. Benzer şekilde Sinif2'yi Sinif2.cs adlı dosyasına taşıdık.
- Uygulamamızdaki 2. sınıfta "Kare()" adında statik olmayan bir metot hazırlamış ve bu metodu ilk class'taki Main() metodu içinde işletebilmek için Sinif2'den yola çıkıp **new** anahtar kelimesi ile Sinif2 tipinde bir nesne hazırlamıştık.
- new anahtar kelimesi ile bir sınıfın örneği alınırken yani o sınıf tipinde nesne hazırlanırken geri planda sınıfın yapıcı metodu işletiliyordu. Programcı sınıfı için sınıfla aynı ada sahip yapıcı metodu hazırlamadıysa geri planda derleyici standart özelliklere sahip bir yapıcı metodu hazırlıyordu.

Yukarıdaki sayfalarda statik olan-olmayan metotlarla ilgili olarak yaptığımız işlemleri bu şekilde özetledikten sonra kaldığımız yerden anlatmaya devam edeceğiz. Şimdi konu üzerinde düşünmenizi sağlamak için yukarıda Sinif2'de hazırladığımız Kare() adlı metodu ilk sınıfa taşıyıp "Sinif2" adını verdiğimiz sınıfı sileceğiz. Uygulamamızın son hali tek kod dosyası olarak aşağıdaki gibidir.

```
using System;
namespace Uzay1
 {
  class Sinif1
   {
    static void Main()
    {
     Console.Write("Karesi Alınacak sayı :");
     string str = Console.ReadLine();
     int sayi = Convert.ToInt32(str);
     int karesi = Kare(sayi); // hata meydana gelir.
     Console.Write("Sayının karesi :");
     Console.Write(karesi);
     Console.Read();
    }
   public int Kare(int number)
    {
     return number * number;
    }
  }
}
```

Bu şekilde düzenlenen uygulama konsolda derlenmek istenirse derleyici hata rapor edip derleme işlemini yapmaz. Hataya, Main() metodu içinde Kare() metodunun işletilmek istendiği satır neden olur. Çünkü Kare() metodu statik olmayan bir metot olmasına rağmen burada statik metotlar gibi işletilmek istenmektedir. Çözüm, "Sinif1" tipinde bir nesne hazırlamaktan geçer. Başka bir anlatımla burada işletmek istediğimiz Kare() metodunu ancak üyesi olduğu sınıftan yola çıkılarak hazırlanan nesne üzerinden işletebiliriz

Sinif1'in içinde iken "Sinif1" tipinde nesne hazırlayıp bu sınıfın metotlarını kullanmak ilk bakışta size ters gelebilir. Bize sorarsanız şimdilik bunun üzerinde durmayın. Elimizde nesne üzerinden işletilebilinecek, yani statik olmayan bir metot var. Dolayısıyla işletmek istediğimiz metodun üyesi olduğu sınıf tipinde bir nesneye ihtiyacımız var. Bu nesneyi nasıl hazırladığımızı aşağıda görebilirsiniz.

```
using System;
namespace Uzay1
{
 class Sinif1
 {
  static void Main()
  {
   Console.Write("Karesi Alınacak sayı :");
   string str = Console.ReadLine();
   int sayi = Convert.ToInt32(str);
   Sinif1 s1 = new Sinif1();
   int karesi = s1.Kare(sayi);
   Console.Write("Sayının karesi :");
   Console.Write(karesi);
   Console.Read();
  }
  public int Kare(int number)
  {
    return number * number;
  }
 }
}
```

Konunun geri planı hakkında net bilgiye sahip olmanız için tekrar niyetini bu uygulama hakkında maddeler halinde biraz konuşacağız:

- Uygulamamız "Uzay1" adını verdiğimiz namespace olarak düzenlenmiştir. C# uygulamalarının namespace olarak düzenlenmesi şarttır. Bu şartı programcı yerine getirmediğinde namespace olarak düzenleme işi derleyiciye kalıyor. Sizler C# uygulaması geliştirirken Visual Studio kullanacağınız için namespace hazırlama işlemini Visual Studio yapacaktır.

- Uygulamamızda bir tek sınıf bulunmaktadır. Dolayısıyla uygulamanın başlangıç noktası Main() metodu bu sınıfta bulunmaktadır. C# programcılarının Main() metodunu string tipte ve "args" adında bir parametreye sahip olması gibi bir tercihleri olmasına rağmen burada Main() metodunu parametresiz olarak düzenledik.

- "Sinif1" adını verdiğimiz sınıfımızda Main'den başka bir de Kare() adında statik olmayan yani nesne üzerinden erişilen bir metodumuz daha var. Geriye int tipinde bilgi gönderen bu metodumuz int tipinde bir parametreye sahiptir.

- Kare() metodunu Main() metodu içinde yani aynı sınıf dahilinde işletmek istediğimiz için metodun üyesi olduğu sınıf tipinde bir nesneye gerek olduğu için new anahtar kelimesinden yararlanıp "Sinif1" tipinde nesne hazırladık.

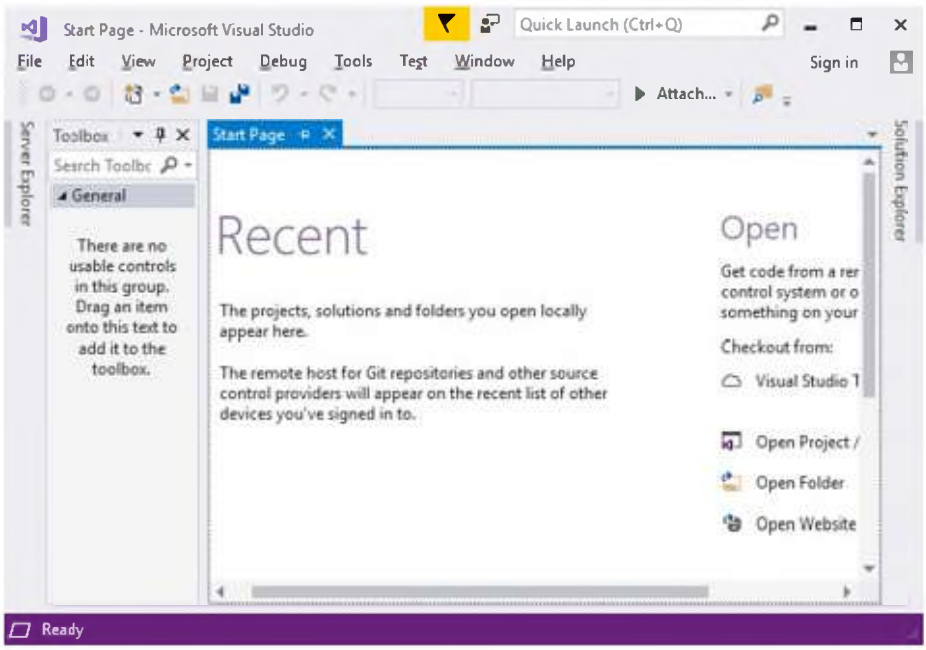
Bu şekilde C# programlama dili hakkında ön bilgi verdikten sonra bir sonraki bölümde Visual Studio ile çalışmaya başlayacağız. Visual Studio, C# programları yazarken size kolaylıklar sağlayan bir araçtır, bir IDE'dir. Kitabın sayfa sayısını azaltmak için Visual Studio'nun anlatıldığı bir sonraki bölümü PDF haline getirip ek olarak verdik.

2. BÖLÜM

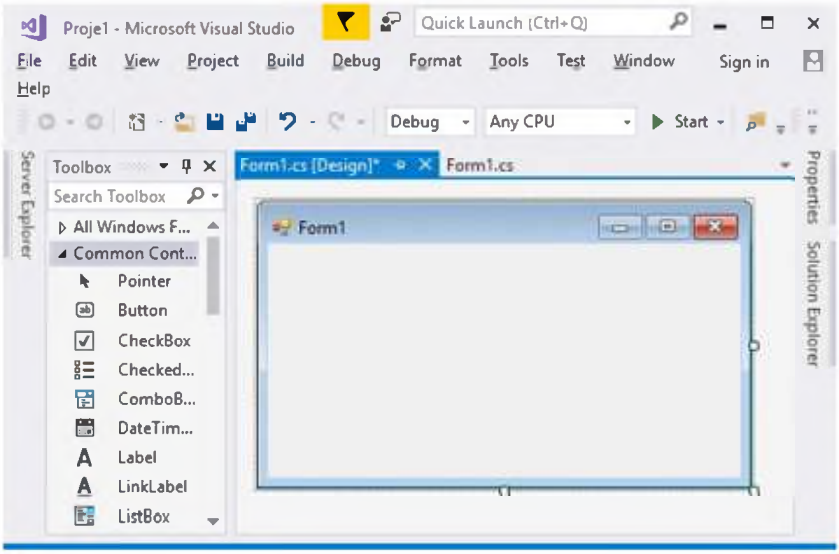
C# İLE PROGRAMCILIĞA GİRİŞ

Bu bölümde, kitaptaki diğer bölümlerinden farklı olarak birden fazla konudan söz edildi. Bazen bir paragrafta her kaynakta bulunabilinen ve kolay anlaşılan bir işlemden söz edilirken bir sonraki paragrafta söz konusu işlemin geri planı hakkında teknik açıklamalar yapıldı. Daha önce Visual Studio'dan yararlanıp ile az çok kod yazanlar bu bölümdeki konu başlıklarına bakıp yazılanları dikkatlice okumadan sayfaları hızlıca geçmek isteyebilirler. Örneğin birçok kişinin nazarında gerektiğinde forma bir düğme yerleştirilir, sonra bu düğme tasarım anında çift tıklanır ve istenen kodlar yazılır. Aslında bu yorum yanlış değildir. Ne var ki programcı işin geri planını bilmek zorundadır.

Visual Studio başlatıldığında aşağıdaki gibi bir pencere ekrana gelmektedir. Bu penceredeki seçeneklerden yararlanıp yeni proje hazırlayabilir veya mevcut projeleri açabilirsiniz. Visual Studio, daha önce açılıp üzerinde çalışılan birkaç projeyi **Start Page**'de listelemektedir.



Açmak istediğiniz proje Start Page'de listelenmiyorsa **Open Project** linkini tıklayıp **Open Project** diyalog kutusunu ekrana getirmelisiniz. Open Project diyalog kutusunda istenen sürücü ve klasördeki "csproj" uzantılı proje veya "sln" uzantılı solution dosyası seçilip **Open** düğmesi tıklanarak proje açılabilir. Aşağıda verilen ekran görüntüsünü daha önce hazırladığımız standart özelliklere sahip "Proje1" adlı projeyi açtıktan sonra aldık.





Burada yaptığımız gibi Visual Studio'yu başlatıp Windows Forms Application şablonu ile yeni proje hazırlasanız bile Visual Studio penceresinin görünümü bundan farklı olabilir. Bize sorarsanız bunu sorun yapmayın. Diğer yandan Visual Studio'nun **Window** menüsünden **Reset Layout Window** komutunu verip Visual Studio'nun pencere yerleşim şeklini orijinal haline döndürebilirsiniz.

Open Project diyalog kutusunda "csproj" uzantılı dosya yerine "sln" uzantılı solution dosyasını seçip **Open** düğmesini tıklarsanız solution ile birlikte ilgili projeleri de açmış olursunuz. Bu şartlarda üzerinde çalıştığımız solution'da sadece bir proje olacağı için projeyi açmakla solution açmak aynı işleve sahiptir.

Şimdi yukarıda verilen ekran görüntüsünden yola çıkarak birkaç cümlelik yorum yapacağız. Çalıştırıldığında bir pencereye sahip klasik Windows programı hazırlamak üzere Visual Studio'nun "New Project" diyalog kutusunda **Windows Forms Application** şablonundan yararlanıp "Proje1" adında bir proje hazırladık. Visual Studio, bu projemize .NET Framework ile gelen ve System.Windows.Forms'de bulunan "Form" sınıfının mirasçısı bir sınıf hazırlayıp dahil etti. Ayrıca bu sınıf pencere olarak ekrana geldiğinde nasıl bir görünüme sahip olacaksa o şekilde Visual Studio penceresi içinde görüntülendi.

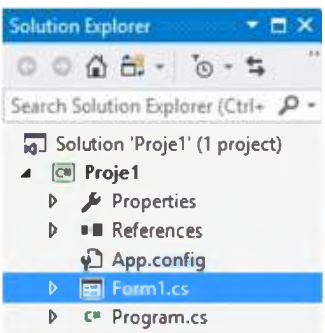
Visual Studio ile hazırlanan projelerde her form için Visual Studio tarafından 3 dosya hazırlanmaktadır. Yukarıda hazırladığımız projedeki "Form1" için konuşmak gerekirse, "Form1.cs" dosyasında programcı tarafından yazılan kodlar olurken Form1.Designer.cs dosyasında ise formun görsel yapısıyla ilgili kodlar bulunmaktadır. "Form1.resx "dosyasında ise formla ilgili resource bilgilere yer verilmektedir.

Visual Studio Sayesinde C# Kodu Yazmak

Bu başlık altında yazılanlar, üzerinde çalışılan uygulamanın Windows Forms Application olduğu varsayımına dayanmaktadır. Yoksa konu konsol uygulamaları olduğunda ilk kod dosyası kendiliğinden editör penceresinde gelmektedir.

Konu Visual Studio ile hazırlanan projeler için kod yazmak olduğunda ilk akla gelen, projedeki mevcut formları temsil eden CS uzantılı kod dosyalarıdır. İlgili formun oluşturulduğu kod dosyasını açmak için kullanabileceğiniz değişik teknikler vardır.

Görsel yapısı Visual Studio penceresi içinde görüntülenen formun oluşturulduğu, daha doğrusu Form sınıfının kodlandığı CS dosyasını açmak için **View** menüsündeki **Code** komutunu verebilirsiniz. Ayrıca **Solution Explorer** penceresinde ilgili formu seçip **View Code** düğmesini tıklayıp **Code Editor** penceresini ekrana getirebilirsiniz.



Visual Studio ile hazırlanan projedeki ilk form için kod yazmayı, **System.Windows.Forms** adlı namespace'te yer alan **Form** sınıfının mirasçısı "Form1" sınıfı içinde değişken tanımlamak, metot hazırlamak vs şeklinde anlamak gerekir.

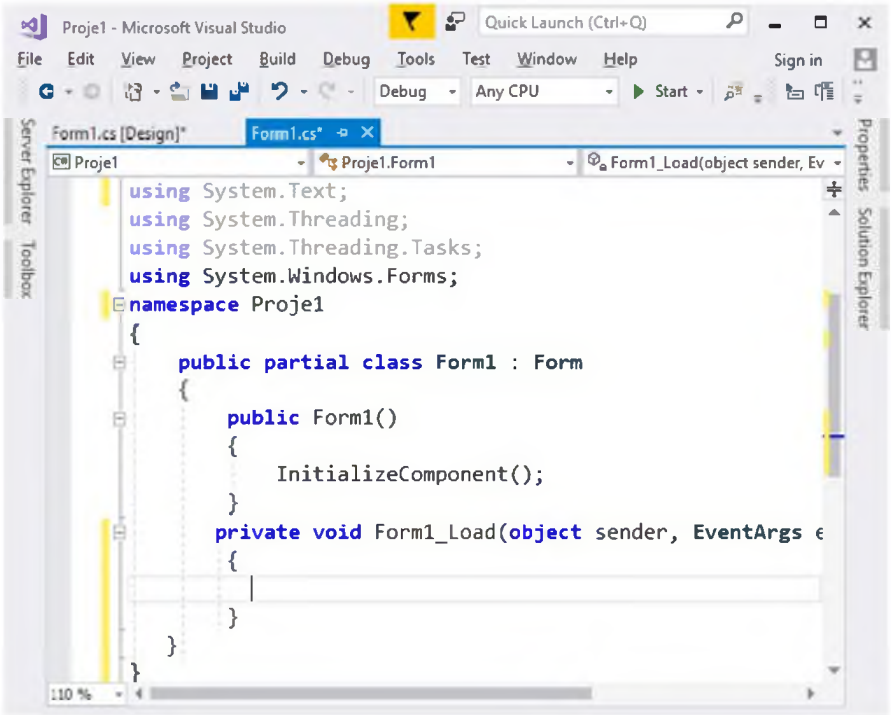
Solution Explorer penceresindeki **View Form** düğmesini tıklarsanız o sırada seçili durumdaki formun görsel yapısı görüntülenir. Solution Explorer penceresindeki **View Code** düğmesi tıklandığında ise seçili durumdaki formla ilgili kod yazılabilinen bir pencere açılır.

Formun görsel yapısını görüntülemek demek, bir bakıma çalışma anında ilgili form ekrana getirildiği zaman nasıl bir görünüme sahip olacağını önceden görmek demektir. Örneğin Visual Studio, projedeki "Form1" adlı formun oluşturulduğu Form1.cs ve Form1.Designer.cs dosyalarındaki kodlara bakıp çalışma anında pencere görevi yapacak formu tasarım anında programcının karşısına getirmektedir.

Visual Studio, söz konusu formun "FormAdı.Designer.cs" dosyasına sürekli bakıp, daha doğrusu bu kod dosyasındaki **InitializeComponent()** metodunu işletip bu formun çalışma anında nasıl bir görünüme sahip olacağı konusunda programcıyı tasarım anında bilgilendirmektedir. Bu sayede programcı formun görünümünü ayarlamak üzere yazdığı satırların etkisini görmek için projesini çalıştırmak zorunda kalmıyor.

Formun görsel yapısıyla ilgili değişiklikleri gidip **xxxDesinger.cs** dosyasında kendiniz yaptığınızda ise Visual Studio bu değişiklikleri formun görsel yapısına anında yansıtmaktadır. Çünkü Visual Studio sürekli olarak formların görsel yapısı ile ilgili kodları gözetip formun son halini tasarım anında kaşınıza getirmektedir.

Görsel yapısı görüntülenen formu çift tıklarsanız Visual Studio hem kod yazılabilinen pencereyi açmakta hem de proje çalıştırılıp form pencere olarak ekrana geldiği zaman meydana gelecek **Load** olayını temsil edecek bir metot hazırlamaktadır. Başka bir deyişle tasarım anında formu çift tıkladığınızda Visual Studio, "Form1" sınıfının Load olayı için bir metot hazırlayıp bu olayla ilişkilendirmektedir. Üzerini çift tıkladığımız formun adı "Form1" olduğu için hazırlanan metodun adı **Form1_Load()** olarak seçildi. Başka bir deyişle Visual Studio, bu metoda ad seçerken olay adının başına ilgili class'ın(Form1) adını ekledi.





Visual Studio tarafından "Form1" sınıfı için hazırlanan koda dikkatlice bakılırsa, Visual Studio tarafından proje ile aynı ada sahip bir namespace'in hazırlandığı ve "Form1" sınıfının bu namespace'e üye yapıldığı görülür.

C#'ta geriye değer göndermeyen metotların başına **void** anahtar kelimesi yazılmaktadır. Burada metot adından önce kullanılan **private** anahtar kelimesi ile, metodun geçerlilik alanı işaret edilmektedir. Private metotlar ancak hazırlandıkları sınıfın içinde kullanılabilinir. Metotların geçerlilik alanı seçilirken **public** ve **protected** anahtar kelimeleri de kullanılabilir. Bu konunun ayrıntılarından daha sonra söz edilecektir.

Visual Studio ile hazırlayıp üzerinde çalıştığımız projede "Form1" adında bir form olduğu için bu proje Visual Studio sayesinde çalıştırılıp (Debug menüsünden komut verilerek) test edildiği zaman "Form1" sınıfının örneği alınıp ekrana getirilir. **Form** nesnesinin bir pencere olarak ekrana getirilmesi sırasında **Load** olayı meydana gelmektedir. Dolayısıyla örneği alınan sınıfta (Form1) Load olayıyla ilişkilendirilmiş bir metot varsa işletilmektedir. Bu metodu yukarıda tasarım anında Form1'in görsel yapısını çift tıklayarak hazırlamıştık.

Şimdi proje çalıştırıldığında bir diyalog kutusu içinde ekrana bir mesaj getirmek için hazırlık yapacağız. Sabit bilgiler diyalog kutusu içinde ekrana getirilirken .NET Framework ile gelen **MessageBox** sınıfının **Show**() metodu kullanılmaktadır.

```
private void Form1_Load(object sender, EventArgs e)
{
 MessageBox.Show("Microsoft Visual C#");
}
```

Bu metotta ilk dikkatinizi metodun tanımlama satırına çekmek istiyoruz. Bu metot **object** ve **EventArgs** tipinde 2 parametreye sahiptir. Bu parametreler sıklıkla karşımıza çıkacakları için şimdiden göz aşinalığı yararlı olacaktır.

İşletilmek üzere metoda yazılan satırdaki MessageBox, System.Windows.Forms'de yer alan bir sınıftır. Show() ise bu sınıfın statik bir metodudur. Show() metodu kendisine parametre olarak verilen bilgiyi bir diyalog kutusu içinde ekrana getirmektedir. Aşağıda verilen ekran görüntüsünü bir forma sahip olup Form1_Load() metodunu yukarıdaki gibi düzenlediğimiz projeyi çalıştırdıktan sonra aldık.



MessageBox sınıfının Show() metodunu formun Load olayını temsil eden metoda yazdığımız için Form henüz pencere olarak ekrana gelmiş değildir. Bu sırada Tamam düğmesi tıklanıp bu diyalog kutusu kapatılırsa projenin ana penceresi durumundaki form ekrana gelir.



Program.cs Dosyası

Visual Studio ile hazırlanan C# projeleri için "Program.cs" adında bir kod dosyası hazırlanmakta ve projeler bu dosyadaki **Main**() metodundan itibaren çalışmaya başlamaktadır. Main() metodunun yerleştirildiği bu dosya ile ilgilenmeyebilirsiniz. Çünkü bu dosya aracılığı ile Main() metodu içinde yalnızca projenin başlangıç formunun örneği alınıp **Application** sınıfının **Run**() metoduna parametre olarak verilmektedir.

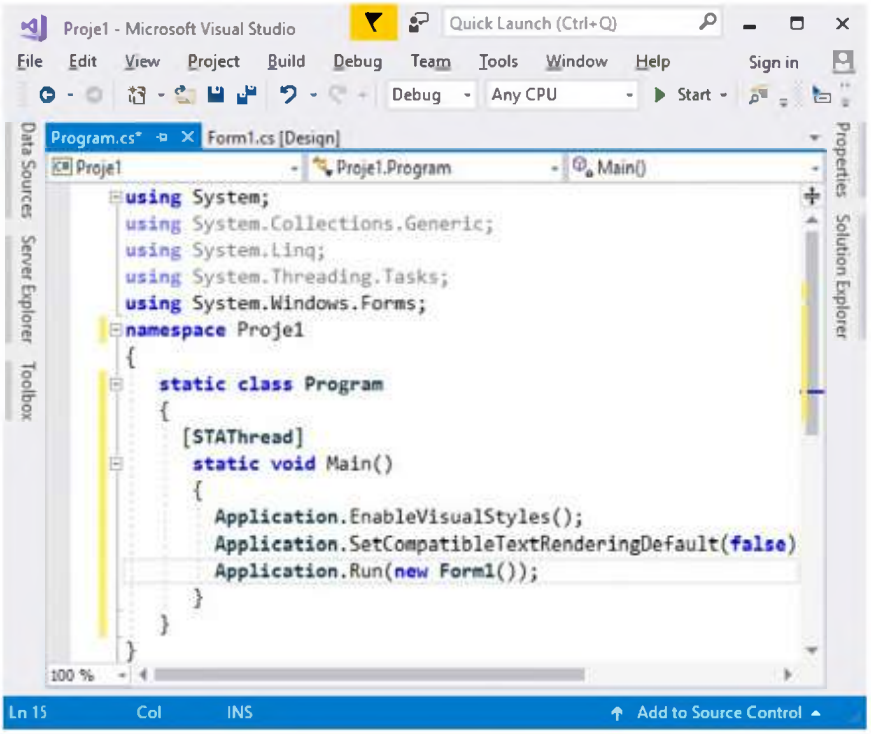


C# uygulamasının Windows'la ilişkisi **Application** sınıfı üzerinden sağlanmaktadır. Bu sınıfla hiç ilgilenmeden usta işi C# uygulamaları geliştirmek mümkündür. Buna rağmen konunun geri planı hakkında kafa yormanızı sağlamak ve .NET programcılığın geri planı hakkında biraz bilgi vermek amacıyla bu kitabın henüz başında olmamıza rağmen Application sınıfıyla ilgilendik.

Bu dosyaya "Program.cs" adı verilmiş olunsa bile sizler başka bir ad verebilirsiniz. Hatta **Main**() metodunu projedeki mevcut formlardan birisinin kodlandığı dosyaya taşıyıp "Program.cs" dosyasını projeden tümden uzaklaştırabilirsiniz. Kodun okunurluğuna zarar vermemek için "Program.cs" dosyasını silmeniz veya adını değiştirmenizi önermeyiz.

"Program.cs" dosyasının en başında C# projelerinde sık kullanılan sınıfları içeren namespace'ler **using** deyimi ile koda dahil edilmektedir. Daha teknik bir söyleyişle; kulla-

nılması muhtemel sınıfları içeren namespace'ler **using** deyimi ile derleyiciye ve Visual Studio'ya en başta söylenmektedir. Kullanılması muhtemel sınıfları içeren namespace'ler **using** deyimi ile tek tek işaret edildikten sonra **Program** adında bir sınıf hazırlanmakta ve bu sınıfta **Main()** metoduna yer verilmektedir.



Aslında **using** anahtar kelimesinden sonra adları yazılan namespace'ler için "koda dahil etmek" demek yerine, "programcının bu namespace'lerden yararlanma isteğini baştan belirtmesi" olarak görmek daha doğru olur. Çünkü using anahtar kelimesinden sonra ilgili namespace'in adını yazmak bir zorunluluk olmamakla birlikte programcının işi kolaylaşmaktadır. Bu sayede kod satırlarının kısa olması sağlanmaktadır.

Hazırladığınız uygulama istenilen sayıda kod dosyasından ve bu kod dosyalarında istenilen sayıda sınıf olabilir. Bu sınıflardan birisi **Main()** metoduna sahip olmalıdır. Main() metodu parametrelere sahip olabilir ve geriye bilgi gönderebilir. Programcılar genellikle "Program.cs" dosyasına başka bir şey yazmıyor olsalar bile sizler bu kod dosyasındaki **Program** adlı sınıfta başka metotlar hazırlayabilir veya bu kod dosyasında başka sınıflara yer verebilirsiniz.

Main() metodundaki **Application.Run(new Form1());** satırı ile projedeki ilk formu temsil eden "Form1" adlı sınıfın örneği **new** anahtar kelimesi ile hazırlanmakta ve uygulamanın Windows'la ilişkisi kurulmaktadır. Bu işlem yapılırken projedeki ilk forma ait CS uzantılı dosyadaki **Form1** sınıfının yapıcı metodundan yararlanılmaktadır. Başka bir deyişle "new Form1()" ile Form1'in örneği hazırlanmaktadır. Yani "Form1" sınıfından yola çıkılıp **new** anahtar kelimesiyle "Form1" tipinde bir nesne hazırlanmaktadır.

C# uygulamalarında yapıcı metotlara ilgili sınıfın adı verilmektedir. Visual Studio'dan yararlanıp **Windows Forms** uygulaması hazırladığınızda bu uygulama başlangıçta "Form1" adında bir sınıfa sahip olmakta ve bu sınıfta **Form1()** adında bir metot hazırlanmaktadır. Başka bir anlatımla Application sınıfının **Run()** metoduna "Form1" sınıfının örneği yani Form1 tipinde bir nesne parametre olarak verilmektedir.

Örnek olması için **Program.cs** dosyasına aşağıdaki gibi ekleme yaptık. Yaptığımız eklemeler sayesinde projenin ilk formu **new** anahtar kelimesi ile oluşturulup ekrana getirilmeden önce **MessageBox** sınıfının **Show()** metodu ile kullanıcıya mesaj verilmektedir.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
namespace Proje1
{
 static class Program
 {
  [STAThread]
  static void Main()
  {
   string mesaj ="Proje çalıştırıldı, Form1 ekrana getirilecek";
   MessageBox.Show(mesaj);
   Application.EnableVisualStyles();
   Application.SetCompatibleTextRenderingDefault(false);
   Application.Run(new Form1());
  }
 }
}
```

Bu kodun başında **using** anahtar kelimesi ile **System.Windows.Forms**'deki sınıflardan yararlanmak istediğimizi belirtmemiş olsaydık **MessageBox** sınıfından önce namespace adını yazmamız gerekirdi. Bu konunun netleşmesini sağlamak için "Program.cs" dosyasındaki **using** bloğunda **System.Windows.Forms**'le ilgili satırı silip yukarıda verilen Main() metodunu aşağıdaki gibi düzenledik. Bu kod işlev bakımından yukarıdakinden farksızdır.

```
static void Main()
{
 string mesaj ="Proje başlatıldı, Form1 ekrana getirilecek";
 System.Windows.Forms.MessageBox.Show(mesaj);
 System.Windows.Forms.Application.EnableVisualStyles();
 System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(false);
 System.Windows.Forms.Application.Run(new Form1());
}
```

using deyiminin işlevi hakkında biraz konuşmak gerekirse şunlar söylenebilir: C# kodunda önce tipler veya class'lar yazılmakta, ardından ilgili değişken veya sınıfın adı belirtilmektedir. Örneğin **string mesaj;** şeklinde bir satır yazıldığında C# derleyicisi ilk yazılan **string**'in bir tip veya bir class olduğunu varsayıp bu tipi using bloğunda işaret edilen namespace'lerde aramaktadır. Tabii hemen uyarmak gerekir, bazı namespace'ler using bloğunda işaret edilmeseler bile bu namespace'lerdeki sınıflar, tipler veya yapılar namespace adı yazılmadan kullanılabilmektedir. **System.Core** böyle bir namespace'tir.

Benzer şekilde **Application.Run()** yazıldığında ilk yazılanın bir class, nokta işaretinden sonra yazılanın ise bir metot olduğu varsayılıp bu class kod dosyasının başında using deyimleri ile işaret edilen namespace'lerde aranır. Söz konusu class (yani **Application**) belirtilen namespace'lerde varsa sorun yoktur. Yararlanılmak istenen class, **using** deyimi ile işaret edilen hiçbir namespace'te yoksa Visual Studio uyarıda bulunur ve derleyici hata rapor eder.

Projenizde birden fazla form varsa "Program.cs" dosyasında değişiklik yapıp projenizi istediğiniz formdan başlatabilirsiniz. **Application** sınıfının **Run()** metoduna hangi form nesnesini parametre olarak verirseniz proje o form ile başlatılır. Farklılık olsun diye Visual Studio'nun **Project** menüsündeki **Add Windows Form** komutundan yararlanıp üzerinde çalıştığımız projeye "Form2" adında 2. bir form dahil edip Program.cs dosyasındaki **Main()** metodunu aşağıdaki gibi düzenledik.

```
static void Main()
{
 MessageBox.Show("Proje Form2 ile başlatılacak");
 Application.EnableVisualStyles();
 Application.SetCompatibleTextRenderingDefault(false);
 Application.Run(new Form2());
}
```

Şimdi proje başlatıldığı zaman ekrana getirilecek formu kod yazarak hazırlayacağız. Form nesnesi hazırlamak üzere "Program.cs" dosyasındaki Main() metodunda yaptığımız değişiklikleri aşağıda görebilirsiniz. "Program.cs" dosyası bu şekilde düzenlenen proje çalıştırıldığında "Form1" adında bir form hazırlanıp bir pencere olarak ekrana getirilir.

```
static void Main()
{
 System.Windows.Forms.Form Form1;
 Form1 = new System.Windows.Forms.Form();
 Form1.Text = "Form1";
 System.Windows.Forms.Application.Run(Form1);
}
```

Şimdi gelelim form hazırlamak üzere yazdığımız bu satırlar hakkında biraz konuşmaya. İlk olarak **System.Windows.Forms.Form** tipinde bir değişken tanımladık. Sözünü ettiğimiz bu değişkenin tanımlandığı satırdaki "System.Windows.Forms" bir **namespace** iken en son yazılan **Form** ise .NET Framework ile gelen bir sınıftır. Ardından **new** anahtar kelimesi ile form nesnesini hazırlayıp **Application** sınıfının **Run()** metoduna parametre olarak verdik.

Bu sırada **using** anahtar kelimesi ile **System.Windows.Forms** adlı namespace'teki class'lara gerek duyacağınızı işaret ettiyseniz bu koddaki bazı satırlar kısalır. Başka bir deyişle class adından önce namespace adını yani "System.Windows.Forms" yazmak zorunda kalmazsınız. Buradaki **System** bir namespace'tir. **Windows** ise System'in içinde yer alan başka bir namespace'tir. **Forms** ise sık kullanılan sınıfların bulunduğu başka bir namespace'tir ve bu namespace, Windows adlı namespace'te bulunmaktadır.

Şimdi bu koda ekleme yapıp kod yazarak hazırlayıp ekrana getirdiğimiz forma bir **TextBox** yerleştireceğiz. Bu amaçla Program.cs dosyasındaki Main() metodunu aşağıdaki gibi düzenledik. Program.cs dosyasında kod yazarak hazırladığımız forma TextBox nesnesi yerleştirmek

için önce **System.Windows.Forms**'da yer alan **TextBox** tipinde bir değişken tanımlayıp **new** anahtar kelimesi ile TextBox nesnesi hazırladık. Devamında bu nesneyi **Add**() metoduyla form nesnesinin **Controls** koleksiyonuna ekledik.

```
static void Main()
{
 System.Windows.Forms.Form Form1;
 Form1 = new System.Windows.Forms.Form();
 Form1.Text = "Form1";
 System.Windows.Forms.TextBox Text_kutu1;
 Text_kutu1 = new System.Windows.Forms.TextBox();
 Text_kutu1.Left = 50;
 Text_kutu1.Top = 50;
 Form1.Controls.Add(Text_kutu1);
 Application.Run(Form1);
}
```

"Program.cs" dosyasındaki Main() metodu bu şekilde düzenlenen proje çalıştırıldığında aşağıdaki gibi bir sonuç alınır. Form sınıfının **Controls** koleksiyonuna eklenmeyen nesneler formun üzerinde görünmezler.



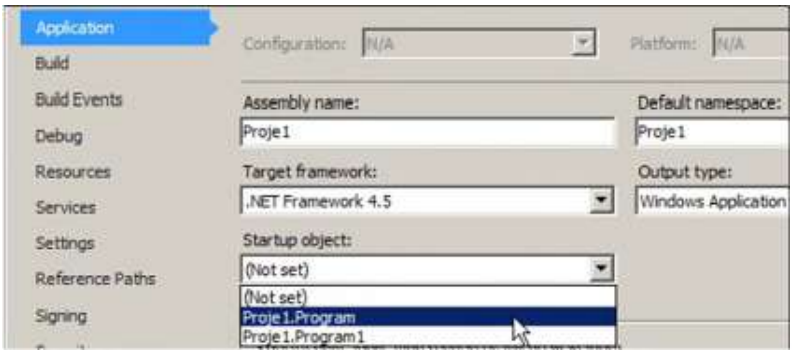
Toolbox'ta **TextBox** kontrolünü seçip forma yerleştirdiğiniz zaman Visual Studio geri planda, yukarıda TextBox'la ilgili olarak yazdığımız satırların benzerini otomatik olarak hazırlayıp formla ilgili **Designer.cs** dosyasına yazmaktadır.

Şimdiye kadar "Program.cs" dosyasıyla ilgili olarak yazılanlardan fark edilmiş olacağı gibi bu kod dosyasının önemi **Main**() metodunu içermesinden dolayıdır. Bir sonraki bölümde konsol uygulamaları hakkında bilgi verilirken görüleceği gibi C# derleyicisi için Main() metodunun hangi kod dosyasında olduğunun önemi yoktur.

C# derleyicisi aynı class'ta aynı parametre yapısına ve aynı ada sahip birden fazla metodun olmasına izin vermez. Bu konuda bilgi vermek için Visual Studio'dan yararlanıp 2 forma sahip bir proje hazırladık. Ardından "Program.cs" dosyasında "Program1" adında 2. bir class'a yer verip bu class'ın da Main() metoduna sahip olmasını sağladık.

```
namespace Proje1
{
 static class Program
  {
   static void Main()
    {
     Application.Run(new Form1());
    }
  }
 static class Program1
  {
   static void Main()
    {
     Application.Run(new Form2());
    }
  }
}
```

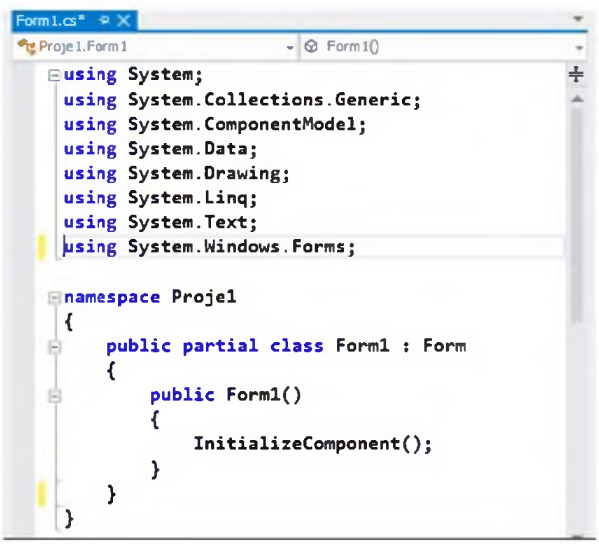
"Program1" adını verdiğimiz ikinci sınıfı başka bir kod dosyasına alabilirdik. Kod uzamasın diye kullanılması zorunlu olmayan bazı satırları Main() metotlarından sildik. Bu şartlarda bu proje derlenirse hata meydana gelir. Çünkü C# derleyicisi hangi sınıftaki Main() metodunu kullanacağına karar veremez. Hangi sınıftaki Main() metodunun uygulamanın başlangıç noktası olması isteniyorsa **Project** menüsünden komut verilerek ekrana getirilen **Project Properties** penceresinde **Startup object** liste kutusunda seçim yapılmalıdır.



Başlangıçta bu liste kutusunda (**Not set**) seçilidir ve C# derleyicisi bulduğu tek Main() metodunu kullanır. Tabii projedeki kod dosyalarının hiç birisinde Main() metodu yoksa hata meydana gelir. Burada listelenen seçeneklerdeki "Proje1" bir namespace iken "Program" ve "Program1" birer class'tır. Bu liste kutusunda hangi class seçilirse projenin başlangıç noktası o class'taki Main() metodu olur.

Formlar İçin Hazırlanan Kodlar

Windows Forms Application şablonundan yararlanıp proje hazırlandığınızda "Form1" adında bir formun projeye dahil edildiğini biliyorsunuz. Formu oluşturan kodları görüntülemek için **View** menüsünden **Code** komutu verilebilir. Aşağıda verilen CS dosyasının en başında kullanılması muhtemel sınıfların yer aldığı namespace'ler **using** ile işaret edilmektedir.



Kod dosyasının başında **using** anahtar kelimesi ile işaret edilen her namespace'i içeren referanstan (DLL dosyası) Visual Studio'nun haberdar edilip **Solution Explorer** penceresinde listelenmesi sağlanmalıdır. Bu işlem çok sayıda namespace için Visual Studio tarafından otomatik olarak yapılmaktadır.

Using bloğunda isimleri listelenen ve .NET Framework ile gelen namespace'lerden ilgilenmediklerıniz varsa bunları silebilirsiniz. Örneğin bu kitapta "System.Data" adlı namespace'te bulunan ve veritabanı işlemleriyle ilgili sınıfları kullanmayı düşünmediğimiz için using bloğunda "System.Data" ile ilgili satırı silebiliriz.

Bu koda dikkatlice bakarsanız using bloğundan sonra **namespace** bloğu bulunmaktadır ve projenin adı aynı zamanda namespace(Proje1) adı olarak kullanılmaktadır. Visual Studio, varsayım olarak bir projedeki bütün class'ları aynı namespace'e üye yapmaktadır. Yani bu sırada bu proje için başka bir kod dosyasında ikinci bir form hazırlanırsa o forma ait kod dosyasının en başına yine "namespace Proje1" yazılır. Tabi, projemdeki bazı sınıfları ayrı bir namespace'de toplamak istiyorum derseniz bu zor değildir.

Devamında **public** ve **partial** olarak işaret edilen class bloğu bulunmaktadır. Hazırlanan sınıfa **Form1** adı verilmiştir. Bu sınıfa ait kodların bir kısmı "Form1.cs" dosyasında iken diğer kısmı "Form1.Designer.cs" dosyasında bulunduğu yani bu sınıfa ait kodlar birden fazla dosya arasında paylaşıldığı için **partial** anahtar kelimesine yer verilmesi gerekmiştir.

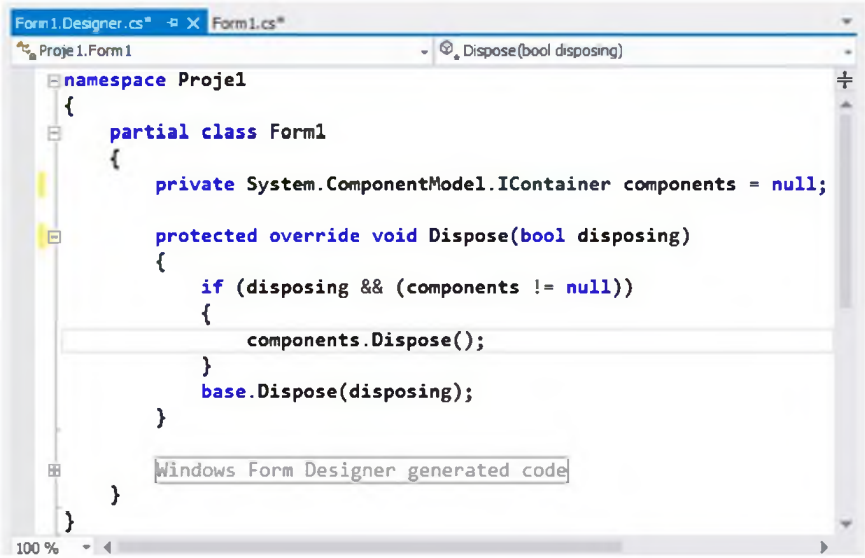
C# programlama dili bir sınıfa ait kodlarında birden fazla dosya arasında paylaşımını desteklediği için böyle bir tercih yapılmıştır. Bir sınıfa ait kodun birden fazla dosya arasında paylaşılmasına **partial class** denilmektedir.

Solution Explorer penceresinden yararlanılıp Form veya CS dosyasının adı değiştirilirse Visual Studio, başlangıçta adı "Form1" olan sınıfın adını kendiliğinden değiştirir. Tabi kod dosyasının adıyla ilgilenmeden class'a farklı bir ad vermek mümkündür.

Hazırlanmak istenen sınıfın adından sonra mirasçı (base class) pozisyonundaki "Form" sınıfının adı yazılmış. Hazırlanmak istenen sınıf ile mirasçı (taban sınıf) sınıfın adını birbirinden ayırmak için ":" kullanılmaktadır. **Form** sınıfı System.Windows.Forms adlı namespace yer aldığı ve bu namespace **using** bloğunda işaret edildiği için "Form" sınıfının adının önüne namespace adının yazılmasına gerek duyulmamış. Bu kod dosyasındaki "Form1" sınıfı için yazılan satırlar az olsa bile **System.Windows.Forms** adlı namespace'teki **Form** adlı sınıfın mirasçısı olduğu için çok sayıda özellik ve metoda sahiptir.

Devamında "Form1" sınıfının yapıcı metoduna yer verilmiş. Class ile aynı ada sahip metotlara yapıcı(**constructor**) metot denilmektedir. Daha sonra yapıcı metotlar hakkında bilgi verileceği için şimdilik "**sınıf ile aynı ada sahip metotlara yapıcı metot denilir**" tanımını bilmeniz yeterlidir.

Form1 adı verilen sınıfın yapıcı metodunda yalnızca **InitializeComponent()** metodu işletilmektedir. Form1'le ilgili **InitializeComponent()** metodu "Form1.Designer.cs" dosyasında yer almaktadır. Konunun devamında bu metot hakkında bilgi verilecektir. Formların ve formların üzerine yerleştirilen nesnelerin özellikleri CS uzantılı **Designer** dosyasında ayarlanmaktadır. Designer dosyasında da "Form1" adında bir class bulunmaktadır. Başka bir deyişle "Form1" adlı sınıfın bir kısmı "Form1.cs" diğer kısmı "Form1.Designer.cs" dosyasında bulunmaktadır. Bu dosyanın içeriğinin bir kısmını aşağıda görebilirsiniz.



"Form1" adlı sınıfı için Visual Studio tarafından hazırlanan "Form1.Designer.cs" dosyasına bakarsanız **Dispose**() adında bir metodun olduğunu görürsünüz. Bu metot, form ve formla ilgili olarak hazırlanan nesnelerin form kapatıldığı zaman bellekten silmesiyle ilgilidir.

Şimdi "Form1" sınıfı için Visual Studio tarafından hazırlanan satırları ve bu satırlarda kullanılan metot ve anahtar kelimelerin bazılarından söz edeceğiz. Visual Studio her form için yukarıda ekran görüntüsü olarak verilen satırlardan daha fazlasını hazırlamaktadır. Tasarım anında Properties penceresinde yapılan değişikliklere genellikle doğrudan müdahale edilmediği için bu satırlar **Windows Form Designer generated code** adlı altında gruplanıp gizlenmektedir. Bu bölgeye (region) yazılan satırları aşağıda görebilirsiniz.

```
namespace Proje1
{
 partial class Form1
 {
  private System.ComponentModel.IContainer components = null;
  protected override void Dispose(bool disposing)
   {
    if (disposing && (components != null))
     {
       components.Dispose();
     }
    base.Dispose(disposing);
   }
   #region Windows Form Designer generated code
   private void InitializeComponent()
   {
    this.SuspendLayout();
    this.AutoScaleDimensions=new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode=System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(292, 146);
    this.Name = "Form1";
    this.Text = "Form1";
```

```
        this.ResumeLayout(false);
      }
      #endregion
    }
}
```

Bu koddaki bölge başını ve sonunu işaret eden **#region** ve **#endregion** satırlarını silerseniz bu bölgedeki satırlar Visual Studio penceresinde sürekli görüntülenir. Region'lar '#region' ile başlayıp '#endregion" ile bitmektedir. **InitializeComponent()** metodundaki satırlar Toolbox'tan yararlanılarak forma yerleştirilen nesnelere göre artabilmektedir. Bu sırada Toolbox'tan yararlanıp forma herhangi bir nesneyi yerleştirir veya Properties penceresinden yararlanıp formun özelliklerini değiştirseniz "Windows Form Designer generated code" bölgesine yeni satırlar eklenir veya mevcut satırlarda değişiklik meydana gelir.

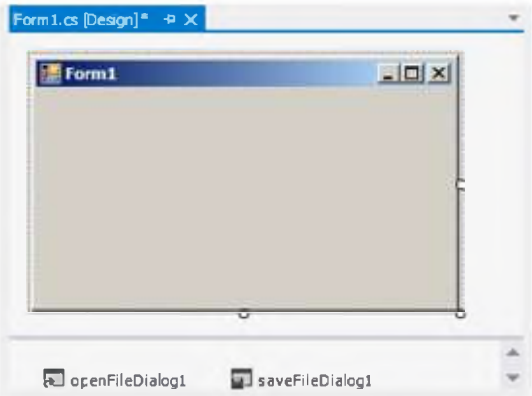
InitializeComponent() metodunda, form ve formun üzerine yerleştirilen nesnelerin özellikleri ayarlanmaktadır. Toolbox'tan yararlanıp projenin ana formuna bir **Button** nesnesi yerleştirince Form1 sınıfının 2. kısmının saklandığı "Form1.Designer.cs" dosyasındaki **InitializeComponent** metodu aşağıda verilen şekle dönüştü.

```
private void InitializeComponent()
{
 this.button1 = new System.Windows.Forms.Button();
 this.SuspendLayout();
 this.button1.Location = new System.Drawing.Point(71, 45);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(75, 23);
 this.button1.TabIndex = 0;
 this.button1.Text = "button1";
 this.button1.UseVisualStyleBackColor = true;
 this.ClientSize = new System.Drawing.Size(294, 141);
 this.Controls.Add(this.button1);
 this.Name = "Form1";
 this.ResumeLayout(false);
}
```

"Form1.cs" dosyasındaki **InitializeComponent()** metodunu işleten satırı silerseniz "Form1.Designer.cs" dosyasındaki bu satırlar işletilmeyeceği için tasarım anında forma yerleştirilen **Button** nesnesi çalışma anında formun üzerinde görünmez ve formun bazı özelliklerinin ayarlandığı satırlar işletilmeyeceği için form standart özelliklere sahip olur.

Toolbox'tan yararlanarak forma yerleştirdiğimiz Button nesnesi ile ilgili olarak Form1.Designer.cs dosyasındaki **InitializeComponent()** metoduna yazılan satırlara dikkat ederseniz Button nesnesinin formun **Controls** koleksiyonuna **Add()** metodu ile eklendiğini görebilirsiniz. Visual Studio, tasarım anında ancak ilgili formun Controls koleksiyonuna eklenen nesneleri formun üzerinde görüntülemektedir. Button nesnesinin Controls koleksiyonuna eklendiği satırı InitializeComponent() metodundan silerseniz button nesnesi hem tasarım anında hem de çalışma anında formun üzerinde görünmez.

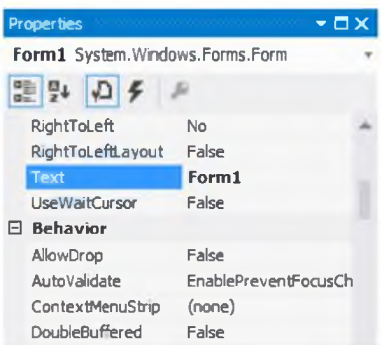
Forma yerleştirilen bazı kontroller görsel kontroller olmadıkları, başka bir deyişle çalışma anında formun üzerinde görünmedikleri için formun altına konulan panele yerleştiriliyor. **OpenFileDialog** ve **SaveFileDialog** görsel olmayan kontrollere örnek olarak gösterilebilir.



OpenFileDialog gibi görsel olmayan nesneler formun **Controls** koleksiyona dahil edilemezler. InitializeComponent() metoduna ekleme yapıp OpenFileDialog nesnesini formun Controls koleksiyonuna eklemek isterseniz hata meydana gelir. Çünkü OpenFileDialog sınıfı **Control** sınıfının mirasçısı değildir.

Olaylar ve Metot Hazırlamak

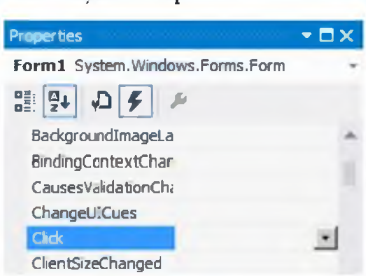
Visual Studio ile gelen Windows Forms Application şablonu ile hazırlanan C# projeleri başlangıçta bir forma sahiptir. Projenin çalıştırılıp ilk formun oluşturulup pencere olarak ekrana getirilmesi olayının Properties penceresindeki adı **Load**'dır. Çalışma anında form tıklandığında ise formla ilgili olarak **Click** olayı meydana gelir.



Formlarla ve formlara yerleştirilen nesnelerle ilgili olarak çalışma anında meydana gelmesi muhtemel çok sayıda olay vardır. Örneğin çalışma anında formu tıklamak, imleç bir TextBox'ın içinde iken klavyenin herhangi bir tuşuna basmak birer olaydır.
Olay kavramını anlatabilmek için **Properties** penceresine ait ekran görüntüsü yanda verildi. Kendi hazırlayacağınız sınıflar için nasıl olay hazırlayabileceğiniz konusunda daha sonra bilgi verilecektir. Bu bölümde sözü edilecek olaylar .NET Framework ile gelen **Button** ve **Form** gibi sınıfların olaylarıdır.

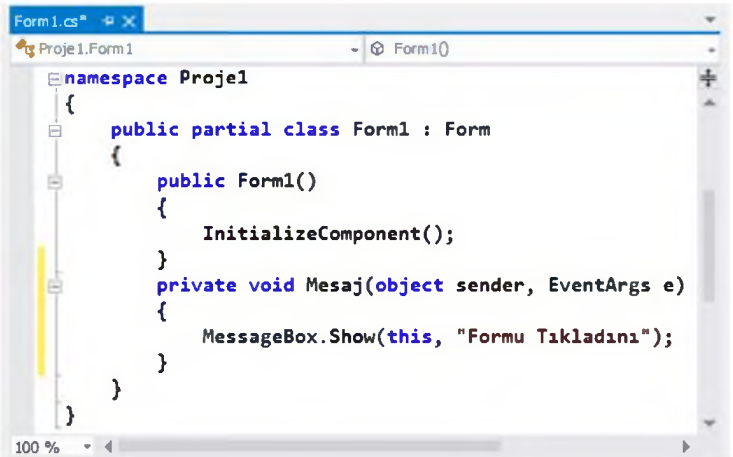
Visual Studio'nun Properties penceresi 2 amaca yönelik olarak işlev görmektedir. Birincisi, form ve Toolbox'tan yararlanılarak formlara yerleştirilen nesnelerin özellikleri hakkında bilgi edinmek ve gerekirse bu özelliklerde değişiklik yapmaktır.

Bu penceredeki **Events** düğmesi tıklandığında ise o sırada seçili durumdaki nesneye kaynaklık eden sınıfın olayları listelenmektedir. Aşağıda verilen ekran görüntüsü "Form1" sınıfının özellikleri listelenirken **Events** düğmesi tıkladıktan sonra alındı. Dolayısıyla burada listelenen olayların hepsi **Form** sınıfınındır.



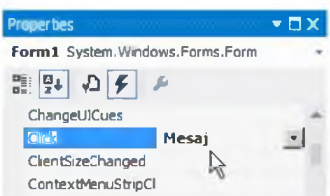
Hangi nesnenin hangi olayı için metot hazırlamak istiyorsanız o nesneye kaynaklık eden sınıfın ilgili olayının parametre yapısını bilmeniz gerekir. Tabii C# uygulamalarını geliştirirken Visual Studio'dan yararlandığımız için söz konusu olayların parametre yapılarını ezbere bilmemiz gerekmiyor.

Şimdi üzerinde çalıştığımız projedeki "Form1" adlı class'ta "Mesaj()" adında ve geriye değer göndermeyen bir metot hazırlayacağız. Geriye değer göndermeyen metotların adının önüne **void** yazıldığını biliyorsunuz. Mesaj() adını verdiğimiz bu metodu aşağıda görebilirsiniz.



Bu metodun **object** ve **EventArgs** tipinde 2 parametreye sahip olmasını sağladık. Çünkü Form1 sınıfının **Click** olayını temsil etmek üzere hazırlanacak metodun **object** ve **EventArgs** tipinde 2 parametreye sahip olması gerekmektedir.

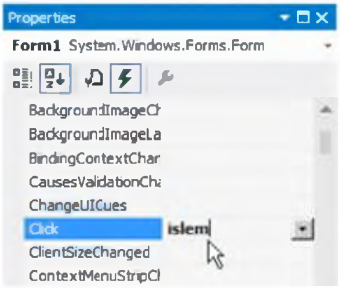
Şimdi sıra bu metodun nasıl işletileceğini belirlemeye geldi. Formun üzeri tıklandığında kullanıcıya mesaj verilsin istediğimiz için bu metodun adını Properties penceresinde **Click** olayının karşısına yazmak gerekiyor.



Başlangıçta Form1 sınıfının olayları için metot hazırlanmadığı için söz konusu olaylar çalışma anında meydana gelse bile C# uygulaması bir tepkide bulunmaz. Örneğin Form1 sınıfının **Click** olayına bir metodun adını aktarmadığınız sürece çalışma anında form istenildiği kadar tıklansın herhangi bir işlem gerçekleşmez.

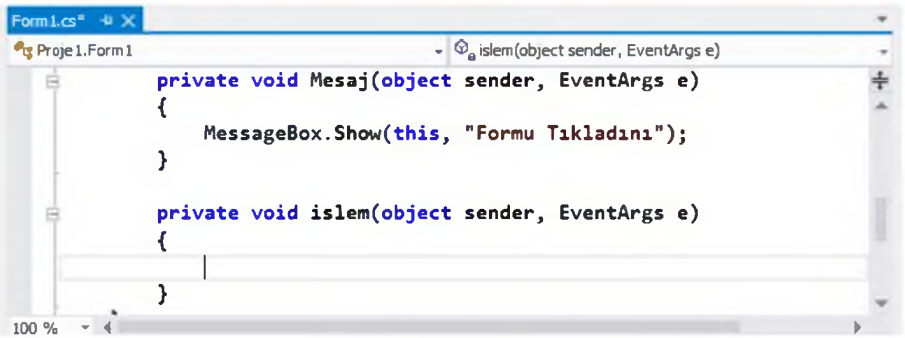
Properties penceresinde ilgili nesnenin olayları listelenirken üzerinde bulunan olay adının hemen yanına bir liste kutusu konulmaktadır. Söz konusu olay meydana geldiği zaman işletilmesini istediğiniz metodu önceden hazırladıysanız bu liste kutusundan seçebilirsiniz. Bu liste kutusunda ancak formun Click olayını temsil edebilecek, başka bir deyişle geriye değer göndermeyip **object** ve **EventArgs** tipinde 2 parametreye sahip metotlar listelenir. **Click** olayının karşısına bu metodun adı yazıldıktan sonra projenin çalışması sırasında formun üzerinde yapılacak her tıklamada bu metot işletilir.

Forma yerleştirilmiş nesnelerin olayları için metot hazırlarken Visual Studio'dan yararlanabilirsiniz. Bu amaçla Properties penceresinde kendisi için metot hazırlanmak istenen olayın üzerine gidilir ve olaya ait kutuya hazırlanmak istenen metodun adı yazılır. Bu işlemin nasıl yapıldığını anlatmak için üzerinde çalıştığımız projenin formuna bir Button nesnesi yerleştirip Properties penceresinde bu Button nesnesinin olaylarının listelenmesini sağladık. Metodun adını kendiniz belirlemek istiyorsanız metodun adını olaya ait kutuya yazmalısınız.

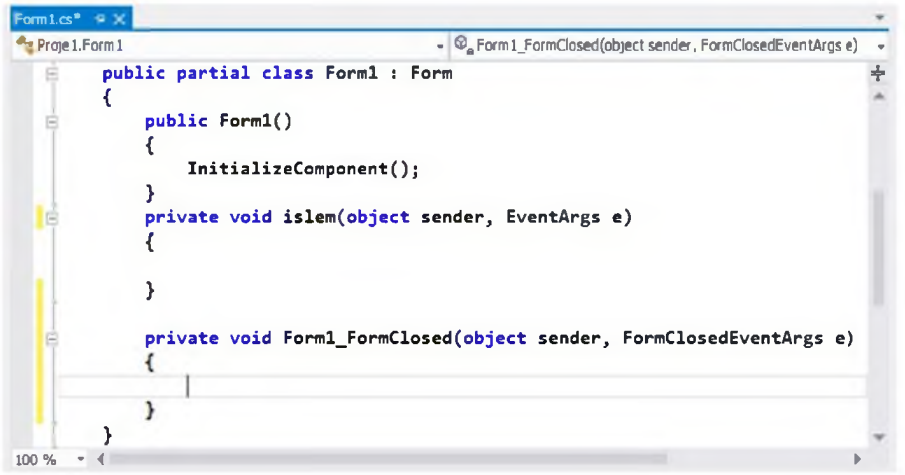


Hazırlamak istediğiniz metodun adını bu şekilde yazıp enter tuşuna basarsanız Visual Studio metodu hazırlar ve olayla ilişkilendirir.

Yeni bir metot hazırlamak üzere bu kutuya metot adını yazmak yerine bu kutuda çift tıklama yapabilirsiniz. İlgili olaya ait kutu çift tıklandığında olayın ve nesnenin adının birleşiminden meydana gelen bir ada sahip bir metot Visual Studio tarafından hazırlanır. Aşağıda verilen ekran görüntüsünü Click olayına ait kutuya "islem" yazıp Enter tuşuna bastıktan sonra aldık.

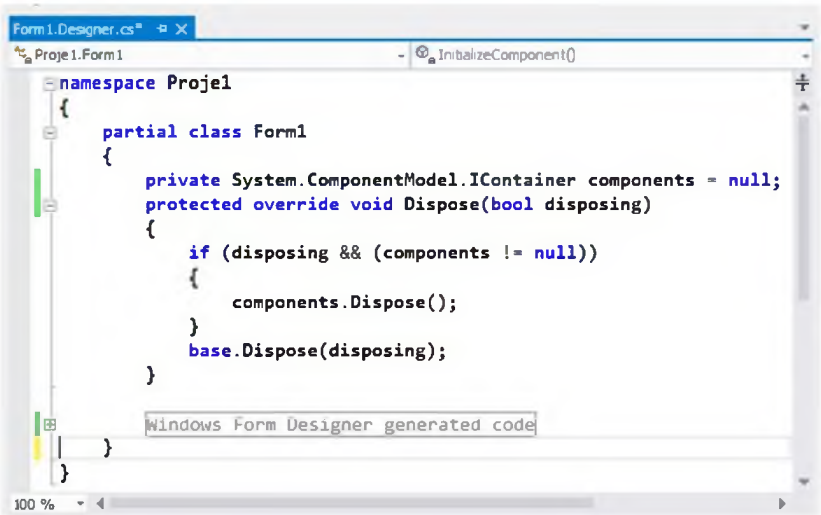


Properties penceresinde forma ait olaylar listelenirken **FormClosed** olayı çift tıklanırsa code penceresi aşağıda verilen şekle dönüşür. Formun **FormClosed** olayını temsil etmek üzere Visual Studio'dan yararlanarak hazırladığımız metodun parametre yapısı Click olayını temsil eden metodun parametre yapısından farklıdır. Bu şekilde hazırlanan metodun adında Form1'e yer verilerek bu metodun Form1 sınıfına ait olduğu işaret edilmektedir.

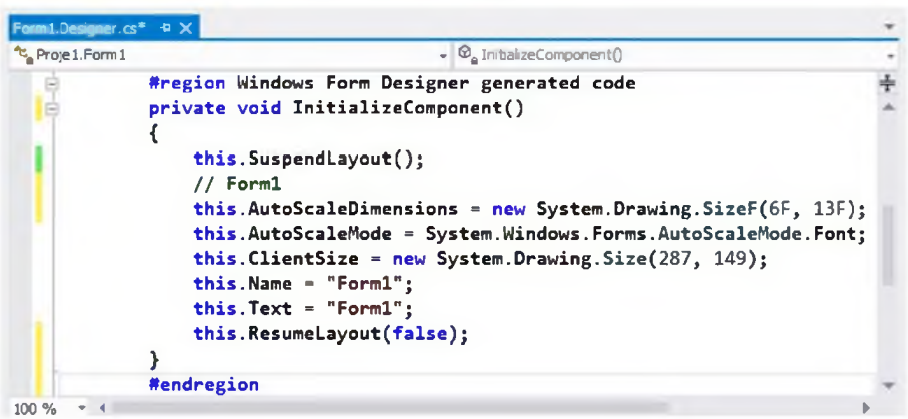


Olaylarla Metotları İlişkilendirmek

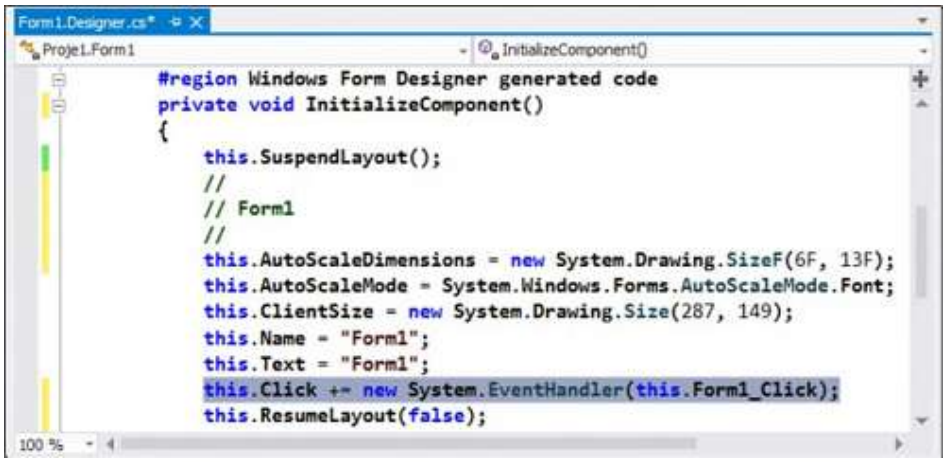
Properties penceresinde ilgili nesnenin olayları listelenirken olayın adı üzerinde çift tıklama yapılarak metot hazırlandığında, formla ilgili "Designer.cs" dosyasındaki **Windows Form Designer generated code** bölgesine bir satır eklenerek metot ilgili olayla ilişkilendirilir. Başlangıçta bu bloktaki kodlar gizlendiği için görünmemektedir. Bu konuda bilgi vermek için yeni bir proje hazırlayıp forma ait "Form1.Designer.cs" dosyasını açtık.



Bu sırada kod penceresindeki "+" işareti tıklanırsa bu pencere aşağıda verilen şekle dönüşür. Visual Studio tarafından hazırlanan **InitializeComponent**() metodunda formun ve formun üzerine yerleştirilen nesnelerin özellikleri ayarlanmaktadır. Bu metot "Form1.cs" adlı kod dosyasındaki Form1 sınıfının yapıcı metodunda işletilmektedir.



Bu ekran görüntüsü alındığı sırada Form1.Designer.cs dosyası içinde hazırlanan tek nesne Form1 (buradaki adıyla **this**) olduğu için **Windows Forms Designer generated code** bloğunda yalnızca Form1'le ilgili satırlar vardı. Bu sırada Properties penceresinde formun **Click** olayı çift tıklanırsa bu olay meydana geldiğinde işletilecek metodun kalıbı hazırlanır ve Click olayıyla ilişkilendirilir. Bu ilişkilendirmenin nasıl yapıldığını aşağıda görebilirsiniz.



Properties penceresinde herhangi bir olayın adı çift tıklandığı zaman 2 ayrı ayarlama yapılmaktadır: Metodun hazırlanması ve bu metodun olayla ilişkilendirilmesi. Daha sonra hazırladığınız metodu silmek istemeniz halinde metodun olayla ilişkilendirildiği satırı "Form1.Designer.cs" dosyasından silmeniz gerekir.

Tekrar etmek gerekirse; form veya forma yerleştirilen nesnelerin Properties penceresinde listelenen olaylarından birisi çift tıklandığında, Visual Studio o nesne için bir metot hazırlayıp bu metodu olayla ilişkilendirmektedir. Bu söylenenlerin anlaşılmasına katkı olması için formun üzerine bir Button yerleştirip aşağıdaki gibi bir metot hazırladık.

```
private void yesil_dugme(object sender, EventArgs e)
{
 button1.BackColor = Color.SpringGreen;
}
```

Bu şekilde düzenlenen metot işletilirse formun üzerine yerleştirilip "Form1" sınıfına üye yapılan "button1" nesnesinin **BackColor** özelliği yani zemin rengi değiştirilir. Fare işareti düğmenin üzerine götürüldüğü zaman bu metodun işletilmesini istediğimiz için, bu metodu "button1" nesnesinin **MouseHover** olayıyla ilişkilendireceğiz.

Bu metodu hazırlama ve ilgili olayla ilişkilendirme işlemini Visual Studio'dan yararlanarak yapmak isteseydik Properties penceresinde **MouseHover** olayını çift tıklayıp hazırlanan metoda yukarıda verilen satırı yazardık. Bu işlemi Visual Studio'dan yararlanmadan yapmak için formun **Load** olayını temsil edecek metodu aşağıdaki gibi düzenledik.

```
private void Form1_Load(object sender, EventArgs e)
{
 this.button1.MouseHover += new EventHandler(this.yesil_dugme);
}
```

Bu hazırlıktan sonra proje çalıştırılıp fare işareti düğmenin üzerine götürüldüğünde düğmenin zemin rengi kendiliğinden yeşil olur. Bu şartlarda fare işareti düğmenin üzerinden uzaklaştırılsa düğme yeşil olarak kalmaya devam eder. Fare işareti düğmenin üzerinden uzaklaştırıldığında düğme ile ilgili olarak **MouseLeave** olayı meydana gelir.

Properties penceresinde **MouseLeave** olayını çift tıklayıp bu olayla ilişkilendirilen bir metot hazırlayıp düğmenin renginin eskisi gibi olmasını sağlayabilirsiniz. Olay kavramanın anlaşılması için Visual Studio'dan yardım almadan üzerinde çalıştığımız Form1 sınıfı dahilinde aşağıdaki gibi bir metot hazırladık.

```
private void gri_dugme(object sender, EventArgs e)
{
 button1.BackColor = Color.Gray;
}
```

Fare işareti düğmenin üzerinden uzaklaştırıldığında meydana gelen **MouseLeave** olayı ile bu metot arasında ilişki kurmak için formun Load olayını temsil eden metoda 2. bir satır yazdık. Bu 2 satır ile yapılan işlemler hakkında daha sonra bilgi verilecektir.

```
private void Form1_Load(object sender, EventArgs e)
{
 this.button1.MouseHover += new EventHandler(this.yesil_dugme);
 this.button1.MouseLeave += new EventHandler(this.gri_dugme);
}
```

MessageBox Sınıfı

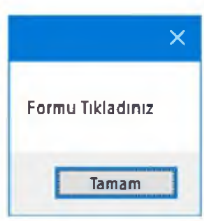
C# uygulaması dahilinde diyalog kutusu içinde kullanıcıya mesaj verilmek istendiğinde **MessageBox** sınıfının **Show()** metodu kullanılmaktadır. Show() metodu, bazısı isteğe bağlı 4 parametreye sahiptir. İlk parametrede kullanıcıya verilecek mesaj, 2. parametrede ise diyalog kutusunun başlığı belirtilmektedir. 3. parametrede diyalog kutusunda yer alacak düğmeler seçilmektedir. MessageBox sınıfıyla ekrana diyalog kutusu getirilip kullanıcıya nasıl mesaj verildiğini anlatmak için aşağıda verilen metodu hazırladık.

```
private void Form1_Click(object sender, System.EventArgs e)
{
 MessageBox.Show("Formu Tıkladınız");
}
```



Burada kullanılan MessageBox bir sınıf, Show() ise bu sınıfın bir metodudur. Dolayısıyla bu sınıf bir namespace'in üyesi olmalıdır. İlgili namespace ise .NET Framework ile gelen bir DLL dosyasında yer almalıdır. MessageBox sınıfı System.Windows.Forms adlı namespace'te tanımlıdır. Bu namespace Visual Studio tarafından hazırlanan formlara ait kod dosyasında **using** bloğunda işaret edildiği için MessageBox sınıfının metotlarını direk kullanabiliyorsunuz.

Bu kodda MessageBox sınıfının **Show()** metodunun yalnızca ilk parametresi kullanıldı. MessageBox sınıfının Show() metodu bu hali ile işletilirse 1. parametre olarak verilen bilgi mesaj olarak ekrana getirilir ve diyalog kutusunda yalnızca Tamam düğmesi olur.



Show() metodu ile ekrana getirilen diyalog kutusunun başlık metnine sahip olmasını istiyorsanız bu metni 2. parametre olarak vermelisiniz. MessageBox sınıfı sayesinde ekrana diyalog kutusu getirmedeki amacınız kullanıcıya yalnızca bilgi vermek ise ilk 2 parametre yeterlidir.

```
private void Form1_Click(object sender, System.EventArgs e)
{
 MessageBox.Show("Formu Tıkladınız", "Onaylama");
}
```

Show() metodu sayesinde ekrana getirdiğiniz diyalog kutusunda kullanıcıya seçim yapma imkanı vereceksiniz 3. parametre ile diyalog kutusunda yer alacak düğmeleri belirtmeniz gerekir. **Show()** metodu aşağıdaki gibi kullanılırsa diyalog kutusunda Evet, Hayır ve İptal düğmeleri yer alır.

```
MessageBox.Show("Bu Satır Silinecek", "Onaylama",
                MessageBoxButtons.YesNoCancel);
```

MessageBox sınıfının Show() metodu ile ekrana getirilmek istenen diyalog kutusunda Durdur, Yeniden Dene ve Yoksay düğmelerinin olması isteniyorsa 3. parametre olarak "MessageBoxButtons.AbortRetryIgnore" sabiti kullanılmalıdır. Yukarıda verilen satır işletildiğinde aşağıdaki gibi bir sonuç alınır.



MessageBox sınıfı ile ekrana getirilen bu diyalog kutusunda tıklanan düğmeyi öğrenip ona göre işlem yapabilirsiniz. **Show()** metodu, tıklanan düğmeye bağlı olarak geriye **DialogResult** tipinde bir bilgi göndermektedir. Tıklanan düğmeyi öğrenmek için bir değişken tanımlayıp yukarıda verilen kodu aşağıdaki gibi değiştirdik.

```
DialogResult Onay = MessageBox.Show("Bu Satır Silinecek","Onaylama",
    MessageBoxButtons.YesNoCancel);
if (Onay == DialogResult.Yes)
    MessageBox.Show("Yes düğmesi tıklandı");
if (Onay == DialogResult.No)
     MessageBox.Show("No düğmesi tıklandı");
```

MessageBox sınıfının Show() metodu sayesinde ekrana getirilen diyalog kutusundaki mesaj metninin önüne ünlem(!) veya soru(?) işaretinin konulmasını istiyorsanız bunu 4. parametre olarak belirtmeniz gerekir. Bu sınıfın Show() metodunun 4. parametresinde aşağıdaki gibi **MessageBoxIcon.Warning** seçeneği kullanılırsa mesajın önüne ünlem(!) konulur.

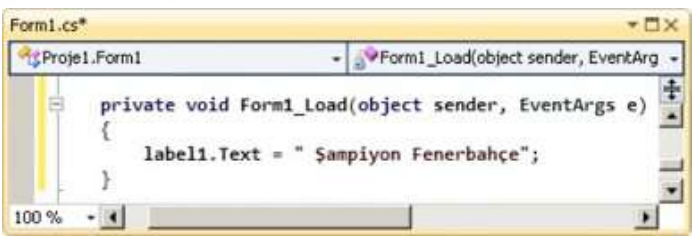
```
DialogResult Onay;
Onay = MessageBox.Show("Bu Satır Silinecek","Onaylama",
       MessageBoxButtons.YesNoCancel,MessageBoxIcon.Warning);
```

Show() metodu ile kullanıcıya verilmek istenen mesaj bazen birden fazla satıra bölünüp öyle görüntülenmek istenir. Mesajı birden fazla satıra bölmek istiyorsanız "\n" karakterlerini mesaj metnine eklemeniz yeterlidir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 string Oyuncular = "Ömer Onan" + "\n";
 Oyuncular = Oyuncular + "İbrahim Kutluay";
 Oyuncular = Oyuncular + "\n" + "Mirsat Türkcan";
 Oyuncular = Oyuncular + "\n"+ "Kaya Peker";
 MessageBox.Show(Oyuncular);
}
```

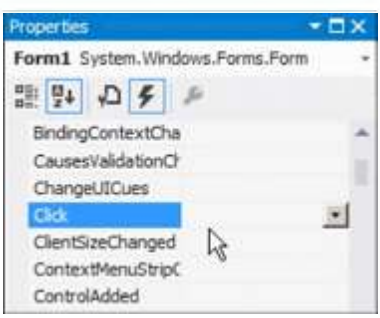
Kod Yazarak Nesnelerin Özelliklerini Değiştirmek

Forma yerleştirdiğiniz Label, TextBox ve Button gibi nesnelerin özelliklerini Properties penceresinden yararlanarak veya kod yazarak ayarlayabilirsiniz. Şimdi forma bir Label nesnesi yerleştirip Text özelliğini kod yazarak değiştireceğiz. Bu değişikliğin proje çalıştırıldığında kendiliğinden olmasını sağlamak için Label nesnesinin **Text** özelliğinde değişikliğe neden olacak satırı formun **Load** olayını temsil eden metoda yazdık.

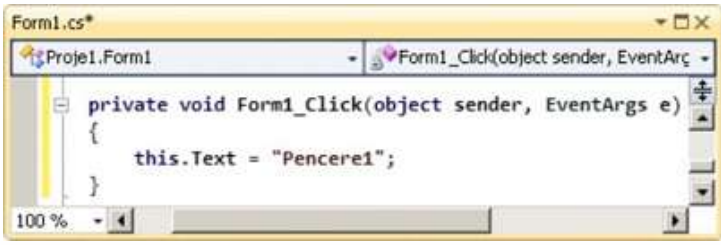


Kod yazarak ilgili nesnenin hangi özelliğini değiştirmek istiyorsanız önce nesnenin, sonra özelliğin adını yazmanız gerekir. İlk formunun **Load** olayını temsil eden metodu bu şekilde düzenlenen proje çalıştırılırsa formun üzerine "Şampiyon Fenerbahçe" yazılır.

Şimdi hazırlayacağız kod sayesinde çalışma anında form tıklandığı zaman formun başlığının değişmesini sağlayacağız. Bu amaçla **Properties** penceresinde formun özellikleri listelenirken **Events** sekmesine geçip **Form** sınıfının olaylarının listelenmesini sağladık.



Çalışma anında form tıklandığında formun başlığının değişmesini sağlamak için **Click** olayını seçtik. Properties penceresinde Click olayı çift tıklanırsa bu olay meydana geldiği zaman işletilecek bir metot hazırlanır. Geri planda bu metot ile Click olayı ilişkilendirilir.



Kendi hazırladığınız metotları silmek istiyorsanız **Code** penceresinde o metoda ait satırları silebilirsiniz. Hazırladığınız metotları kod penceresinde sildiğinizde Properties penceresinde olay adının karşısına yazılan metot adlarını Visual Studio silmektedir.

Yukarıdaki sayfalardan hatırlayacağınız gibi Properties penceresinde herhangi bir olayın adı çift tıklanıp metot hazırlandığı zaman forma ait "Designer.cs" dosyasında **Windows Form Designer generated code** adlı blokta metot ile ilgili olay arasında ilişki kurulmaktadır. Bu nedenle silmiş olduğunuz metot daha önce formun herhangi bir olayı ile ilişkilendirilmişse Designer.cs dosyasındaki ilgili satırı da silmelisiniz. Yoksa hata meydana gelir.

Projelerde Birden Fazla Form Kullanmak

Visual Studio'nun File menüsünden komut verilerek ekrana getirilen **New Project** diyalog kutusundaki **Windows Forms Application** şablonu ile hazırlanan projeler için bir formun otomatik olarak hazırlandığını biliyorsunuz. Üzerinde çalıştığınız projeye 2. bir form eklemek için **Solution Explorer** penceresinde projeye ait kısayol menüsünden **Add** komutu verildiği zaman açılan menüden **Add Windows Form** komutunu verebilirsiniz.

Projeye 2. bir form dahil etmek, "Form2" adında bir class hazırlayıp projeye dahil etmek demektir. Projeye yeni form dahil edilirken hazırlanan sınıf, System.Windows.Forms'deki **Form** sınıfının mirasçısı olduğu için bu sınıfın bütün özellik ve metotlarına sahip olmaktadır. Şimdi Toolbox'tan yararlanıp projedeki ilk formun üzerine bir düğme yerleştirip ve çalışma anında bu düğme tıklandığı zaman 2. formun pencere olarak ekrana gelmesini sağlayacağız.



Projedeki 2. formun ekrana nasıl getirildiğini anlatmadan önce ilk formun nasıl ekrana getirildiğini hatırlatacağız. Visual Studio ile hazırlanan C# uygulamalarında "Program.cs" adında bir kod dosyasının hazırlandığını ve Main() metodunun bu dosyaya konulduğunu biliyorsunuz. Program.cs dosyasındaki Main() metodunda uygulamanın başlangıç formunun örneği alınıp **Application** sınıfının **Run**() metoduna parametre olarak verilerek pencere olarak ekrana getirilmektedir.

Bu işlem için bazı hazırlıkların yapılması ve ekrana getirilmek istenen formdan (class) yola çıkılarak nesne hazırlamak gerekiyor. Bu amaçla "Goster" adını verdiğimiz düğmeyi çift tıklayıp bu düğme çalışma anında tıklandığında işletilecek metodun Visual Studio tarafından hazırlanmasını sağladık. Ardından bu metoda aşağıda verilen satırı yazdık.

```
private void Goster_Click(object sender, EventArgs e)
{
 Form2 F2 = new Form2();
}
```

Projedeki ilk formun üzerindeki düğmenin Click olayını temsil eden metoda yazılan bu satırda Form2'den yola çıkılarak "F2" adında bir nesne hazırlanmaktadır. Sınıflardan yola çıkıp nesne hazırlanırken **new** anahtar kelimesini kullanmak gerektiğini biliyorsunuz. Yukarıdaki sayfalarda belirtildiği gibi C# kodunda sınıfların yapıcı metoduna sınıfın adı verilmektedir. Buna göre Form2 sınıfının yapıcı metodunun adı **Form2()** olmaktadır.

Yukarıda verilen satırda gerçekte 2 işlem yapılmaktadır. İlk işlemde Form2 tipinde bir değişken tanımlanmaktadır. 2. işlemde ise **new** anahtar kelimesi ile Form2 sınıfının yapıcı metodu işletilip bu metodun geriye gönderdiği nesne(yani nesnenin adresi) tanımlanan değişkene aktarılmaktadır. Bu 2 işlemi tek satır yerine aşağıdaki gibi 2 satırda yapabilirsiniz.

```
private void Goster_Click(object sender, EventArgs e)
{
 Form2 F2;
 F2 = new Form2();
}
```

Bu şekilde hazırlanan Form2 tipindeki nesne **Show()** metodu ile pencere olarak ekrana getirilebilir. "Form" sınıfının Show() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Goster_Click(object sender, EventArgs e)
{
 Form2 F2 = new Form2();
 F2.Show();
}
```

2 form içeren bu proje başlatıldığında projenin ilk formu "Program.cs" dosyasında **Application** sınıfının **Run()** metoduna parametre olarak verildiği için kendiliğinden ekrana gelmektedir. Aşağıda verilen ekran görüntüsü Form1'deki düğme tıklanıp yukarıda verilen metodun işletilmesi sağlanıp 2. form ekrana getirildikten sonra alındı.



Yukarıda değişken tanımlayıp nesne hazırlayan satırı düğmenin **Click** olayı ile ilişkilendirilen metoda yazdık. Bu nedenle çalışma anında ekranda Form2 varken Form1'deki düğme 2. kez tıklanırsa aynı nesne tekrar oluşturulur ve 2. form tekrar pencere olarak ekrana getirilir.

Ekranda 2. form varken yukarıda verilen kod işletildiğinde aynı formun 2. veya 3. kopyalarının hazırlanıp ekrana gelmesini engellemek için kullanılabilinecek değişik teknikler olmakla birlikte önceliği ilk akla gelene vereceğiz. Bu amaçla "Göster" düğmesi için hazırladığımız kodu değiştirip **Show()** yerine **ShowDialog()** metodunu kullandık.

```
private void Goster_Click(object sender, EventArgs e)
{
 Form2 F2 = new Form2();
 F2.ShowDialog();
}
```

ShowDialog() metodu ile ekrana getirilen form kapatılmadan projenin ana formuna geçilemez. Dolayısıyla projedeki 2. formun 2. bir kopyasını hazırlayıp ekrana getirmek mümkün

olmaz. Formları ShowDialog() metodu ile ekrana getirmede önemli bir ayrıntı vardır: ShowDialog() metodu ile ekrana getirilen form normal bir şekilde yani Kapat düğmesi tıklanarak kapatıldığında bellekten yok edilmeyip sadece gizlenmektedir.

Bu sorunun önüne geçmek, başka bir deyişle hem 2. formu Show() metodu ile ekrana getirmek hem de ekranda aynı anda 2 kopyasının olmasını engellemek için değişken tanımlama satırı düğmenin Click olayını temsil eden metodun dışına yani class bloğuna alınabilinir.

```
Form2 F2 = new Form2();
private void Goster_Click(object sender, EventArgs e)
{
 F2.Show();
}
```

Bu şartlarda ilk formdaki "Göster" düğmesini tekrar tıklasanız bile 2. form tekrar ekrana getirilmez. Ancak bu şekilde ekrana getirdiğiniz 2. formu kapattıktan sonra **Show()** metodu ile tekrar ekrana getirmek istemeniz halinde hata meydana gelir. Çünkü bu şekilde hazırlanıp Show() metodu ile ekrana getirilen form normal bir şekilde kapatıldığında otomatik olarak bellekten yok edilir. Dolayısıyla yok edilen form tekrar ekrana getirilemez.

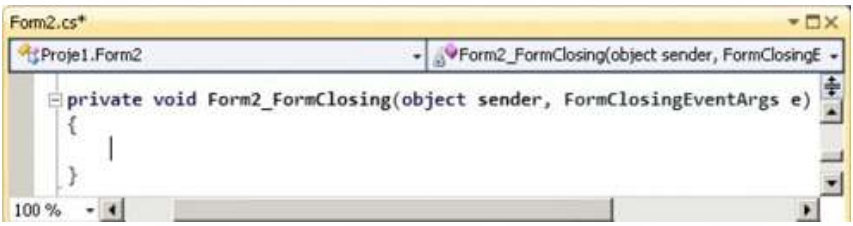
2. formun ekrana getirilmesi sırasında yaşanan bütün sorunların önüne geçmek için formu oluşturan ve Show() metodunun kullanıldığı satırları aynı metoda yazdık ama bu kez form nesnesini **new** anahtar kelimesi ile tekrar oluştururken mevcut olup olmadığını araştırdık.

```
Form2 F2;
private void Goster_Click(object sender, EventArgs e)
{
 if (F2 == null || F2.IsDisposed)
  {
   F2 = new Form2();
   F2.Show();
  }
}
```

Show() metodu ile ekrana getirdiğiniz formları **Hide** metodu ile gizleyebilir veya **Close()** metodu ile kapatabilirsiniz. Close() metodu ile formların nasıl kapatıldığını anlatmak için yukarıda Show() metodu ile ekrana getirdiğimiz 2. formu kapatacağız. Bu amaçla projedeki 2. forma "Kapat" başlıklı bir düğme yerleştirip bu düğmenin Click olayını temsil eden metoda aşağıda verilen satırı yazdık. Bu satır sayesinde 2. form kapatılır.



Çalışma anında aktif form kapatıldığında **FormClosing** olayı meydana gelir. Bu olayı temsil eden metoda kod yazarak formun kapatılması işlemini kontrol altına alabilirsiniz. Söz konusu form kapatıldığında işletilecek metodun yapısını aşağıda görebilirsiniz.



Bu metodun "e" adındaki 2. parametresinin **Cancel** özelliğini **true** yaparak formun kapatılmasını önleyebilirsiniz. "e" parametresinin Cancel özelliğinin nasıl kullanıldığını anlatmak için 2. formun **FormClosing** olayı ile ilişkilendirdiğimiz metodu aşağıdaki gibi düzenledik. Kapatılmak istenen formun "e" parametresinin Cancel özelliği true iken form kapatılmaz.

```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
{
 System.Windows.Forms.DialogResult Cevap;
 Cevap = MessageBox.Show("Form2 kapatılsın mı?", "Kapatma",
                         MessageBoxButtons.YesNo);
 if (Cevap == DialogResult.No)
  {
   e.Cancel = true;
  }
}
```

Projenin Başlangıç Formunu Seçmek

Projenizde birden fazla form varsa proje başlatıldığında ilk olarak hangi formun ekrana geleceği konusunda seçim yapmak isteyebilirsiniz. Projeler, varsayım olarak projeye ilk sırada dahil edilen formdan başlarlar. Başlangıç formu ile ilgili ayarlamalar "Program.cs" dosyasında yapılmaktadır. Aşağıda verilen kod Program.cs dosyasından alındığı sırada üzerinde çalıştığımız proje 2 forma sahipti ve projenin başlangıç formu Form1'di.

```
static void Main()
{
 Application.EnableVisualStyles();
 Application.SetCompatibleTextRenderingDefault(false);
 Application.Run(new Form1());
}
```

Projenin başlangıç formunun Form2 olmasını istiyorsanız **Application.Run(new Form1());** satırındaki Form1'i silip yerine "Form2" yazmanız yeterlidir.

```
Application.Run(new Form2();
```

Program.cs dosyasında **new** anahtar kelimesini birden fazla kez kullanıp birden fazla formun kopyası oluşturulup Application sınfının Run() metodu birden fazla kez kullanılabilir. Bu konuda bilgi vermek için 2 forma sahip bir proje hazırlayıp Program.cs dosyasındaki **Main**() metodunu aşağıdaki gibi düzenledik.

```
static void Main()
{
 Application.EnableVisualStyles();
 Application.SetCompatibleTextRenderingDefault(false);
```

```
        Application.Run(new Form1());
        Application.Run(new Form2());
    }
```

Bu şartlarda bu proje çalıştırılırsa **Run**() metodunun kullanıldığı ilk satırda ilk form ekrana getirilir. İlk form kapatıldığında **Run**() metodunun 2. kez kullanıldığı satır işletilir ve 2. form pencere olarak ekrana getirilir.

Tahmin ettiğiniz gibi Visual Studio ile hazırlanan C# uygulamalarında **Main**() metodunun sonuna gelindiğinde çalışma sona ermektedir. Ancak uygulamanın kontrolü Application sınıfının Run() metoduna parametre olarak verilen forma geçtiği için bu form kapatılana kadar uygulama çalışmaya devam eder. Tıpkı Windows'un çalışmasını sona erdirmek için **Masaüstü** adı verilen ana pencerenin kapatılması gerektiği gibi.

Bu şekilde düzenlenen projedeki ilk form başlangıç formu olarak düzenlenebilir. Şimdi bu projedeki ilk formu kullanıcıdan şifre istenen başlangıç formu olarak kullanacağız. Bu amaçla projenin ilk formunu aşağıdaki gibi düzenledik.



Şifre girilip Tamam düğmesi tıklandığında şifre kontrol edilecek. Şifre geçerli değilse projenin çalışması sona erdirilecek. Tamam düğmesi tıklandığında bu formun kapatılmasını sağlamak için düğmenin Click olayını temsil eden metodu aşağıdaki gibi düzenledik.

```
private void Tamam_Click(object sender, EventArgs e)
{
 this.Close();
}
```

Bildiğiniz gibi **Close**() metodu veya başka bir teknikle form kapatıldığında **FormClosing** olayı meydana gelmektedir. Bu nedenle şifre kontrolü yapılan satırları ilk formun FormClosing olayını temsil eden metoda yazdık. C# projelerinin çalışması genellikle **Application** sınıfının **Exit**() metodu ile sona erdirilmektedir.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
 if (textBox1.Text != "12345")
   Application.Exit();
}
```

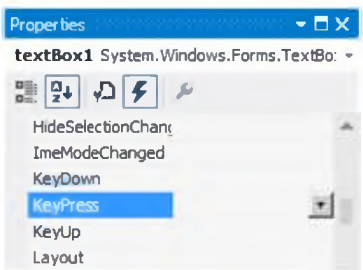
TextBox Kontrolü

Her ne kadar konunun başlığı TextBox olarak seçilmiş bile burada amacımız TextBox kontrolü hakkında bilgi vermekten çok **olay** kavramının anlaşılmasını sağlamaktır. Bu amaçla üzerinde çalıştığımız projenin ilk formunu aşağıdaki gibi düzenledik. Çalışma anında bu TextBox'a girilecek bilgileri **MessageBox** sınıfının **Show**() metodu ile bir diyalog kutusu içinde ekrana getirmek için kod hazırlayacağız.



Yapacağımız ayarlamaya göre çalışma anında Enter tuşuna basılınca TextBox'a o ana kadar girilen bilgiler bir diyalog kutusu içinde ekrana getirilecek. Burada yapılması gereken, basılan tuşun Enter olup olmadığını araştırmaktır.

Çalışma anında imleç TextBox'ın içinde iken klavyenin herhangi bir tuşuna basılması halinde TextBox'la ilgili olarak **KeyPress** olayı meydana gelir. Bu nedenle ilk yapılması gereken, forma yerleştirilen TextBox için **KeyPress** olayı meydana geldiği zaman işletilecek bir metot hazırlamak ve bu metodu KeyPress olayı ile ilişkilendirmektir. Bu amaçla Properties penceresinde TextBox'ın olayları listelenirken KeyPress olayının üzerine gittik.



Properties penceresinde TextBox'a ait KeyPress olayının üzerinde çift tıklama yapılırsa Visual Studio hem **KeyPress** olayı için bir metot hazırlar hem de Code Editor penceresini açar.
TextBox seçili iken Properties penceresinde KeyPress olayının üzerinde çift tıklama yapılırsa Visual Studio aşağıda verilen metodu hazırlar. Metodun adı TextBox'ın adına bağlı olarak farklı olur.

Properties penceresinde KeyPress olayı çift tıklandığında hazırlanan metot için geri planda Designer.cs dosyasında değişiklik yapılarak TextBox1'in **KeyPress** olayıyla ilişkilendirilir. Visual Studio'nun **textBox1_Keypress()** adını verdiği metot KeyPress olayıyla ilişkilendirilmezse tuşa basıldığında bir karşılık alınamaz ve bu metot işletilmez.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
 {
 }
```

Properties penceresinde seçilen olaya bağlı olarak Visual Studio tarafından otomatik olarak hazırlanan bu metotta, metot adından sonra parantezlerin içinde "**object sender, KeyPressEventArgs e**" bilgisi var. Daha açık anlatımla, TextBox sınıfının KeyPress olayı için hazırlanan bu metot **object** ve **KeyPressEventArgs** tipinde 2 parametreye sahiptir.

KeyPress olayı meydana geldikçe basılan tuşla ilgili olarak metoda bu parametreler aracılığıyla bilgi gönderilir. Parametreler, metot adından sonra parantez içine yazılmaktadır. Basılan tuşu temsil eden bilgi metoda gönderileceği için **KeyPressEventArgs e** ile gelecek bilgi belirtiliyor. Buradaki "e", her tuşa basma işlemi sırasında basılan tuşun Ascii kodunun aktarıldığı **KeyPressEventArgs** tipindeki parametredir.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
 if (e.KeyChar == 13)
    MessageBox.Show(textBox1.Text);
}
```

Bu metotta "if" deyimi ile en son basılan tuşun ASCII kodu 13 olan Enter olup olmadığı araştırılıyor. En son basılan tuşun kodu 13 ise **MessageBox** sınıfının **Show()** metodu ile "Text" özelliğinin içeriği ekrana getiriliyor. En son basılan tuşun kodu 13 değilse imleç TextBox'ın içinde beklemeye devam eder.

Şimdi yukarıda verilen metotta değişiklik yapıp TextBox'a girilen karakterlerin ASCII kodunu öğrenip Show() metodu ile ekrana getireceğiz. TextBox'a yazılan karakterin kodunu KeyPress olayından yararlanarak bulmak mümkündür ama basılan tuşun ASCII kodu **KeyDown** olayına ait metot ile daha kolay öğrenilmektedir. Aşağıda verilen kod dikkatle incelenirse TextBox sınıfının KeyDown olayı için hazırlanan metodun parametre yapısı KeyPress olayını temsil edebilen metottan farklıdır.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
 int kod = e.KeyValue;
 MessageBox.Show(Kod.ToString());
}
```

Bu metotta ilk olarak "kod" adında integer bir değişken tanımlanmakta ve **KeyDown** olayının meydana gelmesine neden olan tuşun Ascii kodu **KeyEventArgs** tipindeki "e" parametresi aracılığıyla bu değişkene aktarılmaktadır. Aşağıda verilen ekran görüntüsü çalışma anında TextBox'a "A" harfi girilip bu metodun işletilmesi sağlandıktan sonra alındı.



Bu sonucu KeyPress olayından yararlanarak almak isteseydik aşağıdaki gibi kod yazmak gerekirdi. MessageBox sınıfı sayesinde ekrana getirilen diyalog kutusunu kapatırsanız bastığınız tuşun kendisi (burada "A") TextBox'a yazılır. Burada parantezlerin içine yazılan (byte) sayesinde dönüştürme yapılmaktadır.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
 byte Kod = (byte) e.KeyChar;
 MessageBox.Show(Kod.ToString());
}
```

Şimdi ise forma 2. bir TextBox yerleştirip ve ilk TextBox'a girilen her karakteri aynı zamanda 2. TextBox'a aktaracağız. Bu amaçla ilk TextBox'ın **KeyDown** olayını temsil eden metodu aşağıdaki gibi düzenledik. Bu metoda yazılan satırda KeyEventArgs tipindeki "e" parametrenin **KeyData** özelliği 2. TextBox'ın mevcut içeriğine eklendi.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
 textBox2.Text = textBox2.Text + e.KeyData;
}
```

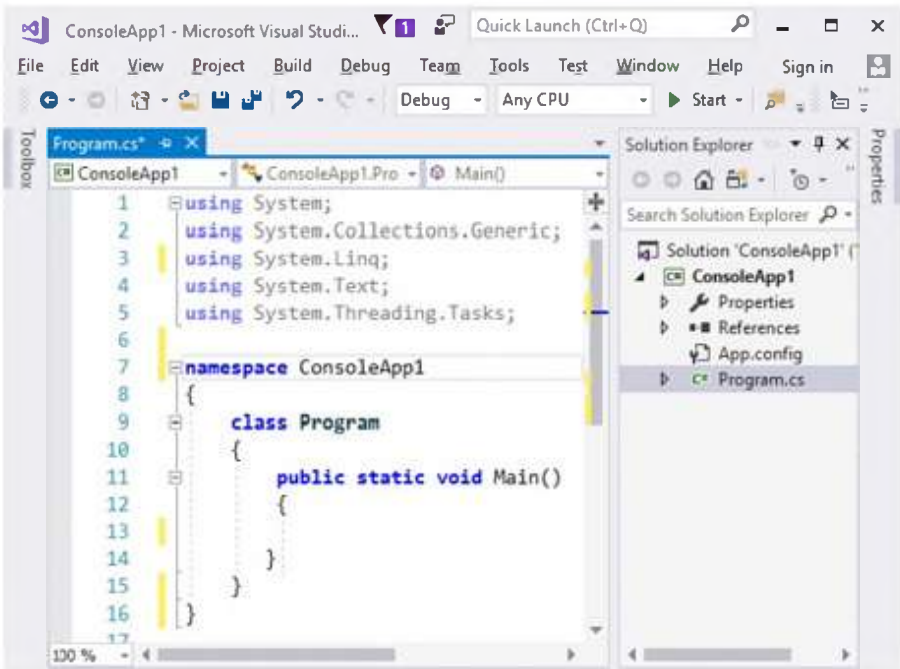
3. BÖLÜM

KONSOL UYGULAMALARI

Konsolda çalıştırılmak üzere hazırlanan .NET uygulamalarına konsol uygulaması adı verilmektedir. Programcılar konsol uygulamalarını özel durum ve ihtiyaçlar için hazırlıyor. Buna rağmen bu bölümde konsol uygulamalarından söz etme nedenimiz bir önceki bölümde anılan referans, namespace, class ve metot kavramlarından tekrar söz etmektir. Çünkü C# programcılığı konusunda henüz yolun başında olan birisi için bu kavramlar önemlidir. Bu kavramlara sahip olunmadan ilerlemek sorunlara neden olabilmektedir.

Bu kitabın ilk bölümünde konsol uygulamalarını Not Defteri'nden yararlanarak hazırlamış ve konsolda kendimiz derlemiştik. Bu bölümde üzerinde konuşacağımız konsol uygulamalarını Visual Studio ile hazırlamak istediğimiz için Visual Studio'nun **New Project** diyalog kutusunu ekrana getirip **Console Application** şablonunu seçtik.

Console Application şablonunu seçip uygulamaya vermek istediğiniz adı **Name** kutusuna yazıp **Location** seçeneği ile uygulamanın tutulacağı klasörü seçip OK düğmesini tıklarsanız Visual Studio aşağıdaki gibi bir kod dosyası hazırlar.



Konsol uygulaması dahilinde hazırlanan kod dosyasına "Program.cs" ve Visual Studio tarafından hazırlanan namespace'e projenin adı verilmektedir. Ayrıca hazırlanan sınıfa "Program" adı verilmekte ve konsol uygulamasının başlangıç olacak noktası Main() metodu bu sınıfın içinde hazırlanmaktadır. "Program.cs" dosyası içinde hazırlanan sınıfa başka bir ad verebilir veya aynı dosya içinde başka sınıflar hazırlayabilirsiniz. Söz konusu uygulama ister

Windows Forms ister Console Application olsun, Main() metodunun hazırlandığı "Program.cs" dosyasına genellikle fazla müdahale edilmez.

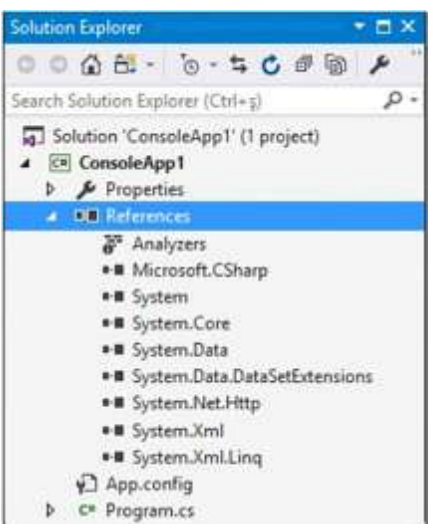
Bu ekran görüntüsünü tekrar niyetine maddeler halinde yorumlarsak;

- Visual Studio ile "ConsoleApp1" adında bir konsol uygulaması hazırladık. Visual Studio kendiliğinden "Program.cs" adında bir kod dosyası ve kod dosyası dahilinde "Program" adında bir class hazırladı.
- "Program" adı verilen class'ta uygulamanın başlangıç noktası olarak kullanılacak Main() metodu Visual Studio tarafından hazırlandı.
- Visual Studio ayrıca uygulama ile aynı ada sahip bir namespace hazırlayıp "Program" adı verilen sınıfı bu namespace'e üye yaptı. Namespace hazırlama işlemi **namespace** anahtar kelimesi ile yapılmaktadır.

Visual Studio tarafından hazırlanıp proje ile aynı ada sahip namespace'in adını "Program.cs" dosyası aracılığıyla değiştirebilirsiniz. Namespace adı değişikliği öncesi düşünülüp uygulamanın işlevini çağrıştıran bir adın seçilmesi yerinde olacaktır.

Namespace'lerin asıl işlevinin uygulama dahilinde hazırlanacak class, struct, enum veya interface'leri gruplamak olduğunu bilmek gerekir. Konunun devamında ve bir sonraki bölümde namespace'ler hakkında bilgi verilmeye devam edilecektir. Namespace'lerle ilgili olarak şimdilik bilmenizi istediğimiz şudur: Visual Studio ile yeni bir uygulama hazırlandığında uygulama ile aynı ada sahip bir namespace hazırlanmakta ve uygulama dahilinde hazırlanan class, struct, interface'ler bu namespace'e üye yapılmaktadır.

Visual Studio tarafından hazırlanan konsol uygulamasına **System** ve **System.Core** gibi çok sayıda referans dahil edilmekte ve bu referanslarda yer alan bazı namespace'lerdeki sınıfları direk kullanılabilmeniz için **using** deyimi ile ayarlama yapılmaktadır. Bu referansları Solution Explorer penceresinde görebilirsiniz. Bu referansların her birisinin .NET Framework ile gelen birer DLL dosyası olduğunu tahmin ediyorsunuzdur.

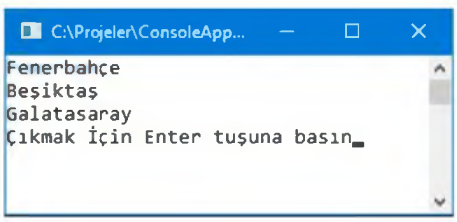


Microsoft firması tarafından C# uygulaması geliştirirken kullanabilmemiz için çok sayıda sınıf, struct, enum ve interface önceden hazırlanmıştır. Sayıları çok fazla olan bu sınıf ve struct'lar işlevlerine göre gruplanıp DLL dosyası yani referans olarak paketlenip .NET Framework ile birlikte bilgisayarınıza kurulmaktadır. Visual Studio ile hazırladığınız C# uygulamasında kullanacağınız sınıfların yer aldığı referansların Solution Explorer penceresinde listelenmesi gerekiyor.

Visual Studio ile hazırlanan konsol uygulamasında herhangi bir değişiklik yapılmadan çalıştırılırsa bir DOS penceresi açılır ve hemen kapanır. Çünkü bu uygulamanın başlangıç noktası olan Main() metodunda işlem yapan herhangi bir satır yoktur ve uygulamanın çalışması Main() metodunun sonunda bitmektedir. Programın çalışmasının hemen sona ermesini engellemek amacıyla **Main**() metoduna birkaç satır yazdık.

```
class Program
{
 static void Main(string[] args)
  {
   Console.WriteLine("Fenerbahçe");
   Console.WriteLine("Beşiktaş");
   Console.WriteLine("Galatasaray");
   Console.WriteLine("Çıkmak İçin Enter tuşuna basın");
   Console.ReadLine();
  }
}
```

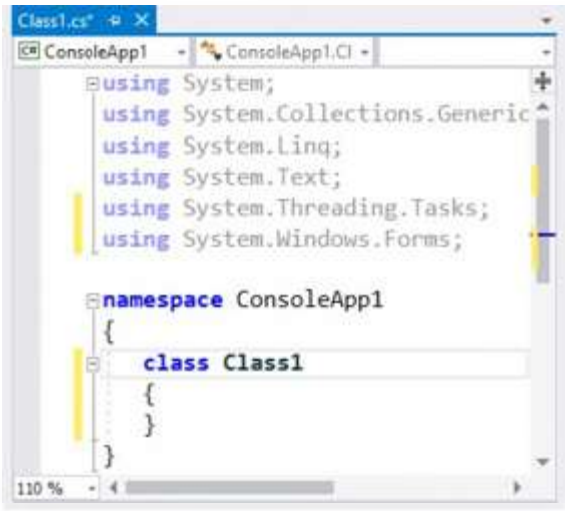
Bu uygulama Visual Studio'nun **Debug** menüsünden komut verilerek çalıştırıldığında aşağıdaki gibi bir sonuçla karşılaşılır. Bu sırada Enter tuşuna basılırsa uygulamanın çalışması sona erer ve tekrar Visual Studio penceresine dönülür.



Uygulamanın birden fazla sınıfında Main() metodu varsa Visual Studio'nun **Project** menüsünden komut verip **Project Properties** diyalog kutusunu ekrana getirip **Startup Object** seçeneği ile ayarlama yapabilirsiniz.

Konsol uygulamasında birden fazla kod dosyası ve birden fazla class olabilir. Örnek olması için bu konsol uygulamasına 2. bir CS dosyası dahil edip bu kod dosyasında başka bir class hazırlayacağız. C# programlama dili aynı kod dosyası içinde birden fazla class'ın hazırlanmasına izin verdiği için her class için ayrı bir kod dosyası hazırlamak şart değildir.

Üzerinde çalıştığınız uygulamaya Class hazırlamak üzere 2. bir CS dosyası dahil etmek istiyorsanız **Project** menüsünden komut verip **Add New Item** diyalog kutusunu ekrana getirmelisiniz. Add New Item diyalog kutusunda **Class** şablonu seçilip, kod dosyasına verilmek istenen ad yazılıp **Add** düğmesi tıklanırsa kod dosyası ile aynı ada sahip bir class hazırlanır.



Add New Item diyalog kutusunda listelenen **Class** şablonu ile hazırlanıp uygulamaya dahil edilen kod dosyasında, kaynaklarından yararlanılması muhtemel 6 namespace **using** deyimi ile işaret edilmektedir.

Bu kod dosyası yakından incelendiğinde "Class1" adlı class'ın "ConsoleApp1" adındaki mevcut namespace'e üye yapıldığı görülür. Bu namespace adı "Program.cs" dosyasında da kullanılmıştı. Bundan şu sonucu çıkarabiliriz: Mevcut şartlarda hem "Program.cs" dosyasındaki "Program" sınıfı hem de "Class1.cs" dosyasındaki "Class1" aynı namespace'e üyedir.

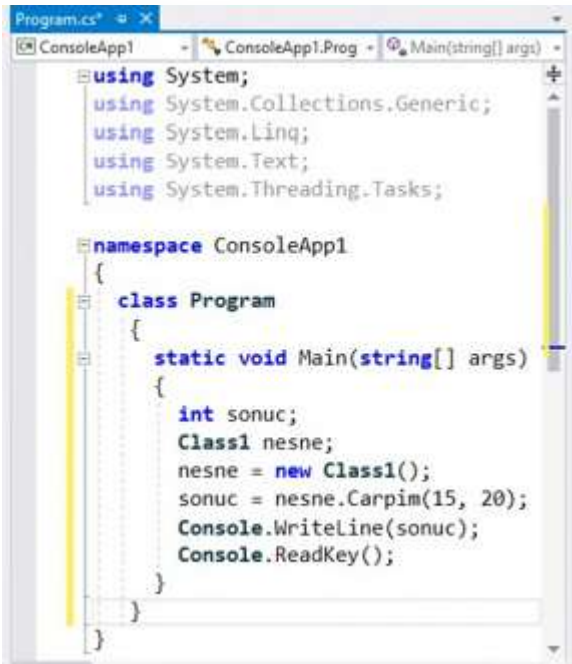
Karşınızda "Class1.cs" dosyasının içeriği varken **namespace** anahtar kelimesinden sonra farklı bir namespace adı yazarsanız bu uygulamanın 2 namespace'e sahip olmasını sağlamış olursunuz.

Şimdi üzerinde çalıştığımız uygulamaya sonradan dahil ettiğimiz 2. kod dosyasındaki sınıfta bir metot hazırlayacağız. Hazırlayacağımız metot, kendisine parametre olarak verilen 2 sayıyı çarpıp bulduğu sonucu geriye gönderecektir.

```
namespace ConsoleApp1
{
 class Class1
 {
  public int Carpim(int x, int y)
   {
    return x * y;
   }
 }
}
```

Buradaki amacımız metotların nasıl hazırlanıp kullanıldığını kısaca anlatmak olduğundan metodun yaptığı işlemin önemi yoktur. Diğer yandan başka bir yazar "daha önce yayınlanan kitabımda 'Carpim' adında ve 2 sayıyı birbirleri ile çarpan bir metoda yer vermiştim" deyip bizi fikrini çalmakla suçlayıp dava edebilir.

Bu kod dosyasındaki class'ta bu şekilde "Carpim" adında ve 2 parametreye sahip metodu hazırladıktan sonra ilk CS dosyasına geçip **Main**() metodunu aşağıdaki gibi düzenledik.



Uygulamanın ilk sınıfındaki Main() metodunda ilk olarak **int** ve kendi hazırladığımız sınıf tipinde (Class1) 2 değişken tanımladık. Devamında **new** anahtar kelimesiyle "Class1" sınıfının örneğini hazırlayıp "nesne" adını verdiğimiz değişkene aktardık. Sınıftan yola çıkıp nesne hazırladıktan sonra bu nesnenin üzerinden **Carpim**() metodunu işleterek 2 sayının çarpımını bulup konsola yazdık.

Şimdi konsol uygulamamızın son durumunu özetleyeceğiz:

- Uygulamamızda "Program" ve "Class1" adında 2 sınıf bulunmaktadır. Bu sınıflar ayrı kod dosyalarında olsalar bile aynı namespace'e üyeler.
- "Class1" adını verdiğimiz sınıfımızı başka bir namespace'e üye yapsaydık bu şekilde sınıftan yola çıkıp nesne hazırlayamazdık. Sınıf adından önce namespace adını yazmamız gerekirdi.
- Class1 adını verdiğimiz sınıfta Carpim() adında bir metot bulunmaktadır. Bu metodu işletmek için Main() metodunun içinde Class1 tipinde bir nesne hazırladık.

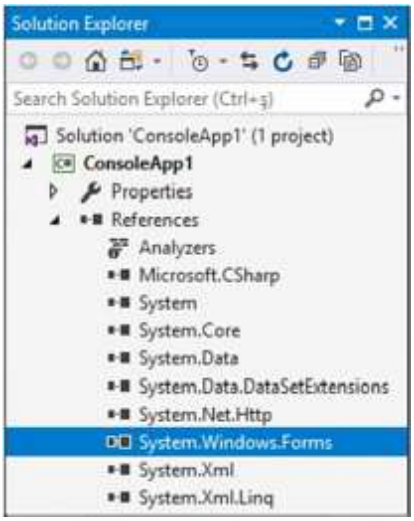
Console Application şablonu ile hazırlanan projenin kodlarına baktığınızda konsolun açılmasına yönelik herhangi bir deyim veya metodun olmadığını görürsünüz. Console Application şablonu ile hazırlanan proje çalıştırıldığında DOS penceresinin açılmasını sağlayan ayarlama **Project** menüsünden komut verilerek ekrana getirilen **Project Properties** penceresinde **Output type** liste kutusunda yapılmaktadır.

Konsol Uygulamasında Pencere Hazırlamak

Şimdi yukarıda uygulamaya sonradan dahil ettiğimiz sınıfta (Class1 adını verip Class1.cs dosyasında kodladığımız) Windows formu hazırlayıp ekrana getireceğiz. Bunu yapabilmek için .NET Framework ile gelen **Application** ve **Form** sınıflarından yararlanacağız.

Application sınıfından yararlanabilmek ve pencere hazırlayıp ekrana getirebilmek için "System.Windows.Forms" referansının **Solution Explorer** penceresinde listelenmesi gerekiyor. Projeyi hazırlarken **Windows Forms Application** şablonundan yararlanmış olsaydık bu referans Solution Explorer penceresinde zaten listelenirdi.

Uygulamaya referans ekleme işlemi Solution Explorer penceresinde **References** seçeneğine ait kısa yol menüsünden **Add Reference** komutu verilerek yapılmaktadır. Aşağıda verilen ekran görüntüsünü "System.Windows.Forms" adlı referansı konsol uygulamıza ekleyip Solution Explorer penceresinde listelenmesini sağladıktan sonra aldık.





Bir önceki bölümden hatırlayacağınız gibi **Windows Forms Application** şablonu ile hazırlanan projelerde Main() metodunda **Application** sınıfının **Run**() metodu kullanılmaktadır. Çünkü C# uygulaması ile Windows arasındaki mesaj alış verişini Application sınıfı sağlamaktadır. Bu nedenle C# uygulaması geliştirmek isteyenlerin Application sınıfı hakkında bilgi edinmeleri gerekmektedir.

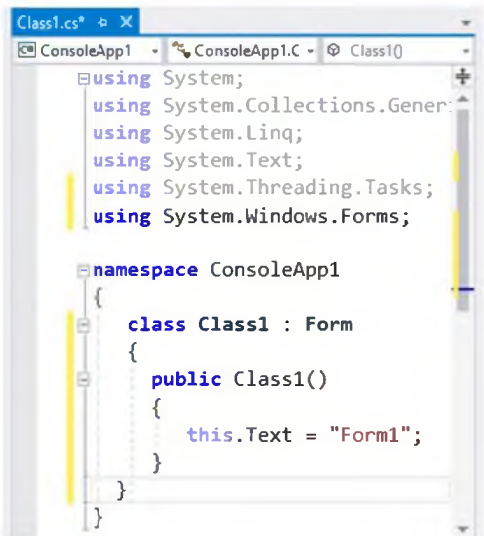
Konsol uygulaması dahilinde Windows uyumlu pencere hazırlama amacımız, konu üzerinde düşünmenizi sağlamak ve bu sayede bazı konulardan söz edebilmektir. Tabi akla şöyle bir soru gelebilir: Madem konsol uygulaması dahilinde Windows uyumlu pencere açıbiliyoruz, Windows Forms Appliacation dahilinde konsolla diyalog kurmak mümkün mü? Evet, bu teknik olarak mümkündür. Ancak zahmetlidir, Windows'un ilgili fonksiyonlarına başvurmak gerekir.

System.Windows.Forms.dll referansı uygulamaya dahil edildikten sonra **usign** deyimi ile namespace'deki sınıfları kullanmak istediğinizi belirtirseniz sınıf adlarından önce "System.Windows.Forms" yazmak zorunda kalmazsınız.

Konsol uygulaması üzerinde çalışırken "System.Windows.Forms" referansının Solution Explorer penceresinde listelenmesi sağlandıktan sonra bu referanstaki bütün namespace ve bu namespace'lerdeki sınıflardan yararlanma imkanını elde ederiz. Örneğin konsol uygulamasında **MessageBox** sınıfından yararlanma imkanı elde ettik.

```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
  {
    MessageBox.Show("Konsol Uygulaması Başlatılacak");
   }
 }
}
```

Şimdi ise konsol uygulaması dahilinde bir form hazırlayıp ekrana getireceğiz. Bu amaçla konsol uygulamasındaki 2. sınıfın System.Windows.Forms'de bulunan **Form** sınıfının mirasçısı olmasını sağladık. Ayrıca bu CS dosyasında class ile aynı ada sahip **public** bir metot hazırlayıp bu metotta formun yani "Class1" sınıfının **Text** özelliğini ayarladık.



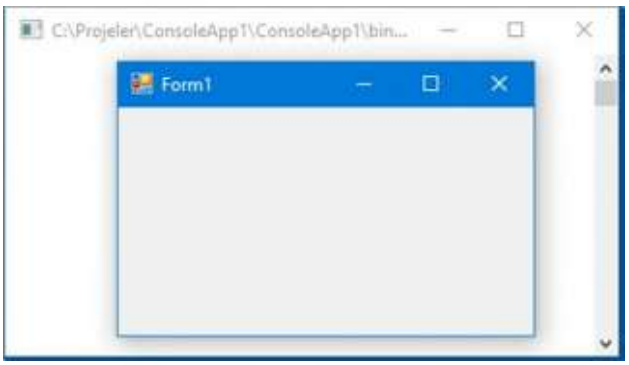
Sınıf ile aynı ada sahip metotlara **yapıcı metot (Constructur)** adı verilmektedir. Bu metotların işlevi hakkında daha sonra bilgi verilecektir.

Hazırlamak istediğiniz sınıf, mevcut sınıflardan hangisinin mirasçısı olacaksa o sınıfın adını hazırlamak istediğiniz sınıfın adından sonra yazmalısınız. "Class1" adlı sınıfımızın .NET Framework ile gelen "Form" sınıfının mirasçısı olmasını istediğimiz için **class** anahtar kelimesinin olduğu satırda bu sınıfın adını yazdık.

Uygulamaya sonradan dahil ettiğimiz CS dosyasındaki "Class1" adlı sınıfı bu şekilde düzenledikten sonra "Program.cs" dosyasına geçtik. Bu dosyadaki Main() metodunda **new** anahtar kelimesi ile Class1'den yola çıkıp bir nesne hazırladık ve bu nesneyi **Application** sınıfının **Run**() metoduna parametre olarak verdik.

```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
  {
   Class1 Form1 = new Class1();
   Application.Run(Form1);
  }
 }
}
```

.NET Framework ile gelen **System.Windows.Forms**'deki **Form** sınıfından yararlanılarak hazırlanan formlar standart özelliklere sahip olurlar. Bu kod işletildiğinde ekrana aşağıdaki gibi standart özelliklere sahip bir pencere(form) getirilir ve bu pencere kapatılıncaya kadar konsol uygulaması çalışmaya devam eder. Başka bir deyişle Application sınıfının Run() metoduna parametre olarak verilen form kapatılana kadar uygulamanın çalışması sürer.



Yukarıda anlatılan şekilde hazırladığımız form nesnesini ekrana getirmek için **Application** sınıfının **Run**() metodundan yararlanmak zorunda değilsiniz. Form1'i ekrana getirmek için **Form** sınıfının **ShowDialog**() metodundan yararlanabilirsiniz.

```
static void Main(string[] args)
{
 Class1 Form1 = new Class1();
 Form1.ShowDialog();
}
```

Konsol uygulaması dahilinde bu şekilde form hazırlamak size mantıklı gelmeyebilir. Buradaki amacımız deneysel. Şimdi kod yazarak bir **ListBox** nesnesi hazırlayıp forma yerleşzireceğiz. Bu amaçla yukarıda verdiğimiz koda ekleme yapıp bir ListBox nesnesi hazırlayıp bazı özelliklerini ayarladık.

```
public class Program
{
 static void Main(string[] args)
  {
   Class1 Form1 = new Class1();
   System.Windows.Forms.ListBox Liste1;
   Liste1 = new System.Windows.Forms.ListBox();
   Liste1.Top = 25;
   Liste1.Left = 25;
  }
}
```

Main() metoduna eklediğimiz bu 4 satırla bir ListBox nesnesi hazırlayıp **Top** ve **Left** özelliklerini ayarladık. .NET Framework ile gelen ListBox sınıfı System.Windows.Forms'de bulunmaktadır. Bu şekilde ListBox nesnesi hazırladıktan sonra sıra bu nesneyi formun üzerine yerleştirmeye gelir. Bu amaçla yukarıda verdiğimiz koda ekleme yapıp ListBox nesnesinin **Parent** özelliğinde ayarlama yaptık. En son olarak ListBox'ta iki F1 pilotu adının

listelenmesini sağladık. Buradaki amacımız ListBox sınıfı hakkında bilgi vermek olmadığı için ayrıntıya girmiyoruz, yaptığımız işlemle ilgili açıklama yapmıyoruz.

```
class Program
{
 static void Main(string[] args)
 {
  Class1 Form1 = new Class1();
  System.Windows.Forms.ListBox Liste1;
  Liste1 = new System.Windows.Forms.ListBox();
  Liste1.Top = 25;
  Liste1.Left = 25;
  Liste1.Parent = Form1;
  Liste1.Items.Add("Fernando Alonso");
  Liste1.Items.Add("Michael Schumacher");
  Form1.ShowDialog();
 }
}
```

Yukarıda konsol uygulamasında 2. bir sınıf hazırlamış ve bu sınıfın **Form** sınıfının mirasçısı olmasını sağlamıştık. Devamında ise bu sınıftan yararlanarak form nesnesi hazırlayıp pencere olarak ekrana getirmiştik. Form nesnesi hazırlamak için ayrıca bir class hazırlamak zorunda değilsiniz. Konunun geri planı hakkında bilgi vermek için yukarıda verilen konsol uygulamasındaki CS uzantılı 2. kod dosyasını sildik ve ilk kod dosyasını(Program.cs) aşağıdaki gibi düzenledik.

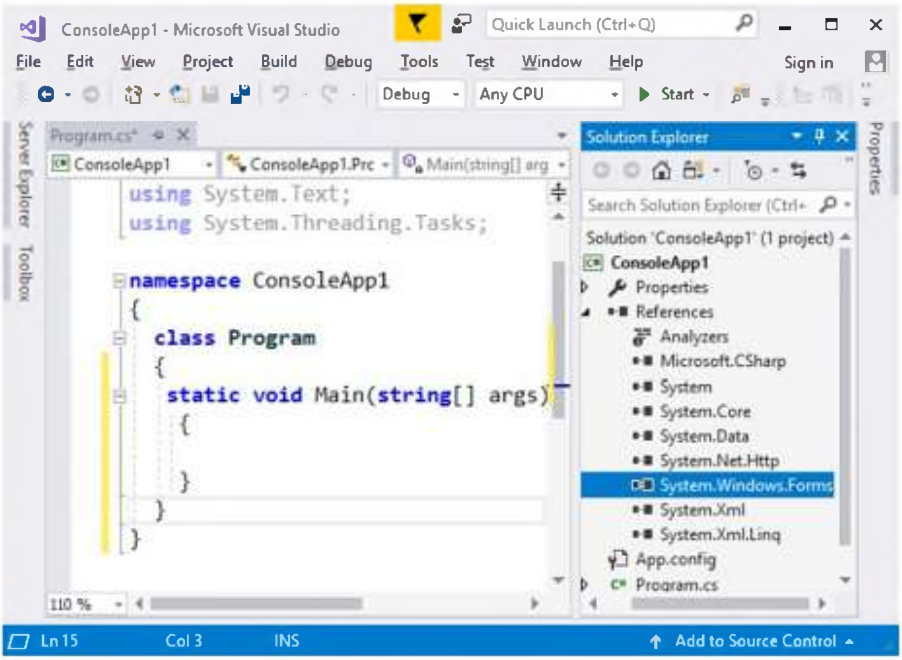
```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
   {
    Form Form1 = new Form();
    Form1.ShowDialog();
  }
 }
}
```

Burada .NET Framework ile gelen Form sınıfının örneği alınıp bu sınıfın ShowDialog() metodu ile diyalog kutusu olarak ekrana getirilmektedir. Bu şekilde ekrana getirilen diyalog kutusu kapatıldığında konsol uygulamasının çalışması sona erer.

Konsol Uygulamaları ve Application Sınıfı

Windows kurulu bilgisayarda işlemler mesajlar aracılığıyla yapılır. Çalıştırılan her program için Windows tarafından bir mesaj kuyruğu hazırlanır. Örneğin herhangi bir uygulamaya ait penceredeki TextBox'a bir harf girildiğinde, Windows ilgili uygulamanın mesaj kuyruğuna kullanıcının TextBox'ın içinde iken bir tuşa bastığı mesajını gönderir. Uygulama bunun üzerine kendisine ait mesajı okuyup gereğini yapar. Ne zaman projenin ana formu kapatılır o zaman uygulamaya ait mesaj kuyruğu silinir. Bu anlattıklarımız geri planda yapılır. Ancak konsol uygulamalarında durum farklıdır ve bu uygulamalar için mesaj kuyruğu hazırlanmaz.

Konsol uygulamalarının bazı özelliklerinden söz etmek için Visual Studio ile bir konsol uygulaması hazırladık ve **Add Reference** diyalog kutusunu ekrana getirip **System.Windows.Forms.dll** referansının Solution Explorer penceresinde listeledik.

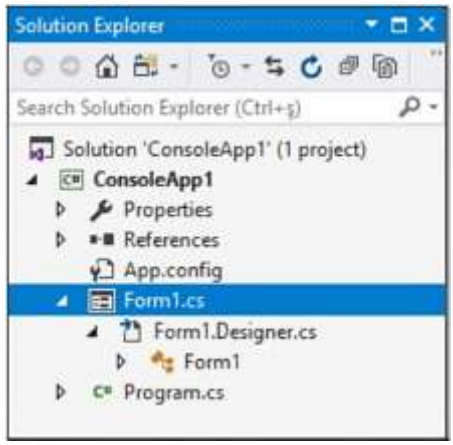


Bu şartlarda konsol uygulamasında System.Windows.Forms'de yer alan sınıflardan yararlanmak mümkün olduğu gibi **Application** sınıfından yararlanabiliriz. Konu Application sınıfı olduğunda durup biraz düşünmek gerekmektedir. Bir önceki bölümde işaret edildiği gibi Windows Forms Application şablonu ile hazırlanan uygulama için **Application** sınıfının **Run**() metodu sayesinde bir mesaj kuyruğu hazırlanmakta ve uygulamanın Windows'la ilişkisi bu mesaj kuyruğu üzerinden kurulmaktadır.



Konsol uygulamaları için burada yapılanların pratik bir değerinin olmadığını ve amacımızın geri planda gelişen olaylar hakkında bilgi vermek olduğunu bilmenizi isteriz.

Solution Explorer penceresinde System.Windows.Forms.dll referansının listelenmesini sağlayıp Application sınıfının bazı metotlarından yararlanmak mümkün olsa bile uygulamanın kontrolü bu şartlarda Application sınıfında değildir. Bu konuyu netleştirmek için Project menüsünden komut verip ekrana **Add New Item** diyalog kutusunu getirip **Windows Forms** şablonundan yararlanıp konsol uygulamasına bir form dahil ettik.



Add New Items diyalog kutusundan yararlanıp "Form1" adında bir formu konsol uygulamasına dahil ettikten sonra bu forma bir düğme yerleştirip bu düğmenin Click olayını temsil eden metotta Application sınıfının **Exit**() metodunu kullandık. Application sınıfının Exit() metodu o sırada çalışan **Windows Forms** uygulamasının çalışmasını sona erdirmektedir.

```
private void button1_Click(object sender, EventArgs e)
{
 Application.Exit();
}
```

Şimdi sırada konsol uygulamasının Main() metodunu düzenleyip uygulama çalıştırıldığı zaman yukarıda button nesnesi yerleştirdiğimiz formu ekrana getirmek var. Bu amaçla Main() metoduna yazdığımız satırları aşağıda görebilirsiniz.

```
namespace Konsol_uygulamasi
{
 class Program
 {
  static void Main(string[] args)
  {
    Form1 F1 = new Form1();
    F1.ShowDialog();
    Console.ReadKey();
  }
 }
}
```

Main() metodu bu şekilde düzenlenen konsol uygulaması çalıştırıldığında "Form1" sınıfının örneği alınıp **ShowDialog**() metodu ile ekrana getirilir. Kullanıcı formdaki düğmeyi tıklarsa form kapatılır ve programın işletimi Main() metodundaki bir sonraki satıra yani Console sınıfının **ReadKey**() metodunun kullanıldığı satıra geçer. Ne zaman ki kullanıcı konsolda bir tuşa basarsa Main() metodundaki bütün satırlar işletilmiş olacağı için konsol uygulamasının çalışması sona erer. Başka bir deyişle Application sınıfının Exit() metodunun konsol uygulamasının çalışmasına bir etkisi olmadı. Exit() metodu ancak konsol uygulaması dahilinde hazırlanan formlar üzerinde etkili olabilmektedir.

Yukarıda konsol uygulaması dahilinde form nesnesini ekrana getirirken **ShowDilaog**() metodunu kullandık. ShowDialog() yerine **Show**() metodunu kullansaydık ne olurdu? Hata meydana gelirdi ve konsol uygulaması çökerdi. Çünkü Show() metodu ile ekrana getirilen form, **Application** nesnesine gerek duyar. Bunun için ya form nesnesi Application sınıfının Run() metoduna parametre olarak verilmeli ya da Show() metodu ile form ekrana getirildikten sonra Application sınıfının **Run**() metodu işletilmelidir.

```
static void Main(string[] args)
{
 Form1 F1 = new Form1();
 F1.Show();
 Application.Run();
 Console.ReadKey();
}
```

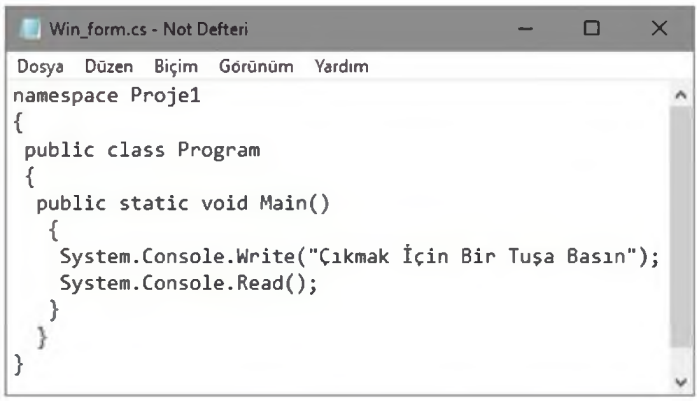
Bu andan itibaren ekrana getirilen pencere yani form normal Windows Forms uygulamasındaki formlardan farksızdır. Bu şekilde ekrana getirilen form kapatılırsa veya Application sınıfının **Exit**() metodu işletilirse uygulamanın çalışması sona erer.

Burada yaptıklarımızı özetlemek gerekirse şu söylenebilir: Konsol uygulaması dahilinde Windows Forms uygulaması hazırlayıp çalıştırdık. Tabi bu önerilecek bir yöntem değildir. Bu bir bakıma Windows Forms uygulaması hazırlayıp sonra da Project menüsünden komut verilerek ekrana getirilen Properties penceresinde uygulamanın **Output type** özelliğini **Console Application** olarak ayarlamak gibidir. Windows Forms uygulamasının çıkış tipini konsol uygulaması olarak ayarlarsanız formdan önce konsol penceresi açılır.

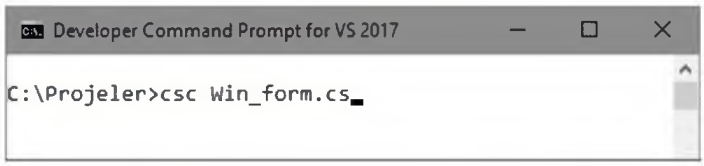
Not Defteri İle Visual Studio Benzeri Proje Hazırlamak

Visual Studio'nun programcılara desteği oldukça fazladır. Örneğin intellisense özelliği sayesinde metotların adlarını ve parametre yapılarını ezbere bilmeniz gerekmiyor. Bu bölümde amaçlarımızdan birisi Visual Studio ile hazırlanan projeleri mercek altına almak olduğu Visual Studio'yu kullanmadan Visual Studio ile hazırlanan projelerin benzerini Not Defteri ile hazırlayacağız. Bu sayede bazı ayrıntılardan söz etme imkanı elde etmiş olacağız.

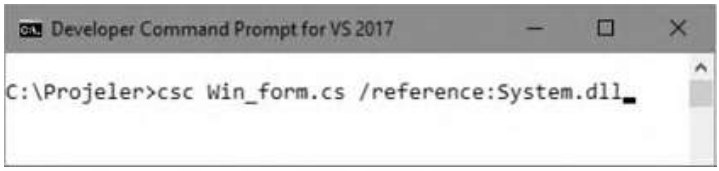
Yukarıdaki sayfalarda yazılanlara dikkat ettiyseniz konsol uygulamasını Visual Studio ile hazırladığınızda ilgilendiğiniz namespace'leri içeren referans yani DLL dosyalarını Solution Explorer penceresinde kolayca listeleyebiliyorsunuz. Ancak Visual Studio'dan yararlanmadan hazırladığınız konsol uygulamasında **System** dışında herhangi bir namespace'in kaynaklarını kullandığınızda derleme yaparken bu namespace'in yer aldığı DLL dosyalarından derleyiciyi haberdar etmek gerekiyor. Bu konu üzerinde konuşabilmek için aşağıdaki gibi basit bir konsol uygulaması hazırladık.



Sizin için yeni bir şey içermeyen bu kodu incelerseniz **System** adlı namespace'te dolayısıyla .NET Framework ile gelen "System.dll" dosyasında yer alan **Console** sınıfından yararlanıldığını görürsünüz. Konsolda derlemeyi aşağıdaki gibi yapmamıza, başka bir deyişle derleyiciyi System.dll'den haberdar etmemiş olmamıza rağmen derleme başarılı bir şekilde gerçekleşti.



Çünkü derleyici "System.dll" dosyasını otomatik olarak refere etmektedir. Bu nedenle bu program aşağıdaki gibi derlenirse değişen bir şey olmaz.



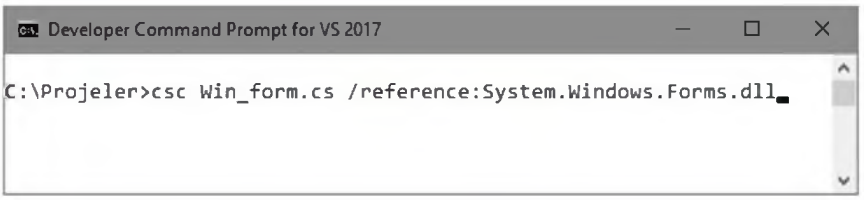
Derleme satırında derlenecek CS dosyasını belirttikten ettikten sonra **reference** parametresi ile ilgili DLL dosyasını işaret ettik. Şimdi yukarıda verilen konsol uygulamasında değişiklik yapıp System.Windows.Forms'deki **Form** ve **Application** sınıflarından yararlanacağız. Bu amaçla kod dosyamızda 2. bir class hazırladık.

```
using System.Windows.Forms;
namespace Proje1
{
 public class Form1 : System.Windows.Forms.Form
 {
  public Form1()
   {
    this.Text ="Form1";
   }
 }
```

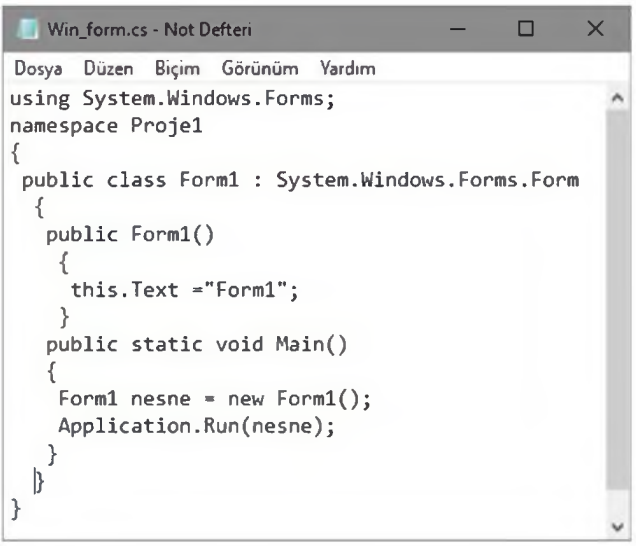
```
public class Program
{
 public static void Main()
  {
   Form1 nesne = new Form1();
   Application.Run(nesne);
  }
 }
}
```

2 sınıfın olduğu bu konsol uygulamasındaki **Form1** sınıfı System.Windows.Forms'deki **Form** sınıfının mirasçısı olup yapıcı metodunda **Text** özelliği ayarlanmaktadır. Main() metodunun olduğu 2. sınıfta ise ilk sınıfın örneği alınıp Application sınıfının **Run**() metoduna parametre olarak verilmektedir. Bu sayede Form1 sınıfı pencere olarak ekrana getirilmektedir.

Bu kodu derleyip çalıştırdığınızda ekrana "Form1" başlıklı bir pencerenin geleceğini tahmin edebilirsiniz. Bu uygulamada "System.Windows.Forms.dll" adlı reference'ten yararlanıldığı için derleme yapılırken bu DLL dosyasının aşağıdaki gibi **reference** parametresi ile işaret edilmesi gerekir. Bu derleme satırında **reference** parametresini 2. kez kullanıp derleyiciyi başka bir referanstan haberdar edebilirdik.



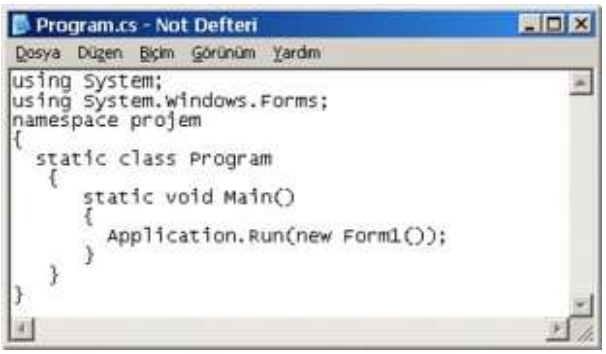
Not Defteri veya başka bir şekilde hazırladığınız uygulamayı DOS penceresinde kendiniz derlerken uygulamanın konsolda çalışacağı varsayılır. Dolayısıyla aşağıdaki gibi kod yazıp derleme yaparsanız ekrana formdan önce DOS penceresi gelir.



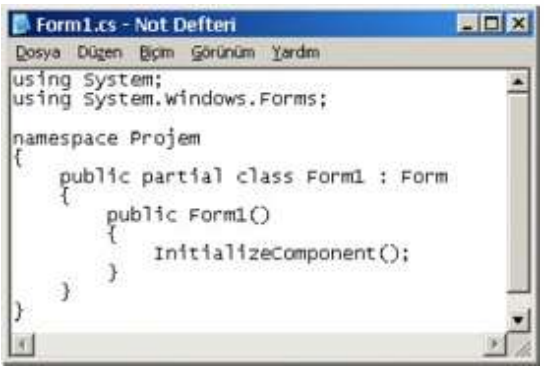
Bu uygulamanın konsolla bir işi olmasa bile bu uygulama yukarıdaki sayfalarda anlatılan şekilde derlenirse **Target** parametresinin "exe" olduğuna karar verilir ve Windows formu ile birlikte DOS penceresi de açılır. DOS penceresinde derlediğiniz uygulama Windows Forms uygulaması ise derlemeyi aşağıdaki gibi yapabilirsiniz.



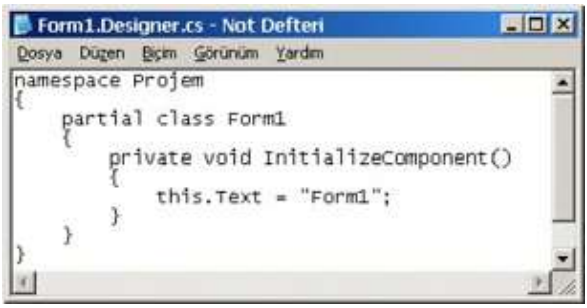
Bu şekilde Visual Studio'dan yararlanmadan Not Defteri ile **Windows Forms** uygulamasının nasıl hazırlandığını gösterdikten sonra şimdi Visual Studio ile hazırlanan uygulamanın benzerini hazırlayacağız. Bu amaçla "C:" sürücüsünün kök klasöründe "\Projem" adında bir klasör hazırlayıp Not Defteri ile aşağıdaki gibi bir "Program.cs" dosyası hazırlayıp bu klasöre kaydettik. Bu dosyaya **Program.cs** yerine başka bir ad verebilirdik. Bu dosyanın önemi **Main**() metodunu içermesidir.



Visual Studio'nun hazırladığından biraz kısıtlı bir "Program.cs" dosyasını bu şekilde hazırlayıp yukarıda adını andığımız klasöre kaydettikten sonra yine Not Defteri aşağıdaki gibi bir "Form1.cs" dosyası hazırladık.



Visual Studio tarafından **using** deyimi ile koda dahil edilen bazı namespace'lerdeki class'lara bu projede gerek duymayacağımız için dahil etmedik. Şimdi sırada "Form1" adlı **Partial Class**'ın diğer kısmını yazmak var. Bu amaçla Not Defteri ile aşağıdaki gibi bir "Form1.Designer.cs" dosyası hazırladık.

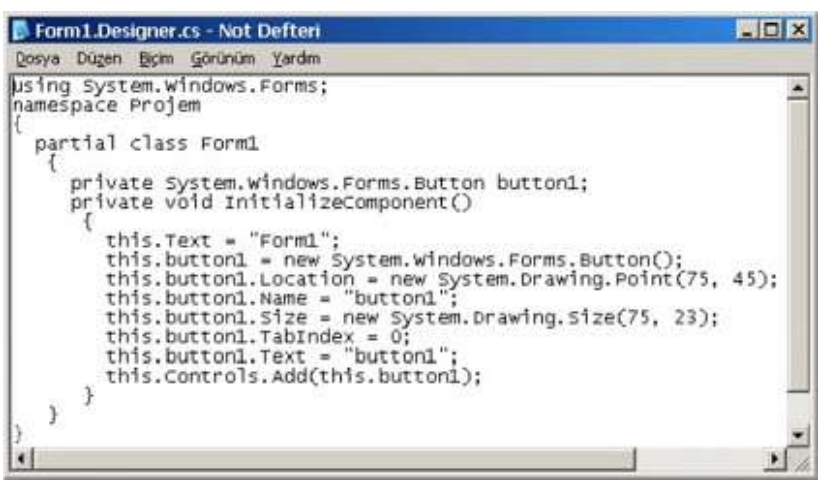


Bu şartlarda hazırladığımız projenin kodları 3 ayrı CS dosyasında bulunmakta ve **System.Windows.Form** adlı namespace'ten yararlanılmaktadır. Bu namespace System.Windows.Forms.dll adlı referansta yer aldığı için derleme satırını aşağıdaki gibi düzenledik. Derleme sırasında "System.dll" dosyası uygulamaya otomatik olarak dahil edildiği için derleme satırına yazmaya gerek duymadık.

```
C:\Projem>CSC /out:Projem.exe Program.cs Form1.cs Form1.Designer.cs /reference:
 System.Windows.Forms.dll
```

Bu şekilde yapılan derleme sayesinde "Projem.exe" adında bir EXE dosya hazırlanır. Bu program dosyası çalıştırıldığında ekrana "Form1" başlıklı bir pencere gelse bile bu uygulama konsol uygulaması sayılmaktadır. Bu EXE dosyanın Windows Form uygulaması olarak değerlendirilmesini sağlamak için **/target:winexe** parametresi ile derlemek gerekmektedir.

Şimdi Not Defteri ile hazırladığımız bu proje dahilinde ekrana getirilen forma bir Button nesnesi yerleştireceğiz. Bu amaçla "Form1" sınıfının **InitializeComponent**() metodunu aşağıdaki gibi düzenledik.



Bu şekilde 3 dosya şeklinde düzenlenen konsol uygulaması derlenip çalıştırıldığında aşağıdaki gibi bir sonuç alınır. Formdaki bu düğme tıklandığı zaman işletilmek üzere kod yazmak mümkün olmakla birlikte Visual Studio dururken Not Defteri ile proje geliştirmek pek mantıklı olmayacağı için konuyu fazla uzatmadık.



Main Metodunu Parametreli Kullanmak

Windows uyumlu programları parametreli olarak kullanma alışkanlığı yaygın olmasa bile Main() metodu hakkında biraz daha bilgi vermek amacıyla bu metodu parametreli olarak kullanmayı deneyeceğiz. Bu amaçla Not Defteri ile yeni bir C# programı yazıp Main() metoduna parametre olarak verilen 2 sayıyı toplayıp konsola yazacağız. Başka bir deyişle kullanıcı toplamak istediği sayıları konsolda program adının yanına yazacak.

Program çalıştırıldığında program adının yanına yazılacak bilgiler Main() metodunun parametresi olarak görüleceği için metot adından sonra parantezlerin içine "**string[] pars**" yazdık. Başka bir deyişle Main() metoduna ait parantezlerin içinde henüz eleman sayısı belli olmayan bir dizi değişken tanımladık.

İlk olarak kullanıcının programı çalıştırırken program adının yanına yazdığı 2 sayıyı elde edip ekrana yazmakla yetineceğiz. Programcılar Main() metodunun string tipindeki dizi değişken olan parametresine genellikle **args** adını veriyorlar. Bizler "pars" adını tercih ettik.

Parametreli.cs - Not Defteri

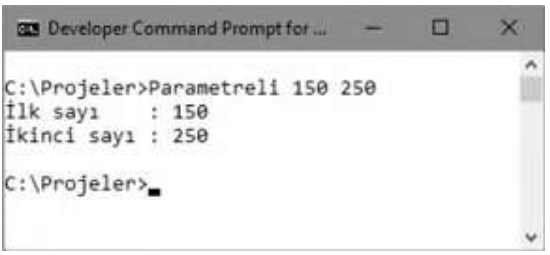
Dosya Düzen Biçim Görünüm Yardım

```
using System;
class Program
{
 static void Main(string[] pars)
  {
   System.Console.Write("İlk sayı    : ");
   Console.WriteLine(pars[0]);
   System.Console.Write("İkinci sayı : ");
   Console.WriteLine(pars[1]);
  }
}
```

Buradaki **pars** string tipinde bir dizi değişkendir. "pars[0]" ile dizi değişkenin ilk, "pars[1]" ile dizi değişkenin 2. elemanına ulaşılmaktadır. Bu şekilde kodladığımız C# programını derledikten sonra çalıştırma satırında program adının yanına aşağıdaki gibi 2 sayı yazdık.



Bu sırada enter tuşuna basıp programı çalıştırırsanız 150 sayısı ilk parametre olarak değerlendirilip dizi değişkenin ilk elemanına ve 250 ise "pars" adını verdiğimiz dizi değişkenin 2. elemanına aktarılır. Aşağıda verilen sonucu bu programı çalıştırdıktan sonra aldık.



Bu örnekte toplanacak sayıları Main() metoduna parametre olarak verdik. Şimdi bu programda değişiklik yapıp toplanacak sayıları önce **int** tipindeki değişkenlere aktarıp sonra birbirleri ile toplayıp bulunan sonucu konsola yazacağız.

```
using System;
class Program
{
 static void Main(string[] pars)
  {
   int sayi1, sayi2, sonuc;
   System.Console.Write("İlk sayı    : ");
   Console.WriteLine(pars[0]);
   System.Console.Write("İkinci sayı : ");
   Console.WriteLine(pars[1]);
   sayi1 = Convert.ToInt16(pars[0]);
   sayi2 = Convert.ToInt16(pars[1]);
   sonuc = sayi1 + sayi2;
   Console.Write("Sayıların toplamı :");
   Console.Write(sonuc);
  }
}
```

Tahmin edeceğiniz gibi burada "pars" adını verdiğimiz dizi değişkenin eleman sayısı programa parametre olarak verilen bilgilerin sayısına bağlıdır. Bu program çalıştırılırken program adının yanına 2 yerine 1 sayı yazılırsa hata meydana gelir. Çünkü programda dizi değişkenin 2. elemanın içeriği okunmak istenmektedir. Benzer şekilde program çalıştırılırken program adının yanına karaktersel bilgi yazılırsa hata meydana gelir.

Şimdi yukarıda verilen kodda değişiklik yapıp Main() metoduna parametre olarak verilen bilgileri **foreach** döngüsü ile elde edip içeriklerini ekrana yazacağız. Bu kitabın ileriki sayfalarında foreach döngüleri hakkında bilgi verilecektir.

```
using System;
class Program
{
 static void Main(string[] pars)
 {
  if (pars.Length > 0)
   foreach(string par in pars)
    {
     Console.WriteLine(par);
    }
  else
    Console.WriteLine("Program parametresiz çalıştırıldı");
 }
}
```

Şimdi bu dizi değişkenin elemanlarının içeriklerini farklı bir teknikle elde edeceğiz. **Environment** sınıfının **GetCommandLineArgs()** metodu, çalıştırılan C# uygulamasına ait parametreler hakkında geriye bilgi göndermektedir. Bu metottan yararlanıp konsol uygulamasının parametreleri aşağıdaki gibi elde edilebilir.

```
using System;
class Program
{
 static void Main()
  {
   string[] dizi = Environment.GetCommandLineArgs();
   int sayi1, sayi2, sonuc;
   System.Console.Write("İlk sayı    : ");
   Console.WriteLine(dizi[1]);
   System.Console.Write("İkinci sayı : ");
   Console.WriteLine(dizi[2]);
   sayi1 = Convert.ToInt16(dizi[1]);
   sayi2 = Convert.ToInt16(dizi[2]);
   sonuc = sayi1 + sayi2;
   Console.Write("Sayıların toplamı :");
   Console.Write(sonuc);
  }
}
```

Bu metot, çalıştırılan programın adını 1. parametre olarak kabul edip bütün parametreleri string tipte bir dizi değişkene doldurup geriye göndermektedir. Örneğimizde program adından sonra yazılacak 2 sayıyı toplamak istediğimiz için **GetCommandLineArgs()** metodunun geriye gönderdiği dizi değişkenin 2. ve 3. elemanı ile ilgilendik.

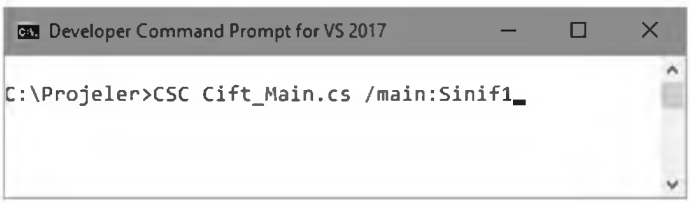
Birden Fazla Main() Metodu Hazırlamak

Uygulamanızda birden fazla sınıf olacağından Main() metodunun hangi sınıfta olduğunun önemi yoktur. Çünkü C# derleyicisi Main() metodunun hangi sınıfta olduğunu arayıp buluyor. Konsol uygulamanızda birden fazla sınıfta Main() metodu varsa hangisinin uygulamanın başlangıç noktası olarak kullanılmak istediğinizi belirtilmeniz gerekir. Bu konuda bilgi vermek için aşağıdaki gibi bir C# programı hazırladık.

```
using System;
class Sinif1
 {
  public static void Main()
  {
    Console.Write("Program İlk sınıftaki Main ile başladı");
    Console.ReadLine();
  }
 }
class Sinif2
{
 public static void Main()
  {
   Console.Write("Program İkinci sınıftaki Main ile başladı");
   Console.ReadLine();
  }
}
```

Bu koda biraz dikkat ederseniz Main() metodunun bu kez parametresiz olduğunu veya parantezin içine herhangi bir değişken adı yazmadığımızı görürsünüz. Ayrıca bu örnekte namespace hazırlamadığımızı fark etmiş olmalısınız.

Bu şekilde hazırlanan C# programı bilinen şekilde derlendiğinde hata meydana gelir. Bu kodun hatasız derlenebilmesi için derleme satırında hangi sınıftaki Main() metodunun kullanılmak istendiği **main** parametresi ile belirtilmelidir. Bu konsol uygulaması aşağıdaki gibi derlenirse uygulama ilk sınıftaki Main() metodundan itibaren çalışmaya başlar.



4. BÖLÜM

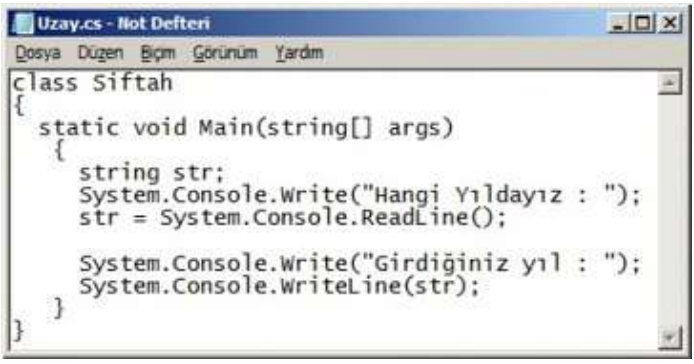
NAMESPACE HAZIRLAMAK ve KULLANMAK

Visual Studio, uygulamanıza verdiğiniz adı aynı zamanda namespace adı olarak kullanıyor. .NET Framework ile gelen class, struct, interface veya enum'ları kullanıp uygulamanızı hazırlarsınız. Uygulamanızı yazarken hazırladığınız bütün sınıf, struct veya interface'ler varsayım olarak aynı namespace'in içinde bulunurlar. Ardından uygulamanızı derleyip exe dosya yaparsınız. Sonra da kullanır veya kullanmaları için başka birilerine verirsiniz. Bu senaryoya göre, namespace'lerle ilgili olarak bilmeniz gereken, uygulamayı yazarken .NET Framework ile gelen hangi class veya interface'in hangi namespace'te yer aldığıdır.

Ne zaman ki kendiniz birden fazla uygulamada kullanmak üzere sınıflar hazırlayıp bunları DLL dosyası olarak düzenlemek istediniz, işte o zaman yukarıda verilen senaryo yeterli gelmemeye başlar. Bu kitapta mümkün mertebe konuların arka planından da söz etmeyi tercih ettiğimiz için kurumsal bir firmada programcı olarak çalışan birisinin namespace'ler hakkında bilmesi gerektiğini düşündüğümüz bilgileri adım adım sunmayı hedefliyoruz. En başta belirtmek gerekirse, C# uygulamaları dahilinde hazırlanacak class ve struct'ların namespace'lere yerleştirilmelerinin en önemli nedeni sınıf adı çakışmalarını önlemektir.

.NET Framework ile gelen sınıflar, yapılar, enum'lar ve interface'ler vs. namespace'lerde yer almaktadır. Daha doğrusu, istenen sayıda class, struct, enum ve interface bir araya getirilip namespace olarak paketlenmektedir. Namespace'ler ise **referans** adı verilen DLL dosyalarına konuluyorlar. Kullanılmak istenen class veya struct hangi namespace'de yer alıyorsa o namespace'i içeren referans (dll dosyası) konsolda derleme yapılırken işaret edilmelidir veya Visual Studio'nun **Solution Explorer** penceresinde listelenmelidir.

Visual Studio, sık kullanılan referansları **Solution Explorer** penceresinde kendiliğinden listelediği için genellikle referans kavramıyla ilgilenilmez. Bu konuda adım adım bilgi vermek için Not Defteri ile aşağıdaki gibi bir konsol uygulaması hazırladık.





Namespace'ler hakkında bilgi verirken özellikle Visual Studio'dan yararlanma yoluna gitmedik. Çünkü Visual Studio birçok işlemi kendiliğinden yapıyor ve sözünü etmek istediğimiz ayrıntıları fark etmek zorlaşıyor. Yani bu kitapta zaman zaman Not Defteri ile kod yazıp konsolda derleme yapma tercihimiz eğitsel bir kaygıdan kaynaklanıyor.

Bu kitabın ilk bölümünde konsolda aşağıdakine benzer şekilde bir derleme yapmıştık. Bu derleme satırında herhangi bir referans işaret edilmemektedir. Yani derleme sonucu hazırlanacak EXE dosyanın çalışma anında gerek duyacağı herhangi bir DLL dosyasından C# derleyicisi haberdar edilmiyor.



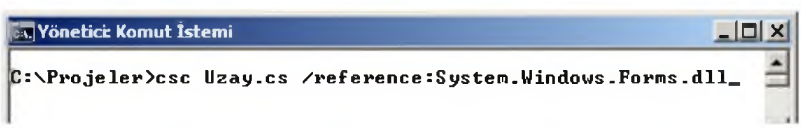
Yine daha önceki bölümlerden hatırlayacağınız gibi konsol uygulamasında **MessageBox** sınıfının **Show**() metodunu kullandığımızda "System.Windows.Forms.dll" adlı referanstan konsolda yapılan derleme sırasında derleyiciyi haberdar etmek (başka bir deyişle derleyiciye bu referanstaki class'lardan yararlanmak istediğimizi söylememiz) gerekiyordu.

Çalışma anında ne zaman ilgili DLL dosyasındaki yani referanstaki bir sınıf kullanılmak istenirse DLL dosyaları doğaları gereği o an belleğe yüklenirler. Daha açık bir ifade ile; derleme yaparken derleyiciye diyeceğiz ki "derlemek istediğimiz koddaki şu sınıf şuradaki şu DLL dosyasında bulunmaktadır. Bu DLL dosyasının yerini not et ki çalışma anında gerektiğinde bulup belleğe yükleyip yararlanabilinsin".

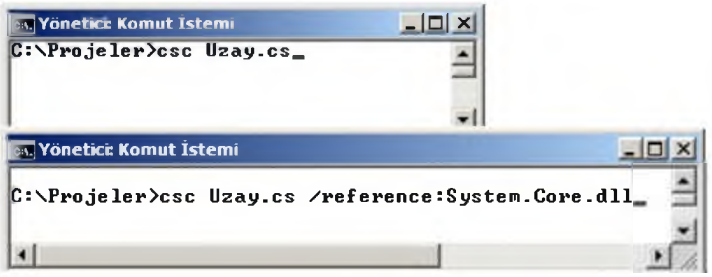
Daha sonra bu EXE dosya çalıştırılıp söz konusu DLL dosyasındaki sınıflardan birisi kullanılmak istendiğinde, EXE dosya gerek duyduğu DLL dosyasını ilgili klasörde bulup belleğe yükler. Başka bir deyişle derleme yaparken söz konusu DLL dosyasının nerede olduğunu söylediysek çalışma anında o DLL dosyası yine aynı yerde olmalıdır.

Şimdi bu konuda bir deneme yapıp bir konsol uygulaması hazırlayıp derleme sırasında **rerefence** parametresi ile herhangi bir referansı işaret etmeyeceği. Yukarıda hazırlayıp "Uzay.cs" adını verip Main() metodunda birkaç satır bulunan uygulama derlenirse "exe" dosya oldukça küçük olur.

Şimdi aynı CS dosyasını bir de "System.Windows.Forms.dll" dosyasını refere ederek derleyeceğiz. Bakalım EXE dosyanın boyutu ne olacak? .NET Framework ile gelen "System.Windows.Forms.dll" dosyasının büyüklüğü yaklaşık 5 MB'tır. Buna rağmen aşağıdaki gibi bir derleme yaptığımızda EXE dosyanın büyüklüğü değişmedi.



O zaman şu soruyu da soralım:Yukarıda verilen konsol uygulaması derlenirken yararlanılan **Console** sınıfının yer aldığı referans neden göz ardı edildi? Çünkü **System** namespace'i **System.core.dll**'de yer almakta ve EXE dosya bu referans hakkında otomatik olarak bilgilendirilmektedir. Bu nedenle aşağıda verilen iki derleme işlemi birbirinden farklı değildir.



Madem "System.Core.dll" adlı referans konsol uygulamalarına otomatik olarak dahil ediliyor (alışkanlık sonucu 'dahil edilir' diyoruz yoksa dahil edilen bir şey yoktur ve DLL dosyası yerinde durmaktadır) o zaman bu referansta yer alan namespace'lerdeki sınıfları kullanmak için derleme sırasında herhangi bir hazırlık yapmak gerekmiyor.

Konsolda derleme yapılırken System.Core.dll'den başka 2. bir DLL dosyası otomatik olarak refere edilmektedir. Çünkü değişken tanımlanırken kullanılan çok sayıda tip .NET Framework ile gelen **System.dll**'de tanımlıdır. Bu nedenle DOS penceresinde aşağıdaki gibi bir derleme hataya neden olmaz ve böyle bir derlemenin artı bir etkisi yoktur.



Bu derleme işleminde yalnızca System.dll dosyasını veya referansını kullandığımızı işaret etmemize rağmen **System.Core.dll** dosyası da otomatik olarak eklenir. Bu nedenle aşağıdaki gibi derleme yukarıdakinden farklı değildir. Tahmin edeceğiniz gibi yararlanılan referansların sayısı fazla olduğunda konsolda derleme yapmak zorlaşmaktadır.



Yukarıda verilen örnekte yararlanılan Console sınıfının **Write()** ve **WriteLine()** metotları statik oldukları için direk kullanılabiliniyor. Diğer yandan Console sınıfı statik bir sınıf olduğu için örneği alınamıyor. Bu nedenle aşağıda yapıldığı gibi Console sınıfından yola çıkılıp nesne hazırlanmak istenirse derleyici hata rapor eder.

```
class Siftah
{
 static void Main(string[] args)
  {
    System.Console konsol = new System.Console(); //Hata oluşur.
  }
}
```

Console veya başka bir sınıfı kullanmak istiyorsanız öncelikle o sınıfın hangi namespace'in içinde yer aldığını belirtmeniz gerekir. Bu nedenle yukarıda verilen kısa örnekte önce **System**, ardından **Console** sınıfını, en son olarak ilgili metodun adını yazdık.

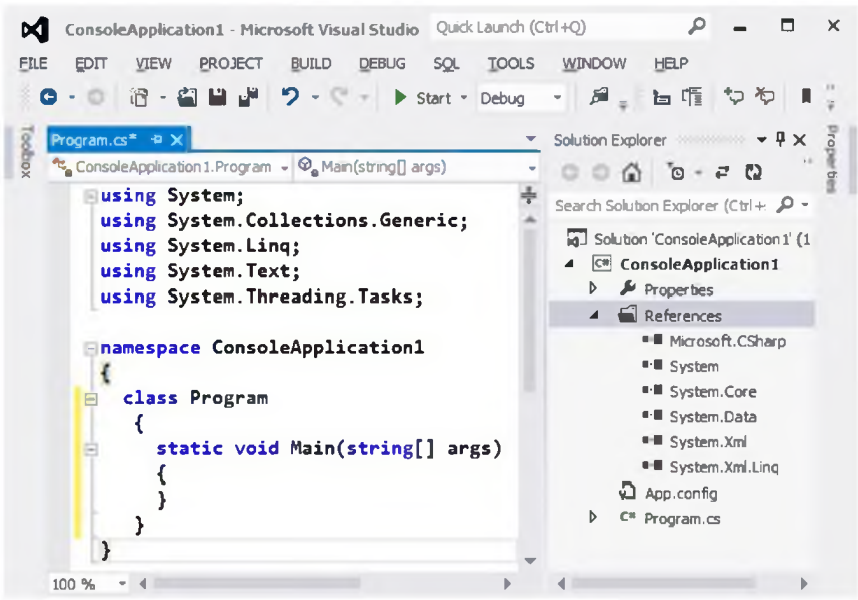
```
System.Console.Write("Hangi Yıldayız :");
```

İlgili class'ın adından önce class'ı barındıran namespace'in adını her seferinde yazmak istemiyorsanız söz konusu namespace'i kod dosyasının en üstünde **using** deyimi ile işaret etmeniz gerekir. Örneğin konsol uygulamasının en üstüne **using System;** yazıp ondan sonra

```
Console.Write("Hangi Yıldayız :");
```

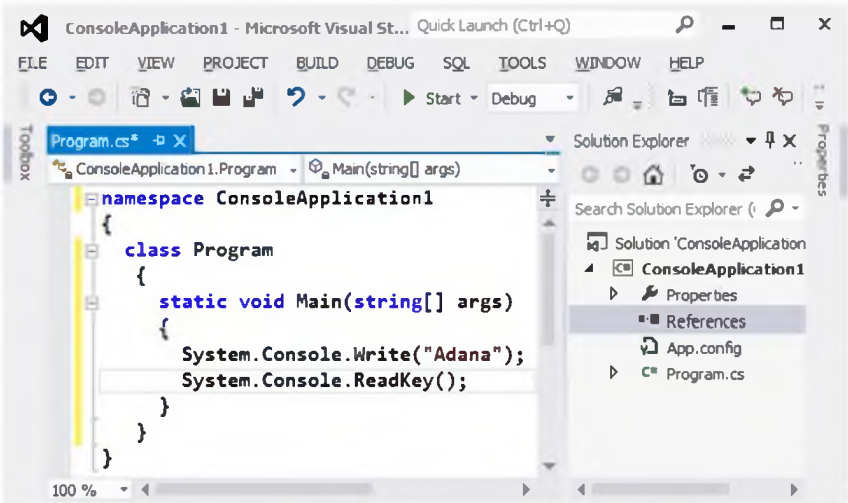
gibi bir satır yazarsanız C# derleyicisi burada ilk yazılanın (yani Console) bir class, tip veya yapı olduğuna hükmedip using deyimi ile işaret edilen namespace'lerde arama yapar. Console sınıfı **System** adlı namespace'te bulunduğu için kod hatasız derlenir.

Şimdi referansları bir de Visual Studio ile hazırlanan konsol uygulaması üzerinde ele alacağız. Bu konuyu incelemek için Visual Studio ile aşağıdaki gibi bir konsol uygulaması hazırladık. Verilen ekran görüntüsünden tespit etmiş olacağınız gibi Visual Studio ile hazırlanan konsol uygulamalarına çok sayıda referans otomatik olarak dahil edilmektedir. Bu referansları Visual Studio'nun **Solution Explorer** penceresinde görebilirsiniz.



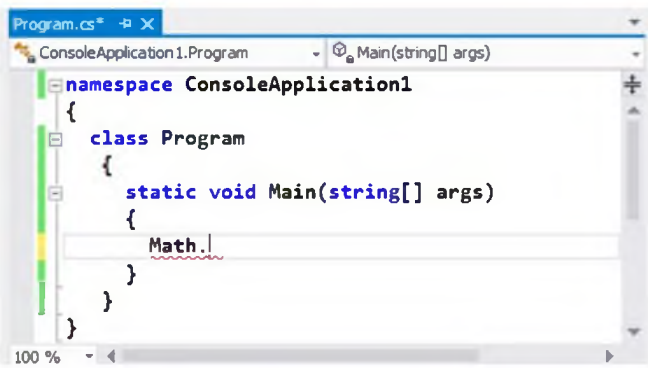
Visual Studio tarafından uygulamaya dahil edilmesine rağmen gerek duymadığınız referanslar varsa bunları projeden uzaklaştırmak için Solution Explorer penceresinde söz konusu referansa ait kısayol menüsünden **Remove** komutunu vermeniz yeterlidir.

Konunun ayrıntıları ortaya çıksın diye Visual Studio tarafından uygulamaya dahil edilen referansların hepsini Solution Explorer penceresinden uzaklaştırdık. Ardından kodun başında **using** deyimin kullanıldığı satırları da sildik. Aşağıda verilen ekran görüntüsü alındığı sırada Solution Explorer penceresinde **References** adı altında listelenen herhangi bir seçenek yoktu.



Sizce bu konsol uygulaması Visual Studio yardımıyla çalıştırılmak istenirse hata meydana gelir mi? **System** ve **System.core** referansları C# uygulamalarına otomatik olarak dahil edildiği için bu referanslar Solution Explorer penceresinden silinse bile bu uygulama sorunsuzca çalışır. Çünkü adı anılan DLL dosyaları her şartta uygulamaya dahil edilmektedir. Alışkanlıktan "dahil etmek" desek bile bunu bir dahil etme işleminden öte yerlerinin not edilmesi olarak anlamak gerekir.

Şimdi konuyu baştan alacağız. Konsol uygulamalarında ek bir hazırlık yapmadan **Sistem** adlı namespace'teki sınıflar ve yapılar kullanılabilmektedir. Örneğin değişken tanımlanırken kullanılan yapıların hepsi System'de yer almaktadır. Bu nedenle yukarıda verilen konsol uygulamasında herhangi bir hazırlık yapılmadan istenen tipte değişken tanımlanabilir. Gelelim Visual Studio'nun sınıfları arama mekanizmasına: Programcının yukarıda verilen konsol uygulamasında Main() metodunun içinde **Math** sınıfının adını yazdığını varsayalım.



Bu sırada **System**'de yer alan **Math** sınıfını içeren referans uygulamaya dahil edilmiş olmasına rağmen Math sınıfı Visual Studio tarafından bulanamaz. Bunun çaresi, kod dosyasının başında içeriği araştırılacak namespace listesine **System**'i dahil etmek veya Math sınıfından önce namespace adını yazmaktır.

Buna göre **System** adlı namespace'te yer alan bazı sınıfları kullanabilmek için bu sınıfların System'de yer aldıklarını bir şekilde işaret etmek gerekiyor. System adlı namespace'te Math ve Console gibi sınıfların dışında çok sayıda yapı bulunmaktadır. Örneğin sayısal bilgileri bellekte tutmak için **int** tipindeki değişkenler aşağıdaki gibi tanımlanmaktadır.

```
class Program
{
 static void Main(string[] args)
  {
   int yil;
  }
}
```

Hemen soralım: Bu **int** hangi namespace'te yer alan bir sınıftır? Cevap şöyledir: int gerçekte **System** adlı namespace'teki **Int32** yapısının takma adıdır. Dolayısıyla Int32 yapısının örneğini almak veya **System.In32** tipinde değişken tanımlamak için System adlı namespace'i **using** deyimi ile işaret etmek gerekmediği gibi değişken tanımlarken veya Int32 yapısının örneğini alırken System adlı namespace'i yapı adından önce yazmak gerekmiyor. System.Int32 yapısının (struct) takma adı olmasaydı yukarıda verilen değişken tanımlama satırını aşağıdaki gibi yazmak zorunda kalırdık.

```
class Program
{
 static void Main(string[] args)
  {
   System.Int32 sayi;
  }
}
```

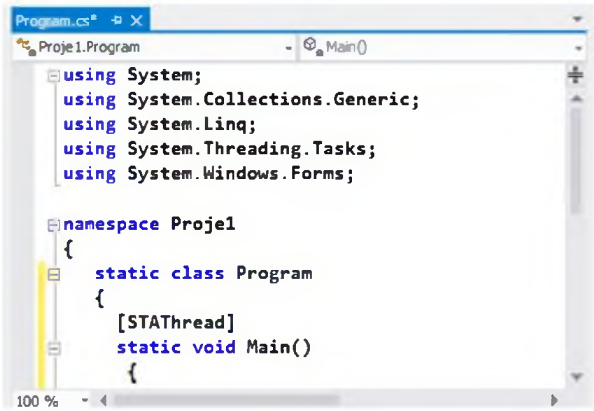
Bu kod dosyasının en üstünde **using** deyimi ile System adlı namespace'in kaynaklarının kullanılacağı işaret edilmiş olunsaydı Int32 yapısının örneğini alırken **System** yazmak zorunda kalmazdık.

```
using System;
namespace ConsoleApplication1
{
 class Program
  {
   static void Main(string[] args)
    {
     Int32 sayi;
    }
  }
}
```

Namespace Hazırlamak ve Kullanmak

.NET Framework ile gelen namespace'lerdeki sınıf ve yapıların nasıl kullanıldığından biraz söz ettikten sonra şimdi sırada kendi hazırlayacağımız namespace'ler üzerinde biraz duracağız. Şimdiye kadar anlatılanlara göre Visual Studio ile hazırlanan C# uygulamaları namespace şeklinde düzenlenmektedir. Namespace'lerin içinde başka namespace'ler, class'lar ve/veya struct'lar olabilmektedir. Hazır namespace'leri kullandığınız gibi Visual Studio'nun otomatik olarak hazırladığı namespace dışında kendiniz namespace hazırlayıp

kullanmak durumunda kalabilirsiniz. Bu konuda bilgi vermek için Visual Studio ile "Proje1" adında bir Windows Forms uygulaması hazırladık. Dolayısıyla bu projedeki bütün class'lar varsayım olarak "Proje1" adlı namespace'e dahil edilecektir.



Hazırlayacağınız namespace'leri ve içerdikleri class'ları başka uygulamalarda kullanmak istiyorsanız .NET Framework ile gelen class'lar gibi DLL olarak düzenleyebilirsiniz. Bu işlemler için Visual Studio ile gelen **Class Library** şablonundan yararlanabilirsiniz.

Bu kod dosyasındaki "Program" sınıfı proje ile aynı ada sahip namespace'e dahil edilmektedir. Yine bildiğiniz gibi Visual Studio ile "Windows Forms App" şablonu kullanılarak hazırlanan projede System.Windows.Forms'da yer alan **Form** sınıfının mirasçısı "Form1" adında bir sınıf hazırlanıyor.

Hatırlayacağınız gibi konsol uygulamaları dahilinde hazırlanan class'lar bir namespace'e dahil edilmediği zaman derleyici kendiliğinden bir namespace hazırlamaktadır. Bir diğer konu ise şudur: .NET Framework ile gelen çok sayıda sınıf namespace şeklinde paketlenip kullanıma öyle sunulmuştur. .NET Framework ile birlikte gelen namespace'ler DLL dosyaları içinde oldukları için öncelikle ilgili DLL dosyasının **Add References** diyalog kutusundan yararlanıp Visual Studio'nun Solution Explorer penceresinde listelemek gerekmektedir.

Uygulama DLL şeklinde düzenlendiği zaman namespace'lerle yakından ilgilenmek gerekse bile aynı proje dahilinde bazen birden fazla namespace hazırlamaya gerek duyulabilir. Şimdi örnek olması için aynı CS dosyası içinde 3 namespace hazırlayacağız.

```
Namespace Proje1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
 }
}
namespace Matematik
{
 public partial class Islemler
 {
 }
}
```

Bu kod dosyasındaki "Proje1" adlı namespace Visual Studio tarafından proje için hazırlanan namespace'tir. Aynı kod dosyası içinde hazırladığımız "Matematik" ve "Aritmetik" adlı namespace'lerde ise "Islemler" adında birer sınıf bulunmaktadır.

Bu koddaki her iki "Islemler" sınıfını da **partial** olarak işaret etmiş olsak bile bu 2 class'ın başka parçaları yoktur. "Form1" sınıfının diğer yarısının Form1.Designer.cs dosyasında olduğunu biliyorsunuz.

```
namespace Aritmetik
{
 public partial class Islemler
 {
 }
}
```

Yanda verilen kodu yazarken asıl amacımız namespace'lerin önemli bir işlevi olan sınıf adı çakışmalarını önlemeye vurgu yapmaktır. Çünkü namespace ve class'lara özensizce isim verilmesi sorunlara neden olabilir.

Bu şartlarda hem "Matematik" hem de "Aritmetik" adlı namespace'teki class'lardan yararlanılmak istenildiği **using** deyimi ile belirtilip "Matematik" adlı namespace'teki "Islemler" sınıfının örneği alınmak istenirse hata meydana gelir. Çünkü derleyici hangi namespace'teki "Islemler" sınıfının kullanılmak istendiğine karar veremez. Bu nedenle aşağıda yapıldığı gibi class adından önce namespace adını yazmanız gerekir.

```
using Matematik;
using Aritmetik;
namespace Proje1
{
 public partial class Form1 : Form
  {
   public Form1()
   {
     InitializeComponent();
   }
   private void Form1_Load(object sender, EventArgs e)
   {
    Matematik.Islemler nesne = new Matematik.Islemler();
   }
  }
}
```

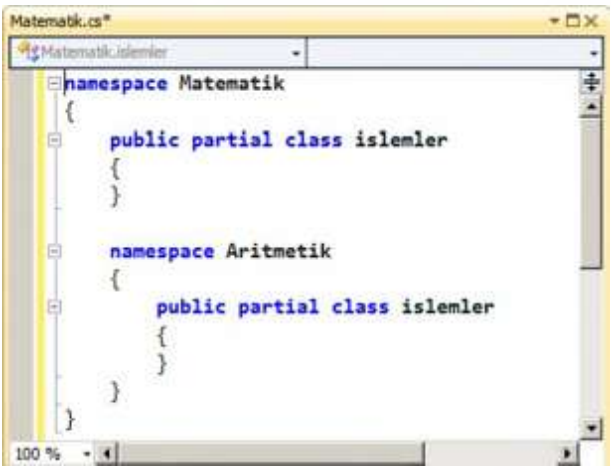
C# derleyicisi iç-içe namespace tanımlanmasına izin verdiği için aşağıda verilen kod hataya neden olmaz. Bu kodda "Matematik" adını verdiğimiz namespace'in içinde "Aritmetik" adında ikinci bir namespace hazırladık.

```
namespace Matematik
{
 public partial class Islemler
  {
  }
 namespace Aritmetik
 {
  public partial class Islemler
   {
   }
  }
}
```

Bu 2 namespace'te de "Islemler" adında birer class olduğu için bu class'ların örnekleri alınırken dikkatli olmak gerekir. Aritmetik'te tanımlı "Islemler" class'ının örneğini hazırlamak üzere aşağıdaki gibi kod yazdık.

```
private void Form1_Load(object sender, EventArgs e)
{
 Matematik.Aritmetik.Islemler nesne;
 nesne = new Matematik.Aritmetik.Islemler();
}
```

Yukarıdaki sayfalarda hazırladığımız namespace'ler aynı kod dosyası içinde olduğu için ek bir işlem yapmaya gerek olmadan yani **using** deyimi ile namespace'i koda dahil etmeye gerek kalmadan kullandık. Konunun anlaşılması için **Add New Item** diyalog kutusundan yararlanıp projeye "Matematik.cs" adında boş bir kod dosyası hazırladık. Ardından projenin ilk formunun kod dosyasında hazırladığımız namespace'teki satırları bu dosyaya taşıdık.



Bu şartlarda bu namespace'lerdeki "Islemler" sınıfının örneği hazırlanabilir. Bu namespace DLL dosyası olarak düzenlenmiş olunsaydı **Add Reference** diyalog kutusundan yararlanıp Solution Explorer penceresinde listelemek gerekirdi.

Using Deyimi İle Namespace'lere Takma Ad Vermek

Şimdiye kadar anlatılanlara göre **using** deyimi ile kodun başında ilgilenilen namespace'ler işaret edilip bu namespace'lerdeki class'lardan yararlanılmak istendiği zaman class adından önce namespace'in adını yazmak bir zorunluluk olmaktan çıkıyor. Ayrıca C# uygulamalarında değişken tanımlama işlemlerinin hem C++'deki gibi olmasını sağlamak için hem de her değişken tanımlandığında ilgili yapının örneğinin kolayca alınmasını sağlamak için bazı yapılara takma adlar verilmiştir. Örneğin System'deki **Int32** yapısının takma adı **int**'tir. Benzer şekilde kendi hazırladığınız namespace'lere takma ad verebilirsiniz.

Şimdi örnek olması için System.Drawing'de yer alan **Imaging** adlı namespace'e takma ad vereceğiz. Her ne kadar Visual Studio ile hazırlanan projelere using deyimi sayesinde **System.Drawing**'deki sınıflara kolayca erişilebilinse bile bu namespace'in içinde ayrıca **Imaging** adında başka bir namespace var. Şimdi bu namespace'e using deyimi ile takma ad vereceğiz.

```
using imge = System.Drawing.Imaging;
```

Bu satır kodun üst kısmına yazıldıktan sonra System.Drawing.Imaging yazmak yerine kısaca "imge" denilebilir. **using** bloğunda System.Drawing.Imaging yazıldıktan sonra

namespace'teki sınıflar zaten kullanılabilmekle birlikte takma isim vermeye daha çok sınıf adı çakışmalarını önlemek için ihtiyaç duyulur. Bu konuda bilgi vermek için aşağıdaki gibi adı aynı olan birer sınıfa sahip 2 namespace hazırladık. Burada "Fonksiyonlar1" ve "Fonksiyonlar2" adını verdiğimiz namespace'leri ve içindeki class'ları siz hazırlamamış olabilirsiniz. Dolayısıyla elinizde kaynak kodları olmayabilir.

```
namespace Fonksiyonlar1
{
 class Class1
  {
   public string buyuk_harf(string str)
    {
     return str.ToUpper();
    }
  }
}
namespace Fonksiyonlar2
{
 class Class1
  {
   public string buyut(string str)
    {
     return str.ToUpper();
    }
  }
}
```

Her iki namespace'te aynı ada sahip birer class olduğu için bu 2 namespace'teki class'ları kullanmak istediğinizi using anahtar kelimesi ile belirtmeniz halinde hata meydana gelir. Ancak aşağıda yapıldığı gibi **using** deyimi ile bu namespace'lerden yalnızca birisini işaret ederseniz derleyici hata rapor etmez.

```
using System.Windows.Forms;
using Fonksiyonlar1;
namespace WindowsFormsApplication1
 {
  public partial class Form1 : Form
   {
    public Form1()
    {
     InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
     Class1 nesne1 = new Class1();
     textBox2.Text = nesne1.buyuk_harf(textBox1.Text);
    }
   }
}
```

Bu kod dosyasındaki "Fonksiyonlar2" adı verilen namespace'in kaynakları **using** deyimiyle kullanılmak istendiği belirtilirse hata meydana gelir. Çünkü derleyici hangi namespace'teki "Class1" sınıfının kullanılmak istendiğine karar veremez. Bu durumda bu namespace'lere aşağıda yapıldığı gibi takma ad vermek gerekir.

```
using System;
using System.Windows.Forms;
using space1 = Fonksiyonlar1.Class1;
using space2 = Fonksiyonlar2.Class1;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
  {
   public Form1()
   {
    InitializeComponent();
   }
   private void button1_Click(object sender, EventArgs e)
   {
    space1 nesne1 = new space1();
    space2 nesne2 = new space2();
    textBox2.Text = nesne1.buyuk_harf(textBox1.Text);
    textBox3.Text = nesne2.buyut(textBox1.Text);
   }
  }
}
```

Bu kodda takma ad, namespace yerine ilgili sınıfa verilmektedir. Sorunsuz çalışabilen bu takma ad verme tekniği ilgili namespace'te çok sayıda sınıf varsa pratik olmamaya başlar. Çünkü bu durumda her class için ayrı bir takma ad hazırlamak gerekir. Bu gibi durumlarda her class için ayrı bir takma ad hazırlamak yerine "::" operatöründen yararlanılır. Her ne kadar bu örnekte hazırlayıp "Fonksiyonlar1" ve "Fonksiyonlar2" adlarını verdiğimiz namespace'lerde birer class olsa bile aynı operatörle sorun yaşamadan istediğiniz class'ı kullanabilirsiniz. Tabi bu durumda takma adı class yerine namespace için hazırlamak gerekir.

```
using space1 = Fonksiyonlar1;
using space2 = Fonksiyonlar2;
private void button1_Click(object sender, EventArgs e)
{
 space1::Class1 nesne1 = new space1::Class1();
 space2::Class1 nesne2 = new space2::Class1();
 textBox2.Text = nesne1.buyuk_harf(textBox1.Text);
 textBox3.Text = nesne2.buyut(textBox1.Text);
}
```

using Deyimi ve IDisposable Arayüzü

using deyiminin yukarıda anlatılanlardan farklı bir kullanım şekli daha var. Bu kullanım şekli ancak **IDisposable** arayüzünü uygulayan, dolayısıyla **Dispose**() metoduna sahip sınıflar tarafından desteklenmektedir. Bu nedenle using deyiminin sözünü edeceğimiz kullanım şekli hakkında bilgi vermeden önce IDisposable arayüzü hakkında kısaca bilgi vermek gerekiyor.

Class'lar ve Interface'ler hakkında temel bilgiye sahip olunmadan IDisposable arayüzü hakkında anlatılanları hemen kavramak zor olacağı için bu konuyu yüzeysel anlatmayı tercih ettik. Öncelikle aşağıdaki gibi basit bir class hazırladık.

```
class Yoketme : IDisposable
{
 void IDisposable.Dispose()
  {
  }
}
```

"Yoketme" adını verdiğimiz bu sınıfa IDisposable arayüzünü uyguladığımız için bu arayüzün gerekleri yerine getirilmezse, başka bir deyişle bu sınıf **Dispose**() adında bir metoda sahip olmazsa hata meydana gelir. Bu nedenle bu sınıfta herhangi bir satıra sahip olmayan Dispose() adında bir metot hazırladık. Ardından bu sınıfın örneğini aldığımızı göstermek için aşağıdaki gibi 2. bir metot hazırladık.

```
class Yoketme : IDisposable
{
 void IDisposable.Dispose()
  {
   MessageBox.Show("Yoketme sınıfının örneği bellekten silindi");
  }
 public void nesne_durumu()
  {
   MessageBox.Show("Yoketme sınıfının örneği bellekte yaşıyor");
  }
}
```

Bu sınıfın Dispose() metodunda, nesnelerin bellekten silinmesini sağlayan herhangi bir işlem yapmadık. Yani bu sınıf bu hali ile dispose işlemini yapmaz. Buradaki amacımız **using** bloğunda hazırlanan nesne için Dispose() metodunun otomatik olarak işletildiğini göstermektedir. Şimdi sırada bu sınıfın örneğini alıp kullanmak var. Bunu yaparken **using** deyiminden yararlanacağız. using deyimini nasıl kullandığımızı aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 using (Yoketme nesne = new Yoketme())
  {
   nesne.nesne_durumu();
  }
}
```

using deyiminden sonra parantezlerin içinde IDisposable arayüzünü uygulayan sınıfın örneğini aldıktan sonra **using** için bir blok hazırlayıp bu bloğun içinde sınıfın normal metodunu işlettik. Şimdi durup düşünürseniz, using deyimine ait blokta ilgili metodun işletileceğini ve using bloğunun sonuna gelineceği için using'in işlevinin biteceğini sanabilirsiniz. Ancak böyle olmuyor; using bloğundan çıkıldığında using deyimine ait parantezlerin içinde hazırlanan nesne için otomatik olarak **Dispose** metodu işletilmektedir.

Burada "Yoketme" adını verdiğimiz sınıfa **IDisposable** arayüzünü uygulamayıp, dolayısıyla bu sınıfın örneğini using deyimine parametre olarak vermeseydik hazırlanan nesne uygun bir zamanda **Garbage Collector** mekanizması tarafından geri planda bellekten temizlenirdi.

5. BÖLÜM
DEĞİŞKEN TANIMLAMAK ve KULLANMAK

Değişkenler hakkında teknik bilgi vermeye başlamadan önce ileri bir tarihte kurumsal bir firmada programcı olarak çalışmaya başladığınızda işinize yarayacağını düşündüğümüz 2 öneride bulunacağız.

- Değişken tanımlarken ilk hedefiniz değişkenin kullanılmayacağı kapsamlarda yaşamasının önüne geçmek olmalıdır. Yani sadece bir metodun içinde kullanacağınız değişkeni o metodun içinde tanımlamalısınız. Class bloğu içinde değişken tanımlarken iyi düşünmek gerekir.
- C# kodu geliştirenlere tanımladıkları değişkenlere ad seçerken, değişken adlarının akılda kalıcı olmaya uygun, içeriği hakkında bilgi verici olmasına, mümkünse kısa olmasına, hatta kolay telaffuz edilebilir olmasına özen gösterilmesi önerilmektedir. Çünkü değişkenin içeriğine ve kullanım amacına uygun olmayan bir ad seçildiğinde kodun okunması zorlaşacaktır. Bu kitapta kullandığımız kısa kodlarda tanımladığımız değişkenlere ad seçerken özenli olmasak bile ekip çalışması olarak geliştirilen kapsamlı projelerde değişken ve sınıflara verilen adlar önemlidir.

C# ve diğer programlama dillerinde tipler genellikle 2 grup halinde incelenmektedir: Söz konusu dille hazır olarak gelen tipler, programcının hazırladığı tipler. Bu bölümde daha çok .NET Framework, dolayısıyla C# ile hazır olarak gelen tiplerden söz edilecektir. Programcının hazırladığı tipler class, struct, enum, delegate, interface vs şeklinde olabilir. Bu tiplerin nasıl hazırlanıp kullanıldığı bu kitabın ileriki sayfalarında anlatılacaktır.

Konu .NET dilleri olduğunda .NET Framework ile hazır olarak gelen tipler; referans tipler (reference types) ve değer tipleri (value types) şeklinde iki gruba ayrılmaktadır. Bu ayrım önemli olduğu içindir ki bu kitabın bir çok sayfasında reference types ve value types kavramlarını sıklıkla duyacaksınızdır.

Konu nesneye yönelik programcılık olduğunda class bloğu içinde değişken tanımlamak demek gerçekte ilgili sınıfın alanlara sahip olmasını sağlamak demektir. Metotların içinde tanımlanan değişkenlere ise yerel değişken adı verilmektedir. Bu bölümde değişkenler konusunu işlerken bazen geleneksel alışkanlığa uyup konuya class penceresinden bakmamayı tercih ettik. Yani hem class bloğu içinde tanımlandığı için alan (field) denilen sınıf üyelerine, hem de metot içinde tanımlandığı için yerel değişkenlere **değişken** adını verdik.

.NET ortamı için geliştirilen bütün diller aynı veri tiplerine sahiptir. Bu bağlamda dillerin arasındaki fark, değişkenleri tanımlama şeklindedir. Örneğin Visual Basic'te tamsayı tipteki değişkenleri tanımlamak için **"Dim sayi As Integer"** şeklinde kodlama yapılmaktadır. Aynı değişken C# projesi dahilinde tanımlanmak istendiğinde ise "**int sayi;**" şeklinde kodlama yapılarak değişken tanımlanır.

Başka bir deyişle, .NET uyumlu bütün dillerde aynı özelliklere sahip değişkenler bulunmaktadır. Çünkü .NET uyumlu her programlama dili **Common Type System** adı verilen ortak tip tanımlama kurallarına bağlıdır. Bu nedenle C# projelerinde tanımlanan her değişkenin Common Type System'de bir karşılığı vardır.

.NET uyumlu her programlama dili Common Type System'de tanımlı türlere kendince takma adlar vermektedir. Örneğin C#'ta tam sayı bilgileri bellekte saklarken kullanılan **int** tipindeki değişkenlerin CTS'deki karşılığı **System.Int32**'dir. Benzer şekilde C#'ta kullanılan **bool** değişkenlerin Common Type System'deki karşılığı **System.Boolean**'dır. **int** gerçekte System.Int32 yapısının (struct) takma adıdır. Kesirli sayıları bellekte tutmak amacıyla kullanılan **float** değişkenlerin Common Type System'deki karşılığı ise **System.Single**'dır.

Değişkenlerin içerikleri bir tipten başka bir tipe **Convert** sınıfı sayesinde dönüştürülürken Common Type System'deki karşılıkları kullanılmaktadır. Örneğin TextBox'a yazılan ve rakamlardan meydana gelen bilgi **int** tipindeki değişkene aktarılmadan önce aşağıdaki gibi bir dönüştürme yapılabilir. Burada kullanılan **ToInt32**(), Convert sınıfının bir metodudur.

```
int sayi;
sayi = Convert.ToInt32(textBox1.Text);
```

Bu nedenle C# projelerinde kullanılan tiplerin Common Type System'de bir karşılığı olduğunu bilmeniz bazı konuları anlamanıza yardım edecektir. Bu bölümde değişken tipleri anlatılırken bazen ilgili değişkenin Common Type System'deki karşılığından söz edilecektir. Common Type System'deki veri tipleri ve C#'taki karşılıkları aşağıda tablo halinde verildi.

.NET Yapısı	C#	Boyut(byte)	Kapasite
System.Byte	byte	1	0-255
System.Sbyte	sbyte	1	-128 ile 127
System.Int16	short	2	-32768 ile 32767
System.Int32	int	4	-2147.483 647 ile 2147.483.647
System.Int64	long	8	9.223.372.036.854.775.808
System.Uint16	ushort	2	0- 65535
System.Uint32	uint	4	0 – 4.294.967.295
System.Uint64	ulong	8	0- 18.446.744.073.709.551.615
System.Single	float	4	$1.5E^{-45}$ ile $3.4E^{+38}$
System.Double	double	8	$5E^{-324}$ ile $1.7E^{+308}$
System.Object	object		Tüm türler için temel sınıf
System.Char	char	2	Unicode karakterler
System.String	string		Karaktersel bilgiler
System.Decimal	decimal	8	Finansal bilgiler –en fazla 28 hane
System.Boolean	bool	1	true veya false

Common Type System kaynaklı değişken tiplerine Microsoft tarafından verilen takma adlar yerine, kendiniz ad verip kullanabilirsiniz. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Kod dosyasının başında **using** anahtar kelimesi ile ilgili yapıyı istediğimiz takma ada eşitledik.

```
using System;
using Tamsayi = System.Int32;
namespace ConsoleApplication1
{
 class Program
  {
    static void Main(string[] args)
    {
      Tamsayi i;
      i = 2017;
    }
  }
}
```

Bu hazırlıktan sonra **Int32** tipinde değişken tanımlarken bu takma addan yararlanabiliriz. Değişkenlere bu şekilde farklı takma adlar verip kullanmak kodun okunurluğunu azaltacağı için önerilmez.

Tekrar etmek gerekirse; int, char, double, byte vs gerçekte birer takma addır. Örneğin **int**, "System.Int32" yapısının (struct) takma adıdır. Bu nedenle değişken tanımlarken mevcut takma adlar yerine ilgili yapıyı kullanabilirsiniz. Aşağıda verilen kodda takma ad kullanılmadan "Int32" tipinde bir değişken tanımlanmaktadır.

```
System.Int32 i;
i = 2017;
textBox1.Text = i.ToString();
```

Daha sonraki konularda göreceğiniz gibi yapılar (struct) tıpkı class'lar gibi yapıcı metotlara sahip olabiliyorlar ve örnekleri alınabilmektedir. Bu nedenle **System.Int32** ve **System.Byte** gibi yapıların örnekleri aşağıdaki gibi bilinen şekilde alınabilir.

```
System.Int32 i = new System.Int32();
i = 2017;
textBox1.Text = i.ToString();
```

C# programcıları yıllar itibarıyla değişken tanımlarken takma adları kullanmayı alışkanlık haline getirdiler. Bu nedenle değişken tanımlarken .NET Framework ile gelen Int32 ve Int64 gibi yapılar yerine bu yapıları temsil eden takma adları kullanmanızı öneririz.

Değişken Tanımlamak ve İlk Değer Aktarmak

C#'ta değişken tanımlanırken önce değişkenin tipi, ardından değişkenin adı belirtilmektedir. Örnek olması için üzerinde çalıştığımız Windows Forms uygulamasının formuna yerleştirdiğimiz button nesnesinin Click olayını temsil eden metodun içinde **string, int** ve **bool** tipinde 3 değişken tanımlayıp bilgi aktardık.

```
private void button1_Click(object sender, EventArgs e)
{
 string ad;
 int yil;
 bool medeniHal;
```

```
    ad = "Fatih Terim";
    yil = 2017;
    medeniHal = true;
}
```



C# derleyicisi değişken adları konusunda büyük-küçük harf ayrımı yapmaktadır. Değişken adının ilk karakteri harf olmalıdır ve değişken adı olarak anahtar kelimeler kullanılamaz. C# programcılarının metotların içinde tanımlanan değişkenlerin ilk harfini küçük olarak seçmek gibi bir tercihleri vardır.

Değişkenleri tanımlarken aynı zamanda ilk değerini aktarabilirsiniz. İlk değer ataması yapılırken aktarılan bilginin, değişkenin tipine uygun olmasına dikkat edilmelidir. Değişken tanımlarken ilk değerin nasıl aktarıldığını aşağıda görebilirsiniz.

```
string takim = "Fenerbahçe";
int yil = 2017;
bool medeniHal = true;
```

Programın yazımı sırasında söz konusu değişkene aktarılacak ilk değer belli değilse sayısal değişkenlere 0 ve **bool** değişkenlere **false** aktarılabilinir. Değişkenlere varsayılan başlangıç değeri aktarılırken **new** anahtar kelimesi kullanılabilir. new anahtar kelimesi ile değişkenlerin nasıl tanımlanıp ilk değer aktarıldığını aşağıda görebilirsiniz.

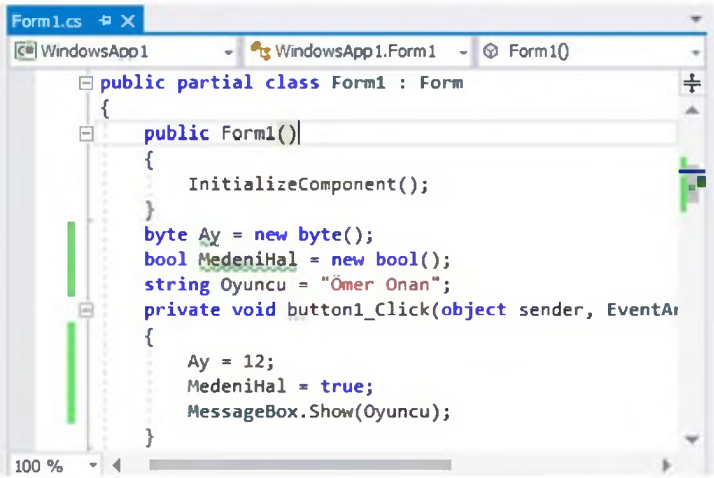
```
int yil = new int();
bool medeniHal = new bool();
char cevap = new char();
float katsayi = new float();
```

new anahtar kelimesi C# programlama dilinde önemlidir. Çünkü new ile sınıflardan yola çıkılarak nesne hazırlanmaktadır.

Yerel Olan-Olmayan Değişkenler

Yukarıda değişkenleri **button1_Click()** metodu içinde tanımladık. Dolayısıyla bu değişkenleri ilgili sınıfın (burada Form1) başka metotlarında kullanmak mümkün değildi. Çünkü herhangi bir metodun içinde tanımlanan değişken o metottan çıkılıp başka bir metoda geçildiğinde tanımsız duruma düşmektedir. C# ve diğer nesneye yönelik programlama dillerinde metot içinde tanımlanan değişkenlere **yerel değişken** adı verilmektedir.

Tanımlamak istediğiniz değişkeni sınıfın her yerinde yani bütün metotlarında kullanabilmek istiyorsanız tanımlama işlemini class bloğunda yapmalısınız. Aşağıda verilen kodda tanımlanan değişkenler bu formdaki (Form1 sınıfı) bütün metotlarda kullanılabilir. Çünkü bu değişkenler class bloğu içinde tanımlandı. Belki bazılarımızın aklına şöyle bir soru gelmiş olabilir: Değişkenler namespace bloğunda tanımlanabilir mi? Bu sorunun cevabı hayırdır.



C# gibi nesneye yönelik programlama dillerinde class bloğu içinde tanımlanan değişkenlere **alan** (field) adı verilmektedir. Buna rağmen bu bölümde alanları da değişken olarak adlandıracağız. Daha sonra alanlar hakkında bilgi verildikten sonra yerel olmayan değişkenleri alan olarak ifade edeceğiz.

Yukarıda değişkenlerin yaşam alanı konusunda anlatılanlardan başka C# derleyicisi yalnızca blok içinde geçerli olabilen değişkenlerin tanımlanıp kullanılmasına izin vermektedir. Aşağıda verilen kodda Form1 sınıfının Form1_Click() metodu içinde "if" deyimi ile sınırlanan blokta "oyuncu" adında **string** bir değişken tanımladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 string takim = "Fenerbahçe";
 if (takim == "Fenerbahçe")
  {
    string oyuncu;
    oyuncu = "Tuncay Şanlı";
    textBox1.Text = oyuncu;
  }
}
```

Bu metot işletilirse "if" deyimi ile kontrol edilen koşul doğru olacağından blok işletilir ve bu bloktaki satırlar sayesinde değişken tanımlanıp kullanılır. Bloktan çıkıldığında burada "oyuncu" adını verdiğimiz değişken tanımsız duruma düşer. Bu nedenle bu metot aşağıdaki gibi düzenlenirse hata meydana gelir.

```
private void Form1_Click(object sender, EventArgs e)
{
 string takim = "Fenerbahçe";
 if (takim == "Fenerbahçe")
  {
    string oyuncu;
    oyuncu = "Tuncay Şanlı";
  }
 textBox1.Text = oyuncu;  // Hata meydana gelir.
}
```

Yukarıdaki sayfada değişken tanımlanırken mümkün olan en dar kapsama sahip olmasının hedeflenmesi gerektiği söylenmişti. Bu açıdan yukarıda "oyuncu" adını verdiğimiz değişken sadece "if" ile sınırlanan blokta kullanılacağı için metot bloğunda tanımlanması önerilmez.

Tanımlanan değişken hiç kullanılmazsa hata meydana gelmez ama derleme sırasında uyarı mesajı alınır. Benzer şekilde ilk değer aktarılmayan değişken herhangi bir şekilde kullanılmak istendiğinde hata meydana gelir. Bu son cümle statik değişkenler için geçerli değildir. Değişkenlerin yaşama alanı konusundaki bir ayrıntıyı ortaya çıkarmak için aşağıda verilen kodu hazırladık.

```
string oyuncu = "Cemil Turan";
private void Form1_Click(object sender, EventArgs e)
{
 string oyuncu = "Tuncay Şanlı";
 textBox1.Text = oyuncu;
}
```

Verilen kodu dikkatlice incelerseniz hem class bloğunda hem de metodun içinde aynı ad ve tipe sahip iki değişken tanımlayıp bilgi aktardık. Hataya neden olmayan bu kodu işletirseniz TextBox'a metodun içinde tanımlanan değişkenin içeriği aktarılır. Eğer class bloğu içinde tanımlanan yani ilk değeri "Cemil Turan" yapılan değişkenle ilgileniyorsanız bu değişkenin adının önüne "bu sınıf" anlamında **this** eklemelisiniz.

```
string oyuncu = "Cemil Turan";
private void Form1_Click(object sender, EventArgs e)
{
 string oyuncu = "Tuncay Şanlı";
 textBox1.Text = oyuncu;
 textBox2.Text = this.oyuncu;
}
```

Tahmin edeceğiniz gibi burada "Tuncay Şanlı" bilgisini aktardığımız değişken bu sınıfın (Form1) başka metotlarında yaşamayacağı için class bloğu içinde tanımlanan değişkeni başka yerde kullanırken adının önüne this'i yazmaya gerek yoktur.

Değişkenleri Public Yapmak

Tanımladığınız değişkeni uygulamanızdaki diğer sınıflarda kullanabilmek istiyorsanız **public** yapmalısınız. Tanımlama satırında, başına public veya private yazılmayan değişkenlerin **private** oldukları kabul edilir ve yalnızca tanımlandıkları sınıfta kullanılabilirler.

Bu paragraftaki son cümleyi lütfen bir kez daha okuyunuz. İlk bakışta bu cümlede herhangi bir hatalı durum görülmese bile kapsamı (scope) belli değildir. Dikkatinizi konuya çekmek için bu cümleyi metinden çıkarmadık. Şimdi konuyu baştan ele alıyoruz.

Amacımız uygulamadaki diğer sınıflarda yaşayacak değişken tanımlamaktır. En başta belirtmek gerekir ki bu amaca hizmet edecek değişkeni metotların içinde tanımlayamayız. Çünkü metotların içinde tanımlanan değişkenlerin yaşama alanları metotlarla sınırlıdır.

Dolayısıyla konu uygulamalardaki diğer sınıflarda yaşaması istenen değişkenler olduğunda tanımlama yeri class bloğudur. Bu söylediğimizi netleştirmek için üzerinde çalıştığımız konsol uygulamasında 2. bir sınıfa yer verdik.

```
public class Program
{
 static void Main(string[] args)
  {
   Console.ReadLine();
  }
}
public class Takimlar
{
 string Takim;
 int Kurulus;
}
```

Dikkat ettiyseniz burada "Takimlar" adını verdiğimiz bu sınıfta class bloğunda tanımlanmış 2 değişken vardır. Bu değişkenlerin başına public veya internal yazılmadığı için ancak "Takimlar" sınıfının metotlarında yaşarlar. Yani class bloğunda tanımlanıp public veya internal olarak işaret edilmeyen değişkenler o sınıfa özel olurlar. Bu değişkenleri uygulamadaki diğer sınıflarda kullanmak istiyorsanız aşağıda yaptığımız gibi tanımlama satırında **public** anahtar kelimesine yer vermelisiniz.

```
public class Takimlar
{
 public string Takim;
 public int Kurulus;
}
```

Bu public değişkenlerin başka sınıflarda kullanılabildiğini göstermek için üzerinde çalıştığımız uygulamada Main() metoduna sahip diğer sınıfta Main() metodunun içinde **new** anahtar kelimesinden yararlanıp "Takimlar" tipinde bir nesne hazırladık.

```
class Program
{
 static void Main(string[] args)
  {
   Takimlar t1 = new Takimlar();
  }
}
```

Bu andan itibaren buradaki adıyla "Program" sınıfı içinde "Takimlar" sınıfının "Takim" ve "Kurulus" adını verdiğimiz değişkenlerine (alan) erişim sağlayabiliriz. Yani Main() metodu aşağıdaki gibi düzenlenirse derleyici hata rapor etmez. Bu sırada "Takimlar" sınıfının "Takim" ve "Kurulus" değişkenleri public olmasaydı bu kod hataya neden olurdu.

```
static void Main(string[] args)
{
 Takimlar t1 = new Takimlar();
 t1.Takim = "Beşiktaş";
```

```
 t1.Kurulus = 1903;
}
```

Public anahtar kelimesinin işlevini göstermek için şimdi 2. bir örnek hazırlayacağız. Bu amaçla üzerinde çalıştığımız Windows Forms uygulamasına 2. bir form dahil ettik. Ardından 2. formu ekrana getirebilmek için ilk formun üzerine bir düğme yerleştirip bu düğmenin Click olayını temsil eden metodu aşağıdaki gibi düzenledik.

```
private void Goster_Click(object sender, EventArgs e)
{
 Form2 f2 = new Form2();
 f2.Show();
}
```

Üzerinde çalışılan projeye 2. bir form dahil etmek denilince; bunu, **System.Windows.Forms** adlı namespace'te yer alan **Form** sınıfının mirasçısı 2. bir sınıf hazırlamak olarak düşünebilirsiniz. Burada yapılan ise şudur: Form2 tipinde bir nesne hazırlanmakta ve **Show**() metoduyla bu nesne pencere olarak ekrana getirilmektedir.

Şimdi "Form1" sınıfında tanımlayıp bilgi aktaracağımız string değişkenin içeriğini Form2'nin başlığında kullanacağız. Bu amaçla **public** anahtar kelimesini kullanarak "Form1" sınıfının class bloğunda aşağıda yapıldığı gibi bir değişken tanımlayıp bilgi aktardık. "Form1" sınıfına ait kodun tamamını vermediğimiz için ilk bakışta burada "Baslik" adını verdiğimiz değişkenin hangi blok içinde tanımlandığı fark edilmeyebilir.

```
public string Baslik;
private void Goster_Click(object sender, EventArgs e)
{
 Baslik = "Fenerbahçe";
 Form2 f2 = new Form2();
 F2.Show();
}
```

Bu şekilde düzenlenen "Form1" sınıfının burada "Baslik" adını verdiğimiz public değişkene Form2'den erişebiliriz ama bu değişkene aktarılan bilgi Form2'de kaybolur. Bu nedenle burada "Baslik" adını verdiğimiz değişkene bilgi aktarmayı "Form1" sınıfının yapıcı metodunda yaptık.

```
public partial class Form1 : Form
{
 public string Baslik;
 public Form1()
 {
   InitializeComponent();
   Baslik = "Fenerbahçe";
 }
}
```

C# programlama dilinde sınıfla aynı ada sahip olan metoda yapıcı metot adı verildiğini ve bu metodun sınıftan yola çıkıp **new** anahtar kelimesi ile nesne hazırlanırken işletildiğini belirtip ayrıntıları daha sonraya bırakıyoruz. Bu şekilde tanımlayıp bilgi aktarılan değişkenin

diğer sınıflarda kullanılabilindiğini göstermek için Form2 sınıfının **Click** olayıyla ilgili metodunu aşağıdaki gibi düzenledik.

```
private void Form2_Click(object sender, EventArgs e)
{
 Form1 f1 = new Form1();
 this.Text = f1.Baslik;
}
```

Buraya aldığımız 2 satırda, "Form2" sınıfı içinde "Form1" tipinde bir nesne hazırladık ve bu nesnenin "Baslik" adlı alanın içeriğini Form2 tipindeki nesnenin **Text** özelliğine aktardık.

C# gibi nesneye yönelik programlama dillerinde değişken tanımlama işlemini class'lardan bağımsız düşünmemek gerekiyor. Buna rağmen hem geleneksel alışkanlığa bağlı kalmak hem de konunun kolay anlaşılmasını sağlamak için değişken tanımlama işlemlerini bu bölümde class penceresinden bakmadan ele aldık. Buna rağmen zaman zaman konuları nesneye yönelik programcılık ilkeleri çerçevesinde anlattık.

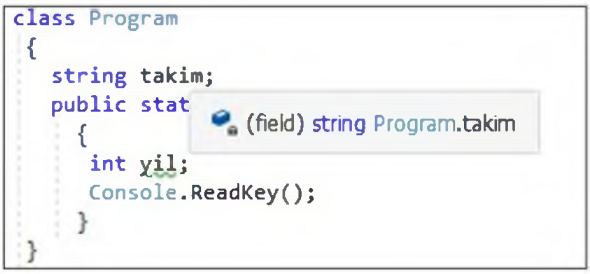
Static Değişkenler

Bu kitabın ilk bölümünde statik metotlardan biraz söz edilmişti. Çünkü C# uygulamalarının başlangıç noktası olan Main() metodunun statik olması şarttır. Bu nedenden dolayı "C# İle Programcılığa Başlamak" adlı bölümde yüzeysel de olsa statik metotlardan söz etmiştik. Bu bölümde ise konu değişkenler penceresinden ele alınacaktır. Yani bu kitapta, statik sınıflardan bir yerde, statik alanlardan 2 yerde, statik metotlardan 3 yerde söz edilmiş olacaktır. Bu sayede bu kitabın sonuna geldiğinizde statik alanlar, metotlar ve sınıflar hakkında ayrıntılar dahil her şeyi kavramış olacaksınız.

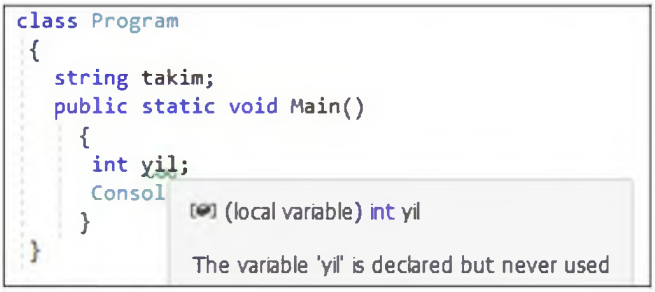
Yukarıdaki sayfalarda işaret edildiği gibi C# uygulamaları dahilinde class bloğunda tanımlanan değişkenlere sınıfın alanları demek gerekir. Metot veya for döngüsü içinde yaşayanlara değişken demekte fayda vardır. Bu sınıflandırma hakkında sizi ikna edebilmek için Visual Studio ile hazırladığımız uygulamadaki "Program" adlı sınıfta class bloğunda ve Main() metodu içinde birer değişken tanımladık.

```
class Program
{
 string takim;
 public static void Main()
 {
  int yil;
  Console.ReadKey();
 }
}
```

Fare işaretini class bloğu içinde tanımlanan değişkenin üzerine götürünce Visual Studio bunun bir **field** olduğunu rapor etti.



Benzer şekilde fare işaretini Main() metodu içinde tanımladığımız yerel değişkenin üzerine götürünce Visual Studio, bunun bir **variable** olduğunu işaret etti.



C# derleyicisi metotların içinde statik değişken tanımlanmasına izin vermediğine göre statik değişken yerine statik alan diye adlandırma yapmak daha doğru olacaktır. Daha önce işaret edildiği gibi C ve C++ gelenekleri nedeniyle C# kodu içinde standart tiplerle yapılan rezervasyon işlemine değişken tanımlama denilmektedir.

Bu kitabı baştan okuyup bu sayfalara kadar gelen birisi ister istemez ileriki sayfalarda class'lar ve metotlar hakkında yazılanlardan henüz habersizdir. Buna rağmen bu sayfada **static** değişkenlerden söz etmeden önce kısaca statik metotlardan söz etmek gerekiyor. Çünkü statik metotlar hakkında yazılanları anlamak daha kolaydır.

Normalde herhangi bir sınıftaki bir metot işletilmek istendiğinde önce **new** anahtar kelimesi ile o sınıfın örneği alınır (yani o class'tan yol çıkılıp nesne hazırlanır). Daha sonra söz konusu metot nesne üzerinden işletilir. Ancak işletilmek istenen metot statikse sınıfın örneğini almaya yani sınıftan yola çıkıp nesne hazırlamaya gerek yoktur. Çünkü statik metotlar sınıfla bağlantılı oldukları için nesne üzerinden değil sınıf adıyla çağrılıp işletilirler.

Bu konuda ön bilgi vermek için .NET Framework gelen ve **System.Collections** adlı namespace'ye yer alan **ArrayList** sınıfının örneğini aldık. Başka bir deyişle ArrayList nesnesi hazırladık. Aşağıda verilen metottaki satır sayesinde başlangıç kapasitesi 10 olan bir ArrayList nesnesi hazırlanır.

```
private void button1_Click(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList(10);
}
```

Bu şekilde **new** anahtar kelimesi ile bu sınıfın örneği alındıktan sonra ArrayList sınıfının statik olmayan bütün metotları bu nesne üzerinden işletilebilinir. Başka bir deyişle Microsoft firmasının programcılarının yazıp ArrayList sınıfına üye yaptıkları bütün statik olmayan metotlar burada "liste" adını verdiğimiz nesne üzerinden işletilmek üzere hazır olarak bekliyorlar. Örneğin bu koda aşağıdaki gibi bir satır yazılıp ArrayList sınıfından "liste" adlı nesneye intikal eden **Add**() metodu ile bu nesneye eleman eklenebilir.

```
private void button1_Click(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList(10);
 liste.Add("Fenerbahçe");
}
```

ArrayList sınıfının Add() gibi statik olmayan metotları olduğu gibi statik metotları da bulunmaktadır. Dolayısıyla statik metotları kullanabilmek için yukarıda yapıldığı gibi **new** anahtar kelimesiyle sınıftan yola çıkıp nesne hazırlamak gerekmiyor. Örneğin ArrayList sınıfının **FixedSize**() metodu statik bir metottur ve kendisine parametre olarak verilen mevcut ArrayList nesnesinin kapasitesi sabitlenmiş halini geriye göndermektedir. FixedSize() metodu aşağıdaki gibi kullanılabilir. Aşağıda FixedSize() metodunu nesne üzerinden değil sınıf üzerinden işlettik.

```
private void button1_Click(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList(2);
 liste.Add("Fenerbahçe");
 liste.Add("Galatasaray");
 liste = ArrayList.FixedSize(liste); //Sınıf üzerinden metot işletme
}
```

FixedSize() metodu ile yapılan bu işlemden sonra ArrayList tipindeki "liste" adını verdiğimiz nesneye Add() metodu ile 3. bir eleman eklenemez. Yine .NET Framework ile gelen **Math** sınıfının bütün metotları statiktir. Örneğin bir sayının karekökü alınmak istendiği zaman Math sınıfının **Sqrt**() metodu aşağıdaki gibi kullanılabilir.

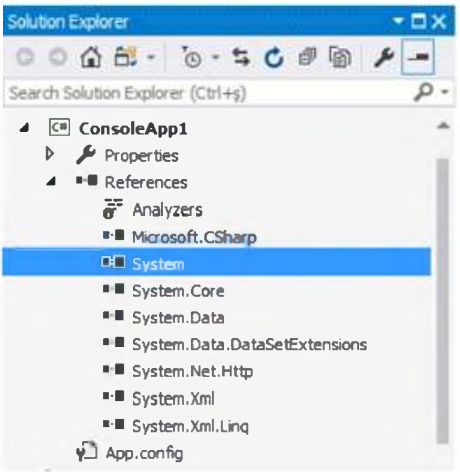
```
private void button1_Click(object sender, EventArgs e)
{
 int sayi = 16;
 double kok = Math.Sqrt(sayi);
}
```

Math sınıfının bütün metotları statik olduğu, dolayısıyla metotları sınıf üzerinden işletildiği için bu sınıftan yola çıkıp nesne hazırlamaya gerek yoktur. Diğer yandan .NET Framework ile gelen Math sınıfı statik bir sınıf olduğu için new anahtar kelimesi ile örneği alınamaz. Bu nedenle aşağıda verilen satır hataya neden olur.

```
private void button1_Click(object sender, EventArgs e)
{
 System.Math m = new System.Math(); // hata meydana gelir.
}
```

Math sınıfı "System" adlı namespace'te yer aldığı ve System, Visual Studio ile hazırlanan uygulamalara otomatik olarak dahil edildiğinden **Math** sınıfının metotları her yerde işletilmek üzere hazır olarak beklemektedirler. Bu son cümleyi daha açık yazmak gerekirse;

.NET Framework ile gelen çok sayıda sınıf "System" adlı namespace'de toplanmıştır. System, kullandığımız bilgisayarda "System.dll" dosyası olarak bulunmakta ve bu DLL dosyası yani reference her C# uygulamasına Visual Studio tarafından dahil edilmektedir. Uygulamaya dahil edilmiş referansları **Solution Explorer** penceresinde görebilirsiniz.



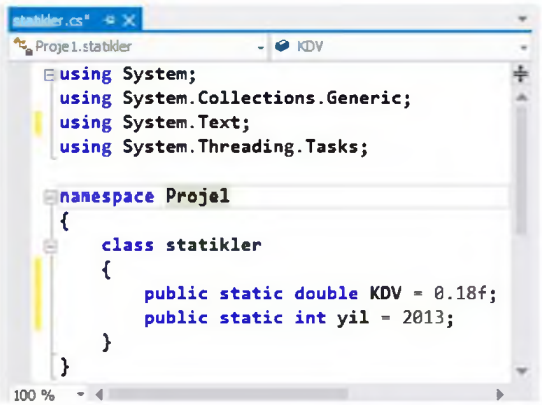
"System.dll" dosyasının başka bir deyişle "System" adlı referansın uygulamaya dahil edilmesinin yanında System nasmespace'teki sınıfları namespace adını yazmaya gerek kalmadan kullanabilirsiniz. Çünkü Visual Studio, her uygulamaya ait using bloğunda "using System;" satırına yer vermektedir.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApp1
{
    class Program
    {
        public static void Main()
        {

        }
    }
}
```

Sınıflarda statik metotlar olduğu gibi statik değişkenler, nesneye yönelik programcılık terimi ile söylemek gerekirse "statik alanlar" olabilmektedir. Statik alanlar tıpkı statik metotlar gibi ilgili sınıfın örneğini almaya gerek olmadan sınıf üzerinden kullanılabilmektedir. Başka bir deyişle statik değişkenlere nesne üzerinden değil sınıf üzerinden erişilmektedir.

Statik değişkenler hakkında bilgi vermek için üzerinde çalıştığımız Windows Forms uygulamasına CS uzantılı yeni bir kod dosyası dahil edip bu dosyanın içeriğini aşağıdaki gibi düzenleyip bir class hazırladık.



Bu bölümde tercih edilen değişken adları başka yerli kitaplarda kullanılmış olabilir. Olası değişken adı benzerlikleri tesadüften ibarettir. Yoksa "yil" değişkenine "year" adını vermek marifete dahil değildir.

Ayrı bir CS dosyası olarak düzenlenen bu sınıfta class bloğu içinde tanımlanmış 2 statik değişken vardır. Metotların içinde tanımlanamayan statik değişkenler doğaları gereği **public** olmalıdır. Bu şekilde 2 statik değişkene sahip bir sınıf hazırladıktan sonra KDV adını verdiğimiz değişkeni diğer sınıfta aşağıdaki gibi kullandık.

```
private void button1_Click(object sender, EventArgs e)
{
 int fiyat = 1200;
 double kdv = statikler.KDV;
 double kdv_dahil = fiyat + fiyat * kdv;
}
```

Başka bir sınıftaki statik değişkeni bu şekilde kullandığınız gibi mevcut içeriğini değiştirebilirsiniz. Aşağıda yapıldığı gibi statik değişkenin içeriği değiştirilirse, uygulama sona erene veya aynı şekilde değiştirilene kadar bu içerik korunur.

```
private void button1_Click(object sender, EventArgs e)
{
 statikler.KDV = 0.10f;
}
```

Statik değişkenlerle ilgili bir ayrıntıdan söz etmek için üzerinde çalıştığımız konsol uygulamasında aşağıdaki gibi kısıtlı özelliklere sahip bir class hazırladık. Bu class'ın farkı, yapıcı metoda sahip olması ve yapıcı metotta statik değişkenin içeriğinin değiştirilmesidir.

```
public class Katsayilar
{
 public static double KDV = 0.18;
 public Katsayilar()
 {
  KDV = 0.15;
 }
}
```

Bu sınıfın KDV adını verdiğimiz statik alanı sınıf tipinde nesne hazırlanmadığı veya doğrudan alanın içeriğinde değişiklik yapılmadığı sürece mevcut içerik korunur. Ne zaman ki aşağıda yapıldığı gibi new anahtar kelimesi bu sınıf tipinde nesne hazırlanır o zaman statik alanın içeriği değişmiş olur.

```
class Program
{
 public static void Main()
 {
  Console.WriteLine(Katsayilar.KDV); // 0.18
  Katsayilar nesne = new Katsayilar();
  Console.WriteLine(Katsayilar.KDV); // 0.16
 }
}
```

Statik değişkenlerle ilgili önemli bir ayrıntıdan söz etmek için aşağıda verilen kodu hazırladık. Bu kodda class bloğunda tanımlanan statik değişkene herhangi bir değer atanmadan Main() metodu içinde içeriği konsola yazılmaktadır.

```
public class Program
{
 public static int Yil;
 public static void Main()
  {
   Console.WriteLine(Yil);
  }
}
```

Bu kod hata üretmez ve konsola 0 yazılır. Çünkü C# derleyicisi ilk değeri atanmayan statik değişkenlere varsayılan değeri otomatik olarak atamaktadır. int değişkenlerin varsayılan değeri 0 olduğu için konsola 0 yazılır. Statik olmayan değişkenleri ilk değer atamadan kullanamadığımızı biliyorsunuz.

Sabit Tanımlamak ve Readonly Değişkenler

Programcılar, tanımladıkları değişkenin içeriğinin bazen değiştirilmesini istemezler. Bu gibi durumlarda normal değişken yerine **const** anahtar kelimesi ile sabit tanımlanabilir. Aşağıda 2 sabit tanımlama örneği bulunmaktadır. Bazı programcıların sabitlere ad seçerken büyük harfleri kullanmak gibi bir alışkanlıkları vardır.

```
const string Takim = "Fenerbahçe";
const double PI = 3.14;
```

const anahtar kelimesi ile sabit tanımlarken ilk değerin (aynı zamanda son) aktarılması gerekiyor. Sabitin ilk değeri aktarılmazsa hata meydana gelir. Başka bir deyişle derleme sırasında sabitin alacağı değerin belli olması gerekiyor.

Sabitlere, tıpkı statik değişkenlerde olduğu gibi sınıf üzerinden ulaşılmaktadır. Tanımlanan sabite başka sınıflardan erişim sağlamak için erişim belirleyici olarak public'i kullanmak gerekir. Sabitlerin başka sınıflarda kullanmaya örnek olması için aşağıda verilen kodu yazdık.

```
public class Sabitler
{
 public const float PI = 3.14f;
}
class Program
{
 public static void Main()
 {
  Console.WriteLine(Sabitler.PI);
 }
}
```

Tanımlamak istediğiniz sabit için seçeceğiniz tip **value type** olmak zorundadır. Bu nedenle aşağıda verilen sabit tanımlama satırı hataya neden olur. Object yerine **string** seçseydik hata meydana gelmezdi. Çünkü string değişkenler referans tip olsalar bile özel durumları vardır.

```
public class Sabitler
{
 public const object Sehir = "Ankara"; // Hata meydana gelir.
}
```

Burada const olarak tanımlanan üyeler aynı zamanda static'tir. Çünkü statik değişkenlere nasıl erişiliyorsa sabitlere aynı şekilde erişiliyor. const olarak ayarlanan üyeleri ayrıca static anahtar sözcüğüyle belirtmek hataya neden olur.

const değişkenin içeriği kod yazılırken bilinmelidir. Bazen programın yazımı sırasında değil de nesne oluşturulurken içeriği belirlenebilinen sabitlere gerek duyulur. Nesne oluşturma işlemi sırasında yapıcı metotta içeriği ayarlanılabilinen değişkenlere **readonly** denilmektedir. Bu değişkenler üzerine konuşabilmek için aşağıdaki gibi bir sınıf hazırladık.

```
public class Sabitler
{
 public readonly int Yil;
 public Sabitler()
 {
  this.Yil = 2017;
 }
}
```

Bu sınıfta **readonly** özelliğini verdiğimiz değişken veya alanın içeriği sınıfın yapıcı metodunda değiştirilmektedir. Readonly değişkenlerin içerikleri yalnızca yapıcı yani constructor metotlarda değiştirebildiği için başka bir metotta bu değişkenin içeriği değiştirilmek istenirse hata meydana gelir. Bu readonly değişkenden yararlanabilmek için uygulamadaki diğer sınıfta yukarıda verilen sınıfın örneğini hazırladık.

```
class Program
{
 public static void Main()
 {
  Sabitler nesne = new Sabitler();
  Console.WriteLine(nesne.Yil);
 }
}
```

```
public class Sabitler
{
 public readonly int Yil;
 public Sabitler()
  {
    this.Yil = 2017;
  }
}
```

Bu anlatılanlara göre readonly değişkenlerin içerikleri **new** anahtar kelimesinden yararlanıp nesne hazırlanırken yapıcı metotlar aracılığı ile değiştirilebilmektedir. Bu örnekteki "Sabitler" sınıfının yapıcı metodunun bir parametreye sahip olması sağlanıp **readonly** değişkenin içeriği kullanıcıdan istenebilir. Bunun nasıl yapıldığını aşağıda görebilirsiniz.

```
class Program
{
 public static void Main()
 {
  Sabitler nesne1 = new Sabitler(2016);
  Console.WriteLine(nesne1.Yil);
  Sabitler nesne2 = new Sabitler(2017);
  Console.WriteLine(nesne2.Yil);
  Console.ReadKey();
 }
}
public class Sabitler
{
 public readonly int Yil;
 public Sabitler(int i)
 {
  this.Yil = i;
 }
}
```

Uygulamanızda çok sayıda sabit tanımlamak durumunda kaldıysanız ve bu sabitleri ileri tarihlerde değiştirme ihtimali varsa, bu sabitleri yukarıda yaptığımız gibi tanımlamanız önerilmiyor. Çünkü yazıp kullanıma sunduğunuz programı güncellemeniz gerekebilir. Size önerimiz, sabitlerde tuttuğunuz bilgileri resource dosyalarında tutmanız şeklindedir.

Object Değişkenler

Bu kitabın birçok yerinde ifade edildiği üzere C# programlama dilinde bütün sınıflar **Object** sınıfının mirasçıdır. Dolayısıyla Object tipindeki bir nesneye herhangi tipteki başka bir nesnenin referansı aktarılabilir. Benzer şekilde Object tipindeki nesnelere herhangi bir tipteki değişkenin içeriği aktarılabilir. Nasıl ki String sınıfının örneğini alırken "string" anahtar kelimesinden yararlanılıyorsa Object tipinde nesne hazırlanırken "object" anahtar kelimesinden yararlanılabilir. object anahtar kelimesi ile değişken tanımlandığında derleyici geri planda Object sınıfının örneğini hazırlamaktadır. Yani, "Object" bir sınıf iken "object" bir anahtar kelimedir. object değişkenlere istenen tipteki bilgiler aktarılabildiği için aşağıda verilen satırlar hataya neden olmaz.

```
object sayi = 123.456;
sayi = "Galatasaray";  // Hata meydana gelmez
```

Object değişkenler, şimdiye kadar hakkında bilgi verilen int, float ve byte gibi değişkenlerin aksine belleğin **Heap** adı verilen kısmında tutuluyorlar. Başka bir deyişle int, float, double, char ve byte gibi değişkenler belleğin **Stack** adı verilen kısmında tutulan **Value Type** iken object değişkenler belleğin Heap bölgesinde tutulan **Reference Type**'dır.

int, float ve byte gibi stack'te tutulan bir değişkenin (Value Type) içeriğinin Heap'te tutulan **object** tipli bir değişkene aktarılması işlemine **Boxing** (kutulama) adı verilmektedir. Aşağıda verilen kodda Stack'te tutulan int tipindeki değişkenin içeriği Heap'te tutulan object değişkene aktarılmaktadır.

```
int sayi = 12345;
object obje = sayi; // Boxing işlemi
```



Belleğin **Stack** adı verilen kısmında tutulan bool, btye, sbyte, char, decimal, double, long, ulong, short, ushort, int, uint ve float değişkenlere **Değer tipleri** yani **Value Type** adı verilmektedir. Belleğin **Heap** adı verilen kısmında saklanan Class, Object, String ve Delegate tipindeki değişkenler **Reference Type** olarak adlandırılmaktadır. Bu ayrım .NET programcılığında çok önemli olduğu için sıklıkla vurgulanmaktadır.

Bazen yukarıda yapılan işlemin tersine gerek duyulur; yani Reference Type olarak adlandırılan ve Heap'te tutulan değişkenlerin içerikleri stack'te tutulan **Value Type** değişkenlere aktarılmak istenir. Aşağıda verilen kodda bu tip bir işlem yapılmaktadır. Bu işleme **UnBoxing** denilmektedir. UnBoxing işlemi yapılırken object değişkenin içeriği hangi tipteki değişkene aktarılmak isteniyorsa o tipin adı parantezlerin içine alınıp object tipindeki değişkenin adının önüne yazılmalıdır.

```
object obje = 12345;
int sayi = (int)obje;  // UnBoxing işlemi
```

Bu koddaki object değişkenin içeriği int değişkenin kapasitesini aşmadığı için hata meydana gelmez ve bilgi kaybı olmaz. **object** değişkene büyükçe bir sayıyı aktarıp bu değişkenin içeriğini int değişkene aktarmak istemeniz halinde çalışma anında hata meydana gelebilir.

int, byte ve long gibi değer tiplerinin bellekte kaplayacakları yer miktarı önceden belli olduğu için bu değişkenler stack'te tutulur. Object gibi referans tiplerin içerecekleri bilginin bellekte kaplayacağı yer miktarı önceden bilinemeyeceği için Heap'te tutulurlar. Ancak bu bilginin Heap'ın neresinde olduğu konusunda bilgi Stack'te tutulur. Şimdi 2. bir int değişken tanımlayıp object değişkenin içeriğini bu değişkene aktarıp **UnBoxing** işleminin gerçekleşmesini sağlayacağız. Bu amaçla yukarıda verilen koda ekleme yaptık.

```
int yil = 2017;
object obje = yil;  // Boxing işlemi
int yeniYil = (int)obje; // Unboxing işlemi
```

Yukarıda Value Types değişkenlerin, .NET uygulaması tarafından kullanılan belleğin Stack adı verilen kısmında tutulduğu, Reference Types olarak görülen diğer tipleri temsil eden değişkenlerin ise belleğin Heap adı verilen kısmında tutulduğu ifade edildi. Bu tür cümlelere C# konulu her kaynakta rastlayabilirsiniz. Direk söyleyelim, bu ayrıma yerel değişkenler dahil değildir. Çünkü metot içinde tanımlanan değişkenler metodun kapsam alanı dışında çıkıldığında zaten erişilmez olacaklardır. Dolayısıyla metot içinde tanımlanan değişken reference types olsa bile .NET uygulamasına ayrılan belleğin stack kısmında tutulurlar.

GetType() Metodu

Object değişkene aktarılmış bilginin tipini öğrenmek istiyorsanız Object sınıfının **GetType** metodundan yararlanabilirsiz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
object sayi = 123.456;
object tip = sayi.GetType(); // System.Double
MessageBox.Show(tip.ToString());
```

Bu kodda object tipinde 2 değişken tanımlayıp "sayi" adını verdiğimiz değişkene sayısal bilgi aktardık. Bu değişkende kesirli bilgi varken **GetType**() metodu ile bu değişkenin içeriğinin tipini öğrendik. Yukarıda verilen kodu aşağıdaki gibi düzenleyip object değişkene false bilgisini aktarmış olsaydık **GetType**() metodu geriye **System.Boolean** bilgisini gönderirdi.

```
object durum = false;
object tip = durum.GetType();  // System.Boolean
```

GetType() metodu, burada kullanıldığı gibi yalnızca object tipindeki değişkenin hangi tipte bilgi içerdiği öğrenilirken kullanılmıyor. Örneğin mevcut bir değişkenin tipi öğrenilmek istendiğinde kullanılabilir. Aşağıda verilen kod işletildiğinde TextBox'a **System.Byte** yazılır.

```
byte ay = 11;
System.Type tip = ay.GetType();
textBox1.Text = tip.ToString(); // TextBox'a System.Byte yazılır.
```

Şimdi bu 3 satırda neler yapıldığını anlatacağız. Önce byte tipinde bir değişken tanımlayıp sayısal bilgi aktardık. Devamında **System.Type** tipinde bir değişken tanımladık. Çünkü GetType() metodu geriye **Type** tipinde bilgi göndermektedir. En son olarak Object sınıfı kökenli **GetType**() metodundan yararlanıp değişkenin içeriğinin tipini öğrenip **Type** nesnesine aktardık.

Burada kullanılan Type sınıfı çok sayıda özellik ve metoda sahiptir. Bu sınıfı bütün yönleri ile burada anlatmayı düşünmemekle birlikte konu üzerinde düşünmenizi sağlamak için aşağıda verilen kodu yazdık.

```
private void Form1_Click(object sender, EventArgs e)
{
 System.Type tip = typeof(string);
 textBox1.Text = tip.ToString();
 textBox2.Text = tip.Namespace;
 textBox3.Text = tip.Name;
}
```

Bu kod işletildiğinde aşağıdaki gibi bir sonuç alınır. **ToString**() metodu burada **base class**'ın adını geriye göndereceği için ilk TextBox'a **System.String** yazılır. Bu tipin yani string tipinin namespace'i **System** olduğu için 2. TextBox'a "System" yazılır.



GetType() metodu üzerinde biraz daha konuşmak ve **typeof** operatöründen söz edebilmek için aşağıdaki gibi kısıtlı özelliklere sahip bir sınıf hazırladık.

```
public class Takimlar
{
 public string Takim;
 public int Kurulus;
}
```

Şimdi ise bu sınıftan yola çıkıp new anahtar kelimesi ile bir nesne hazırlayacağız. Devamında GetType() metodunu kullanıp bu nesnenin tipini ve base class'ını öğreneceğiz.

```
private void button1_Click(object sender, EventArgs e)
{
 Takimlar t1 = new Takimlar();
 Type tip = t1.GetType();
 textBox1.Text = tip.Name;
 textBox2.Text = tip.BaseType.ToString();
}
```

Bu kod işletildiğinde ilk TextBox'a "Takimlar" ve 2. TextBox'a ise "System.Object" yazılır. Bu bilgileri elde ederken GetType() metodu yerine **typeof** operatörünü kullanabilirdik. Typeof operatörüne daha çok Type nesnesi hazırlanmak istediğinde başvurulmaktadır.

```
private void button1_Click(object sender, EventArgs e)
{
 Type tip = typeof(Takimlar);
 textBox1.Text = tip.Name;
 textBox2.Text = tip.BaseType.ToString();
}
```

var Anahtar Kelimesi İle Değişken Tanımlamak

C# 3.0'dan itibaren değişken tanımlama tekniğine ekleme yapılıp **var** anahtar kelimesi ile değişkenin tipi işaret edilmeden tanımlama yapma imkanı programcılara verildi. Var ile tanımlanan değişkenlere **Implicitly Typed Local Variables** adı verilmektedir.

Normalde C# kodu dahilinde değişken tanımlanmak istendiği zaman önce değişkenin tipi, sonra değişkene verilmek istenen ad yazılmaktadır. var anahtar kelimesi ile tanımlanan değişkenin tipi ise değişkene aktarılan bilgiye göre derleyici tarafından derleme anında seçilmektedir. Var ile değişkenlerin nasıl tanımlandığını aşağıda görebilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 var str = "Fenerbahçe";
}
```

Bu satırda **var** anahtar kelimesi ile tipi başlangıçta belli olmayan bir değişken tanımladık. Var ile yapılan tanımlama satırında değişkene string bilgi aktardığımız için bu andan itibaren bu değişken string tipte olur ve string tipte olmayan bilgilerin aktarılmasına izin verilmez. Bu nedenle aşağıdaki gibi bir satır hataya neden olur. Çünkü C# derleyicisi "var" anahtar kelimesi ile tanımlanan değişkenin tipinin ilk aktarılan bilgi ile aynı olmasını istemektedir.

```
private void button1_Click(object sender, EventArgs e)
{
 var str = "Fenerbahçe";
 str = 2017; // hata meydana gelir
}
```

Aşağıda verilen değişken tanımlama satırında değişken sayısal bilgi aktardığımız için derleme sırasında bu değişkenin tipi int olarak ayarlanır. Var anahtar kelimesiyle ilgili diğer ayrıntılardan daha sonra söz edilecektir.

```
private void button1_Click(object sender, EventArgs e)
{
 var yil = 2017;
}
```

Char Değişkenler

Yukarıdaki sayfalarda C# kodu yazarken değişkenlerin nasıl tanımlandığı, geçerlilik alanlarının nasıl ayarlandığından söz edildi. Şimdi sırada değişken tanımlarken sıklıkla kullanacağımız built-in tipler (char, int, byte, long string, double) hakkında konuşmak var.

Tek karakterlik bilgileri bellekte tutma gereği duyulduğunda **char** tipinden yararlanarak değişken tanımlanabilir. Char değişkenlerde ASCII ve **Unicode** karakter kümesindeki karakterlerden birisi tutulabilir. Char değişkenler bellekte 2 byte yer kaplamaktadır. char değişkenlere aktarılan karakterler tek tırnak(' ') içine alınmalıdır.

```
char karakter = 'A';
karakter = "A"; // Hata oluşur.
```

char değişkene aktarılmak istenen bilgi birden fazla karaktere sahipse hata meydana gelir. C# derleyicisi birden fazla karakterden meydana gelen bilginin char değişkene aktarılmasına izin vermez. String bilgi tek karakterden meydana geliyorsa aşağıda yapıldığı gibi dönüştürülüp char değişkene aktarılabilinir. String bilgiler çift tırnak içine alınmaktadır.

```
char cevap = Convert.ToChar("E");
```

char değişkene aktarmak istediğiniz karakterin Unicode veya ASCII kodunu biliyorsanız dönüştürme yaparak aktarabilirsiniz. Aşağıda verilen metot işletildiğinde "cevap" değişkenine "A" harfi aktarılır. Çünkü 'A' harfinin ASCII kodu 65'tir.

```
private void Form1_Click(object sender, EventArgs e)
{
 char cevap = Convert.ToChar(65);
 textBox1.Text = cevap.ToString();
}
```

Bu kodda char değişkenin içeriğini TextBox'ta görüntüleyebilmek için **ToString()** metodu ile dönüştürdük. 65 sayısını char tipine dönüştürürken burada yapıldığı gibi **Convert** sınıfını kullanmak yerine dönüştürülecek bilginin önüne aşağıdaki gibi ilgili tipi yazıp açık dönüştürme yapılabilir. Bu satırlarda 65 sayısının önüne parantez içine **char** yazıldığı için 65 sayısı char tipine dönüştürüp değişkene öyle aktarılır.

```
char Cevap = (char) 65;  // Cevap A olur.
textBox1.Text = Cevap.ToString();
```

Enter veya geriye silme tuşunu temsil eden kodu tırnak içine yazıp char değişkene aktarmak mümkün olmadığı için ilgili ASCII kodlarını char değişkenlere aynı şekilde aktarabilirsiniz.

```
char Tus = (char)13;
```

Şimdi örnek olması için bazı Ascii kodlarını elde edip bir ListBox'ta görüntüleyeceğiz. Bu amaçla forma bir ListBox yerleştirip aşağıda verilen kodu hazırladık.

```
string Str;
for (int i=32; i<=255; i++)
{
 Str = i.ToString() + " = " + (char)i;
 listBox1.Items.Add(Str);
}
```

char değişkene aktarmak istediğiniz karakterin **Unicode** standardına göre kodunu biliyorsanız bu kodun önüne "u" harfini yazmalısınız. "A" harfinin Unicode standardına göre kodu "0041" olduğu için aşağıda verilen kod işletilirse TextBox'a "A" aktarılır. Benzer şekilde "B" harfinin unicode'u \u0042'dir.

```
char Cevap = '\u0041';
textBox1.Text = Cevap.ToString();
```

char değişkenlerde saklanan karakterin ASCII kodunu öğrenmek istiyorsanız **Convert** sınıfının **ToByte()** metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Bu satırlar bir metoda yazılıp işletildiğinde **sayi** değişkenin içeriği 66 olur.

```
char karakter = 'B';
byte sayi = Convert.ToByte(karakter);   // sayi 66 olur.
textBox1.Text = sayi.ToString();
```

char değişkenin rakam mı yoksa harf mi içerdiğini öğrenmek istiyorsanız **Char** sınıfının **IsDigit**() ve **IsLetter**() metotlarından yararlanabilirsiniz. IsLetter() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 char karakter = 'A';
 bool sonuc = Char.IsLetter(karakter);
 if (sonuc == true)
   MessageBox.Show("Değişken harf içeriyor");
}
```

IsLetter() metoduna parametre olarak verilen char değişken harf içeriyorsa geriye true, rakam veya noktalama işareti içeriyorsa false gönderir. Benzer şekilde **IsDigit**() metodu parametre olarak aldığı char değişkenin içeriği rakam ise geriye true göndermektedir.

Char değişkenler hakkında bilgi edinirken **Char** sınıfının **IsLower**(), **IsUpper**() ve **IsSymbol**() metotlarından yararlanabilirsiniz. char tipindeki değişkenin içerdiği harfin küçük olup olmadığı IsLower() metodu ile öğrenilebilinir.

Char sınıfının **IsLetterOrDigit**() metodu ise kendisine parametre olarak verilen bilgi rakam veya harf ise geriye true göndermektedir. **IsSymbol**() metodu söz konusu karakterin $, € ve £ gibi bir simge olup olmadığı araştırılmak istendiği zaman kullanılmaktadır. Char değişkenin içerdiği karakteri büyük harfe çevirmek istiyorsanız Chat sınıfının **ToUpper**(), küçük harfe çevirirken **ToLower**() metodunu kullanabilirsiniz.

String Değişkenler

Birden fazla karakteri bellekte tutmak için **string** değişkenlerden yararlanılır. String değişkenlere istenilen uzunlukta karaktersel bilgi aktarılabilinir. String değişkenler **Reference Type**'ler olduklarından CLR tarafından kontrol edilen belleğin **Heap** adı verilen kısmında tutulmaktadır. Yukarıda ifade edildiği gibi bu ayrıma metot içinde tanımlanan yerel değişkenler tabi değildir.

String değişkenlerin içeriklerini artı(+) operatörü ile birleştirebilirsiniz. Aşağıda verilen metotta önce 3 string değişken tanımladık. Devamında bu değişkenlerden ilk ikisine bilgi aktardık. En son olarak bu 2 değişkenin içeriğini birleştirip 3. değişkene aktardık.

```
private void Form1_Click(object sender, EventArgs e)
{
 string Ad, Soyad, Ad_soyad;
 Ad = "Ömer ";
 Soyad = "Onan";
 Ad_soyad = Ad + Soyad;
 textBox1.Text = Ad_soyad;
}
```

String değişkenleri tanımlarken aynı zamanda bir karakterin birden fazla kopyasının aktarılmasını istiyorsanız String sınıfının yapıcı metodunun 2 parametreli sürümünü kullanabilirsiniz.

```
string str;
str = new String('-', 25);
```

String değişkenler gerçekte char tipinde bilgi içeren dizi değişkenler oldukları için String bilginin istenilen karakteri char değişkene aktarılabilir. Aşağıda verilen satırda string değişken adından sonra köşeli parantezin içine sıra noyu yazıp istenilen karakteri elde edebilmenin nedeni String sınıfının indeksleyiciye sahip olmasıdır.

```
string str = "Beşiktaş";
char kar = str[0];  // kar değişkeni B harfini içerir.
```

Ancak bu işlemin tersi yapılamaz. Yani string değişken tanımlanıp bilgi aktarıldıktan sonra içeriğinin herhangi bir kısmı indeksleyici ile değiştirilemez. Bu nedenle aşağıda verilen kod hataya neden olur. Çünkü String sınıfının indeksleyicisi yalnızca **get** bloğuna sahiptir.

```
string Str = "Beşiktaş";
Str[0] = 'F'; // Hata meydana gelir…
```

Yukarıda belirtildiği gibi string değişkenler **System**'deki **String** sınıfından yararlanılarak tanımlanmakta ve bu sınıfa üye metot ve özellikleri kullanabilmektedir. String sınıfının diğer özellikleri ve metotları hakkında daha sonra ayrıntılı bilgi verilecektir.

String değişkenler referans tipler olup belleğin **Heap** adlı kısmında tutulsalar bile kullanım şekilleri bakımından int ve double gibi değer tipindeki değişkenlerden farklı değildir. Örneğin aşağıda verilen 2 satırda sizin için yeni bir şey yoktur.

```
int yil = 2017;
string takim = "fener";
```

Bu satırlar işletildiğinde belleğin stack adı verilen kısmında **int** tipindeki değişken için 4 byte yer ayrılır ve 2017 sayısı sağ taraftan başlamak üzere bu byte'lara yerleştirilir. int değişkenler bu sayıdan daha büyük değerleri saklayabildikleri için ayrılan yerin baş tarafındaki çok sayıda byte 0 ile doldurulur. Sonra çalışma anında bir şekilde "yil" adı verilen bu int değişkenin içeriği değiştirildiğinde bellekte ayrılan yerdeki mevcut sayı silinip yenisi yazılır.

Aynı durum string değişkenler için geçerli değildir. Çünkü string değişkenler sabit uzunluklu değildir ve belleğin Heap kısmında tutulmaktadır. Bu nedenle yukarıda verilen 2 satıra aşağıdaki gibi bir satır eklendiğinde geri planda değişkenin son uzunluğuna uygun ayrı bir yer ayrılıp yeni içerik burada saklanılır. "fener" bilgisini saklamak için yapılan değişken tanımlama işlemi sonucu bellekte ayrılan yer bir süre sonra kendiliğinden silinir.

```
string takim = "Fener";
this.Text = takim;
takim = "Fener" + "bahçe";
```

Mevcut string bir değişkenin içeriği değiştirildiği zaman bu değişkenin belleğin başka bir yerine otomatik olarak taşındığına dikkatinizi çekmek için aşağıda verilen kodu yazdık.

```
private void Form1_Click(object sender, EventArgs e)
{
 string takim = "Şampiyon Fenerbahçe";
 takim.Replace("Fenerbaçe", "Galatasaray");
 this.Text = takim;
}
```

Bu kod hatasız olarak çalışır ancak beklenen sonucu vermez. Burada kullanılan **Replace**() metodu kendisine 1. parametre olarak verilen string bilgiyi 2. parametrede verilen bilgi ile değiştirir. Yani beklentimiz burada formun başlığına "Şampiyon Galatasaray" yazılmasını sağlamaktı. Ne var ki bu kod işletildiğinde formun başlığına yine "Şampiyon Fenerbahçe" yazılır. Çünkü Replace() metodu "takim" değişkenin içeriğini değiştirdiğinde bu değişken belleğin başka bir yerine taşınır. Belleğin yeni adresi ile "takim" değişkeni arasında henüz bir bağlantı olmadığı için bu değişken ilk tanımlandığı bellek bölgesindeki bilgiyi tutmayı sürdürür. Replace işleminin sonucunu almak istiyorsanız replace işlemi sonucu elde edilen string bilginin adresini "takim" değişkenine aşağıdaki gibi aktarmanız gerekir.

```
string takim = "Şampiyon Fenerbahçe";
takim = takim.Replace("Fenerbahçe", "Galatasaray");
this.Text = takim;
```

Bu sırada "Şampiyon Galatasaray" bilgisini saklayan "takim" değişkeni artık belleğin başka bir yerinde yaşamaktadır. Ya içeriğinde "Şampiyon Fenerbahçe" bilgisini içeren değişken ne oldu diye soranlara cevabımız şudur: O bellek bölgesi yani içinde "Şampiyon Fenerbahçe" bilgisini içeren bellek bölgesi postanede adresi silinmiş bir mektup gibidir.

Escape Karakterler

String değişkenlere bazen özel işlevi olup **Escape karakter** adı verilen karakterler aktarılmak istenir. Bu karakterlerin önüne "\" işareti konulmaktadır. Örneğin C# kodu içinde "C:" sürücüsündeki "Belgelerim" klasöründeki "Adres.dat" dosyası ile ilgilenildiğinde ilgili değişkene "C:\\Belgelerim\\Adres.dat" aktarılmalıdır. Bu konuda bilgi vermek için aşağıda verilen kodu hazırladık. Burada klasörleri birbirinden ayırmak amacıyla kullandığımız "\" özel bir karakter olduğu için önüne ikinci bir "\" konulması gerekti.

```
private void Form1_Click(object sender, EventArgs e)
{
 string yol = "C:\\Belgelerim\\Adres.dat";
 textBox1.Text=yol; //TextBox'a 'C:\Belgelerim\Adres.dat' yazılır
}
```

Escape karakterleri bu şekilde string bilgiye dahil etmek yerine "@" işareti (verbatim) ile söz konusu bilginin escape karakter içerdiği ve bu karakterlerin escape karakter olarak değerlendirilmesini istemediğiniz konusunda C# derleyicisini bilgilendirebiliriz. Bu karakterin nasıl kullanıldığını aşağıda görebilirsiniz.

```
string yol = @"C:\Belgelerim\Adres.dat";
textBox1.Text = yol;
```

C# kodu içinde string bilginin başına "@" karakteri konulmuşsa bu string bilgiye dayanıklı (verbatim) string denilmektedir. Dayanıklı string'ler birden fazla satıra bölünebilir ve bu string'lerin içerdikleri ters bölü ve diğer escape karakterler özel anlam ifade etmez.

C# programlama dilinde "\" dışında çok sayıda escape karakter bulunmaktadır. Örneğin "\n" karakteri satır başı yapmaya ve "\r" ise paragraf başı yapmaya yaramaktadır. Bu 2 Escape karakteri birlikte kullanarak TextBox'ın içeriğini birden fazla satıra bölebilirsiniz. Bu karakterlerin nasıl kullanıldığını göstermek için aşağıda verilen kodu hazırladık.

```
string firma = "Vatan \r\nBilgisayar";
textBox1.Text = firma;
```

Bu kod işletildiğinde 2 kelimeden meydana gelen string bilgi 2 satıra bölünüp öyle TextBox'a aktarılır. Ancak bu kodun işlevsel olabilmesi için TextBox'ın **MultiLine** özelliği bu sırada **true** olmalıdır. String bilgi içinde "\" karakterine yer verip yanlış escape karakteri kullanırsanız hata meydana gelir. Örneğin boşluk oluşturmak için kullanılan "\0" yerine "\O" kullanırsanız hata meydana gelir. C# derleyicisi "\0" karakterinden sonrasını silmektedir. Bu nedenle aşağıda verilen kod işletildiğinde TextBox'a yalnızca "Vatan" bilgisi aktarılır.

```
string firma = "Vatan \0\nBilgisayar";
textBox1.Text = firma;
```

Bool Değişkenler

Bool değişkenler bellekte 2 byte yer kaplarlar ve **true** ve **false** değerlerini alabilirler. Doğru ve yanlış gibi iki değer alabilen değişkenlere gerek duyulduğunda bool tipinden yararlanılır.

```
bool evetHayir;
bool dogruYanlis;
```

Bu tip değişkenlerin içeriği sayısal değişkenlere aktarıldığı zaman false yerine 0 ve true yerine 1 aktarılır. Aşağıda verilen kodda birisi bool olmak üzere 2 değişken tanımladık ve bool değişkenin içeriğini **int** tipindeki değişkene aktardık.

```
bool durum = false;
int sonuc = Convert.ToInt32(durum);
textBox1.Text = sonuc.ToString(); // TextBox'a 0 yazılır.
```

bool bilgilerin nasıl kullanıldığına örnek olması için "Form1" sınıfının Click olayını temsil eden metodu aşağıdaki gibi düzenledik.

```
private void Form1_Click(object sender, EventArgs e)
{
 string final = textBox1.Text;
 int final_not = Convert.ToInt32(final);
 bool sonuc = final_not > 50;
 if (sonuc)
   MessageBox.Show("Geçer not");
 else
   MessageBox.Show("Zayıf not");
}
```

Byte, Integer, Short ve Long Değişkenler

Tamsayı bilgileri bellekte tutmak için değişken tanımlarken **byte**, **int**, **short**, **long**, **uint**, **short**, **ushort** ve **ulong** tipleri kullanılmaktadır. Başında "u" harfi olmayan tipteki değişkenler hem pozitif hem de negatif sayıları tutabilirler. Bunu yaparken en soldaki bitten yararlanıyorlar. En soldaki bit 0 ise değişkenin içeriği pozitif, 1 ise içeriği negatiftir.

long değişkenler bellekte 8 byte, int'ler 4 byte, short değişkenler ise 2 byte yer kaplamaktadır. **short** değişkenlere -32768 ile +32767 arasında değişebilen değer aktarılabilinirken **byte** değişkenlere 0 ile 255 arasında değişen değer aktarılabilinir. **sbyte** tipindeki değişkenlerde ise -128 ile 127 arasında değişen bilgiler tutulabilmektedir.

int'lere -2.147.483.648 ile +2.147.483.647, **long**'lara ise -9.223.372.036.854.775.808 ile +9.223.372.036.854.775.807 arasında değişen bilgiler aktarılabilinir. Başka bir deyişle byte değişkenler 2^8 büyüklüğünde sayıları saklayabilirken int'lerde 2^{32} büyüklüğündeki sayıları tutabilirler. Adında "u" harfi olan işaretsiz tam sayı değişkenlere eksi değerlerin aktarılmasına izin verilmez. Örneğin C# derleyicisi aşağıda verilen satırda hata olduğunu rapor eder.

```
uint sayi = -10; // hata meydana gelir.
```

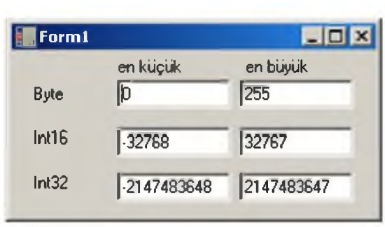
İşaretsiz tamsayı değişkenlere sabit bilgileri aktarırken bilginin sonuna "unsigned" anlamında "u" harfini eklemek iyi bir alışkanlıktır. Burada bilgi aktarmak istediğimiz işaretsiz tamsayı değişken long tipinde olsaydı "u" harfinden başka ayrıca "l" harfine de yer verirdik.

```
ulong sayi = 100ul;
```

Değişkenlerin alabilecekleri en büyük veya en küçük değerleri öğrenmek istiyorsanız söz konusu değişkeni temsil eden yapının **MaxValue** ve **MinValue** özelliklerine bakabilirsiniz. Bu özelliklerin nasıl kullanıldığını göstermek için forma 6 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
textBox1.Text = (System.Byte.MinValue).ToString();
textBox2.Text = (System.Byte.MaxValue).ToString();
textBox3.Text = (System.Int16.MinValue).ToString();
textBox4.Text = (System.Int16.MaxValue).ToString();
textBox5.Text = (System.Int32.MinValue).ToString();
textBox6.Text = (System.Int32.MaxValue).ToString();
```

Bu satırlar bir metoda yazılıp işletildiğinde byte, Int16 ve Int32 tipindeki değişkenlerin **MinValue** ve **MaxValue** özelliklerinin içerikleri TextBox'lara aktarılır.



Tamsayı bilgileri benzer tipteki değişkenlere aktarırken dikkatli olmak gerekir. Örneğin **int** tipindeki değişkene byte değişkenin içeriğini aktardığınızda C# derleyicisi otomatik olarak dönüştürme yapar bilgi ve kaybı olmaz.

```
byte b = 225;
int i = b;     // Bu tip dönüştürmeye kapalı dönüştürme deniliyor.
```

int değişkenin içeriği, byte veya short değişkene aktarıldığında bilgi kaybı ihtimaline karşı derleyici hata mesajı verir. Bu nedenle aşağıda işaret edilen satır hataya neden olur.

```
int i = 255;
byte b;
b = i; // hata meydana gelir.
```

Burada int değişkene aktarılan bilgi byte değişkene sığacak büyüklükte olmasına rağmen derleyici bu işleme izin vermez. Burada yapılması gereken, aktarma sırasında cast operatörü ile açık dönüştürme yapılmasıdır. Bu dönüştürmenin nasıl yapıldığını aşağıda görebilirsiniz.

```
int i = 255;
byte b = (byte)i; // Bu tip dönüştürmeye açık dönüştürme deniliyor
```

Bu kod işletildiğinde hata ve bilgi kaybı olmadan int değişkenin içeriği byte değişkene aktarılır. Aktarma sırasında ilgili değişkenin içeriği hangi tipteki değişkene aktarılıyorsa hedef değişkenin tipi parantez içine alınarak içeriği aktarılan değişkenin önüne yazılmalıdır.

Bu örnekte int tipindeki değişkende 255'ten daha büyük bir sayı tutuluyor olsaydı C# derleyicisi hata mesajı vermezdi ama bilgi kaybı olurdu. Bu gibi durumlarda karşılaşılabilinecek bilgi kayıplarının önüne geçmek için **checked** anahtar kelimesinden yararlanabilirsiniz. Bu konuda bilgi vermek için aşağıda verilen basit örneği hazırladık.

```
int i = 65535;
byte b = (byte)i;
textBox1.Text = b.ToString(); // TextBox'a 255 aktarılır
```

Bu kod işletildiğinde hata meydana gelmez ve TextBox'a 255 aktarılır. Çünkü integer bilgiler bellekte 32 bit yer kaplarken byte değişkenler 8 bitliktir. Bu durumda 32 bitlik int değişkenin içeriği byte değişkene aktarılırsa integer değişkenin son 24 bitinin içeriği kaybolur. Bunun önüne geçmek için **checked** anahtar kelimesi aşağıdaki gibi kullanılabilir.

```
int i = 65535;
byte b;
checked
 {
  b = (byte)i;
 }
textBox1.Text = b.ToString();
```

Bu şartlarda proje hatasızca derlenir ancak programın işletimi integer değişkenin içeriğinin byte değişkene aktarıldığı satıra geldiğinde hata meydana gelir ve çalışması kırılır. Benzer şekilde aşağıda verilen kod hatasızca derlenir ama sonuç beklendiği gibi çıkmaz.

```
byte sayi = 255;
sayi += 20;
textBox1.Text = sayi.ToString();  // TextBox'a 19 aktarılır.
```

Şimdi ise işaretsiz bir int değişkenin içeriğini işaretli bir int değişken ile toplayıp çıkan sonucu yine işaretli bir int değişkene aktaracağız. Bu amaçla hazırlayıp aşağıda verdiğimiz kod işletildiğinde hata meydana gelir.

```
private void Form1_Click(object sender, EventArgs e)
{
 uint sayi1 = 12345;
 int sayi2 = 54321;
 int sonuc = sayi1 + sayi2; // Hata meydana gelir.
}
```

uint ile **int** değişkenin toplanması sonucu çıkan sonuç ancak **long** değişkenlere aktarılabilinir. Bu kodun hata vermesini önlemek istiyorsanız açık dönüştürme yapabilirsiniz. Aşağıda yapıldığı gibi yalnızca **uint** değişkeni dönüştürmek yeterlidir.

```
uint sayi1 = 12345;
int sayi2 = 54321;
int sonuc = (int)sayi1 + sayi2;
```

Şimdiye kadar anlatılanlara göre sayısal bilgilerin değişkenlere aktarılması ve dönüştürülmesi, üzerinde önemle durulması gereken bir konudur. Bu konudaki diğer ayrıntıları ortaya çıkarmak için aşağıda verilen satırları yazdık.

```
ushort us = 4567;
int i = 532;
int s = us + i;
```

Burada toplama işlemi yapılmadan önce **ushort** tipteki sayı 32 bitlik işaretli tam sayıya dönüştürülür. Çünkü toplama işlemi ancak aynı tipteki iki sayı arasında yapılabilir. Bu satırlar aşağıdaki gibi değiştirilip toplama işleminden sonra açık dönüştürme yapılabilir.

```
uint us = 4567;
int i = 532;
int s = (int) (i + us);
```

Burada toplama işlem sonucu bulunan değeri açık dönüştürmeye tabi tuttuk. Bazen de her değişkeni ayrı ayrı açık dönüştürmeye tabi tutmak isteyebilirsiniz. Aşağıda verilen satırlarda toplama işlemi yapılmadan önce uint tipindeki sayı işaretli tam sayıya dönüştürüldü.

```
uint us = 4567;
int i = 532;
int s = i + (int)us;
```

Şimdi byte, integer ve long tipindeki değişkenlerle ilgili bir ayrıntıdan tekrar söz edeceğiz. C# derleyicisi sayısal tipteki herhangi bir değişkene aktarılmak üzere aktarma operatörünün sağına yazılan sayı tamsayı ise int, ondalık nokta içeriyorsa double tipte olduğunu varsaymak-

tadır. Örneğin aşağıda verilen satırda float değişkene double bilgi aktarılmak istendiği için hata meydana gelir. İlk bakışta eşitliğin sağına yazılan sayının float tipe uygun olduğu düşünülür.

```
float ivme = 9.81; // hata meydana gelir.
```

Gerçekte "=" operatörünün sağına yazılan bilginin double olduğu varsayılmaktadır. Burada karşılaşılan hatayı gidermenin yolu, değişkene aktarılmak istenen bilginin float tipte olduğunu sonuna "f" harfini ekleyerek işaret etmektir.

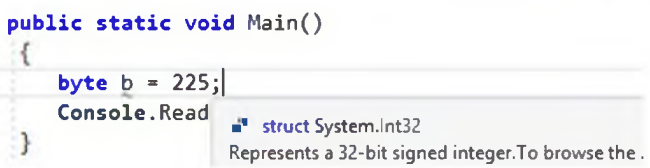
```
float ivme = 9.81f;
```

Konu tamsayı bilgiler olduğunda varsayılan tip ingeter'dır. Bu nedenle aşağıda verilen satırda aktarma operatörünün sağına yazılan tamsayı bilginin int tipinde olduğu varsayılır.

```
long sayi = 12345;
```

int tipindeki tamsayı bilgiler long değişkenlere kayıpsız aktarılabildiği için bu bilginin sonuna long anlamında "l" veya "L" konulmazsa sorun çıkmaz. Bazı programcıların long değişkenlere aktardıkları bilginin sonuna "L" veya "l" eklemek gibi bir alışkanları vardır.

Elinizdeki tamsayı bilgiyi byte tipindeki değişkene aktarıyorsanız durum biraz değişiktir. Çünkü byte değişkene aktarmak istediğiniz tamsayı bilginin sonuna byte anlamında "b" veya "B" harfini eklemeyi C# derleyicisi desteklemiyor. Bu nedenle aşağıda verilen satırda ilk bakışta C# derleyicisinin hata rapor edeceğini düşünebilirsiniz. Çünkü byte değişkene int bilgi aktarılmak istenmektedir. Aktarma operatörünün sağına yazılan sayısal bilginin int yani Int32 tipinde olduğu varsayımını size göstermek için Visual Studio'dan yararlandık.



Fare işaretini eşitliğin sağ tarafında byte değişkene aktarılmak istenen tam sayı bilginin üzerinde görünce Visual Studio bu bilginin Int32 tipinde olduğunu bu şekilde işaret etti. Buna rağmen C# derleyicisinin bu satıra bir itirazı olmaz. Burada byte değişkene aktarılma istenen tamsayı bilgi byte değişkenin kapasitesini aşıyor olsaydı hata meydana gelirdi.

Dönüştürme işlemlerine örnek olması için şimdi 10 tabanlı bir sayıyı 16 tabanlı sayıya dönüştüreceğiz. Aşağıda verilen kod bir metoda yazılıp işletilirse byte değişken içeriği 16 tabanlı olarak ifade edilip TextBox'a yazılır.

```
byte Sayi = 255;
for (int i=7; i>=0; i--)
{
 int j = (Byte)(Sayi >> i & 1);
 textBox1.Text += j.ToString();
}
```

Elinizdeki 16 tabanlı sayıyı 10 tabanlı olarak ifade etmek isteyebilirsiniz. Dönüştürmek istediğiniz sayının 16 tabanlı olduğunu işaret etmek için başına "0x" konulmaktadır. Aşağıda verilen metot işletildiği zaman TextBox'a 65535 aktarılır.

```
private void Form1_Click(object sender, EventArgs e)
{
 int sayi = 0xFFFF;
 textBox1.Text = sayi.ToString();
}
```

Yukarıdaki sayfalarda string değişkenin içeriğini char tipindeki dizi değişkene aktarmıştık. Benzer şekilde string değişkenin içeriğini byte dizisine aktarabilirsiniz.

```
string str = "Fenerbahçe";
byte[] byte_dizi = new byte[10] ;
for (int i = 0; i < str.Length; i++)
{
 byte_dizi[i] = (byte) str[i];
 textBox1.Text += (char)byte_dizi[i];
}
```

Şimdi sırada byte değişkenlerle ilgili iki işlem yapmak var. String değişkenin içeriğini byte dizisine aktarırken **Encoding** sınıfının **GetBytes()** metodundan yararlanabilirsiniz. **Encoding** sınıfı **System.Text**'te bulunmaktadır. String değişkeni parametre olarak alan bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string str = "Fenerbahçe";
byte[] byte_dizi;
byte_dizi = Encoding.Unicode.GetBytes(str);
for (int i = 0; i < str.Length; i++)
{
 byte_dizi[i] = (byte) str[i];
 textBox1.Text += (char)byte_dizi[i];
}
```

Şimdi ise byte dizisini char dizisine aktaracağız. Bu amaçla **Encoding** nesnesi hazırladık ve Encoding sınıfının **GetChars()** metodu ile byte dizisini char dizisine aktardık.

```
Encoding kod = Encoding.ASCII;
byte[] byte_dizi;
char[] char_dizi;
byte_dizi = new byte[3];
byte_dizi[0] = 65;
byte_dizi[1] = 66;
byte_dizi[2] = 67;
char_dizi = kod.GetChars(byte_dizi);
textBox1.Text = Convert.ToString(char_dizi[1]);
```

Binary (2 Tabanlı) Sayılar

C# 7.0 kullanıyorsanız C# kodu içinde 2 tabanlı sayıları kullanıp üzerinde direk işlem yapabilirsiniz. Burada sözünü edeceğimiz ve **Binary Literals** adı verilen binary gösterim desteği

C# 7.0 ile geldi. İki tabanlı sayıların C# kodu içinde nasıl kullanıldığına örnek olması için aşağıda verilen kodu yazdık.

```
static void Main(string[] args)
{
 byte b1 = 0b10000000;
 byte b2 = 0b01111111;
 int sonuc = b1 + b2;
 Console.WriteLine(sonuc);
}
```

Verilen koddan fark etmiş olacağınız gibi binary bilgilerin başına "0b" eklenmektedir. C# kodu içinde binary bilgiler üzerinde işlem yaparken yine C# 7.0 ile gelen üçer hane ayırıcı karakterden (Digit Separator) yararlanmak isteyebilirsiniz. Üçer hane ayırıcı olarak "_" karakterinin nasıl kullanıldığını aşağıda görebilirsiniz. Tabi, üçer hane ayırıcı karakterin sadece binary bilgilere özel olmadığını tahmin edebilirsiniz.

```
static void Main(string[] args)
{
 byte b1 = 0b10_000_000;
 byte b2 = 0b01_111_111;
 int sonuc = b1 + b2;
 Console.WriteLine(sonuc);
}
```

float ve double Değişkenler

Bu bölümün başında verilen tabloda .NET Framework ile gelen "System.Int32" tipinde nesne hazırlarken "int" takma isminin kullanıldığı ifade edilmişti. Benzer şekilde "System.Int64" yapısının örneği alınırken "long" takma ismi kullanılabiliniyor. Bu takma adlar sayesinde hem değişken tanımlama kolaylaşmış hem de C veya C++ dilleriyle geçmişte çalışmış olanların alışkanlıkları korunmuş oluyor. Int32 yapısıyla ilgili bu hatırlatmayı yapıktan sonra gelelim asıl konumuz olan float ve double değişkenlere.

.NET Framework ile birlikte ondalıklı bilgilerle ilgili 2 yapı (struct) gelmektedir: "System.Double" ve "System.Single". Double yapısının örneği alınmak istendiğinde bu yapının takma adı olan "double" kullanılırken "System.Single" yapısına "float" takma adı verildiği için Single tipinde nesne hazırlanmak istendiğinde **float** anahtar kelimesi kullanılmaktadır. Çünkü C# programlama dilinin System.Single'ın C ve C++ karşılığı float olduğu için C# programcıları ondalık nokta içeren bilgiler için değişken tanımalarken 4 byte'ın yeterli olduğu durumlar için **float** anahtar kelimesinden yararlanıyorlar.

Bu nedenle ondalık nokta içeren sayısal bilgileri bellekte tutmak için **float** veya **double** tipinde değişkenler tanımlayabilirsiniz. Ondalık nokta içeren sayısal bilgi çok büyük veya çok küçük ise **double** tipi kullanılabilir. float değişkenler bellekte 4 byte yer kaplarken double değişkenler 8 byte yer işgal ederler. float değişkenlere aktarılan sayının sonuna "f" veya "F" harfini eklemek gerekiyor.

```
float sayi1 = 12345.6789f;
double sayi2 = 987.6543212;
```

Benzer şekilde double değişkenlere bilgi aktarırken bilginin sonuna "d" veya "D" harfini koyabilirsiniz. Float değişkenlerin aksine double değişkenlere aktarılan bilginin sonuna "d" harfi konulmadığı zaman hata meydana gelmemektedir.

Tam bu noktada C# programlama dilinin bir özelliğinden tekrar söz edeceğiz. Herhangi bir tipteki değişkene değer aktarmak üzere "=" operatörünün sağ tarafına yazılan bilgi ondalık nokta içermiyorsa integer, ondalık nokta içeriyorsa double olduğu varsayılıyor. Bu varsayımdandır ki aşağıda verilen satır hataya neden olur.

```
static void Main(string[] args)
{
 float ivme = 9.81; // hata meydana gelir.
}
```

Çünkü atama operatörünün sağına yazılan sayısal bilginin double olduğu varsayılmasına rağmen burada operatörün sol tarafında float tipte bir değişken vardır. Dolayısıyla float tipteki değişkene double tipte bilgi aktarılmayacağına göre bu satır hataya neden olur. Bu hatanın önerilen çözümü sayısal bilginin sonuna float anlamında "f" veya "F" harfini eklemektir. Bu hatayı engellemek için aşağıda verilen 2. satırdaki gibi açık dönüştürme yapılabilir.

```
float ivme1 = 9.81f;.
float ivme2 = (float)9.81;
```

Tamsayı bilgi içeren değişkenlerin içerikleri double değişkenlere aktarıldığı zaman herhangi bir hata meydana gelmez ve bilgi kaybı olmaz. Aşağıda verilen kodda **int** tipindeki değişkenin içeriğini double değişkene aktardık. Bu şekilde yapılan aktarımda **Implicit** adı verilen kapalı bir dönüştürme yapılır.

```
int i = 123456;
double sayi = i; // kapalı dönüştürme
```

Ancak tamsayı bilgi içerebilen değişkenlere float veya double değişkenin içeriği aktarıldığı zaman bilgi kaybı olabileceği için C# derleyicisi buna izin vermez. Bu nedenle aşağıda verilen kod hataya neden olur.

```
double sayi = 1234.876;
int i = sayi; // hataya neden olur
```

double veya float değişkenin içeriğinin virgülden sonrası ile ilgilenmiyorsanız açık dönüştürme yaparak double değişkenin içeriğini int gibi tamsayı değişkenlere aktarabilirsiniz. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Bu kod işletildiğinde double değişkenin içeriğinin tam kısmı **int** değişkene aktarılır.

```
double sayi = 1234.876;
int i = (int)sayi; // i 1234 olur
textBox1.Text = Convert.ToString(i);
```

Float ve double değişkenler virgülden sonra belli sayıda haneyi tutabilmektedir. Float değişkene aktarılan sayısal bilgide virgülden sonra 8 hane varsa 8. hane yuvarlatılır. Bu konuyu netleştirmek için üzerinde çalıştığımız projenin formuna 2 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
float sayi1 = 1.2345678956f;
double sayi2 = 1.2345678956d;
textBox1.Text = sayi1.ToString();
textBox2.Text = sayi2.ToString();
```

Bu kodda önce float ve double değişkenlere virgülden sonra 10 haneye sahip birer sayı aktardık. Ardından bu değişkenlerin içeriklerini TextBox'lara aktardık. Float değişkenler toplam 7 haneyi gösterebildikleri için virgülden sonraki 6. haneden sonrası yuvarlatıldı. Double değişkenler ise 15 haneyi gösterebildikleri için herhangi bir yuvarlatma yapılmadı. Bu kodu bir metoda yazıp işletirseniz aşağıdaki gibi bir sonuç alırsınız.



Float ve double değişkenlere aktarılan sayıdaki ondalık noktayı kaldırırsak, başka bir deyişle bu değişkenlere büyük bir değeri aktarmış olsaydık, float 7. haneden sonrasını, double değişken ise 15. haneden sonrasını üslü olarak görüntülerdi. Float değişkene aktardığınız sayı 7'den daha fazla haneye sahipse sayının "E" formatında görüntülenmesini istemiyorsanız double veya decimal değişken kullanmalısınız.

.NET Framework ile gelen Single yapısı System'de yer almaktadır. Single yapısı diğer yapılar gibi **System.ValueType** tipinden türemiştir. ValueType tipinin de System.Object tipinden türediğini tahmin edebilirsiniz. Bir de Single yapısının IComparable, IFormattable ve IConvertible gibi çok sayıda arayüzü uyguladığını düşününce çok sayıda metot ve özelliğe sahip olabileceğini tahmin etmek zor olmaz.

DateTime Değişkenler

Tarih ve zaman bilgilerini bellekte tutmak istiyorsanız **DateTime** tipinde değişken tanımlamalısınız. DateTime tipindeki değişkenler bellekte 8 byte yer kaplamakta ve 1 Ocak 1 ile 31 Aralık 9999 arasında kalan tarih bilgilerini saklayabilmektedir. DateTime tipindeki değişkeni zaman bilgisini saklamak üzere tanımladıysanız 0:00:00 ile 23:59:59 arasında değişen değerler aktarabilirsiniz.

```
DateTime tarih;
DateTime saat;
```

DateTime tipinde değişken tanımlanırken kullanılan DateTime yapısı **System**'de bulunmaktadır.. DateTime değişkenlere tarih ve zaman bilgisi olarak değerlendirilebilinecek bilgiler Convert sınıfının **ToDataTime**() metodu ile dönüştürülebilmektedir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
DateTime Tarih = Convert.ToDateTime("10.12.2007");
DateTime Saat = Convert.ToDateTime("12:55");
```

DateTime değişkenlerin içeriklerini **Convert** sınıfının ilgili metotlarından yararlanarak aşağıda yapıldığı gibi **String** değişkenlere aktarabilirsiniz. Daha sonra tarih ve zaman bilgileri üzerinde yapılan işlemler hakkında bilgi verilecektir.

```
private void Form1_Click(object sender, EventArgs e)
{
 DateTime tarih = Convert.ToDateTime("10.12.2007");
 DateTime saat = Convert.ToDateTime("12:55");
 textBox1.Text = Convert.ToString(tarih);
 textBox2.Text = Convert.ToString(saat);
}
```

Bu kod işletildiğinde formdaki ilk TextBox'a tarih bilgisi ile birlikte zaman başlangıcı (00:00:00) aktarılır. 2. TextBox'a ise buradaki "saat" değişkenine aktarılan zaman bilgisi ve sistem tarihi yazılır. Bu nedenle aşağıda verilen ekran görüntüsü sizi şaşırtmasın.



String bilgileri tarih ve saate dönüştürürken DateTime sınıfının Parse() metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. DateTime sınıfının Parse() metodu 3 parametreye sahiptir. 2. parametre ile elde edilmek istenen tarih ve zaman bilgisinin formatı belirtilebilir.

```
DateTime Tarih;
Tarih = DateTime.Parse(textBox1.Text);
```

Decimal Değişkenler

Özellikle parasal bilgiler söz konusu olduğunda büyük değerleri saklayabilen **decimal** değişkenlere gerek duyulur. **Decimal** değişkenler bellekte 8 byte yer kaplamaktadır. Ayrı şekilde virgülden sonraki hane sayısı fazla olan sayılar decimal değişkenlerde saklanabilir.

Decimal değişkenler en fazla 29 haneye sahip sayıları tutabilirler. Başka bir anlatımla, decimal değişkenlerin hassasiyetleri oldukça iyi olmasına rağmen double değişkenler daha büyük sayıları tutabilirler.

```
decimal bilanco;
bilanco = 998200100200;
MessageBox.Show(bilanco.ToString());
```

Her ne kadar parasal bilgiler söz konusu olduğunda virgülden sonrası pek önemli olmasa bile decimal değişkenler ondalık nokta içerebilir. Decimal değişkene ondalık nokta içeren bilgi aktarılmak istendiğinde sayının sonuna "m" veya "M" harfi konulmalıdır.

```
decimal katsayi;
katsayi = 0.12345678m;
```

Ondalık nokta içeren bilgiye "m" harfi eklenmeden decimal değişkene aktarılmak istenirse hata mesajı alınır. Tamsayı bilgiler dönüştürme yapılmadan decimal değişkene aktarılabilir.

```
int sayi1, sayi2;
sayi1 = 10;
sayi2 = 100;
decimal sonuc = sayi1 * sayi2;
```

Bu işlemin tersine yani decimal değişkenin içeriğinin int değişkene aktarılmasına C# derleyicisi izin vermez. Decimal değişkenin içeriğini dönüştürüp ondan sonra integer değişkene aktarmak gerekir.

```
decimal sayi1, sayi2;
sayi1 = 10;
sayi2 = 100;
int sonuc = Convert.ToInt16(sayi1 * sayi2);
```

Integer tipteki bilgileri birbirine bölüp herhangi bir casting işlemi yapmadan decimal değişkene aktarmak mümkündür. Örneğin aşağıda verilen satırlar hataya neden olmaz.

```
int sayi1 = 100;
int sayi2 = 30;
decimal sonuc = sayi1/sayi2;  // sonuc 3 olur.
```

Bölme işlemi sonucu elde edilecek değerin kesirli olmasını istiyorsanız aşağıda yapıldığı gibi dönüştürme yapmanız gerekir. Çünkü daha önce işaret edildiği üzere "=" operatörünün sağına yazılan sayısal bilginin int tipinde olduğu varsayılmaktadır. Dolayısıyla burada "sayi1" ile "sayi2" birbirine bölündüğünde çıkan sonuç int yapılıp öyle eşitliğin sol tarafına aktarılır.

```
int sayi1 = 100;
int sayi2 = 30;
decimal sonuc = (decimal)sayi1/sayi2;  //sonuc 3.3333... olur
```

Benzer şekilde float değişkenlerle yapılan işlemlerin sonucunu aşağıda yapıldığı gibi decimal değişkenlere aktarmak mümkündür.

```
float sayi1 = 66.88f;
float sayi2 = 44.55f;
decimal sonuc = (decimal) (sayi1 * sayi2);
```

decimal tipindeki değişkenin içeriği int ve long gibi tamsayı değişkenlere aktarılırken sayının noktadan sonraki kısmı atılır. Elde edilen tam kısım, hedef türün sınırları içerisine sığmıyorsa **System.OverflowException** hatası meydana gelir.

Tip Dönüşümleri

Bazen bir değişkenin içeriği başka tipteki bir değişkene aktarılmak istenir. Diğer yandan bazı değişkenlere kendi tipinden farklı tipte bilgi aktarmak soruna neden olmaz. Örneğin aşağıda verilen kodda **Single** değişkene **int** değişkenin içeriğini doğrudan aktardık.

```
int sayi2 = 2017;
Single sayi1 = sayi2; // kapalı dönüştürme (implicit)
```



Tam bu noktada araya girip ek açıklama yapmak istiyoruz. Bu satırda kullanılan int, System.Int32 yapısının takma adı olan bir anahtar kelime iken, "Single", System'deki Single yapısıdır. Yukarıdaki sayfalarda işaret edildiği üzere System.Single yapısının takma adının "float" olduğunu biliyorsunuz.

Ama tersi bir işlem yapsaydık, başka bir deyişle **int** değişkene **Single** değişkenin içeriğini aktarmak isteseydik hata meydana gelirdi. Single değişkenlere kesirli sayı aktarılırken sayının sonuna float anlamında "f" harfini eklemek gerekmektedir.

```
Single sayi1 = 1234.543f;
int sayi2 = sayi1; // hata meydana gelir.
```

Tip dönüşümünün kontrolünüz altında yapılmasını istiyorsanız hangi tipe dönüştürme yapılacaksa o tipin adı parantezlerin arasına yazılmalıdır. Aşağıda verilen örnekte **int** değişkenin içeriği **byte** değişkene aktarılmakta ve bu sırada dönüştürme yapılmaktadır. Bu şekilde yapılan tip dönüştürmeye açık tip çevrimi(**explicit**) adı verilmektedir.

```
int sayi1 = 125;
byte sayi2 = (byte)sayi1; // açık dönüştürme
```

Bu örnek aktarma işlemi sırasında açık dönüştürme yapıldığı için derleme işlemi sırasında hata meydana gelmez. Ayrıca byte tipindeki değişkene aktarılmak istenen sayı 255'ten büyük olmadığı için bilgi kaybı olmaz. İçeriği byte değişkene aktarılan bu int değişkenin içeriği 255'ten büyük olsaydı bilgi kayıpları olurdu.

```
int sayi1 = 257;
byte sayi2 = (byte)sayi1; // sayi2 1 olur
```

Benzer dönüştürme işlemini **Single** değişkenin içeriğini **int** değişkene aktarırken yapabilirsiniz. Aşağıda verilen satırlar işletildiğinde Single değişkenin içeriği int değişkene aktarılır ama Single sayının virgülden sonrası kaybolur.

```
Single sayi1 = 1224.553f;
int sayi2 = (int)sayi1; // sayi2 1224 olur.
```

Bu şekilde yapılan dönüştürme işlemi sırasında bilgi kaybı olmasına rağmen C# derleyicisi herhangi bir hata rapor etmez. Bu gibi dönüştürme işlemleri sırasında meydana gelen bilgi kayıpları konusunda kullanıcıyı bilgilendirmek istiyorsanız **checked** deyiminden yararlanabilirsiniz. checked deyiminin nasıl kullanıldığını aşağıda görebilirsiniz.

```
int sayi1 = 257;
byte sayi2;
checked
 {
  sayi2 = (byte)sayi1;
 }
```

Bu kodda içeriği **byte** tipindeki değişkene aktarılacak int değişkenin içeriğinin 255'ten büyük olması nedeniyle bilgi kaybı olacağından **checked** deyimi devreye girer ve ekrana hata mesajı getirilip programın çalışması kırılır.

Yukarıdaki sayfalarda belirtildiği gibi **object** değişkenlere dönüştürme işlemine gerek duyulmadan herhangi tipteki bir değişkenin içeriği aktarılabilinmektedir. Ancak object tipli değişkenlerin içerikleri ne tür bilgi içerirlerse içersinler dönüştürülmeden int veya float gibi belleğin Stack adı verilen kısmında tutulan değişkenlere aktarılamaz. Bu nedenle aşağıda verilen satırlar hataya neden olur. Çünkü object değişkenler int, Single, byte gibi değişkenlerden farklı olarak belleğin **Heap** adı verilen kısmında saklanmaktadır.

```
object obje = 1234;
int sayi = obje;  // hata meydana gelir
```

Bu kodda int değişkene object değişkenin içeriği aktarılmaktadır. Bu işleminin başarılı olması için object değişkenin içeriğinin aktarılacağı değişkenin tipi aşağıda verildiği parantezin içine yazılıp açık dönüştürme yapılmalıdır. int değişkene içeriğini aktarmak istediğimiz object değişkende tamsayı bilgi olduğu için parantezin içine **int** yazdık.

```
object obje = 1234;
int sayi = (int)obje;
```

ToString() Metodu İle Dönüştürme Yapmak

Şimdiye kadar birkaç kez sayısal bilgileri TextBox'lara aktarmak için aşağıdaki gibi dönüştürme işlemleri yaptık. Bu satırlarda önce "number1" adında bir float ve "number2" adında **int** tipte 2 değişken tanımlayıp bilgi aktardık. Ardından bu değişkenlerin içeriklerini **ToString**() metodu ile string tipe dönüştürüp TextBox'lara aktardık.

```
float number1 = 2578.987f;
int number2 = 12345;
textBox1.Text = number1.ToString();
textBox2.Text = number2.ToString();
```

Kullanıcı tarafından TextBox'a yazılan bilgilerin int veya başka bir sayısal tipe dönüşüme uygun olup olmadığını öğrenmek isteyebilirsiniz. Gerekli tedbirler alınmazsa bu satırlar programın çalışmasının kırılmasına neden olabilirler.

Bu konuda bilgi vermek için forma bir TextBox ve bir düğme yerleştirip aşağıda verilen metodu hazırladık. TextBox'a bilgi girilip düğme tıklandığında TextBox'ta rakam olmayan karakter olup olmadığı "Uygun" adı verilen metot ile araştırılmaktadır.

```
private void Donustur_Click(object sender, EventArgs e)
{
 int i;
 if(Uygun(textBox1.Text))
   i = Convert.ToInt32(textBox1.Text);
 else
   MessageBox.Show("Uygun bilgi Girilmemiş");
}
```

"Uygun()" adını verdiğimiz metot geriye true gönderirse TextBox'ın içeriğinde yalnızca rakamlar var demektir ve Convert sınıfının ToInt32() metodu ile string bilgi int tipine dönüştürülür. Kullanıcı çalışma anında TextBox'a int tipine dönüştürülemeyecek bilgi girmişse yani bu bilginin içinde harfler varsa Uygun() metodu geriye false göndereceği için dönüştürme yapılamaz. Kendi hazırlayıp Uygun() adını verdiğimiz metot aşağıdadır.

```
public static bool Uygun(string Str)
{
 int sayi = Str.Length;
 char Kar;
 for (int i = 0; i < sayi; i++)
  {
   Kar = Str[i];
   if (!char.IsNumber(Kar))
     return false;
  }
 return true;
}
```

Burada yazdığımız metodun işlevi hemen anlaşılsın diye satır azaltma yoluna gitmedik. for yerine **foreach** döngüsünü kullanıp daha az satırla aynı işi yapmak mümkündür. Aşağıda kullanılan **ToCharArray**() metodu string bilgideki karakterleri char tipindeki dizi değişkene doldurup bu dizi değişkeni geri göndermektedir.

```
public static bool Uygun(string Str)
{
 foreach(char Kar in Str.ToCharArray())
  if (!char.IsNumber(Kar))
    return false;
  return true;
}
```

Single veya Double yapısının ToString() metodu ile ondalık nokta içeren sayısal bilgileri String tipe dönüştürürken dikkatli olmak gerekir. Çünkü ToString() metodu geçerli kültür ve dilden etkilenebilmektedir. Bu konuda bilgi vermek için aşağıda verilen kodu yazdık.

```
static void Main(string[] args)
{
 float ivme = 9.81f;
 string str = ivme.ToString();
 Console.WriteLine(str);
}
```

Main() metodu bu şekilde düzenlenen uygulama çalıştırılıp bu satırlar işletildiğinde konsola "9,81" yani ondalık karakter olarak virgülün (,) kullanıldığı bir string bilgi yazılır. Çünkü bu metni hazırladığımız sırada ("tr-TR") geçerli kültür ve dildi. ToString() metodunun İngiliz kültürüne ve Amerikan diline göre dönüştürme yapmasını istemiş olsaydık aşağıdaki gibi düzenleme yapardık.

```
static void Main(string[] args)
{
 float ivme = 9.81f;
```

```
 string str = ivme.ToString(
       CultureInfo.GetCultureInfo("en-US").NumberFormat);
 Console.WriteLine(str);
}
```

Convert Sınıfı İle Dönüştürme Yapmak

Değişik tipteki değişkenlerin içeriklerini dönüştürürken **Convert** sınıfının metotlarından yararlanabilirsiniz. Bu sınıf sayesinde dönüştürme yaparken dönüştürme işlemi sonucu elde edilecek bilginin aktarılacağı değişkenin tipine göre metot seçmek gerekir. Sayısal bilgi olarak değerlendirilebilecek string bilgileri int tipine dönüştürme işlemi **ToInt32**() metodu ile yapılmaktadır. Bu metot, dönüştürülmek istenen bilgiyi parametre olarak almaktadır.

```
string str = "1234";
int sayi = Convert.ToInt32(str); // sayi 1234 olur.
```

Bu örnekte string bilginin içeriği int tipine uygun olduğu için dönüştürme işlemi başarılı olur. Bu string bilgi yalnızca rakamlardan meydana gelmiyor olsaydı program hatasızca derlenirdi ama çalışma anında hata meydana gelirdi. String bilgiyi Single (float anahtar kelimesi ile tanımlanan değişken tipi) bilgiye dönüştürmek istiyorsanız **ToSingle**() metodundan yararlanabilirsiniz.

```
float sayi;
string str = "1234,876";
sayi = Convert.ToSingle(str); // sayi 1234,876 olur.
```

String bilgiyi double tipine dönüştürmek istemiş olsaydık ToSingle() yerine **ToDouble**() metodunu kullanırdık. Şimdi ise int bilgiyi **ToByte**() metodu ile byte tipine dönüştüreceğiz.

```
int sayi1 = 128;
byte sayi2 = Convert.ToByte(sayi1); // sayi2 128 olur.
```

Bu dönüştürme işleminde byte tipindeki değişkene aktarılan 128 sayısı byte değişkenlerin kapasitesini aşmadığı için hata meydana gelmez ve bilgi kaybı olmaz. İçeriği dönüştürülen int değişken 255'ten büyük sayı içeriyor olsaydı çalışma anında hata meydana gelirdi.

Yukarıda ToByte() metoduna **int** tipinde bilgiyi parametre olarak verip btye tipine dönüştürdük. İsterseniz ToByte() metoduna bool bir değişkeni parametre olarak verebilirsiniz.

```
bool durum = false;
byte sayi = Convert.ToByte(durum);   // sayi 0 olur.
```

Bu koddaki "durum" adındaki bool değişken false bilgisini içerdiği için dönüştürme sonucu byte değişkenin içeriği 0 olur. Bu bool değişken true içeriyor olsaydı dönüştürme işlemi yapıldığında "sayi" adı verilen byte değişken 1 değerini sahip olurdu.

Herhangi bir tipteki bilgiyi bool tipine dönüştürmek istiyorsanız Convert sınıfının **ToBoolean**() metodunu kullanabilirsiniz. Aşağıda verilen dönüştürmede byte tipindeki "sayi" değişkeni 0 bilgisini içerdiği için dönüştürme sonucu bool değişkenin içeriği false olur.

```
byte number = 0;
bool Durum = Convert.ToBoolean(number); // Durum false olur.
```

byte tipindeki bu değişken 0'dan büyük bir değere sahip olsaydı bool değişkene true aktarılırdı. Benzer mantıkla Convert sınıfının **ToChar**(), **ToDecimal**(), **ToUint32**() ve **ToInt64**() metotları ile istediğiniz dönüştürme işlemlerini yapabilirsiniz.

Parse() ve TryParse() Metotları

Değişik türdeki bilgiler **Convert** sınıfının metotları ile çevrilirken istenildiği gibi biçimleme yapılabilindiğini biliyorsunuz. Bazen bu işlemin tersine gerek duyulur. Yani string bir bilgi ayrıştırılıp içerdiği sayısal veya tarihsel bilgi elde edilmek istenebilir. Örneğin bir TextBox'a yazıp biçimlendirdiğiniz string bilgiyi sayısal bilgiye dönüştürmek isteyebilirsiniz. Bu dönüştürme **Int16** ve **Int32** gibi yapıların **Parse**() metotlarıyla yapılabilir.

```
string str = "123";
int sayi = System.Int16.Parse(str);
```

Bu kodda önce string ve int tipte 2 değişken tanımlayıp string değişkene yalnızca rakamlardan meydana gelen bilgi aktardık. Ardından bu string değişkenin içeriğini **Int16** yapısının **Parse**() metoduna parametre olarak verip int değişkene aktardık. String bilgide küçük bir değer saklandığı için **Int16** yapısının Parse() metodunu kullandık. String bilgide büyükçe bir sayı saklanıyor olsaydı bu kez Int32 yapısının **Parse**() metodunu kullanmak gerekirdi.

Bu nedenle String bilgileri Parse() metodu ile dönüştürürken sayısal bilginin tipine ve büyüklüğüne göre **struct** seçmelisiniz. Örneğin string değişkende kesirli sayı olarak değerlendirilecek bilgi varken Int16 veya Int32 yapılarının(struct) **Parse**() metodu ile ayrıştırma yaparsanız hata meydana gelir. Aşağıda verilen kod işletildiğinde string değişkenin içeriği Parse() metodu ile **Single** bilgiye dönüştürülür.

```
string str = "123,456";
float sayi = System.Single.Parse(str);
textBox1.Text = Convert.ToString(sayi);
```

Bu kodda Single yapısının Parse() metodu kendisine parametre olarak verilen bilgiyi kayıpsız olarak float tipine dönüştürür. Şimdi ise üçer hanelere ayrılmış tamsayı bilgi içeren String değişkenin içeriğini Parse() metodu ile dönüştürüp long değişkene aktaracağız.

```
string str = "1.234.567";
long sayi = System.Int32.Parse(str);
```

İlk bakışta Parse() metodunun dönüştürmeyi başarılı bir şekilde yapacağı beklenir. Ancak bu şekilde üçer hane ayırıcı karakter içeren bilgiyi Parse() metodu ile dönüştürmek istiyorsanız Parse() metodunun 2. parametresinde bunu belirtmeniz gerekir.

```
long sayi;
string str = "1.234.567";
sayi = System.Int32.Parse(str,
       System.Globalization.NumberStyles.AllowThousands);
```

Bu örneği denerken kullanılan bilgisayarda ondalık ayırıcı karakter olarak virgül(,), binlik basamak ayırıcı karakter olarak nokta(.) geçerliydi. Bu şartlarda Parse() metodu aşağıdaki gibi kullanılırsa hata meydana gelir.

```
string str = "1,234,567";
long sayi = System.Int32.Parse(str,
              System.Globalization.NumberStyles.Any);
textBox1.Text = Convert.ToString(sayi);
```

Kullanılan bilgisayarda üçer hane ayırıcı karakter olarak virgül(,) geçerli olsaydı hata meydana gelmezdi. Bu gibi durumlarda Parse'nin 3. parametresi ile kültür ve dil seçimi yapılabilir. Bu seçimin nasıl yapıldığını aşağıda görebilirsiniz.

```
string str = "1,234,567";
System.Globalization.CultureInfo ulke;
ulke = new System.Globalization.CultureInfo("en-US");
long sayi=System.Int32.Parse(str,
       System.Globalization.NumberStyles.Any, ulke);
textBox1.Text = Convert.ToString(sayi);
```

Bu kodda Parse() metoduna parametre olarak vermek üzere **CultureInfo** tipinde bir değişken tanımladık. Parametre olarak "en-US" kullanıldığı için Parse() metoduna 1. parametre olarak verilen bilgi hatasızca long tipine dönüştürüldü.

Şimdi ise 16 tabanlı bir sayıya çevrilebilecek string bilgiyi Parse() metoduna parametre olarak vereceğiz. Bu amaçla hazırladığımız kodu aşağıda görebilirsiniz.

```
string str = "FFFF";
long sayi = System.Int32.Parse(str,
           System.Globalization.NumberStyles.HexNumber);
textBox1.Text = Convert.ToString(sayi); //TextBox'a 65535 yazılır
```

Convert sınıfının **ToString**() metodu ile kolayca 10 tabanlı sayıları 16 tabanlı olarak String bilgiye dönüştürülebilir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
int sayi = 255;
string str = Convert.ToString(sayi, 16);
textBox1.Text = str;
```

String bilginin içerdiği bilgiyi ayrıştırıp ilgili türe dönüştürürken kullanılan Parse() metodu ayrıştırmada başarılı olamazsa exception fırlatılır ve tedbir alınmadıysa programın çalışması kırılır. Bu söylediklerimizi örneklemek için aşağıda verilen kodu yazdık.

```
static void Main(string[] args)
{
 string str = "Fenerbahçe 1907";
 int i = int.Parse(str);
 Console.WriteLine(i);
}
```

Burada kullandığımız Parse() metodu Int32 yapısına ait bir metot olduğu için parametre olarak int tipine çevrilebilir bilgi ister. Bu nedenle bu kod hataya neden olur. Parse() metodu ile dönüştürme yaparken meydana gelmesi muhtemel hataları önlemek için aşağıda yapıldığı gibi try-catch blokları hazırlanabilir.

```
static void Main(string[] args)
{
 string str = "Fenerbahçe 1907";
 try
  {
   int i = int.Parse(str);
   Console.WriteLine(i);
  }
 catch (Exception h)
  {
   Console.WriteLine("Dönüştürme başarısız");
  }
}
```

Bu şekilde Parse() metodunu try-catch bloğuna alıp yapılmak istenen dönüştürme işleminin seyrini kontrol altına almak yerine aynı sonucu **TryParse**() metodu ile almak mümkündür. TryParse'nin Parse'den en önemli farkı dönüştürme başarısız olduğunda Exception nesnesi hazırlamıyor oluşudur. Dolayısıyla bu metot kullanıldığında try-catch bloklarını hazırlamak gerekmiyor. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 int yil = 0;
 string str = "Fenerbahçe 1907";
 bool sonuc = int.TryParse(str, out yil);
 if (sonuc)
   Console.WriteLine(yil);
 else
   Console.WriteLine("Dönüştürme başarısız");
}
```

TryParse() metodunu kullanmadan önce parse işlemi sonucu elde edilecek bilginin aktarılacağı değişkenin tanımlanması gerekir. Burada Int32 yapısına ait TryParse() metodunu kullandığımız için int tipinde bir değişken tanımladık. TryParse() metodu 1. parametre olarak ayrıştırmaya konu edilecek bilgiyi almakta ve işlem başarılı olursa geriye true göndermektedir. Programcı, geriye gönderilen bilgiye bakıp işlemin başarılıp olup olmadığını öğrenmektedir.

Dönüştürme işlemi başarısız olduğunda TryParse() metoduna 2. parametre olarak verilen değişkene 0 aktarılmaktadır. Bu nedenle dönüştürme işlemin seyri 2. parametrenin içeriğine bakılıp öğrenilebilir.

```
int yil = 0;
string str = "Fenerbahçe 1907";
bool sonuc = int.TryParse(str, out yil);
if (yil == 0)
  Console.WriteLine("Dönüştürme işlemi başarısız");
```

Nullable Tipler

C#'ın 2.0'dan önceki sürümünde int, long, bool, struct gibi değer (Value) tiplere **null** bilgisini aktarmak mümkün değildi. Referans tipler ise null olabiliyordu. Çünkü referans tiplerin aksine value tipler direk verilerle çalışmaktadır. C# 2.0'dan itibaren int ve long gibi value tipler null değerini içerebilmektedir.

Null değer içerebilen değişkenlere daha çok veritabanı işlemleri yapılırken gerek duyulmaktadır. Örneğin veritabanına kaydı yapılan arabanın ağırlığı boş bırakılabilinir. Ağırlık alanı boş bırakıldığında veritabanına 0 yerine null yazılması daha anlamlıdır. Bu konuda bilgi vermek üzere aşağıda verilen 2 satırı yazdık.

```
string takim;
takim = null;
```

Bu kod bütün C# sürümlerinde çalışır. Çünkü burada tanımlayıp null bilgisi aktarılan değişken referans tipte bir değişkendir. Bu değişken int veya long gibi value bir değişken olsaydı hata meydana gelirdi. Bu nedenle aşağıda verilen satırlar hataya neden olur.

```
int yil;
yil = null; // hata meydana gelir.
```

C#'ın 2.0 sürümünden itibaren int, byte ve long gibi Value tiplere null bilgisinin aktarılmasını desteklemektedir. Söz konusu value tip değişkene null bilgisinin aktarılmasına izin verilsin istiyorsanız değişken tanımlarken tip adına soru(?) işaretini eklemelisiniz.

```
int? yil;
yil = null;
```

Bu satırlar C#'ın 2.0 ve sonraki sürümlerinde hataya neden olmazlar. Null değer alabilen değişkenleri bu şekilde ? işaretini kullanarak tanımlayabildiğiniz ilgili yapıdan yararlanabilirsiniz. Örnek olması için **byte** ve **int** tipinde 2 nullable değişken tanımlayıp bilgi aktardık.

```
System.Nullable<byte> sayi1;
System.Nullable<int> sayi2;
sayi1 = null;
sayi2 = null;
```

Hemen hatırlatmak gerekirse, zaten null değer alabilen String gibi referans tipler için değişken tanımlarken "?" ile nullable olarak işaret etmek istemeniz halinde hata meydana gelir.

```
string? takim; // hata meydana gelir
takim = null;
```

Daha önce tanımladığınız değişkenin null değer içerip içermediğini öğrenmek istiyorsanız **HasValue** özelliğine bakabilirsiniz. HasValue özelliği yalnızca Nullable değişkenlerde bulunmaktadır. Bu özellik hakkında bilgi vermek için aşağıda verilen kodu hazırladık.

```
int? yil = null;
if (yil.HasValue)
```

```
    MessageBox.Show("Değişken normal değer içeriyor");
else
    MessageBox.Show("Değişken null değer içeriyor");
```

Nullable değişkenin **null** olup olmadığını öğrenirken "!=" ve "==" operatörlerini kullanabilirsiniz. Bu amaçla yukarıda verilen kodu aşağıdaki gibi değiştirdik.

```
int? yil = null;
if (yil != null)
    MessageBox.Show("Değişken normal değer içeriyor");
if (yil == null)
    MessageBox.Show("Değişken null değer içeriyor");
```

Daha önce null değer alabilecek şekilde tanımlanan değişken normal değer içeriyorsa **HasValue** özelliği true olmaktadır. Yukarıda verilen örnekte "yil" değişkenine null aktarıldığı için HasValue özelliği false olur. Null değer alabilecek şekilde tanımlanan değişkenin içeriğini öğrenmek istiyorsanız **Value** özelliğine bakabilirsiniz.

```
int? yil = 2012;
if (yil.HasValue)
    textBox1.Text = yil.Value.ToString();
```

Bu kodda "yil" değişkenine daha önce sayısal bilgi aktarıldığı için **HasValue** özelliği true olur ve değişkenin **Value** özelliği TextBox'a aktarılır. null bilgi içeren Nullable değişkenlerin Value özelliği okunamaz. Bu nedenle aşağıda verilen kod hataya neden olur.

```
int? yil;
yil = null;
textBox1.Text = yil.Value.ToString(); // hata meydana gelir.
```

Bu nedenle nullable değişkenlerin Value özellikleri okunmadan önce **HasValue** özelliklerine bakıp null olup olmadığını öğrenmek gerekir. null içerikli değişkenlerin matematiksel işlemlere etkisini göstermek için aşağıda verilen basit örneği hazırladık. Null değer içeren Nullable değişken başka bir Nullable değişkenle toplandığında sonuç null olur. Bu nedenle aşağıda verilen kod işletildiğinde TextBox'a "Sonuc null" bilgisi aktarılır.

```
int? sayi1, sayi2, sonuc;
sayi1 = 20;
sayi2 = null;
sonuc = sayi1 + sayi2;
if (sonuc.HasValue)
   textBox1.Text = sonuc.Value.ToString();
else
   textBox1.Text ="Sonuç null";
```

Bu örnekteki "sayi2 null" olmasaydı toplama işlemi sonucu bulunan değer null olmazdı. Bu örnekte Nullable 2 değişkenin içeriğini toplayıp dönüştürme yapmaya gerek kalmadan başka bir Nullable değişkene aktardık. Aşağıda verilen örnekte ise "sonuc" değişkeni Nullable olmadığı için dönüştürme yaptık.

```
int? sayi1, sayi2;
sayi1 = 20;
sayi2 = 10;
int sonuc = (int)(sayi1 * sayi2);
textBox1.Text = sonuc.ToString(); // TextBox'a 200 yazılır
```

Bu kodda birbirleri ile çarpılan her iki nullable değişken null olmadıkları için hata meydana gelmez ve çarpma sonucu bulunan değer TextBox'a aktarılır. Birbirleri ile çarpılan bu 2 nullable değişkenden birisi null olsaydı çarpma sonucu bulunan değer de null olurdu. Çarpma işlemi sonucu elde edilen null değerin aktarılmak istendiği "sonuc" adlı değişken Nullable olmadığı için bu kod aşağıdaki gibi düzenlendiğinde hata meydana gelir.

```
int? sayi1, sayi2;
sayi1 = 20;
sayi2 = null;
int sonuc = (int) (sayi1 * sayi2); // Hata meydana gelir
```

Daha önce belirtildiği gibi 2 ihtimalin söz konusu olduğu karşılaştırma işlemleri için "if" yerine ? operatörü kullanılabilmektedir. Benzer durum **Nullable** değişkenler için de geçerlidir. Nullable değişken null ise başka, değilse başka işlem yapılmak istenebilir. Bu konuda bilgi vermek için aşağıda verilen basit örneği hazırladık.

```
int? sayi1, sayi2, sonuc;
sayi1 = 10;
sayi2 = 20;
if (sayi1 == null)
  sayi1 = 1;
if (sayi2 == null)
  sayi2 = 1;
sonuc = sayi1 * sayi2;
textBox1.Text = sonuc.ToString();
```

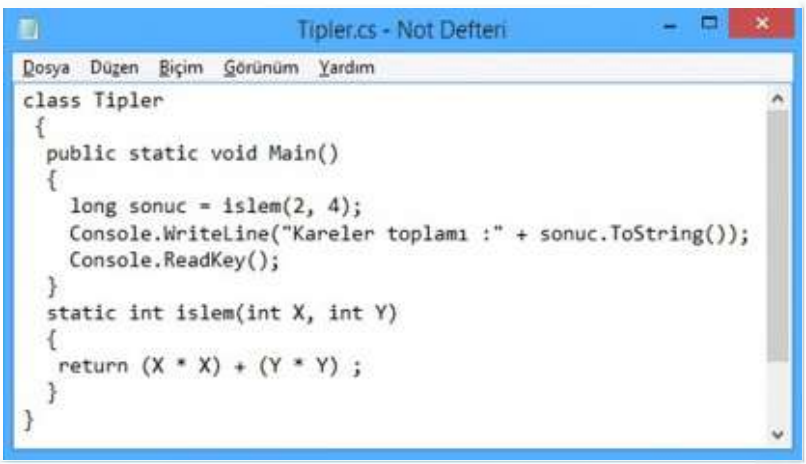
Bu kodda 3 adet **nullable** değişken tanımlayıp ilk ikisine bilgi aktardık. İlk 2 değişkeni birbiri ile çarptıktan sonra bulunan değeri "Sonuc" adlı değişkene aktardık. Çarpma işlemi yapıldığında çıkacak sonucun null olmasını engellemek için önceden bu değişkenlerin içeriklerini kontrol ettik. Benzer işlemi "??" operatörü ile yapmak daha pratiktir.

```
int? sonuc;
int? sayi1, sayi2;
sayi1 = 10;
sayi2 = null;
sayi1 = sayi1 ?? 1;
sayi2 = sayi2 ?? 1;
sonuc = sayi1 * sayi2; // Sonuc 10 olur.
textBox1.Text = sonuc.ToString();
```

Bu örnekte "sayi1 = sayi1 ?? 1" satırında??'dan önce yazılan değişken (burada sayi1) Null değilse eşitliğin sol tarafına aktarılır, Null ise eşitliğin sol tarafındaki değişkene ??'den sonra yazılan (burada 1) aktarılır.

Dynamic Değişkenler - System.Dynamic

C# programlama dilinin 4.0'dan önceki sürümlerinde kod dahilinde kullanılan tipler önceden bellidir. Yani derleyici derleme yapmadan önce kullanılan tipler hakkında net bilgiye sahiptir. Bu tip dillere statik tipli diller denilmektedir. Dolayısıyla C# dilinin ilk 3 sürümünde geçerli olmayan herhangi bir tipin kullanıldığı kodu derlemek ve çalıştırmak mümkün değildi. Bu konuyu en temelden anlatmak istediğimiz için Not Defteri ile aşağıdaki gibi bir konsolda çalışan uygulama yazdık.



Bu koda yakından baktığınızda "sonuc" adını verdiğimiz long tipinde bir değişken tanımlayıp kullandığımızı fark edersiniz. Ayrıca hazırlayıp "islem" adını verdiğimiz metot "int" tipinde 2 parametreye sahiptir. Bu kodda herhangi bir hata olmadığı ve kullanılan tipler derleyici tarafından bilindiği için konsolda yapılacak derleme başarılı olur.

Bu koddaki long tipi yerine yanlışlıkla "Long" yazarsanız veya Main() metodu sınırları için işletmek üzere "islem" yerine Islem() yazarsanız derleme başarılı olmaz ve EXE kod üretilmez. Not Defteri ile kod yazan programcı yaptığı hatalardan ancak derleme yaparken haberdar olur. C# uygulaması geliştirenler genellikle Visual Studio kullandıkları için bu kodda "long" yerine "Long" yazdıklarında Visual Studio anında uyarıda bulunur. Visual Studio, kod yazımı sırasında yazılan kodun derlemeden geçip geçmeyeceğini her an kontrol etmektedir. Böylece desteklenmeyen bir tip kullanıldığında veya olmayan bir metot çağırıldığında daha uygulama çalışma aşamasına gelmeden derleme hatası alınıyor.

Son yıllarda Iron ve Python gibi dinamik dillerin yaygınlaşması sonucu bu dillerle ilişki kurulup kaynak alış verişi yapabilmek için C# programlama diline dinamik tip desteği kazandırıldı. Dinamik tipler **dynamic** anahtar kelimesi ile tanımlanmaktadır.

Daha önce hakkında bilgi verilen **var** anahtar kelimesi hakkında söylenenler hatırlanırsa "var" ile tanımlanan değişkenlerin tipi derleme sırasında aktarılan ilk değere bakılıp belirlenmektedir. Dolayısıyla var anahtar kelimesi ayrı bir tip olmayıp sadece değişken tanımlama ve tip seçimi konusunda derleyiciden yardım alma yolunun açılmasını sağlamaktadır. Ayrıca var anahtar kelimesiyle ilgili olarak yapılan yanlış bilgi aktarma ve

işlemler Visual Studio tarafından daha kod yazılırken rapor edilmektedir. Burada hakkında bilgi verilecek dinamik tiplerde ise tip kontrolü derleme anında değil çalışma anında yapılmaktadır. Bu nedenle konu dinamik tipler olduğunda Visual Studio'dan IntelliSense desteği alınamıyor.

Tanımlanmak istenen değişkenin tipinin "çalışma anında" değişkene aktarılacak bilgiye göre tespit edilmesi isteniyorsa veya böyle bir ihtiyaç varsa **dynamic** anahtar kelimesinden yararlanıp değişken tanımlanır. Tekrar etmek gerekirse, dynamic anahtar kelimesi ile değişken tanımlamak demek, değişkenin tipinin çalışma anında tespit edilmesini istemek demektir. Aşağıda verilen kodda önce dynamic bir değişken tanımladık. Sonra bu değişkene string bilgi aktarıp tipini öğrenmeye çalıştık.

```
private void button1_Click(object sender, EventArgs e)
{
 dynamic takim;
 takim = "Fenerbahçe";
 textBox1.Text = takim.GetType().ToString(); //TextBox'a System.String
}
```

Burada tanımladığımız dynamic değişkene string bilgi aktarıldığı için çalışma anında değişkenin tipinin string olması sağlanır. Şimdi bu koda ekleme yapıp önce string bilgi aktarıp tipinin string olmasını sağladığımız değişkene sonradan sayısal bilgi aktaracağız.

```
private void button1_Click(object sender, EventArgs e)
{
 dynamic takim;
 takim = "Fenerbahçe";
 textBox1.Text = takim.GetType().ToString(); //TextBox'a System.String
 takim = 2017;
 textBox2.Text = takim.GetType().ToString(); //TextBox'a System.Int32
}
```

Bu kod hatasız derlenir ve çalışma anında beklenen sonucu verir. Başka bir deyişle çalışma anında "takim" değişkenin tipinin önce string olması sağlanır. Ne zaman ki bu değişkene sayısal bilgi aktarılır o zaman değişkenin tipi çalışma anında Int32 olarak değiştirilir. Dolayısıyla dynamic değişkenlere, üzerinde işlem yapılacak bilginin tipinin kodlama sırasında bilinmediği durumlarda başvurmak gerekir.

Dynamic değişkenlerin tiplerinin çalışma anında tespit edildiğini işaret etmek üzere bir de aşağıda verilen kodu yazdık. Bu kod hataya neden olmaz ve beklenen sonucu verir.

```
dynamic number1 = 100;
dynamic number2 = 200;
int netice = number1 + number2;
textBox1.Text = netice.ToString(); //TextBox'a 300 yazılır
```

Burada "number1" ve "number2" adını verdiğimiz bu değişkenlere aktarılacak sayısal bilgileri kodlama sırasında aktarmış olsak bile kod yazılırken Visual Studio ve derleme yapılırken

derleyici dinamik değişkenlere aktarılan bilgilerle ilgilenmezler. Bu nedenle bu kod aşağıdaki değiştirilirse neler olacağı konusunda düşünün biraz.

```
private void button1_Click(object sender, EventArgs e)
{
 dynamic number1 = "Adana Demirspor";
 dynamic number2 = 1940;
 int netice = number1 + number2;
 textBox1.Text = netice.ToString();
}
```

Tahmin edeceğiniz gibi bu kod yazılırken ve derlenirken Visual Studio'nun bir itirazı olmaz. Derleme sonucu hazırlanan EXE dosya çalıştırıldığında başlangıçta bir hata rapor edilmez. Başka bir deyişle formdaki "button1" nesnesi tıklanıp bu metot işletilene kadar her şey yolunda gider. Çalışma anında button nesnesi tıklanıp bu metot işletilirse dinamik değişkenlere bu bilgiler aktarılır. Ne zaman ki "number1" ile "number2" toplanmak istenir hata meydana gelir.

C# derleyicisinin dinamik tiplerle ilgili ayrıntılardan söz edebilmek için aşağıda verilen kodu hazırladık. Bu kodda geriye gönderilecek bilginin tipinin çalışma anında belirleneceği bir metot bulunmaktadır.

```
static dynamic Kare(double X)
{
 return (X * X) / 2;
}
private void button1_Click(object sender, EventArgs e)
{
 dynamic netice = Kare(5);
 textBox1.Text = netice.ToString();
 textBox2.Text = netice.GetType().ToString();
}
```

Burada Kare() adını verdiğimiz metodun parametresi hangi tipte ise geriye o tipte bilgi gönderilmektedir. Bu koda bakıp şunu söyleyebiliriz: Hazırladığınız metodun geriye göndereceği bilginin çağrılma şeklinden etkilenmesini istiyorsanız geri dönüş tipini dynamic yapabilirsiniz.

Dinamik tiplerle ilgili başka bir ayrıntıyı ortaya çıkarmak için üzerinde çalıştığımız konsol uygulamasında aşağıdaki gibi 2 public alana sahip bir class hazırladık.

```
public class Takim
{
 public string adi;
 public int kurulusu;
}
```

Ardından uygulamanın Main() metodunda bu sınıfın örneğini alıp nesnenin alanlarına bilgi aktarıp içeriklerini konsola yazdık. Verilen kodda "Takim" adını verdiğimiz sınıfın örneğini alıp dinamik değişkene aktarırken eşitliğin sol tarafına sınıf adını yazmadık.

```
public static void Main()
{
 dynamic T = new Takim();
 T.adi = "Adana Demirspor";
 T.kurulusu = 1940;
 Console.WriteLine("Takım  :" + T.adi);
 Console.WriteLine("Kuruluş yılı : " + T.kurulusu.ToString());
}
```

Bu uygulama derlenip çalıştırılırsa herhangi bir hatalı durumla karşılaşılmaz. Şimdi kodda değişiklik yapacağız. Sınıfın public olan alanlarını **private** yapacağız ve Main() metoduna fazladan satır ekleyip uygulamanın çalışmasını dynamic tipin kullanıldığı satırdan önce askıya alacağız.

```
class Program
{
 public static void Main()
 {
  Console.ReadKey();
  dynamic T = new Takim();
  T.adi = "Adana Demirspor";
  T.kurulusu = 1940;
  Console.WriteLine("Takım  :" + T.adi);
  Console.WriteLine("Kuruluş yılı : " + T.kurulusu.ToString());
}
}
public class Takim
{
 private string adi;
 private int kurulusu;
}
```

Bu sınıfın örneğini aldığımız satırda dynamic anahtar kelimesine yer vermemiş olsaydık Visual Studio kodlama sırasında hata rapor ederdi. Çünkü "Takim" sınıfının private alanlarına erişilmek istenmektedir. "Takim" tipindeki nesnenin dynamic olmasını sağladığımız için bu kod başarıyla derlenir ve çalışmaya başlar. Ne zaman ki kodun işletimi "Takim" sınıfının alanlarına erişim sağlanan satırına gelir o zaman hata meydana gelir ve programın çalışması kırılır. Burada hazırladığımız kısa kodda sınıfımız çok sayıda alana sahip olsaydı ve başka bir kod dosyasında bulunsaydı Visual Studio'nun IntelliSense hizmetine alışmış programcıları zorlardı. Çünkü konu dinamik tipler olduğunda Visual Studio bu hizmeti vermiyor.

Object –Dynamic Karşılaştırması

Şimdiye kadar dynamic tipler hakkında anlatılanlara göre dinamik değişkenler object tipli değişkenlerden oldukça farklıdır. Bir çok kaynakta Object tipler ile Dynamic'ler karşılaştırıldığı için bu konuda birkaç cümle yazmak gerekmektedir. Önce benzerliklerin üzerinde duracağız: Dynamic değişkenlere tıpkı Object değişkenlerde olduğu gibi istenen zamanda istenen tipte bilgi aktarılabiliniyor. Bu 2 değişken tipi hakkında bilgi vermek için aşağıda verilen kodu yazdık.

```
public static void Main()
{
 object number1 = 2017;
 dynamic number2 = 2017;
 Console.WriteLine("1. Değişken tipi: " + number1.GetType().ToString());
 Console.WriteLine("2. Değişken tipi: " + number2.GetType().ToString());
}
```

Bu uygulama derlenip çalıştırıldığında konsola her iki değişkenin tipinin System.Int32 olduğu yazılır. En başında belirtmek gerekirse, burada 2 kez kullandığımız GetType() metoduna biraz odaklanmak gerek. Yorumlamaya object tipli değişkenden başlayacağız: Burada "number1" adını verdiğimiz object değişkene sayısal bilgi aktardığımız için GetType() metodu bize değişkenin içerdiği bilginin tipini söyler. Geriye gönderdiği bilgi değişkenin tipi olmayıp içerdiği bilgiyle ilgilidir. Çünkü değişkenin tipi tanımlama sırasında object olarak zaten seçilmiştir, sonradan değiştirilemez. Örneğin yukarıda verilen koda ekleme yapılıp object değişkene başka tipte bilgi aktarılsa bile değişken object olarak kalmaya devam eder.

Burada "number2" adını verdiğimiz dynamic değişken çalışma anında kodun işletimi bilgi aktarılan satıra gelinceye kadar herhangi bir tipe sahip değildir. Ne zaman ki çalışma anında "dynamic number2 = 2017" satırı işletilir bu değişkenin artık bir tipi vardır. Bu nedenle GetType() metodunun bu değişkenle ilgili olarak geriye gönderdiği bilgi tipini ve içeriğini işaret etmektir. Ancak bu tipi aynı şekilde çalışma anında başka bir tipte bilgi aktararak değiştirmek mümkündür.

```
public static void Main()
{
 dynamic number1 = 2017;
 object number2 = 2017;
 Console.WriteLine("1. Değişken tipi: " + number1.GetType().ToString());
 Console.WriteLine("2. Değişken tipi: " + number2.GetType().ToString());
 number1 = "Adana Demirspor";
 number2 = "Fenerbahçe";
 Console.WriteLine("1. Değişken tipi: " + number1.GetType().ToString());
 Console.WriteLine("2. Değişken tipi: " + number2.GetType().ToString());
}
```

Koda sonradan eklediğimiz 4 satır sayesinde object değişkenin içeriği string yapıldığı için GetType() metodu geriye "System.String" bilgisini gönderir. Değişkenin tipi object olmasına rağmen String bilgi içerdiği için GetType() metodu böyle bir sonuç üretir. Dynamic değişkene ise sonradan String bilgi aktardığımız için bu değişkenin tipi aynı şekilde değiştirilmediği sürece artık string'tir.

6. BÖLÜM
DİZİ DEĞİŞKENLER

Aynı tipte olup bellekte yan yana duran birden fazla değişkenin hepsine birden dizi değişken adı verilmektedir. Dizi değişkenlerin normal değişkenlerden en önemli farkları indekslenip elemanlara erişimde "[]" operatörünün kullanabilinmesidir.

Şimdi ise bu tanımı genişletip maddeler halinde bir şeyler yazacağız.

- Konu dizi değişkenler olunca aynı ad altında gruplanan birden fazla değişkenin söz konusu olduğunu anlamak gerekir. C# derleyicisi tek elemanlı dizi değişken tanımlamanıza bir şey demese bile dizi değişkenden söz edildiğine dizide birden fazla elemanın olduğu düşünülür.
- Madem dizi değişkenlerde birden fazla eleman var ve bunlar bellekte yan yana duracaklar, tipilerinin aynı olması gerekiyor. Her türlü bilginin aktarılabildiği object değişkenlerin olması bu durumu değiştirmiyor. Çünkü değişkenin tipinden söz ediyoruz, içeriğinden değil.
- Madem dizi değişkenlerin elemanları bellekte yan yana yerleştiriliyor ve sayıları birden fazla oluyor; o zaman bu elemanlara erişim sorunu vardır. Erişim sorunu indeksleme yapılıp "[]" operatörü kullanılarak çözülmektedir. Dizi değişkenin hangi elemanıyla ilgileniliyorsa o elemanın dizi içindeki sıra numarası "[]" operatörüne parametre olarak verilmektedir.

C# programları dahilinde **int** tipinde bir değişken tanımlandığında bellekte 4 byte'lık yer rezerve edilir. int yerine **long** tipinde değişken tanımlandığında ise bu kez ilgili değişken için bellekte 8 byte'lık yer ayrılır. Aşağıda verilen metotta aynı tipe sahip 3 değişken tanımladık.

```
private void button1_Click(object sender, EventArgs e)
{
 int ogrenci1;
 int ogrenci2;
 int ogrenci3;
}
```

Bu değişken tanımlama örneğinde her seferinde **int** tipinde bir değişken tanımlanıyor. İstenirse aynı tipte birden fazla bilgiyi saklayabilen, daha doğrusu aynı tipteki birden fazla değişkenin yerine kullanabilinecek **dizi değişken** tanımlanabilir.

Birden fazla değişken veya bilginin oluşturduğu topluluğa veri yapısı denilmektedir. Örneğin; yukarıda tanımlanan int tipindeki değişkenler bir birinden bağımsızdır ve bu değişkenler veri yapısı değildir. Struct'lar, enum'lar ve class'lar birden fazla bilgiyi içerebildikleri için veri yapısı olarak görülürler. Elemanları aynı türden olan ve indeks yardımıyla erişilebilen veri yapılarına dizi (array) denilmektedir.

C#'da dizi değişken tanımlama işlemi 2 aşamada yapılabilmektedir. İlk aşamada tıp seçilip dizi değişken tanımlanmaktadır. Tanımlanan değişkenin "dizi" olduğunu işaret etmek için tip adından sonra köşeli parantezlere "[]" yani indeksleme operatörüne yer veriliyor.

```
int[] ogrenci;
```

Bu satır sayesinde sayısal bilgileri saklayabileceğiniz "ogrenci" adında bir dizi değişken tanımlanır. Bu dizi değişkenin eleman sayısı henüz belli değildir. Dolayısıyla bu değişken henüz belleğin heap adı verilen kısmında yer almamaktadır. Dizi değişkenin eleman sayısı **new** anahtar kelimesinden sonra belirtilmektedir.

```
int[] ogrenci;
ogrenci = new int[10];
```

Burada yapıldığı gibi dizi değişkeni 2 satır yerine tek satırda tanımlayabilirsiniz. Bu 2 satırın işlevini aşağıda verilen satır tek başına yerine getirir.

```
int[] ogrenci = new int[10];
```

Bu şekilde tanımlanan değişkenlere **dizi değişken** adı verilmektedir. Bu dizi değişken için bellekte yan yana duran her biri 4 byte olan 10 adet yer (kutu) ayrılır. C# programlama dilinde dizi değişken tanımlamak demek bir bakıma **System** adlı namespace'deki **Array** sınıfının örneğini almak demektir.

Array sınıfı **abstract** bir sınıf olduğu için new anahtar kelimesi ile örneği alınamaz. Array sınıfı **IList** arayüzünü uygulamış olsa bile Add(), Remove() ve Insert() gibi metotlara sahip değildir. Dolayısıyla mevcut dizi değişkenler üzerinde yapılabilinecek işlemler ArrayList ve List sınıfları sayesinde hazırlanan koleksiyon nesnelerine göre sınırlıdır.

Dizi değişkenlerin elemanlarına bilgi aktarmak veya dizi değişkenin mevcut içeriğine erişip kullanmak için, elemanın dizi içindeki sıra numarası değişken adına köşeli parantez içinde eklenmelidir. Örneğin 10 elemanlı bu dizi değişkenin ilk elemanına bilgi aktarmak için aşağıdaki gibi bir satır yazılabilir. Dizi değişkenlerin elemanları otomatik olarak indekslendiği için elemanlara bu şekilde indeks numarası ile erişilebilmektedir.

```
int[] ogrenci = new int[10];
ogrenci[0] = 65;
```

Bundan çıkarılması gereken sonuç şudur: Verilen örnekte olduğu gibi benzer işlevleri olan ve aynı tipteki birden fazla değişkeni ayrı ayrı tanımlamak yerine bu değişkenleri bir seferde dizi olarak tanımlamak daha mantıklıdır. Dizi değişkenin ilk elemanın sıra numarası 0 olduğu için aşağıda işaret edilen satır hataya neden olur.

```
int[] ogrenci = new int[10];
ogrenci[10] = 70;  // hata meydana gelir
```

Her ne kadar burada dizinin olmayan elemanına erişim sağlanmak istendiğinde hata meydana geleceği işaret edilmiş olsa bile bu hata derleme sırasında meydana gelmez. Hatta dizinin olmayan elemanıyla ilgili işlem yapıldığında kod yazımı sırasında Visual Studio herhangi bir uyarıda bulunmaz. Ta ki uygulama derlenip çalıştırılıncaya kadar. Bu nedenle dikkatli olmak gerek.

Dizi değişkenin elemanlarına farklı tipteki bilgileri aktarmak istiyorsanız değişkenin tipini **object** olarak seçebilirsiniz. **char** tipindeki dizi değişkene string değişkenin içeriğini aktarabilirsiniz. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. string bilginin uzunluğunu öğrenip ondan sonra char tipindeki dizi değişkeni boyutlandırabilirsiniz.

```
string str = "Fenerbahçe";
char[] kar = new char[10] ;
for (int i = 0; i < 10; i++)
{
 kar[i] = str[i];
}
```

Burada verilen 4 satırlık koda bir kez daha bakmanızı istiyoruz. Çünkü bu kodu dizi değişkenlere örnek olması için verdik. Bu kodda ne mi yaptık? Önce string bir değişken tanımlayıp 10 karakter uzunluğunda bilgi aktardık. Devamında 10 elemanlı bir char tipinde dizi değişken tanımladık.

Kodun bizce önemli kısmı for döngüsündeki satırdır. Dikkat ederseniz string değişken sanki bir char tipinde dizi değişkenmiş gibi "[]" operatörü ile karakterlere tek tek erişilip char tipindeki dizi değişkene aktarılmaktadır. Derleyicinin bu satıra itiraz etmeme nedeni şudur: Her string değişken gerçekte indekslenmiş birer char dizisidir. Bu nedenle string değişkenlerin içeriğine char dizisine erişildiği gibi erişilebilip okunabilir. Char dizisinin elemanlarını içeriğini değiştirir gibi string değişkeni değiştirmenize derleyici izin vermez. Yani derleyici okumaya evet, yazmaya hayır diyor.

String değişkenin içeriğini char tipindeki dizi değişkene aktarırken **String** sınıfının **ToCharArray()** metodundan yararlanabilirsiniz. Bu metot string bilgileri char tipindeki dizi değişkene aktarmadan önce boyutlandırmaktadır.

```
string str = "Fenerbahçe";
char[] kar = str.ToCharArray();
int uzunluk = kar.Length;
for (int i = 0; i < uzunluk; i++)
{
 textBox1.Text += kar[i].ToString();
}
```

Dizi Değişkenlerin Varsayılan İlk Değerleri

Bir önceki bölümde statik değişkenler hakkında bilgi verilen sayfalarda ilk değer atanmayan statik değişkenleri kullanmanın hataya neden olmadığı ifade edilmişti. Çünkü derleyici statik değişkenlere, ilgili tipin varsayılan değerini atamaktadır. Benzer durum dizi değişkenler için de geçerlidir. Bu nedenle henüz herhangi bir değer atanmayan dizi değişkenin elemanları söz konusu tipin varsayılan değerini içeriyorlar. Örneğin aşağıda verilen kod işletildiğinde konsola önce 0 sonra false yazılır.

```
public static void Main()
{
 int[] ogrenci = new int[10];
 bool[] durum = new bool[5];
 Console.WriteLine(ogrenci[1]);
 Console.WriteLine(durum[1]);
}
```

Burada tanımlanan dizi değişken string gibi referans bir tip olsaydı elemanlara varsayılan değer olarak null aktarılırdı. Bu nedenle aşağıda verilen kod işletildiğinde konsola 2 kez boşluk yazılır.

```
public static void Main()
{
 string[] sehir = new string[5];
 Console.WriteLine(sehir[0]);
 Console.WriteLine(sehir[1]);
}
```



Dizi değişkenler hakkında yapılan teknik bir değerlendirmeyi "İndeksleyici Hazırlamak ve Kullanmak" adlı bölümde bulabilirsiniz. İndeksleyiciler üzerinde çalışmak, nesneye yönelik programcılıkla ilgili temel kavramlar hakkında bilgi sahibi olduktan sonra anlamlı olduğu için bu bölümden indeksleyicilerin anlatıldığı bölüme atlamanızı önermeyiz.

SetValue() ve GetValue() Metotları

Dizi değişkenlere bilgi aktarırken Array sınıfının **SetValue**() metodunu kullanılabilirsiniz. Bu metot indeksleyiciden yani "[]" operatöründen yararlanamadığı için kullanılmak için daha fazla kod yazmak gerekmektedir. SetValue() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] ilk_5 = new string[5];
ilk_5.SetValue("Kerem", 0);
ilk_5.SetValue("İbrahim", 1);
ilk_5.SetValue("Hidayet", 2);
ilk_5.SetValue("Mehmet", 3);
ilk_5.SetValue("Mirsat", 4);
```

Bu metot dizi değişkene aktarılmak istenen bilgiyi ve elemanın sıra numarasını parametre olarak almaktadır. Bu metodun ilk parametresi object tipinde olduğu için dizi değişkenin elemanlarına dönüştürme yapmadan istenen bilgi aktarılabilir.



Bir sonraki konu başlığında Array sınıfının **CreateInstance**() metodundan söz edilmektedir. Bu metotla hazırlanan Array nesnesine bilgi aktarırken **SetValue**, mevcut içerikleri okurken **GetValue** metotlarını kullanmak gerekmektedir. Bu 2 metodun asıl kullanım alanı Array nesnelerine yapılacak yazma ve okuma işlemleridir.

Tanımlayıp elemanlarına bilgi aktardığınız dizi değişkenlerin elemanlarının içerikleri okunurken Array sınıfının **GetValue**() metodundan yararlanabilirsiniz. GetValue(), tıpkı SetValue() metodu gibi dizi değişkenin elemanlarına erişimde indeksleyiciden yararlanamıyor. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void button2_Click(object sender, EventArgs e)
{
 string[] ilk_5 = new string[5];
 ilk_5.SetValue("Kerem", 0);
 ilk_5.SetValue("İbrahim", 1);
```

```
 ilk_5.SetValue("Hidayet", 2);
 ilk_5.SetValue("Mehmet", 3);
 ilk_5.SetValue("Mirsat", 4);
 listBox1.Items.Add(ilk_5.GetValue(0));
 listBox1.Items.Add(ilk_5.GetValue(1));
 listBox1.Items.Add(ilk_5.GetValue(2));
 listBox1.Items.Add(ilk_5.GetValue(3));
 listBox1.Items.Add(ilk_5.GetValue(4));
}
```

String değişkenin içeriğini bir byte dizisine aktarmak istiyorsanız **GetBytes()** metodundan yararlanabilirsiniz. GetBytes() metodu parametre olarak aldığı String bilginin içeriğine bağlı olarak byte dizisini boyutlamaktadır. Aşağıda verilen kod işletildiğinde ilk TextBox'a "A", ikinci TextBox'a ise A'nın ASCII kodu olan 65 yazılır.

```
private void Form1_Click(object sender, EventArgs e)
{
 string Str = "Ankara";
 byte[] Dizi = Encoding.UTF8.GetBytes(Str);
 textBox1.Text = Convert.ToString((Char)Dizi[0]);
 textBox2.Text = Convert.ToString((Byte)Dizi[1]);
}
```

CreateIntance() Metodu İle Dizi Değişken Tanımlamak

Dizi değişkenleri tanımlarken, başka bir deyişle Array nesnesi hazırlarken Array sınıfının **CreateInstance()** metodunu kullanabilirsiniz. Yukarıda ifade edildiği üzere Array sınıfı abstract bir sınıf olması nedeniyle yapıcı metodu yoktur ve **new** anahtar kelimesi kullanılarak Array nesnesi hazırlanamaz. Buna rağmen "bana mutlaka Array nesnesi lazım" diyorsanız CreateInstance() metodunu kullanabilirsiniz.

CreateInstance() metodunu kullanarak 4 elemanlı ve string tipinde bir dizi değişken tanımlamak istiyorsanız bu metodu aşağıdaki gibi kullanabilirsiniz. İlk satır ile tek boyutlu, ikinci satır ile 2 boyutlu dizi değişken olarak kullanabileceğiniz Array nesneleri hazırlanır.

```
Array dizi1 = Array.CreateInstance(typeof(string), 4);
Array dizi2 = Array.CreateInstance(typeof(string), 4, 4);
```

Bu şekilde hazırlanan Array nesnelerini dizi değişken niyetiyle kullanmak mümkündür ancak zahmetlidir. Çünkü CreateInstance() metodu ile hazırlanan Array nesnesinin elemanlarına erişimde indeksleyici kullanılamıyor. Array nesnesinin elemanlarına bilgi aktarırken **SetValue()**, okuma yaparken **GetValue()** metodunu kullanmak gerekiyor.

Şimdi bu dizi değişkenin elemanlarına bilgi aktarıp içeriklerini bir ListBox'a aktaracağız. Bu amaçla forma bir ListBox yerleştirip aşağıda verilen kodu hazırladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 Array dizi = Array.CreateInstance(typeof(string), 4);
 dizi.SetValue("Fenerbahçe", 0);
 dizi.SetValue("Beşiktaş", 1);
 dizi.SetValue("Galatasaray", 2);
```

```
    dizi.SetValue("Denizlispor", 3);
    foreach (string takim in dizi)
      listBox1.Items.Add(takim);
}
```

Burada Array nesnesinin elemanlarının içeriğini okurken foreach döngüsünü kullandığımız için **GetValue**() metodunu kullanmak durumunda kalmadık. Bu kodda foreach yerine for döngüsü hazırlasaydık GetValue() metodunu kullanmak durumunda kalırdık.

```
private void button2_Click(object sender, EventArgs e)
{
 Array dizi = Array.CreateInstance(typeof(string), 4);
 dizi.SetValue("Fenerbahçe", 0);
 dizi.SetValue("Beşiktaş", 1);
 dizi.SetValue("Galatasaray", 2);
 dizi.SetValue("Denizlispor", 3);
 for (int i=0; i < dizi.Length; i++)
 {
  listBox1.Items.Add(dizi.GetValue(i));
 }
}
```

Overlaod edilmiş çok sayıda versiyonu bulunan GetValue() metodu dizilerden okuduğu bilgiyi geriye object tipinde göndermektedir. Bize sorarsanız özel bir nedeniniz yoksa **CreateInstance**() metodu ile Array nesnesi hazırlamak yerine klasik şekilde dizi değişken tanımlayıp kullanın.

Herhangi bir nedenle **CreateInstance**() metodu ile hazırladığınız Array nesnesine **SetValue**() metodu ile istediğiniz sayıda eleman eklemiş olabilirsiniz. Bu dizi değişkene indeksleyici ile erişmek istiyorum diyorsanız aşağıda yapıldığı gibi casting operatörünü kullanarak bu array nesnesini klasik dizi değişkene dönüştürebilirsiniz. Bu işlemi nasıl yapabileceğinizi aşağıda görebilirsiniz.

```
public static void Main()
{
 Array dizi = Array.CreateInstance(typeof(string), 4);
 dizi.SetValue("Fenerbahçe", 0);
 dizi.SetValue("Beşiktaş", 1);
 dizi.SetValue("Galatasaray", 2);
 dizi.SetValue("Denizlispor", 3);
 string[] takimlar = (string[]) dizi;
 for (int i = 0; i < dizi.Length; i++)
  {
    Console.WriteLine(takimlar[i]);
  }
}
```

Dizi Değişkenlerin Eleman Sayısını Çalışma Anında Belirlemek

Dizi değişkenin eleman sayısını çalışma anında belirlemek mümkündür. Aşağıda verilen kodda **Ogrenci** adında **int** tipinde bir dizi değişken tanımlayıp eleman sayısını çalışma anında belirledik. Bu satırlar bir metoda yazılıp işletildiğinde eleman sayısı TextBox'ın mevcut içeriğine göre belirlenir.

```
int sayi = Convert.ToInt32(textBox1.Text);
int[] ogrenci = new int[sayi];
```

Programın ileri aşamalarında **new** anahtar kelimesini tekrar kullanıp dizi değişkenin eleman sayısını değiştirebilirsiniz. Ancak bu durumda dizi değişkenin mevcut elemanlarının içerikleri silinir. Aşağıda verilen kodda dizi değişkenin eleman sayısını çalışma anında belirledikten sonra ilk elemana bilgi aktardık. Daha sonra bu dizi değişkeni yeniden boyutlandırdık.

```
int sayi = Convert.ToInt32(textBox1.Text);
int[] ogrenci = new int[sayi];
ogrenci[0] = 10;
MessageBox.Show(Convert.ToString(ogrenci[0])); // sonuç 10 olur.
sayi = Convert.ToInt32(textBox2.Text);
ogrenci = new int[sayi];
MessageBox.Show(Convert.ToString(ogrenci[0]));  // sonuç 0 olur.
```

Dizi değişkenin eleman sayısı **new** anahtar kelimesi ile değiştirildiğinde mevcut elemanların içerikleri silineceği için bu örnekte Show() metodunun 2. kullanımında 0 mesajı alınır. Çünkü **int** tipindeki değişkenlerin başlangıç içerikleri sıfırdır.

Array sınıfının **Resize**() metodu ile dizi değişkenlerin eleman sayısını değiştirebilirsiniz. Bu metot eleman sayısı değiştirilmek istenen dizi değişkenin referansını ve yeni eleman sayısını parametre olarak almaktadır. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Form1_Load(object sender, EventArgs e)
{
 string[] iller = new string[2];
 iller[0] = "Adana";
 iller[1] = "Ankara";
 Array.Resize(ref iller, 3);
 iller[2] = "Antalya";
 listBox1.Items.Add(iller[0]);
 listBox1.Items.Add(iller[1]);
 listBox1.Items.Add(iller[2]);
}
```

Bu kodda önce 2 elemanlı bir dizi değişken tanımladık. Devamında bu 2 elemana bilgi aktarıp **Resize**() metodu ile bu dizi değişkenin eleman sayısını 3 yaptık. Ardından diziye sonradan eklenen elemana bilgi aktardık. Asıl vurgulamak istediğimiz nokta şudur: Resize() metodu ile eleman sayısı arttırılan dizi değişkenin elemanların mevcut içerikleri korunur.

İlk Değer Atayarak Dizi Değişken Tanımlamak

Dizi değişkenleri ilk değer atayarak tanımlamak mümkündür. Dizi değişkenlere tanımlama sırasında ilk değerleri aktarmak istiyorsanız satırın sonunu işaret eden noktalı virgülden hemen önce süslü parantezlere "{}" yer yerip elemanlara aktarılacak bilgiler aralarına virgül konulmak suretiyle yazılmalıdır. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
string[] gunler = new string[4]
         {"Pazar", "Pazartesi", "Salı", "Çarşamba"};
```

Bu dizi değişkenin 4 elemana sahip olacağı işaret edildiği için süslü parantezlerin arasında 4 adet bilgi yazılmalıdır. Köşeli paranteze yazılan sayıdan daha az veya daha fazla bilgiyi süslü parantezlerin arasına yazarsanız hata meydana gelir.

Dizi değişken tanımlama ve ilk değer aktarma satırının kolay okunur olmasını sağlamak için aşağıda yapıldığı gibi birden fazla satıra bölmek mümkündür.

```
string[] gunler = new string[4]
 {
  "Pazar",
  "Pazartesi",
  "Salı",
  "Çarşamba" };
```

İlk değer ataması yapılarak tanımlanan dizi değişkenlerin eleman sayısı belirtilmeyebilir. Çünkü dizi değişkenin eleman sayısı istenirse ilk değerlerin sayısına göre belirlenebilir.

```
string[] gunler = new string[]
 {
  "Pazar",
  "Pazartesi",
  "Salı",
  "Çarşamba" };
```

Bu kodda "gunler" adında bir dizi değişken tanımladık ve tanımlama satırında parantezlerin arasına 4 günün adını yazdık. Parantezlerin arasına 4 bilgi yazıldığı için bu dizi değişken 4 elemana sahip olur. İlk değer ataması yapılarak tanımlanan dizi değişkenin elemanlarına sonradan başka bilgi aktarılabilir.

Çok Boyutlu Dizi Değişkenler

Çok boyutlu dizi değişken denildiği zaman genellikle 2 veya 3 boyutlu dizi değişkenler anlaşılmaktadır. 2 boyutlu dizi değişkenleri Excel ve Access tablolarına benzetmek mümkündür. Aşağıda verilen iki satır ile 2 boyutlu ve 6 elemanlı dizi değişken tanımlanır.

```
int[,] Tablo;
Tablo = new int[2, 3];
```

Bu şekilde tanımlanan dizi değişkeni 2 satır ve 3 sütuna sahip bir tablo gibi düşünebilirsiniz. Tablonun 1. satırındaki 1. ve 2. sütuna aşağıdaki gibi bilgi aktarabilirsiniz.

```
int[,] Tablo;
Tablo = new int[2, 3];
Tablo[0, 0] = 10;
Tablo[0, 1] = 15;
```

Tek boyutlu dizi değişkenlerde olduğu gibi çok boyutlu dizi değişkenleri tanımlarken ilk değer ataması yapabilirsiniz. Aşağıda verilen satır sayesinde 2 satır ve 3 sütuna sahip 2 boyutlu bir dizi değişken tanımlanır ve elemanlarına ilk değer ataması yapılır.

```
string[,] Tablo ={{"Kerem","Kaya", "Alper"},{"İbrahim", "Kerem", "Tutku"}};
```

Dizi değişkenlerin kaç boyutlu olduğu öğrenilmek istendiği zaman **Rank**() metodu kullanılmaktadır. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[,] Tablo;
Tablo = new string[4, 5];
int boyut = Tablo.Rank; // boyut 2 olur.
```

Var ve Dynamic Deyimi İle Dizi Değişken Tanımlamak

Bir önceki bölümde işaret edildiği gibi C# 3.0'dan itibaren **var** anahtar kelimesi ile tanımlanan değişkenlerin tipleri değişkene aktarılan ilk bilgiye göre belirlenebilmektedir. Normal değişkenlerde olduğu gibi **var** anahtar kelimesinden yararlanılarak dizi değişkenler tanımlanabilir. Aşağıdaki gibi bir dizi değişken tanımlanırsa **int** tipinde olduğuna hükmedilir.

```
var yillar = new[] {2011, 2012, 2013, 2014};
```

var ile yapılan bu dizi değişken tanımlama satırında ilk değer olarak daha küçük sayısal değerler kullanılsaydı dizi değişkenin tipi yine Int32 olarak seçilirdi. **var** anahtar kelimesi ile aşağıdaki gibi dizi değişken tanımlanması halinde tipinin string olduğuna karar verilir.

```
var mevsimler = new[] {"ilkbahar", "yaz", "sonbahar", "kış"};
```

Var anahtar kelimesi ile dizi değişken tanımlanırken aşağıda yapıldığı gibi new anahtar kelimesinden sonra tip seçimi yapılabilir. Böyle değişken tanımlamakla gerçekte derleyiciye demiş oluyoruz ki tip seçimini bana bırak.

```
var gunler = new string[4]
 {
   "Pazar",
   "Pazartesi",
   "Salı",
   "Çarşamba" };
```

Bu şekilde yapılan değişken tanımlamasında tipin string olmasını istedik. Tip konusunda bir tercihte bulunmamış olsaydık derleyici bu değişkenin tipini yine string olarak seçerdi. Aşağıdaki gibi dizi değişken tanımlaması yapılırsa tip seçimi yapılmadığında derleyicinin tercihi Int32'den yana olur.

```
var sayilar1 = new byte [] {50, 100, 150, 200 };
            Console.WriteLine(sayilar1.GetType());
var sayilar2 = new [] {50, 100, 150, 200 };
            Console.WriteLine(sayilar2.GetType());
```

C# 4.0'dan itibaren **dynamic** anahtar kelimesi ile değişken tanımlamak mümkün hale geldi. Dynamic anahtar kelimesi ile tanımlanan dizi değişkenin elemanlarının tipi çalışma anında aktarılan bilgiye göre tespit edilmektedir. Bu konuda bilgi vermek için aşağıdaki gibi 3 elemanlı dizi değişken tanımlayıp elemanlarına farklı tipte bilgiler aktardık.

```
dynamic[] takimlar = new dynamic[3];
takimlar[0] = 1907;
takimlar[1] = "Fenerbahçe";
takimlar[2] = 9.81;
```

Bu kod işletildiğinde çalışma anında bu dizi değişkenin 1. elemanın Int32, 2. elemanın **string** ve 3. elemanın **double** tipte olması sağlanır. Bu nedenle bu koda aşağıdaki gibi ekleme yapılırsa ilk TextBox'a System.Int32, 2. TextBox'a ise System.String yazılır. Dizi değişkenin her elemanın tipi çalışma anında aktarılan bilginin tipine göre belirlendiği için sonradan ilk aktarılan bilgiler değiştirilebilir.

```
private void button1_Click(object sender, EventArgs e)
{
 dynamic[] takimlar = new dynamic[3];
 takimlar[0] = 1907;
 takimlar[1] = "Fenerbahçe";
 takimlar[2] = 9.81;
 textBox1.Text = takimlar[0].GetType().ToString();
 textBox2.Text = takimlar[1].GetType().ToString();
 textBox3.Text = takimlar[2].GetType().ToString();
}
```

Length Özelliği İle Eleman Sayısını Öğrenmek

Dizi değişkenlerin eleman sayısını öğrenirken Array sınıfının **Length** özelliğinden yararlanabilirsiniz. Bu özellik dizi değişkenin eleman sayısını içermektedir. Length özelliğinin nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] takimlar = {"Bursaspor", "Trabzonspor", "Adanaspor"};
int x = takimlar.Length; // x 3 olur
```

"takimlar" adını verilen dizi değişkenin eleman sayısını ilk değer atamasına göre belirledik. Dikkat edilirse burada dizi değişkeni tanımlarken eşitliğin sağ tarafında **new** anahtar kelimesine yer vermedik. Burada new anahtar kelimesini kullansaydık "[]" operatörüne de yer vermemiz gerekirdi. Bu dizi değişkeni aşağıdaki gibi tanımlamış olsaydık **Length** özelliği 10 değerini içerirdi.

```
string[] takimlar = new string[10];
int x = takimlar.Length; // x 10 olur
```

Length özelliği ile eleman sayısı öğrenilmek istenen dizi değişken çok boyutlu ise geriye dizi değişkenin toplam eleman sayısını içerir. Aşağıdaki gibi tanımlanan dizi değişken 3*3=9 elemana sahip olacağı için Length özelliği 9 değerini içerir.

```
string[,] takimlar = new string[3, 3];
int x = takimlar.Length; // x 9 olur.
```

Bazen dizi değişkenin toplam eleman sayısı yerine istenilen boyutun eleman sayısı öğrenilmek istenir. Bu gibi durumlarda Array sınıfının **GetLength()** metodu kullanılabilinir. Aşağıda verilen satırlar sayesinde dizi değişkenin 2. boyutunun eleman sayısı öğrenilir.

```
string[,,] takimlar;
takimlar = new string[3, 4, 3];
int x = takimlar.GetLength(1); // x 4 olur.
```

Şimdi dizi değişkenlerin kullanılmasına örnek olması için formdaki TextBox'a küçük harflerle yazılan karaktersel bilgileri büyük harfe çevireceğiz. TextBox'lara küçük harflerle yazılan karaktersel bilgileri kolayca büyük harfe çevirmek mümkün olmakla birlikte buradaki amacımız dizi değişkenlerin nasıl kullanıldığına örnek vermektir. Aşağıda verilen kodda TextBox'a yazılan bilgiye göre char tipinde dizi değişken tanımladık. Ardından TextBox'ın içeriğinin her karakterini bu dizi değişkenin ayrı bir elemanına aktardık.

```
int uzunluk = textBox1.Text.Length;
char[] dizi = new char[uzunluk];
for (int i = 0; i < uzunluk; i++)
  dizi[i] = textBox1.Text[i];
```

Bu şekilde TextBox'ın içeriğinin her karakterini char tipindeki dizi değişkene aktardıktan sonra dizi değişkenin her elemanın içeriğinin alfabetik karakter olup olmadığını araştırmak gerekecek. Elemanın içerdiği karakterin Ascii kodu 96'dan büyük ve 123'ten küçükçe dizi değişkenin ilgili elemanın Ascii kodundan 32 çıkaracağız. Bu kodun Türkçeye özgü küçük harfleri büyük harfe çevirmeyeceğini tahmin etmiş olmalısınız.

```
private void Form1_Click(object sender, EventArgs e)
{
 int uzunluk = textBox1.Text.Length;
 char[] Dizi = new char[uzunluk];
 for (int i = 0; i < uzunluk; i++)
   Dizi[i] = textBox1.Text[i];
 for (int i=0; i<uzunluk; i++)
 {
  int harf = (byte)Dizi[i];
  if (harf > 96 && harf < 123)
    Dizi[i] = (char)(harf - 32);
 }
 for (int i = 0; i < uzunluk; i++)
   textBox2.Text += Dizi[i];
}
```

Bu kitabın daha sonra verilecek bölümlerinde ayrıntılı bir şekilde anlatılacağı üzere C# 3.0'dan itibaren LINQ sorgularına destek sağlamak üzere Enumerable sınıfında çok sayıda genişletme metodu hazırlanıp IEnumrable<> arayüzü aracılığıyla kullanıma sunuldu. Array sınıfı bu arayüzü uyguladığı için bu arayüzdeki bazı metotlardan dizi değişkenler üzerinde işlem yaparken yararlanmak mümkündür. Örneğin dizi değişkenlerin eleman sayısını bulurken **Count**() metodundan yararlanmak mümkündür. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] takimlar = new string[10];
int x = takimlar.Count<string>();
Console.WriteLine(x);
```

Burada eleman sayısını öğrenmek istediğimiz dizi değişken string tipte olduğu Count() metoduna ait parantezden önce "<string>" operatörüne yer verdik. Çünkü Count() generic bir metottur. Henüz generic sınıf ve metotların üzerinde durmadığımız için dizi değişkenler üzerinde işlem yaparken kullanabileceğiniz genişletme metotlarıyla ilgilenmeyi ertelemenizi öneriyoruz. Generic sınıflar, genişletme metotlar ve LINQ sorguları hakkında bilgi edindikten sonra Count() gibi metotlarla ilgilenebilirsiniz.

Bazı genişletme metotları var ki bunları kullanmak için ek bilgiye gerek yoktur. Örneğin elemanları sayısal bilgi içeren dizi değişkenlerin elemanlarının ortalamasını bulurken **Average** metodundan yararlanılabilir. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
int[] puanlar = new int[] {10, 12, 14, 17, 18, 20 };
double ortalama = puanlar.Average();
Console.WriteLine(ortalama);
```

Benzer şekilde dizi değişkenin elemanlarının toplamını bulmak istediğinizde **Sum**() adlı genişletme metodundan yararlanabilirsiniz.

```
int[] puanlar = new int[] {10, 12, 14, 17, 18, 20 };
int toplam = puanlar.Sum();
Console.WriteLine(toplam);
```

Dizi Değişkenlerin İçeriklerini Silmek

Yukarıda belirtildiği gibi dizi değişkenlerin eleman sayısında sonradan değişiklik yapıldığında elemanların mevcut içerikleri siliniyor. Dizi değişkenlerin içeriklerini silmek için **Clear()** metodunu kullanabilirsiniz. Array sınıfının bu metodu 3 parametreye sahiptir. İlk parametrede elemanları sıfırlanacak değişken belirtilmektedir.

```
Array.Clear(Dizi Değişken, Başlangıç, Eleman Sayı)
```

2. parametrede içeriği silinecek ilk eleman belirtilmektedir. Başka bir deyişle, içerik silme işleminin başlayacağı eleman işaret edilmektedir. 3. parametrede ise içeriği silinecek eleman sayısı verilmektedir. Clear() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 string[] iller = {"Ankara", "Konya", "İzmir", "Bursa"};
 Array.Clear(iller, 1, 1);
}
```

Bu metotta önce "iller" adında 4 elemana sahip bir dizi değişken tanımlanmaktadır. Ardından Array sınıfının **Clear**() metodu ile 2. elemandan başlamak üzere 1 elemanın içeriği silinmektedir.

Clear() metodu ile içeriği silinen eleman sayısal tipte ise içeriği 0 olur. Clear() metodu ile içeriği silinen string elemanların içeriğine boşluk, bool tipindeki elemanlara ise false aktarılmaktadır. Bu nedenle aşağıda verilen satırlar bir metoda yazılıp işletilirse dizi değişkenin ilgili elemanlarına 0 aktarılır.

```
int[] Dizi = new int[3];
Dizi[0] = 45;
Dizi[1] = 42;
Dizi[2] = 40;
Array.Clear(Dizi, 0, 3);
```

Dizi değişkenler hakkında şimdiye kadar öğrendiklerinizi tekrarlama niyetini aşağıda verilen kodu hazırladık. Bu kodda önce 4 elemanlı bir değişken tanımladık ve elemanlarını foreach döngüsü ile konsola yazdık. Sonra bu dizi değişkenin bir elemanını **Clear**() metodu ile sildik.

```
public static void Main()
{
 string[] dizi = new string[4];
 dizi[0] = "Fenerbahçe";
 dizi[1] = "Beşiktaş";
 dizi[2] = "Galatasaray";
 dizi[3] = "Bursaspor";
 foreach (var item in dizi)
 {
   Console.WriteLine(item);
 }
 Array.Clear(dizi, 1, 1);
 string[] temp_dizi = new string[dizi.Length-1];
 int sayac = 0;
 for (int i = 0; i < dizi.Length; i++)
 {
  if (dizi[i] != null)
  {
   temp_dizi[sayac] = dizi[i];
   sayac = sayac + 1;
  }
 }
 dizi = temp_dizi;
 Console.WriteLine("-------------");
 foreach (var item in dizi)
  {
   Console.WriteLine(item);
  }
}
```

Kodun Clear() metodundan sonraki kısmında içeriği silinen elemanları diziden tümden uzaklaştırmak için geçici bir değişken daha tanımladık. Clear() metodu ile içeriği silinen elemanların **null** olmasından yararlanıp bunları yeni diziye taşımadık. Bu kod üzerinde çalışırsanız yukarıda öğrendiklerinizi pekiştirmiş olursunuz.

Dizi Değişkenlerin İçeriklerini Kopyalamak

Bir dizi değişkenin elemanlarının içeriklerini başka bir dizi değişkene aktarmak istiyorsanız Array sınıfının **Copy**() metodunu kullanabilirsiniz. Copy() metodu, kopyalamaya kaynaklık edecek dizi değişkeni 1. parametre olarak almaktadır.

```
Copy(Kaynak Dizi, Hedef Dizi, Sayi)
```

Kopyalama sırasında hedef durumunda olan dizi değişken Copy() metoduna 2. parametre olarak verilmeli ve içeriği kopyalanacak elemanların sayısı 3. parametrede belirtilmelidir. Bu metodun nasıl kullanıldığını anlatmak için aşağıda verilen kodu hazırladık.

```
string[] ilk_5 = new string[5];
string[] kenar = new string[5];
ilk_5[0] = "Kerem";
ilk_5[1] = "İbrahim";
ilk_5[2] = "Hidayet";
ilk_5[3] = "Mehmet";
ilk_5[4] = "Mirsat";
Array.Copy(ilk_5, kenar, 3);
for (int X=0; X<3; X++)
  listBox1.Items.Add(kenar[X]);
```

Bu kodda önce aynı tip ve sayıda elemana sahip 2 dizi değişken tanımladık. Ardından ilk dizi değişkenin elemanlarına bilgi aktarıp ilk 3 elemanın içeriğini 2. dizi değişkenin aynı sıradaki elemanlarına kopyaladık.

Yukarı verilen Copy() metodu kullanımında kaynak dizi değişkenin ilk elemanından başlamak üzere belirtilen sayıda elemanın içeriği hedef diziye kopyalandı. Kaynak diziden hedef diziye yapılacak kopyalamanın ilk elemandan itibaren yapılmasını istemiyorsanız kopyalamanın hangi elemandan itibaren yapılacağını belirtmelisiniz. Kaynak dizinin 2. ve 3. elemanlarını hedef dizi değişkenin aynı sıra numarasına sahip elemanlarına kopyalamak istiyorsanız Copy() metodu aşağıdaki gibi kullanılabilir.

```
string[] ilk_5 = new string[5];
string[] kenar = new string[5];
ilk_5[0] = "Kerem";
ilk_5[1] = "İbrahim";
ilk_5[2] = "Hidayet";
ilk_5[3] = "Mehmet";
ilk_5[4] = "Mirsat";
kenar[0] = "Ender";
kenar[1] = "Tutku";
kenar[2] = "Haluk";
kenar[3] = "Kerem";
kenar[4] = "Kaya";
Array.Copy(ilk_5,1, kenar, 1, 2);
for (int X=0; X<5; X++)
  listBox1.Items.Add(kenar[X]);
```

Bu örnekte kaynak dizinin 2. elemanından itibaren 2 eleman hedef dizinin 2. ve 3. elemanlarına aktarılır. Array sınıfının dizi değişkenler arası kopyalama yaparken kullanılan Copy() metodundan başka statik olmayan **CopyTo**() metodu bulunmaktadır. Bu metot aşağıdaki gibi kullanıldığında kaynak dizinin içeriği hedef diziye ilk elemandan itibaren kopyalanır.

```
ilk_5.CopyTo(kenar, 0);
```

Dizi Değişkenlerin Elemanlarını Sıralamak

Dizi değişkenlerin içeriklerini sıralamak için kendiniz kod yazabilir veya Array sınıfının **Sort**() metodundan yararlanabilirsiniz. Bu metot sıralanacak dizi değişkeni parametre olarak almaktadır.

```
Array.Sort(Dizi Değişken)
```

Sort() metodu kendisine parametre olarak verilen dizi değişkenin elemanlarını küçükten büyüğe sıralar. Bu metot ile dizi değişkenlerin nasıl sıralandığını anlatmak için 'Gunler' adında bir dizi değişken tanımlayıp haftanın günlerini bu dizi değişkene aktardık.

```
string[] gunler = new string[7];
gunler[0] = "Pazar";
gunler[1] = "Pazartesi";
gunler[2] = "Salı";
gunler[3] = "Çarşamba";
gunler[4] = "Perşembe";
gunler[5] = "Cuma";
gunler[6] = "Cumartesi";
Array.Sort(gunler);
for (int i=0; i<7; i++)
  listBox1.Items.Add(gunler[i]);
```

Bu metot işletildiğinde haftanın günleri alfabetik sıraya konulur ve sıralanmış hali ListBox nesnesine aktarılır. Sort() metodu ile dizi değişkenin bütün elemanları yerine belli sıradaki ve sayıdaki elemanı sıralama işlemine tabi tutmak istiyorsanız sıralanacak ilk elemanı 2. parametre, son elemanı ise 3. parametre olarak vermelisiniz. Sort() metodu aşağıdaki gibi kullanıldığında dizi değişkenin son 4 elemanı sıralama işlemine tabi tutulur.

```
Array.Sort(gunler, 2, 3);
```

Dizi değişkenin ilk elemanın son eleman, son elemanın ise ilk eleman olmak üzere yeniden düzenlemek istiyorsanız Array sınıfının **Reverse**() metodundan yararlanabilirsiniz.

```
Array.Reverse(dizi değişken);
```

Dizi Değişkenlerde Arama Yapmak – BinarySearch Metodu

Dizi değişkenlerde arama yapılırken genellikle Array sınıfının **BinarySearch**() metodundan yararlanılmaktadır. BinarySearch() metodunun aşırı yüklenmiş 4 sürümü bulunmaktadır. En basit sürümü 2 parametreye sahiptir. BinarySearch() metodu ile dizi değişkenlerde arama yapabilmek için değişkenin daha önceden **Sort**() metodu ile sıralanması gerekiyor.

```
Array.BinarySearch(Dizi Değişken, Aranacak Bilgi)
```

Bu metot kendisine 2. parametre olarak verilen bilgiyi dizi değişkende arar. Aranan bilgi bulunamazsa geriye sıfırdan küçük bir değer gönderilir. Aranan bilgi dizide varsa bulunan elemanın dizi içindeki sıra numarası geriye gönderilir. BinarySearch() metodu ile nasıl arama yapıldığını anlatmak için projenin formunu aşağıdaki gibi düzenledik.



Formun **Load** olayını temsil eden metoda yazılacak satırlar sayesinde tanımlanan dizi değişkenin elemanları ListBox'a aktarılacaktır. Dizi değişkeni formdaki bütün metotlarda kullanabilmek için tanımlama satırını metodun dışına aldık.

```
string[] Basket;
private void Form1_Load(object sender, EventArgs e)
{
 Basket = new string[5];
 Basket[0] = "Kaya Peker";
 Basket[1] = "Kerem Tunçeri";
 Basket[2] = "İbrahim Kutluay";
 Basket[3] = "Mirsat Türkcan";
 Basket[4] = "Hidayet Türkoğlu";
 for (int i = 0; i < Basket.Length; i++)
 {
  listBox1.Items.Add(Basket[i]);
 }
}
```

Kullanıcı dizi değişkende aramak istediği oyuncunun adını "Aranan" başlıklı TextBox'a yazıp "Arama Yap" düğmesini tıklayacak. Bu düğme tıklandığında **BinarySearch()** ile arama yapmak için aşağıda verilen kodu yazdık.

```
private void Arama_Yap_Click(object sender, EventArgs e)
{
 Array.Sort(Basket);
 int Sonuc = Array.BinarySearch(Basket, textBox1.Text);
 if (Sonuc <= -1)
   textBox2.Text = "Bulunamadı";
 else
   textBox2.Text = Basket[Sonuc];
}
```

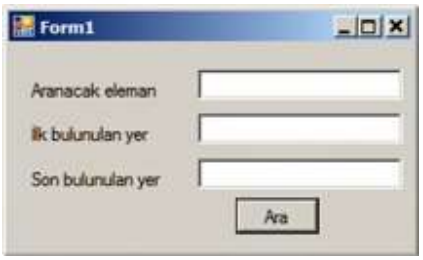
BinarySearch metodu ancak sıralanmış dizilerde arama yapabilir. Dolayısıyla dizi değişkende aynı içeriğe sahip birden fazla eleman varsa bu elemanlar arka arkaya sıralanacaktır.

IndexOf() Metodu İle Arama Yapmak

Dizi değişkenlerde arama yapılırken **BinarySearch()** yerine **IndexOf()** metodu kullanılabilir. Bu metot ile aramaya konu edilen dizi değişkenin sıralı olması şart değildir. Bu metot ile nasıl arama yapıldığını anlatmak için aşağıda verilen kodu yazdık. IndexOf() metodu, aranan elemanın sırasını geriye göndermektedir. Aranan eleman dizide yoksa geriye -1 gönderilir.

```
string[] ilk_5 = new string[5];
ilk_5[0] = "Kerem";
ilk_5[1] = "İbrahim";
ilk_5[2] = "Hidayet";
ilk_5[3] = "Mehmet";
ilk_5[4] = "Mirsat";
string aranan = textBox1.Text;
int sira_no = Array.IndexOf(ilk_5, aranan);
if (sira_no > 0)
  MessageBox.Show("Aranan eleman mevcut");
```

Dizi değişkende içeriği aynı olan birden fazla eleman varsa aynı içeriğe sahip son elemanın sıra numarasını öğrenmek için **LastIndexOf()** metodunu kullanabilirsiniz. Bu metodun nasıl kullanıldığını anlatmak için aşağıdaki gibi bir form hazırladık.



Kullanıcı aramak istediği bilgiyi ilk TextBox'a girip "Ara" düğmesini tıkladığında aranan elemanın dizideki ilk ve son yeri öğrenilip ekrana getirilecek. Bu amaçla hazırladığımız kod aşağıdadır.

```
private void ara_Click(object sender, EventArgs e)
{
 int aranan, ilk_sira, son_sira;
 int[] dizi = new int[5];
 dizi[0] = 15;
 dizi[1] = 20;
 dizi[2] = 40;
 dizi[3] = 30;
 dizi[4] = 20;
 aranan = Int32.Parse(textBox1.Text);
 ilk_sira = Array.IndexOf(dizi, aranan);
 if (ilk_sira < 0)
   MessageBox.Show("Aranan eleman mevcut değil");
 else
   textBox2.Text = ilk_sira.ToString();
 son_sira = Array.LastIndexOf(dizi, aranan);
 if (ilk_sira > 0 && ilk_sira != son_sira)
    textBox3.Text = son_sira.ToString();
}
```

Pratik değeri olmayan bu örnekte önce 5 elemanlı int tipinde bir dizi değişken tanımlayıp elemanlarına bilgi aktardık. Ardından kullanıcının ilk TextBox'a girmiş olduğu elemanı **IndexOf()** metodu ile arayıp bu elemanın sıra numarasını 2. TextBox'a yazdık. En son olarak aynı elemanı dizide bir kez de **LastIndexOf()** metodu ile aradık.

Dizi Değişkenin İçeriğini Birleştirmek – Join() Metodu

Tanımlayıp elemanlarına bilgi aktarmış olduğunuz dizi değişkenin bütün elemanlarının içeriğini **String** sınıfının **Join()** metodundan yararlanıp String değişkende toplamanız mümkündür. Join() metodu 2 parametreye sahiptir.

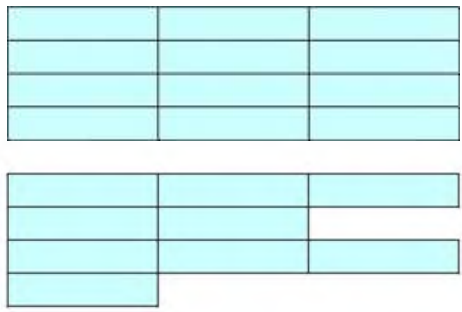
Dizi değişkenin elemanlarını birbirinden ayırmak amacıyla kullanılan 1. parametrede birden fazla karakter kullanılabilinir. Join() metodu geriye dizi değişkenin bütün elemanlarının birleştirilmiş halini göndermektedir. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] Basket = new string[5];
Basket[0] = "Kaya";
Basket[1] = "Kerem";
Basket[2] = "İbrahim";
Basket[3] = "Harun";
Basket[4] = "Mirsat";
string sonuc = String.Join(", ", Basket);
```

Bu şekilde düzenlenen metot işletilince "Basket" adı verilen dizi değişkenin bütün elemanları "sonuc" adlı string değişkene aktarılır. Elemanları birbirinden ayırmak için virgül(,) ile birlikte boşluk karakterini kullandık.

Düzensiz Diziler

Aşağıda verilen 2 tabloyu Word ile hazırladık. İlk tabloda 3 sütun ve 4 satır bulunmaktadır ve her satırda aynı sayıda hücre bulunmaktadır. Aynı sütun sayısına sahip olan 2. tablo ilki gibi düzenli olmayıp her satırda farklı sayıda hücre bulunmaktadır.



Satırlarında farklı sayıda eleman olan dizi değişkenlere düzensiz dizi adı verilmektedir. Şimdi bu tablo ile aynı sayıda satır ve sütuna sahip düzensiz bir dizi değişken tanımlayacağız. Bu tabloda 4 satır olduğu için aşağıda verilen satır ile 4 elemanlı bir dizi değişken tanımladık.

```
public static void Main()
{
 string[][] basket = new string[4][];
}
```

Aslında bu satır ile 4 satırlı ve 2 boyutlu(matris) dizi değişken tanımlanmaktadır. 2 boyutlu bu dizi değişkenin sütunları henüz tanımlı değildir. Şimdi 2 boyutlu bu dizi değişkenin sütunlarını tanımlayacağız. Sütunların eleman sayısı birbirinden farklı olduğu için her sütunu ayrı tanımlamak gerekiyor.

```
string[][] basket = new string[4][];
basket[0] = new string[3];
basket[1] = new string[2];
basket[2] = new string[3];
basket[3] = new string[1];
```

Bu satırlar sayesinde tanımlanan düzensiz dizi değişkenin ilk satırında 3, 2. satırında 2 eleman bulunmaktadır. Bu örnekte düzensiz dizi ve dizinin satırlarını ayrı zamanda tanımla-

dık. Düzensiz dizi değişkenin satırlarının eleman sayısı kod yazılırken belli ise bu işlemi aşağıda yapıldığı gibi tek satırda çözebilirsiniz.

```
string[][] basket={new string[3], new string[2], new string[3], new string[1]};
```

Bu satır, yukarıda verilen 5 satır ile aynı işleve sahiptir. Anlatılan şekilde tanımlanan dizi değişkenin elemanlarına istediğiniz gibi bilgi aktarabilirsiniz. Aşağıda verilen satırlar ile elemanlara bilgi aktarılmaktadır.

```
basket[0][0] = "Kerem";
basket[0][1] = "Harun";
basket[0][2] = "Tutku";
basket[1][0] = "İbrahim";
basket[1][1] = "Hidayet";
basket[2][0] = "Alper";
basket[2][1] = "Mirsat";
basket[2][2] = "Kaya";
basket[3][0] = "Mehmet";
```

Bu şekilde tanımlayıp bilgi aktardığınız dizi değişkenin elemanlarının içeriklerini istediğiniz gibi kullanabilirsiniz. Aşağıda verilen satırlar sayesinde dizi değişkenin ilk satırın içeriği konsola yazılmaktadır.

```
public static void Main()
{
 string[][] basket = new string[4][];
 basket[0] = new string[3];
 basket[1] = new string[2];
 basket[2] = new string[3];
 basket[3] = new string[1];
 basket[0][0] = "Kerem";
 basket[0][1] = "Harun";
 basket[0][2] = "Tutku";
 basket[1][0] = "İbrahim";
 basket[1][1] = "Hidayet";
 basket[2][0] = "Alper";
 basket[2][1] = "Mirsat";
 basket[2][2] = "Kaya";
 basket[3][0] = "Mehmet";
 Console.WriteLine(basket[0][0]);
 Console.WriteLine(basket[0][1]);
 Console.WriteLine(basket[0][2]);
}
```

Konunun iyice anlaşılmasını sağlamak için konsol uygulamasına yazılan bu satırları Visual Studio ile hazırladığımız C# projesine taşıyıp elemanların içeriklerini 4 ayrı ListBox'a aktaracağız. Bu amaçla forma 4 ListBox yerleştirip formun Click olayını temsil eden metoda aşağıda verilen satırları yazıp düzensiz dizinin elemanlarını ListBox'a aktardık.

```
listBox1.Items.Add(basket[0][0]);
listBox1.Items.Add(basket[0][1]);
listBox1.Items.Add(basket[0][2]);
listBox2.Items.Add(basket[1][0]);
listBox2.Items.Add(basket[1][1]);
```

```
listBox3.Items.Add(basket[2][0]);
listBox3.Items.Add(basket[2][1]);
listBox3.Items.Add(basket[2][2]);
listBox4.Items.Add(basket[3][0]);
```

Bu örnekte düzensiz dizi üzerinde işlem yapmak üzere kod yazarken dizi değişkenin satır ve sütun sayısını gözettik. Ancak tanımlanan dizi değişkenin eleman sayısı önceden bilinmeyeceği için kod yazarak düzensiz dizileri bu şekilde kullanmak pratik değildir. Bu satırları metot olarak düzenleyip işletince aşağıdaki gibi bir sonuç elde ettik.



Düzensiz dizi değişkenlerin elemanlarına tanımlama satırında ilk değer atayabilirsiniz. Aşağıda verilen satır bir metoda yazılıp işletilirse düzensiz dizi değişkenin ilk satırında 2, 2. satırında ise 3 elemanı olan bir düzensiz dizi tanımlanmış olur.

```
string[][] basket = new string[][] {new string[]
       {"İbrahim", "Hidayet"}, new string[]
              {"Mehmet", "Mirsat", "Kerem"}};
```

Bu şekilde ilk değer ataması yapılarak tanımlanan düzensiz dizi değişkenin elemanlarının içerikleri aşağıda verilen satırlar ile ekrana yazılabilinir.

```
Console.WriteLine(basket[0][0]);
Console.WriteLine(basket[1][1]);
Console.WriteLine(basket[1][0]);
Console.WriteLine(basket[1][1]);
Console.WriteLine(basket[1][2]);
```

Tanımlayıp değer aktardığınız düzensiz dizi değişkenin kaç satıra sahip olduğunu öğrenmek istiyorsanız Array sınıfının **GetLength**() metodunu kullanabilirsiniz. Aşağıda verilen satırlar bir konsol uygulamasında metot olarak düzenlenip işletilirse 4 değeri elde edilir.

```
string[][] basket = new string [4][];
basket[0] = new string[3];
basket[1] = new string[2];
basket[2] = new string[3];
basket[3] = new string[1];
Console.WriteLine(Convert.ToInt32(basket.GetLength(0)));
```

7. BÖLÜM

TARİH ve ZAMAN İSLEMLERİ

Bu bölümde tarih ve zaman bilgileri üzerinde nasıl işlem yapıldığı konusu üzerinde durulacaktır. Tarih ve zaman bilgileri üzerinde işlem yapılırken **DateTime** yapısından yararlanılmaktadır. Tarih ve zaman bilgileri sıklıkla kullanılmakta, tarih ve zaman bilgilerini başka tiplere dönüştürmeye hep ihtiyaç duyulmaktadır.

Today Özelliği

Tarih ve zaman bilgileri **DateTime** tipindeki değişkenlerde tutulmakta ve bu bilgiler üzerinde işlem yapılırken DateTime yapısının metot ve özelliklerden yararlanılmaktadır. Sistem tarihini elde etmek istiyorsanız DateTime yapısının **Today** özelliğine bakabilirsiniz.

```
textBox1.Text = DateTime.Today.ToString();
```

Bu satır sayesinde TextBox'a sistem tarihi aktarılır. Bu şekilde elde edilen sistem tarihine ayrıca zaman başlangıcı eklenmektedir. **Today** özelliğinden okunan tarih bilgisini TextBox yerine **DateTime** tipindeki değişkene aktarabilirsiniz.

```
DateTime tarih = DateTime.Today;
```

DateTime tipindeki değişkenin içeriğini istediğiniz gibi kullanabilirsiniz. TextBox'a yalnızca tarih bilgisinin aktarılmasını sağlamak için String sınıfının **Substring()** metodunu kullandık.

```
DateTime tarih = DateTime.Today;
string tarihStr = Tarih.ToString();
textBox1.Text = tarihStr.Substring(0, 10);
```

Today özelliği sayesinde elde edilen tarih ve zaman bilgisinin yalnızca tarih kısmı ile ilgileniyorsanız Substring() yerine DateTime yapısının **ToShortDateString()** metodundan yararlanabilirsiniz.

```
DateTime tarih = DateTime.Today;
textBox1.Text = tarih.ToShortDateString();
```

DateTime yapısının Today özelliği ile elde edilen tarih bilgisinin uzun halini kullanmak istiyorsanız ToShortDateString() yerine **ToLongDateString()** metodunu kullanmalısınız. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime tarih = DateTime.Today;
textBox1.Text = tarih.ToShortDateString();
textBox2.Text = tarih.ToLongDateString();
```



ToShortDateString() ve ToLongDateString() metotları geriye string bilgi gönderdiği için bu bilgileri TextBox'a aktarırken dönüştürme yapmaya gerek yoktur.

Tarih bilgilerini string bilgiye dönüştürürken ToShortDateString() ve ToLongDateString() metotlarını kullandığınız gibi **ToString**() metodunu kullanabilirsiniz. DateTime nesnesinden kısa tarih bilgisini almak istiyorsanız "d", uzun tarih elde etmek istiyorsanız "D" biçimleme karakterini ToString() metoduna parametre olarak vermelisiniz. Benzer şekilde "m" biçim karakteri ile gün ve ay bilgisi, "y" karakteri ile ay ve yıl bilgisi elde edilir.

```
DateTime tarih = DateTime.Today;
textBox1.Text = tarih.ToString("d");
textBox2.Text = tarih.ToString("D");
textBox3.Text = tarih.ToString("m");
textBox4.Text = tarih.ToString("y");
```

Now Özelliği

Sistem saatinden hem tarih hem de zaman bilgisini elde etmek istiyorsanız **DateTime** yapısının **Now** özelliğine bakabilirsiniz. Now özelliğinin nasıl kullanıldığını anlatmak için aşağıda verilen satırı yazdık.

```
textBox2.Text = DateTime.Now.ToString();
```

Bu satır sayesinde formdaki 2. TextBox'a hem sistem tarihi hem saati yazılır. Bu satırın işlevini aşağıda görebilirsiniz. Yukarıda yapıldığı gibi **Substring**() metodu ile DateTime tipindeki bilginin istediğiniz kısmını alıp başka amaçlarla kullanabilirsiniz.



Now özelliği sayesinde ile elde edilen sistem tarih ve saatinin yalnızca zaman kısmı ile ilgileniyorsanız **ToShortTimeString**() metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime zaman = DateTime.Now;
textBox1.Text = zaman.ToShortTimeString();
```

Bu metot işletildiğinde sistem saatinin zaman bilgisi TextBox'a aktarılır. ToShortTimeString() metodu ile elde edilen zaman bilgisinde saniye olmaz. Zaman bilgisinin saniye kısmı ile ilgileniyorsanız **ToLongTimeString**() metodunu kullanmalısınız.

```
DateTime zaman = DateTime.Now;
textBox1.Text= zaman.ToShortTimeString();
textBox2.Text= zaman.ToLongTimeString();
```

String Bilgiyi DateTime Tipine Dönüştürmek

Yukarıda DateTime yapısının Today ve Now özelliklerinin içeriklerini DateTime tipindeki değişkenlere aktardık. TextBox'lara girilen bilgileri DateTime tipine dönüştürebilirsiniz. Bu işlemi yaparken **Convert** sınıfının **ToDateTime**() metodunu aşağıdaki gibi kullanabilirsiniz.

```
DateTime tarih = Convert.ToDateTime("30.12.2017");
DateTime saat = Convert.ToDateTime("12:45");
```

Zaman bilgilerini String bilgiye dönüştürürken **ToLongTimeString** ve **ToShortTimeString** metotlarını kullandığınız gibi ToString() metodunu kullanabilirsiniz. ToString() metoduna "t" karakteri parametre olarak verilirse DateTime bilgisinden kısa saat elde edilir. Benzer şekilde "T" karakteri ile uzun zaman bilgisi elde edilir.

```
DateTime zaman = DateTime.Now;
textBox1.Text = zaman.ToString("t");
textBox2.Text = zaman.ToString("T");
textBox3.Text = zaman.ToString("f");
textBox4.Text = zaman.ToString("F");
textBox5.Text = zaman.ToString("g");
```

DayOfWeek Özelliği

Sistem tarihinden veya tarihsel bir bilgiden yararlanarak haftanın gününü öğrenmek istiyorsanız DateTime yapısının bu özelliğine bakabilirsiniz. **DayOfWeek** özelliğinin nasıl kullanıldığını göstermek için aşağıda verilen kodu hazırladık.

```
DateTime Tarih = DateTime.Today;
int gun = Convert.ToInt32(Tarih.DayOfWeek);
textBox1.Text = Tarih.ToString().Substring(0, 10);
textBox2.Text = Convert.ToString(gun);
```



Bu kodda Today özelliğinin içeriğini DateTime tipindeki değişkene aktardıktan sonra sistem tarihini formdaki ilk TextBox'a aktardık. En son olarak "Tarih" nesnesinin **DayOfWeek** özelliğinin içeriğini okuyup 2. TextBox'a aktardık.

Bu şekilde haftanın günü öğrenildikten sonra kullanıcı günün Türkçe adı konusunda bilgilendirilebilinir. Bunun için Türkçe gün adlarından meydana gelen bir dizi değişken tanımlanır. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
string[] Gunler={ "Pazar", "Pazartesi", "Salı",
                  "Çarşamba", "Perşembe","Cuma", "Cumartesi" };
DateTime Tarih = DateTime.Today;
textBox1.Text = Tarih.ToString().Substring(0, 10);
int gun = Convert.ToInt32(Tarih.DayOfWeek);
textBox2.Text = Convert.ToString(gun);
textBox3.Text = Convert.ToString(Gunler[gun]);
```

DayOfYear Özelliği

Elinizdeki tarih bilgisinin yılın kaçıncı günü olduğunu öğrenmek istiyorsanız **DayOfYear** özelliğine bakabilirsiniz. Bu özelliğin nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime Tarih = DateTime.Today;
int gun = Tarih.DayOfYear;
textBox1.Text = Tarih.ToShortDateString();
textBox2.Text = Convert.ToString(gun);
```

Bu kod işletildiğinde geçerli tarih bilgisi ilk TextBox'a, bu tarih bilgisinin gün kısmı ise 2. TextBox'a aktarılır. Aşağıda verilen ekran görüntüsünü bu kodu işlettikten sonra aldık.



Bu örnekte DateTime tipindeki değişkenin **DayOfYear** özelliğiyle ilgilendik. Dolayısıyla yılbaşından bu güne kadar geçen gün sayısını öğrendik. DateTime tipindeki değişkene başka bir tarihi aktarıp yılbaşından o tarihe kadar geçen gün sayısını öğrenebilirsiniz. Bu özellikten yararlanarak 2 tarih arasındaki gün sayısı öğrenilebilir. Bu işlemin nasıl yapıldığını anlatmak için forma 3 TextBox yerleştirdik.



Formdaki ilk TextBox'a geçerli sistem tarihini, 2. TextBox'a ise en son ödeme tarihini yazacağız. Ödemenin kaç gün geç yapıldığını bulup 3. TextBox'a aktaracağız. Bu amaçla formun Load olayını temsil eden metoda aşağıda verilen satırları yazıp proje çalıştırıldığında geçerli tarihin ilk TextBox'a yazılmasını sağladık.

```
private void Form1_Load(object sender, EventArgs e)
{
 DateTime Tarih = DateTime.Today;
 textBox1.Text = Tarih.ToShortDateString();
}
```

Projeyi çalıştırılıp son ödeme tarihi 2. TextBox'a girilip form tıklandığında ödemenin kaç gün geciktiğini öğrenip ilgili TextBox'a yazmak için aşağıda verilen kodu hazırladık. Bu kodda **DayOfYear** özelliğine 2 kez bakılmaktadır. İlk kullanımda yılbaşından son ödemeye kadar gün sayısı öğreniliyor. DayOfYear özelliğinin 2. kez kullanıldığı satırda ise yılbaşından bugüne kadar geçen gün sayısı öğrenilmektedir. Bu 2 tarih arasındaki gün farkı gecikmedir.

```
private void Form1_Click(object sender, EventArgs e)
{
 DateTime Bugun = DateTime.Today;
 textBox1.Text = Bugun.ToShortDateString();
 DateTime Son_gun = Convert.ToDateTime(textBox2.Text);
 int ilk = Bugun.DayOfYear;
 int son = Son_gun.DayOfYear;
 textBox3.Text = Convert.ToString(ilk-son);
}
```

Day, Month ve Year Özellikleri

Sistem tarihinin gün bilgisini öğrenmek için DateTime yapısının **Day** özelliğini bakabilirsiniz. Geçerli tarihin ay bilgisini öğrenmek istiyorsanız **Month** özelliğine, yıl bilgisini öğrenmek için ise **Year** özelliğine bakabilirsiniz. Bu özelliklerin nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime Tarih = DateTime.Today;
textBox1.Text = Tarih.ToString().Substring(0,10);
textBox2.Text = Tarih.Day.ToString();
textBox3.Text = Tarih.Month.ToString() ;
textBox4.Text = Tarih.Year.ToString();
```

Bu kod sayesinde tarih bilgisi gün, ay ve yıl şeklinde parçalanmış oldu. Tarih bilgisinden yola çıkıp ay adını öğrenmek istiyorsanız Visual Basic kökenli **MonthName()** metodunu kullanabilirsiniz. Bu metot parametre olarak ay bilgisini alıp geriye ay adını göndermektedir.

```
DateTime Tarih = DateTime.Today;
int ay = Tarih.Month;
textBox4.Text = Microsoft.VisualBasic.DateAndTime.MonthName(ay, false);
```

MonthName() metoduna 2. parametre olarak true vermeniz halinde ay adının kısa hali elde edilir. Ayın Türkçe adını Visual Basic'in **MonthName** metodu sayesinde öğrenmek yerine Türkçe ay adlarını bir dizi değişkene aktarıp bu dizi değişkenden yararlanabilirsiniz.

DateSerial() Metodu

Gün, ay ve yıl bilgilerinden yola çıkıp tarih bilgisi oluşturmak istiyorsanız Visual Basic'in **DateSerial()** metodundan yararlanabilirsiniz. Bu metot yıl, ay ve gün bilgilerini parametre olarak alıp geriye tarih bilgisini göndermektedir.

```
int gun=24;
int ay =03;
int yil=2012;
DateTime Tarih;
Tarih = Microsoft.VisualBasic.DateAndTime.DateSerial(yil,ay, gun);
textBox1.Text = Convert.ToString(Tarih);
```

DateSerial() metodu sayesinde elde edilen tarih bilgisine zaman başlangıcı eklenmektedir. Bu nedenle bu satırlar bir metoda yazılıp işletilirse aşağıdaki gibi bir sonuç elde edilir.



Substring() veya **ToShortDateString()** metodundan yararlanıp tarihe eklenen zaman başlangıcını silebilirsiniz. Yukarıda verilen koddaki son satır aşağıdaki gibi değiştirilirse TextBox'a yalnızca tarih bilgisi aktarılır.

```
textBox1.Text = Convert.ToString(Tarih).Substring(0,10) ;
```

AddDays(), AddMonths() ve AddYears() Metotları

Değişik şekilde elde edilmiş tarih bilgisine gün ekleyip veya çıkarıp başka bir tarih bilgisini elde ederken DateTime yapısının bu metotlarını kullanabilirsiniz. Bu metotlar gün, ay veya yıl eklenmek veya çıkarılmak istenen tarih bilgisinin yanına yazılmaktadır.

```
DateTime Bugun, Sonra, Once;
Bugun = DateTime.Now;
textBox1.Text = Convert.ToString(Bugun).Substring(0, 10);
Sonra = Bugun.AddDays(5);
textBox2.Text = Convert.ToString(Sonra).Substring(0, 10);
Once = Bugun.AddDays(-10);
textBox3.Text = Convert.ToString(Once).Substring(0, 10);
```

Bu kod işletilirse formdaki ilk TextBox'a geçerli sistem tarihi ve 2. TextBox'a ise 5 gün sonrasının tarihi yazılır. **AddDays**() metoduna eksi(-) bir değeri parametre olarak verirseniz tarih bilgisine gün eklenmeyip çıkarılır.

Tarih bilgisine ay eklemek veya çıkarmak istiyorsanız **AddMonths**(), yıl eklemek veya çıkarmak istiyorsanız **AddYears**() metodunu kullanabilirsiniz. Bu metotların nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime Bugun, Tarih1, Tarih2;
Bugun = DateTime.Now;
Tarih1 = Bugun.AddMonths(2);
Tarih2 = Bugun.AddYears(1);
```

DaysInMonth() ve IsLeapYear() Metotları

Herhangi bir yılın herhangi bir ayının kaç güne sahip olduğunu öğrenmek istiyorsanız DateTime yapısının **DaysInMonth**() metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını anlatmak için forma 3 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
int yil = Convert.ToInt32(textBox1.Text);
int ay = Convert.ToInt32(textBox2.Text);
int gun = DateTime.DaysInMonth(yil, ay);
textBox3.Text = gun.ToString();
```

DaysInMonth() metodu 2 parametreye sahiptir. İlk parametrede yıl, 2. parametrede gün sayısı öğrenilmek istenen ay verilmektedir. Formdaki ilk TextBox'a yılı, 2. TextBox'a gün sayısını öğrenmek istediğimiz ayı yazıp bu kodu işletince aşağıdaki gibi bir sonuç elde ettik.



4 yılda bir şubat ayı 29 çekmekte ve dolayısıyla o yılda 366 gün olmaktadır. Belirtilen yılın 366 güne sahip olup olmadığını öğrenmek istiyorsanız **IsLeapYear**() metodunu kullanabilirsiniz. Bu metot parametre olarak aldığı tarih bilgisinin yılı 366 gün çekiyorsa geriye true, değilse false göndermektedir. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
int yil = Convert.ToInt32(textBox1.Text);
bool artik = DateTime.IsLeapYear(yil);
if (artik == true)
  MessageBox.Show("Bu yıl şubat 29 çekiyor");
```

Hour, Minute ve Second Özellikleri

DateTime yapısının bu özelliklerden yararlanarak zaman bilgisinin saat, dakika ve saniye bileşenlerini öğrenebilirsiniz. Bu özelliklerin nasıl kullanıldığını göstermek için aşağıda verilen kodu hazırladık.

```
DateTime Zaman = DateTime.Now;
textBox1.Text = Zaman.ToString().Substring(11, 8);
textBox2.Text = Zaman.Hour.ToString();
textBox3.Text = Zaman.Minute.ToString();
textBox4.Text = Zaman.Second.ToString();
```

Bu satırlarda önce **DateTime** tipinde bir değişken tanımladık ve bu değişkene **Now** özelliğinden yararlanarak sistem tarih ve zaman bilgisini aktardık. Saat, dakika ve saniye bilgilerini Visual Basic kaynaklı **TimeSerial**() metoduna parametre olarak verip zaman bilgisi elde edebilirsiniz. Bu metodun nasıl kullanıldığını anlatmak için forma 4 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
int saat, dakika, saniye;
DateTime Zaman = DateTime.Now;
saat = Convert.ToInt32(textBox1.Text);
dakika = Convert.ToInt32(textBox2.Text);
saniye = Convert.ToInt32(textBox3.Text);
Zaman=Microsoft.VisualBasic.DateAndTime.TimeSerial(saat,
              dakika, saniye);
textBox4.Text = Convert.ToString(Zaman.ToLongTimeString());
```

Kullanıcı zaman bilgisini saat, dakika ve saniye cinsinden TextBox'lara girecek. Girilen bu bilgiler **TimeSerial**() metodu ile birleştirilip formdaki en son TextBox'a aktarılacaktır. Zaman bilgisinin mili saniye kısmını elde etmek istiyorsanız zaman nesnesinin **Milisecond** özelliğinden yararlanabilirsiniz.

AddHours(), AddMinutes() ve AddSeconds() Metotları

Zaman bilgisine saat, dakika veya saniye ekleme ve çıkarma gereğini duyduğunuzda bu metotları kullanabilirsiniz. Bu metotlar zaman bilgisini içeren değişken ile birlikte kullanılır ve eklenecek veya çıkarılacak saat, dakika ve saniyeyi parametre olarak alırlar.

```
Zaman Bilgisi.AddHours(Artım-Eksiltme Değeri )
```

Bu metotların nasıl kullanıldığını anlatmak için forma 3 TextBox yerleştirip aşağıda verilen satırları hazırladık. Bu satırlarda **DateTime** tipinde 3 değişken tanımlayıp ve **Now** özelliği ile sistem saatini "simdi" adını verdiğimiz değişkene aktardık. Sistem saatine saat eklemek için **AddHours()** ve 30 dakika çıkarmak için **AddMinutes()** metodunu kullandık.

```
DateTime Simdi, Zaman1, Zaman2;
Simdi = DateTime.Now;
Zaman1 = Simdi.AddHours(1);
Zaman2 = Simdi.AddMinutes(-30);
```

Parse() Metodu İle String Bilgileri Tarih ve Saate Çevirmek

Kullanıcının TextBox'a veya başka bir nesne aracılığıyla girmiş olduğu string bilgiyi tarih ve zaman olarak değerlendirmek istiyorsanız DateTime yapısının **Parse()** metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
DateTime Tarih = DateTime.Parse(textBox1.Text);
```

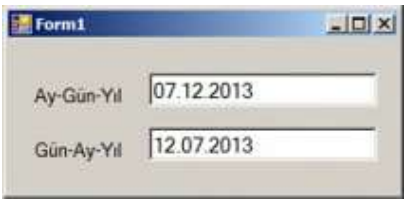
TextBox'a girilen bilgi tarihsel değilse bu satır işletildiğinde hata meydana gelir. Geçersiz tarih girilip Parse() metodu ile dönüştürme yapılırken meydana gelecek hatalardan dolayı programın kırılmasını engellemek için aşağıdaki gibi **try-catch** bloğu hazırlanabilir.

```
DateTime Tarih;
try
 {
  Tarih = DateTime.Parse(textBox1.Text);
 }
catch
 {
  MessageBox.Show("Girilen tarih geçersiz");
 }
```

Bu satırları yazarken kullandığımız bilgisayarda tarih formatı gün-ay-yıl şeklindeydi. Dolayısıyla TextBox'a Ay-Gün-Yıl formatına uygun tarih girildiğinde **Parse()** metodu bu şekli ile kullanıldığında dönüştürme işlemini yapamaz. Parse() metoduna 1. parametre olarak verilecek tarih bilgisinin formatını 2. parametrede belirtebilirsiniz. Bu konuda bilgi vermek için forma 2. bir TextBox yerleştirip aşağıda verilen kodu yazdık.

```
System.Globalization.CultureInfo ulke;
ulke = new System.Globalization.CultureInfo("en-US");
DateTime Tarih;
try
 {
  Tarih = DateTime.Parse(textBox1.Text, ulke);
  textBox2.Text = Tarih.ToShortDateString();
 }
catch
 {
  MessageBox.Show("Girilen tarih geçersiz");
 }
```

Bu kod sayesinde formdaki ilk TextBox'a ay-gün-yıl formatında girilmiş tarih bilgisi DateTime yapısının **Parse()** metodu ile DateTime tipindeki değişkene aktarılır. Ardından bu değişkenin içeriği String'e çevrilip 2. TextBox'a aktarılmaktadır.



Ticks Özelliği

DateTime yapısının bu özelliğinden yararlanılarak bir işlemin ne kadar sürdüğünü öğrenebilirsiniz. Bu özellik hakkında bilgi vermek için aşağıda verilen 2 satırı yazdık.

```
long Tick_sayi = DateTime.Now.Ticks;
textBox1.Text = Tick_sayi.ToString();
```

Ticks özelliği 1 Ocak 0001'den içinde bulunulan ana kadar geçen süreyi tick cinsinden içermektedir. Bu özelliğin içeriği her 100 nano saniyede 1 artmaktadır.



Şimdi 1 Ocak 0001'den bu güne kadar geçen süreyi saniye, dakika, saat, gün ve yıl ile ifade edeceğiz. Bu amaçla forma 6 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
long Saniye, Dakika, Saat, Gun, Yil;
long Tick_sayi = DateTime.Now.Ticks;
textBox1.Text = Tick_sayi.ToString();
Saniye = Tick_sayi / 10000000;
textBox2.Text = Saniye.ToString();
Dakika = Saniye / 60;
textBox3.Text = Dakika.ToString();
Saat = Dakika / 60;
textBox4.Text = Saat.ToString();
Gun = Saat / 24;
textBox5.Text = Gun.ToString();
Yil = Gun / 365;
textBox6.Text = Yil.ToString();
```

Ticks özelliğinin içeriği her 100 nano saniyede bir arttığı için içeriğini 10.000.000'e bölerek 1 Ocak 0001'den bugüne kadar geçen süreyi saniye olarak ifade ettik. Devamında saniyeyi dakika, saat, gün ve yıla çevirdik.

Şimdi ise bir işlemin ne kadar sürdüğünü **Ticks** özelliğinden yararlanarak bulacağız. Bu amaçla bir for döngüsü hazırlayıp işlemin başında ve sonunda Ticks özelliğine bakıp geçen süreyi hesapladık. Ticks özelliğinden yararlanıp bekleme durumları meydana getirebilirsiniz.

```
long Tick_ilk, Tick_son, Tick_fark;
long Sayi = 0;
Tick_ilk = DateTime.Now.Ticks;
for (int i = 0; i < 1000000000; i++)
 {
  Sayi = Sayi + i;
 }
Tick_son = DateTime.Now.Ticks;
Tick_fark = Tick_son - Tick_ilk;
long Saniye = Tick_fark / 10000000;
textBox1.Text = Saniye.ToString();
```

TimeSpan Yapısı

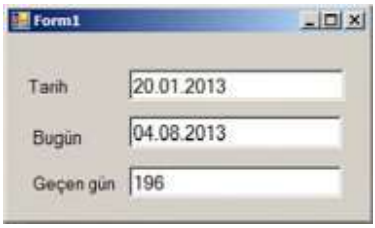
Ticks özelliğinin içeriğini kullanırken ve tarihsel bilgiler üzerinde işlem yaparken **TimeSpan** yapısından yararlanabilirsiniz. Bu konuda bilgi vermek için aşağıda verilen kodu hazırladık. Bu satırlar sayesinde 2 tarih arasındaki farkı gün birimi ile ifade edebileceğiz. Bu amaçla forma 2 TextBox yerleştirip aşağıda verilen odu hazırladık.

```
DateTime Tarih = DateTime.Parse(textBox1.Text);
DateTime Bugun = DateTime.Today;
textBox2.Text = Bugun.ToString("d");
```

Şimdi ise bir TimeSpan nesnesi hazırlayacağız. TimeSpan yapısının aşırı yüklenmiş 4 yapıcı metodu bulunmaktadır. İlk olarak **Ticks** bilgisini parametre olarak alan yapıcı metodu kullanacağız. Bu amaçla DateTime yapısının **Ticks** özelliğinden yararlanarak kullanıcının formdaki ilk TextBox'a girmiş olduğu tarihten yola çıkarak bir Tick değeri elde ettik.

```
DateTime Tarih, Bugun;
Tarih = DateTime.Parse(textBox1.Text);
Bugun = DateTime.Today;
textBox2.Text = Bugun.ToString("d");
long ilk_tick, Bugun_tick;
ilk_tick = Tarih.Ticks;
Bugun_tick = Bugun.Ticks;
System.TimeSpan Zaman;
Zaman = new TimeSpan(Bugun_tick -ilk_tick);
textBox3.Text = Zaman.Days.ToString();
```

DateTime sınıfının **Now** özelliğinden yararlanıp elde ettiğimiz geçerli tarihten de ikinci bir Tick değeri elde ettik. Bu 2 tick değerinin arasındaki farkı TimeSpan'ın yapıcı metoduna parametre olarak verip iki tarih arasındaki farkı TimeSpan olarak elde ettik. TimeSpan nesnesinden gün bilgisi elde ederken **Days** özelliğinden yararlandık. Formdaki ilk TextBox'a bilgi girilip bu kod işletildiğinde aşağıdaki gibi bir sonuç alınır.



TimeSpan'dan yararlanarak tarih ve zaman bilgilerini değişik şekilde ifade etmek mümkündür. Bu 2 tarih arasındaki farkı saat birimi ile ifade etmek istemiş olsaydık TimeSpan nesnesinin **TotalHours** özelliğinden yararlanırdık. TimeSpan yapısı ayrıca **TotalMinutes**, **TotalSeconds** ve **TotalMiliSeconds** özelliklerine sahiptir.

Yukarıda TimeSpan nesnesini hazırlarken parametre olarak iki tarih arasındaki zaman farkını Tick birimi ile vermiştik. Aşağıda ise TimeSpan nesnesini hazırlarken DateTime sınıfının **Subtract** metodunu kullandık. Bu metot geriye TimeSpan nesnesi göndermektedir.

```
DateTime Tarih, Bugun;
Tarih = DateTime.Parse(textBox1.Text);
Bugun = DateTime.Today;
textBox2.Text = Bugun.ToString("d");
System.TimeSpan Zaman = Bugun.Subtract(Tarih);
textBox3.Text = Zaman.TotalDays.ToString();
```

Şimdi ise forma 3 TextBox ve 2 düğme yerleştirip kullanıcının "Baslat" adı verilen düğmeyi tıklaması ile "Durdur" düğmesini tıklaması arasında geçen süreyi mili saniye olarak hesaplayıp 3. TextBox'a aktaracağız. Baslat düğmesi için hazırlanan kodu aşağıda verdik.

```
private void Baslat_Click(object sender, EventArgs e)
{
 DateTime Zaman = DateTime.Now;
 textBox1.Text = Zaman.ToLongTimeString();
}
```

Bu kod sayesinde o anki zaman bilgisi ilk TextBox'a aktarılır. "Durdur" adı verilen 2. düğmeyi tıklanıncaya kadar aradan geçen süreyi hesaplamak üzere aşağıdaki kodu yazdık.

```
private void Durdur_Click(object sender, EventArgs e)
{
 DateTime Zaman = DateTime.Now;
 textBox2.Text = Zaman.ToLongTimeString();
 DateTime saat = DateTime.Parse(textBox1.Text);
 DateTime simdi = DateTime.Now;
 long ilk_tick, son_tick;
 ilk_tick = saat.Ticks;
 son_tick = simdi.Ticks;
 System.TimeSpan fark;
 fark = new TimeSpan(son_tick - ilk_tick);
 textBox3.Text = fark.Milliseconds.ToString();
}
```

Format() Metodu İle Tarih ve Zaman Bilgilerini Biçimlemek

Tarih ve zaman bilgilerini biçimlerken Format() metodundan yararlanabilirsiniz. Aşağıda verilen kod bir metoda yazılıp işletilirse sistem tarihi belirlenen formata dönüştürülüp TextBox'lara aktarılır.

```
DateTime Tarih = DateTime.Today;
string str1 = String.Format("{0:d}", Tarih);
string str2 = String.Format("{0:D}", Tarih);
textBox1.Text = str1;
textBox2.Text = str2;
```

Format() metodu 2 parametreye gerek duymaktadır. İlk parametrede biçimleme yapılırken kullanılacak biçim kodu verilmektedir. 2. parametrede ise biçimlenecek veya dönüştürülecek tarih, zaman ve sayısal bilgi verilmektedir. Bu örnekte DateTime tipinden kısa tarih elde etmek için d'yi, uzun tarihi elde etmek için ise D'yi kullandık.

DateTime tipindeki değişkene tarih ile birlikte zaman bilgisini aktarırken **"t"** ve **"T"** biçim kodları ile zaman bilgisini kısa ve uzun olarak gösterebilirsiniz. **"f" ve "F"** biçim kodu ile hem tarih hem de zaman bilgisi elde edilmektedir.

```
DateTime Tarih = DateTime.Now;
string str1 = String.Format("{0:t}", Tarih);
string str2 = String.Format("{0:T}", Tarih);
string str3 = String.Format("{0:F}", Tarih);
```

Yalnızca sistem saatine gerek duyuyorsanız **TimeOfDay**() metodunu kullanabilirsiniz. Tarih ve zaman bilgilerini formatlarken kullanılan bazı kodlar aşağıda tablo halinde verildi.

Format Kodu	İşlevi	Örnek
D	Kısa tarih	23.12.2013
D	Uzun tarih	23 Aralık 2013 Pazartesi
F	Uzun tarih ve kısa zaman birlikte	23 Aralık 2013 Pazartesi 15:24
F	Uzun tarih ve uzun zaman birlikte	23 Aralık 2013 Pazartesi 15:24:43
Dd	Sadece tarihin günü	23
Ddd	Günün kısa adı	Pzt
Dddd	Günün tam adı	Pazartesi
M	Ay bilgisi	23 Aralık
MM	Ay tek basamaklı ise 01 şeklinde	12
MMM	Ayın kısa adı	Ara
MMMM	Ayın tam adı	Aralık
Yy	Yıl bilgisi 2 haneli olarak gösterilir	23
Yyyy	Yıl bilgisi 4 haneli olarak gösterilir	2013
T	Zaman bilgisinin saati gösterilir	15:33
Mm	Zaman bilgisinin dakikası gösterilir	33

8. BÖLÜM
OPERATÖRLER

Operatörler; bir, iki veya üç bilgi (operand) üzerinde işlem yapıp bir sonuç üreten sembol veya işaretlerdir. Bazı operatörler sadece bir bilgi üzerinde işlem yapıp bir sonuç üretirken bazıları ise bir sonuç üretmek veya değerlendirme yapabilmek için 3 bilgiye ihtiyaç duymaktadır. Bir bilgi üzerinde işlem yapılabilinen operatörlere **unary** (tekli), değerlendirme yaparken 2 bilgiye gerek duyan operatörlere **binary** (ikili) ve 3 parametreye gerek duyan operatörlere **ternary** (üçlü) denilmektedir.

Örneğin aşağıda verilen kodda atama (=) operatörü kullanılmıştır. Eşittir işaretinin sağında (1907) ve solundakiler (i değişkeni) birer operanddır. Dolayısıyla atama (=) operatörü ikili bir operatördür.

```
public static void Main()
{
 int i;
 i = 1907;
 Console.WriteLine(i);
}
```

Bazen bir sayıyı eksi yapmak için önüne eksi (-) operatörü konulur. Başka bir deyişle eksi operatörü tek operanda sahip olduğu için tekli operatörlere örnektir.

```
public static void Main()
{
 sbyte N1 = 20;
 sbyte N2 = -10; // tekli operatör
 byte toplam = (byte) (N1 + N2);
 Console.WriteLine(toplam);
}
```

Bu koddan şu sonucu çıkarabiliriz: Günlük hayatta olduğu gibi "–" operatörü 2 amaçla kullanılmaktadır. Bu operatör tek operand ile kullanıldığında sağına yazılan operandın içeriğini eksi yaparken yani işaret operatörü olurken, 2 operand ile kullanıldığında ise çıkarma işlemini yapmaktır.

C# derleyicisi, toplama ve çıkarma operatörlerini ifade içerisindeki konumlarına bakarak artı(+) ve eksinin(-) hangi amaçla kullanıldığını doğru tahmin etmektedir. Bu konu üzerinde düşünmenizi sağlamak için aşağıda verilen kodu yazdık. Burada yapılan işlemin yorumunu size bırakıyoruz.

```
public static void Main()
{
 int number1 = 10;
 int number2 = 20;
 int sonuc = -number1+-number2;
 Console.WriteLine(sonuc);
}
```

Yukarıda operatörleri operand sayılarına göre gruplad ık. Operatörler genellikle işlevlerine göre gruplanmaktadır. Çünkü kod yazarken sıklıkla kullanmak durumunda kalacağınız operatörleri işlevlerine göre gruplamak iyi bir fikirdir: Matematiksel, karşılaştırma, mantıksal, atama, bitsel ve özel operatörler ilk akla gelen grup adlarıdır.

Karşılaştırma operatörleri aşağıda tablo halinde verildi. Operatörlerin öncelik sıraları genellikle matematikteki gibidir. C# kodunda karşılaştırma ve kontrol işlemleri parantezlerin içinde yapılmaktadır. Bu nedenle aşağıda tabloda verilen örnek kullanımlar parantezlerin içinde gösterildi.

Operatör	İşlevi	Örnek
=	Atama	sayi = 10
==	Eşittir	if (a == b)
!=	Eşit değilldir	if (a != b)
<	Küçüktür	if (a < b)
>	Büyüktür	if (a > b)
<=	Küçük eşittir	if (a <= b)
>=	Büyük eşittir	if (a >= b)

Bu operatörlerden eşitlik operatörüne (==) dikkat edilmesi gerekmektedir. Çünkü bu operatör karşılaştırma yaparken nesnelerin bellekteki adreslerine bakmaktadır. Başka bir deyişle bu operatörle eşitlik kontrolü yapılırken referans tipteki değişkenlerin aynı nesneyi referans edip etmediğinin kontrolü yapılmaktadır. Karşılaştırılan iki değişken aynı nesneyi referans ediyorsa geriye true yoksa false gönderilmektedir. Referans tipteki 2 değişkenin ayni nesneyi gösterip göstermediğinin kontrolü yapılırken "==" operatörü yerine **Equals**() metodu kullanılabilir. Bu konudaki ayrıntıları "Object Sınıfı" adlı bölümünde bulabilirsiniz.

Burada listelenen operatörlerin C# uygulamaları dahilinde kullanılan her türlü bilgiyi parametre olarak alamadığını bilmek gerekir. Örneğin büyüktür (>) operatörü int, byte, double, char gibi standart veri tiplerini parametre olarak alabilirken **List** nesnelerini, string ve dizi değişkenleri parametre olarak alıp karşılaştırma yapmayı kabul etmez. Tabi derseniz ki "ben '<' operatörü ile 2 string bilgi veya 2 dizi değişkeni mutlaka karşılaştıracağım"; o zaman bu operatörü aşırı yüklemeniz gerekecektir. Bu konuda bilgiyi ilgili bölümde bulabilirsiniz.

```
public static void Main()
{
 string str1 = "Adana";
 string str2 = "Ankara";
 bool sonuc = str1 > str2; // hata meydana gelir.
 int[] dizi1 = new int[] {100, 200, 300 };
 int[] dizi2 = new int[] { 10, 20, 30 };
 bool kucuk = dizi1 > dizi2; // hata meydana gelir.
}
```

Aynı işlem veya satırda birden fazla karşılaştırma yapılmak istendiğinde mantıksal operatörlerden yararlanılır. Mantıksal operatörler aşağıda tablo halinde verildi.

Operatör	İşlevi	Örnek
&&	Ve	if (a > b && c=10)
||	Veya	if (a <= b || C>100)
!	Değil	if (a != b)

Yukarıda tablo halinde verilen karşılaştırma ve mantıksal operatörlerden başka matematiksel operatörler vardır. Aşağıda tablo halinde verilen bu operatörler matematiksel işlemlerde kullanılan operatörler ile aynı işleve sahiptir. Matematiksel operatörleri kullanırken parantezlere başvurmak şart değildir.

Operatör	İşlevi	Örnek
+	Toplama	a = b + c;
-	Çıkarma	a = b – c;
*	Çarpma	sonuc = a * b;
/	Bölme	sonuc = 100 / 10;
%	Kalan bulma	sonuc = 40 % 6; // sonuc 4 olur.

Bu listedeki artı (+) operatörüne operand olarak sayısal bilgiler verildiğinde bu bilgiler matematikteki gibi toplanırken, kendisine 2 string bilgi operand olarak verildiğinde bu bilgiler birleştirilmektedir. Başka bir deyişle "+" opetatörü string bilgiler için aşırı yüklenmiştir. Yani "+" operatörü şartlara göre 2 farklı işlemi yapabilmektedir.

Şimdi bu listede verilen toplama (+) operatörü üzerinden C#'ın önemli bir özelliğinden söz edeceğiz. Daha önce "Değişkenler" adlı bölümde söz edildiği üzere byte değişkenlere en büyük 255 sayısı aktarılabilir. Buna rağmen aşağıda verilen kod hataya neden olur. Çünkü byte tipinde 2 bilgi toplandığında çıkan sonucun integer olacağı varsayılmaktadır.

```
static void Main(string[] args)
{
 byte N1 = 20;
 byte N2 = 10;
 byte toplam = N1 + N2; // hata meydana gelir.
}
```

Burada karşılaşılan hata için şu söylenebilir: C#'ta aritmetik operatörler 2 sayısal bilgi üzerinde işlem yaptıklarında çıkacak sonuç en küçük **Integer** olabilir. Bu hatayı gidermek için byte tipindeki 2 operanda sahip atama operatörünün geriye gönderdiği integer bilgi byte bilgiye aşağıdaki gibi dönüştürülebilir. Burada içeriği toplanan değişken veya sayılardan birisi **long** olsaydı sonuç **long** olurdu.

```
static void Main(string[] args)
{
 byte N1 = 20;
 byte N2 = 10;
 byte toplam = (byte) (N1 + N2);
}
```

Benzer sorunla bölme işlemi yapılırken de karşılaşılır. Örneğin aşağıda yapıldığı gibi iki byte bilgi birbirine bölünmek istendiğinde hata meydana gelir. Çünkü toplama işleminde olduğu gibi bölünmek istenen bilgiler byte olsa bile sonuç integer olur.

```
byte N1= 20;
byte N2= 10;
byte sonuc = N1/N2;  // hata meydana gelir.
```

Burada **integer** bilgiyi **byte** değişkene aktarmak istediğimiz için hata meydana gelir. Bu nedenle bu hatayı gidermek için yukarıda yapıldığı gibi cast işleminin yapılması gerekir. Bölme operatörüyle ilgili ayrıntılardan söz etmek amacıyla aşağıda verilen kodu hazırladık.

```
static void Main(string[] args)
{
 int i1 = 12;
 int i2 = 5;
 int sonuc = i1/i2;
 Console.WriteLine(sonuc);
}
```

Bu kod işletildiğinde tahmin edileceği üzere sonuç 2 olur. Çünkü hem bölen ve bölünen hem de çıkan sonucun aktarıldığı değişken tam sayıdır. Çıkan sonucun aktarıldığı değişken **float** olsaydı sonuç 2.0 olurdu. Başka bir deyişle int tipindeki 2 sayı bir birine bölündüğünde sonuç tamsayı çıkar. Bu kod aşağıdaki gibi olsaydı yani bölen veya bölünenlerden birisi float olsaydı sonuç ondalıklı olurdu.

```
static void Main(string[] args)
{
 int i1 = 12;
 float i2 = 5;
 float sonuc = i1/i2;
 Console.WriteLine(sonuc); // sonuç 2.4 olur.
}
```

Atama operatörüyle ilgili önemli bir ayrıntı daha vardır. Yukarıda verilen kısa kodda byte tipindeki değişkene 20 ve 10 sayılarını sorunsuzca aktarabilirken aşağıda yapıldığı gibi önüne eksi (-) işareti konulan negatif sayılar aktarılmak istendiğinde hata meydana gelir. Bu hatayı gidermek için dönüştürme (cast) işlemi yapılabilir.

```
static void Main(string[] args)
{
 sbyte sayi1 = 10;
 sbyte sayi2 = sayi1;
 sbyte sayi3 = -sayi1; // Hata meydana gelir.
}
```

Matematiksel operatörler üzerinde konuşmaya devam ediyoruz. Bu amaçla aşağıda verilen kodu yazdık. Bu kodda önce int tipinde bir değişken tanımlayıp bilgi aktardık. Devamında bu değişkenin içeriğini long bir sayı ile toplayıp başka bir int değişkene aktarmaya çalıştık.

```
static void Main(string[] args)
{
 int sayi1 = 10;
 int sonuc = sayi1 + 100L; // hata meydana gelir.
}
```

En başta ifade edelim ki bu kod hata üretir. Çünkü burada kullandığımız aktarma operatörünün sağında birisi int, diğeri long tipinde olan 2 operand vardır. Derleyici geri planda ilk olarak 2 operandın aynı tipte olmasını sağlar. Burada "+" operatörüne parametre olarak verdiğimiz sayılardan int olanı long tipine yani büyük operandın tipine dönüştürür. Toplama işlemi bundan sonra yapılır. Dolayısıyla toplama işlemi sonucu bulunan long değer eşitliğin sol tarafındaki int değişkene aktarılmak istenir ki bu hataya neden olur.

C# programlama dilinin operatörlerle ilgili tutumlarından söz etmek amacıyla şimdi birisi işaretli diğeri işaretsiz 2 tamsayıyı toplayıp başka bir işaretsiz sayıya aktarmaya çalışacağız. Bu amaçla aşağıda verilen kodu hazırladık.

```
static void Main(string[] args)
{
 int sayi1 = 100;
 uint sayi2 = 200;
 ulong sonuc = sayi1 + sayi2; // hata meydana gelir.
}
```

Burada "= " operatörünün sağındaki iki sayı '+' operatörü ile toplama işlemine sokulmadan önce küçük tip büyük tipe dönüştürülmek istenir. Ancak işaretli int sayıyı işaretsiz int sayıya dönüştürmek mümkün olmadığı için her ikisi de long tipine dönüştürülür. 2 long sayı toplandığında tahmin edeceğiniz gibi sonuç long olur. long sayıyı işaretsiz long değişkene aktarmak mümkün olmadığına göre bu kod hata üretir. Bu hatayı önlemek için ya eşitliğin sol tarafındaki işaretsiz değişkeni işaretli yapmak ya da cast işlemine başvurmak gerekir.

```
public static void Main()
{
 int sayi1 = 100;
 uint sayi2 = 200;
 ulong sonuc = (ulong) sayi1 + sayi2;
 Console.WriteLine(sonuc);
}
```

Yukarıda verilen örneklere bakılırsa C# derleyicisi operatörlere operand olarak verilen değişik tipteki veriler üzerinde işlem yaparken nasıl davrandığı konusu üzerinde önemle durmak bir zorunluluk gibidir. Yoksa, beklenmedik hatalı durumlarla karşılaşmak sürpriz olmayacaktır. Örneğin float ve double tipteki veriler "+" ile toplandığında sonuç double olur. Ancak float veya double veriyi decimal ile toplamayı ret eder. Bu nedenle aşağıda verilen kod hataya neden olur.

```
public static void Main()
{
 float sayi1 = 9.81f;
 double sayi2 = 3.14;
 decimal sayi3 = 3.67m;
 decimal sonuc = sayi1 + sayi2 + sayi3; // Hata meydana gelir.
}
```

Burada eşitliğin sağ tarafında farklı tipte 3 sayısal bilgi birbirleriyle toplanmak isteniyor. Derleyici float ve double sayıları toplamak istediğinde küçüğü büyüğe dönüştürüp toplama işlemini öyle yaptığını biliyorsunuz. İşlemin devamında double ile decimal tipleri birbirleri toplama noktasına gelindiğinde hata meydana gelir. Çünkü derleyici double tipini decimal tipe dönüştürmeyi ret eder. Bu işlemin sonucu mutlaka alınmak isteniyorsa double tipinin decimal'e dönüştürmesi gerekir. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
public static void Main()
{
 float sayi1 = 9.81f;
 double sayi2 = 3.14;
 decimal sayi3 = 3.67m;
 decimal sonuc = (decimal) (sayi1 + sayi2) + sayi3;
 Console.WriteLine(sonuc);
}
```

Operatörlerin Öncelik Sıraları

Operatörlerin kullanıldığı karşılaştırmalarda ve matematiksel işlemlerde operatörlerin öncelikleri matematikte gibidir. Örneğin aşağıda verilen satırlar işletildiğinde TextBox'a 50 yazılır. Çünkü çarpma(*) operatörünün önceliği olduğundan "number2" adlı değişkenin içeriği 3 ile çarpılır ve bulunan değer number1'in içeriğine eklenir.

```
int number1 = 20;
int number2 = 10;
textBox1.Text = Convert.ToString(number1 + number2 * 3); // 50
```

Bu örnekte number1 ile number2'yi toplayıp bulunan değeri 3 ile çarpmak istemiş olsaydık number1 ile number2'yi parantezin içine alıp işlemlerin öncelik sıralarına müdahale ederdik. Aşağıda verilen şekilde düzenlenen kod işletildiğinde TextBox'a 90 aktarılır.

```
int number1 = 20;
int number2 = 10;
textBox1.Text = Convert.ToString((number1 + number2) * 3); // 90
```

Matematiksel operatörlerinin işlem önceliği karşılaştırma operatörlerden yüksek olduğu için parantez kullanılmadan yan yana kullanıldığında öncelik matematiksel operatörün olmaktadır. Aynı mantıkla karşılaştırma operatörleri mantıksal operatörlere göre işlem önceliğine sahiptir. Aşağıda verilen tabloda operatörleri işlem önceliği yönünden sıraladık.

Aritmetik	Karşılaştırma	Mantıksal
^	=	Not
*, /	<>	And
Eksi işareti(-)	<	Or
\	>	Xor
Mod	<=	Eqv
+, -	>=	Imp
&	Like	

Arttırma ve Eksiltme Operatörleri

C# kodu yazılırken arttırma ve eksiltme operatörleri sıklıkla kullanılmaktadır. Aşağıda verilen satırda gerçekte 2 işlem yapılmaktadır. İlk işlemde "X" değişkenin mevcut içeriği 1 sayısı ile toplanmaktadır. 2. işlemde ise bulunan yeni değer X değişkenine aktarılmaktadır.

```
X = X + 1;
```

Değişkenlerin içeriklerini 1 arttırırken arttırma operatörünü (++) kullanabilirsiniz. Aşağıda verilen satır ile X değişkenin mevcut içeriği yine 1 arttırılır.

```
X++;
```

Aynı teknikle değişkenlerin içeriğini 1 azaltabiliriz. Aşağıda verilen her 2 yazılış şeklinde de "X" değişkenin mevcut içeriğini 1 azaltılır.

```
X = X - 1 veya X--
```

Arttırma ve eksiltme operatörlerinin içerikleri artırılacak veya azaltılacak değişkenlerin önüne veya arkasına yazılması operatörün farklı etkide bulunmasına neden olmaktadır.

```
X = 5;
Y = ++X + 5;  // Y = 11 ve X = 6 olur.
```

Bu kullanımda önce X'in içeriği 1 artırılır. Ardından X'e 5 eklenip "Y" değişkenine aktarılır. Burada artım operatörü X'in önüne yazıldığı için önce X'in içeriği arttırılır ondan sonra toplama işlemi yapılır. Aşağıda verilen kullanımda ise önce X ile 5 toplanıp sonra Y değişkenine aktarılır. Ardından X'in içeriği 1 arttırılır.

```
X = 5;
Y = X++ + 5;  // Y = 10 ve X = 6 olur.
```

Atama İşlemleri

C#'ta atama işlemi "=" operatörü ile yapılmaktadır. Ancak atama işlemi ile birlikte bazı ek işlemleri yapabilirsiniz. Örneğin aşağıda verilen 2. ve 3. satır aynı işleve sahiptir. Her 2 satırda da X değişkenin mevcut içeriği 5 arttırılmaktadır.

```
X = 10;
X = X + 5; // X 15 olur.
X += 5; // X 20 olur.
```

İlk satırın nasıl çalıştığını biliyorsunuz. 2. satırda X değişkenin mevcut içeriği 5 ile toplanıp tekrar X'e aktarılmaktadır. 3. satırda aynı işlem daha az kod ile yapılmaktadır. Benzer mantıkla aşağıda verilen 2 satır işlev bakımından birbirinden farklı değildir. Bu satırlarda X değişkenin mevcut içeriğinden 5 çıkarılıp bulunan yeni değer X değişkenine aktarılmaktadır.

```
X = 25;
X = X-5; // X 20 olur.
X -= 5;  // X 15 olur.
```

Yukarıda atama işlemi ile birlikte toplama ve çıkarma yaptık. Aynı şekilde değişkenlere sabit bilgi veya başka bir değişkenin içeriğini aktarırken çarpma ve toplama yapabilirsiniz.

```
X = 10;
X = X * 5; //  X 50 olur.
X *= 5;   //  X 50 olur.

X = 50;
X = X/5;  // X 10 olur.
X /= 5;   // X 10 olur.
```

Kaydırma Operatörleri

C# projelerinde sayıların kuvvetlerini almak istiyorsanız ">>" operatörünü kullanabilirsiniz. Aşağıda verilen 2 satır bir metoda yazılıp işletildiğinde X değişkenin içeriği 16 olur.

```
int X = 0;
X = 2 << 3; // X 16 olur.
```

Burada 2'nin karesini almak veya 4 ile çarpmak istemiş olsaydık "X = 2<<2" şeklinde yazardık. Atama işlemi sırasında sayının üssünü alabilirsiniz. Aşağıda atama işlemi sırasında ayrıca X'in üssü alınmaktadır.

```
int X = 2;
X <<= 3; // X 16 olur.
```

Atama işlemi sırasında kullandığımız "<<" operatörü sayıların istenen sayıda 2 ile çarpılmasını sağlamaktadır. Çarpma yerine bölme yapılacaksa bu kez ">>" operatörünü kullanmak gerekir. Aşağıda verilen 2 satır işletildiğinde X değişkeni 4 olur.

```
int X = 16;
X >>=2;  // X 4 olur.
```

Kaydırma işleminin anlaşılmasını sağlamak için 16 sayısının 2 tabanlı karşılığını aşağıda verilen kutulara yerleştirdik. int değişkenler bellekte 4 byte yer kaplasalar bile bu int değişkene aktardığımız sayı küçük olduğu için yalnızca en sağdaki 8 bitle ilgilendik.

0	0	0	1	0	0	0	0

Bu sırada bu sayı bitsel düzeyde bir kez sola kaydırılırsa yani 2 ile çarpılırsa söz konusu sayının 2 tabanlı karşılığı aşağıdaki gibi olur. 16 x 2= 32 olur.

0	0	1	0	0	0	0	0

Tam bu sırada sayı 2 kez sağa kaydırılırsa yani 32 sayısı 2 kez 2'ye bölünürse 8 sayısı elde edilir ve 2 tabanlı karşılığı aşağıdaki gibi olur.

0	0	0	0	1	0	0	0

as ve is Operatörleri

Bu operatörler, değişik türdeki bilgiler dönüştürülürken ve tip kontrolü yapılırken kullanılmaktadır. Tipler arası dönüştürme yaparken gerek duyacağınız **as** operatörünün nasıl kullanıldığını anlatmak için aşağıda verilen 2 satırı hazırladık. Burada **object** tipindeki değişkenin içeriği bilinen yöntemle dönüştürülüp string değişkene aktarılmaktadır.

```
object nesne = "Beşiktaş";
string takim = (string)nesne;
```

Aşağıda verilen kodda ise sayısal bilgi içeren object değişkenin içeriği dönüştürülüp **int** tipindeki değişkene aktarılıyor. Bu dönüştürme **Convert** sınıfının metotları ile yapılabilinirdi.

```
Object nesne = 2017;
int yil = (int)Nesne;
```

Tipler arası dönüştürme yapılırken **as** operatörü kullanılabilir. Bu operatörün nasıl kullanıldığını aşağıda görebilirsiniz. Aşağıda verilen örnekte **as** operatörü ile **object** değişkenin içeriği dönüştürülüp string değişkene aktarılmaktadır.

```
object nesne = "Beşiktaş";
string takim = nesne as string;
```

"as" operatörü ile string tipe dönüştürülen **object** değişkenin içerdiği bilgi string tipte olmasaydı hata meydana gelmezdi ama "as" operatörü geriye **null** bilgisini gönderirdi. as operatörünün ürettiği bilgi referans tipte olduğu için aşağıdaki gibi kullanıldığında hata meydana gelir.

```
Object nesne = 2017;
int yil = nesne as int;  // hata meydana gelir.
```

Forma yerleştirilmiş nesneler hakkında bilgi edinilirken **as** operatörü kullanılabilir. Aşağıda verilen kodda as operatörü ile forma daha önce yerleştirdiğimiz TextBox'ın referansını TextBox tipindeki değişkene aktardık.

```
private void Form1_Load(object sender, EventArgs e)
{
 TextBox Text1;
 Text1 = textBox1 as TextBox;
 Text1.Text = "Fenerbahçe";
}
```

Bu kod işletildiğinde formun üzerindeki TextBox'ın içeriği değişir. Şimdi **as** operatöründen yararlanıp formdaki bütün TextBox'ların içeriklerini temizleyeceğiz. Forma yerleştirilen nesneler formun **Controls** koleksiyonunda tutulmaktadır.

```
TextBox text1;
int sayi = this.Controls.Count;
for (int i = 0; i < sayi; i++)
{
 text1 = this.Controls[i] as TextBox;
 text1.Text = "";
}
```

Formun üzerindeki nesnelerin hepsi TextBox tipinde ise bu kod hatasız çalışır. **as** operatörüne parametre olarak verilen nesne TextBox tipinde değilse yani dönüştürme işlemi başarılı değilse geriye **null** gönderilmektedir. Bu kod aşağıdaki gibi düzenlenirse formun üzerindeki nesnelerin hepsi TextBox tipinde olmasa bile hatasızca çalışır.

```
private void button1_Click(object sender, EventArgs e)
{
 TextBox text1;
 int sayi = this.Controls.Count;
 for (int i = 0; i < sayi; i++)
 {
  text1 = this.Controls[i] as TextBox;
  if (text1 != null)
    text1.Text = "";
 }
}
```

Bu şekilde "as" operatörünün üzerinde durup nasıl kullanıldığından söz edildikten sonra şimdi sırada "is" operatörü var. Burada hakkında bilgi verilen **as** operatörü ile uygun referans türleri birbirine dönüştürülüyor. Dönüştürme başarısız olduğunda geriye **null** gönderildiği için yukarıda yapıldığı gibi yalnızca belli tipteki nesneler üzerinde kolaylıkla işlem yapılabiliniyor. **is** operatörü ise kendisine parametre olarak verilen 2 nesnenin aynı türe dönüşüme uygun olup olmadığı konusunda geriye mantıksal sonuç göndermektedir. Başka bir deyişle is operatörü, as operatörü kullanılmadan önce kullanılıp söz konusu tipilerin dönüşüme uygun olup olmadıkları konusunda araştırma yapılırken kullanılabilir. "is" operatörünün nasıl kullanıldığını aşağıda görebilirsiniz.

```
int yil = 2017;
bool sonuc;
sonuc = yil is int;
```

Bu kodda önce int ve bool tipinde 2 değişken tanımladık. **"is"** operatörünün soluna yazılan değişkenin tipi sağ tarafa yazılan tip ile aynı ise geriye true gönderilir. Burada "yil" adı verilen değişken int tipinde olduğundan "sonuc" değişkenine true aktarılır. is operatörü aşağıdaki gibi kullanıldığında ise geriye false gönderilir.

```
int yil = 2017;
bool sonuc = yil is string;
```



Yukarıda yaklaşık yarım sayfa boyunca "is" operatöründen söz edildi. Bu sayede is operatörünün nasıl kullanıldığı ve ne işe yaradığı anlaşılmış oldu. Dolayısıyla "is" operatörü hakkında yazacaklarımız bu kadardır deyip başka konuya geçebilirdik. Bizce o zaman konu eksik kalırdı. Çünkü "is" operatörünü kalıtım bağlamında incelememiş olurduk ki "is" operatörünün asıl kullanım alanı, aralarında base-derived class ilişkisi olan tiplerin üzerinde işlem yapıldığı kodlardır.

is operatörü hakkında bilgi vermeye devam etmek üzere aşağıda verilen kodu hazırladık. Bu 3 satırı bir metoda yazıp denemeden is operatörünün geriye ne göndereceğini tahmin etmeye çalışmanızı istiyoruz.

```
byte puan = 127;
bool sonuc;
sonuc = puan is int;
```

İlk bakışta, byte ile int değişkenlerin aynı gruba girdikleri ve aralarındaki en önemli farkın boyut olması nedeniyle "is" operatörünün geriye true göndereceği düşünülebilir. Bu düşüncenin bir diğer kaynağı, byte tipindeki değişkenlerin dönüştürme yapmaya gerek olmadan int değişkenlere aktarılabiliniyor olmasıdır. Derleyicinin nazarında byte tipi int'ten türetilmediği için burada is operatörü geriye false gönderir. Bu konunun ayrıntılarını ortaya çıkarmak için aşağıdaki gibi en kısıtlı halleriyle 2 sınıf hazırladık.

```
class Araba
{
}
class Ford : Araba
{
}
```

Üzerinde çalıştığımız uygulamada bu hazırlığı yaptıktan sonra uygulamadaki bir metodun içinde "Ford" tipinde bir nesne hazırladık. Ardından bu nesne ile "Araba" tipini is operatörüne parametre olarak verdik.

```
private void button1_Click(object sender, EventArgs e)
{
 Ford f = new Ford();
 bool sonuc;
 sonuc = f is Araba;
 MessageBox.Show(sonuc.ToString());
}
```

Burada "f" adını verdiğimiz "Ford" tipindeki nesne ile "Araba" tipi arasında bir kalıtım ilişkisi olması, başka bir deyişle Araba'nın, "Ford" tipinin base class'ı olması nedeniyle "is" operatörü geriye true gönderir.

Yukarıda is operatörüne parametre olarak verdiğimiz byte ve int tiplerinin base class'larını hatırlatmak için aşağıda verilen satırları hazırladık. Bu satırların yorumunu size bırakıyoruz.

```
private void button1_Click(object sender, EventArgs e)
{
 byte puan = 125;
 int yil = 2017;
 bool sonuc1, sonuc2;
 sonuc1 = puan is System.ValueType;
 sonuc2 = yil is System.ValueType;
 MessageBox.Show(sonuc1.ToString());
 MessageBox.Show(sonuc2.ToString());
}
```

Şimdiye kadar "is" operatörüyle ilgili olarak yazdıklarımız operatörün nasıl çalıştığını anlatmaya yönelikti, verilen örneklerin pratik değeri yoktu. Şimdi ise is operatörünü pratik değeri olan bir işlem üzerinde anlatmayı deneyeceğiz. Amacımız Windows Forms uygulamasının formuna yerleştirilmiş bütün TextBox nesnelerinin içeriklerini temizlemektir.

Bu amaçla yazdığımız kod aşağıdadır. Bu kodda formdaki nesnelerin sayısını öğrendikten sonra sıra ile bütün TextBox'ların içeriklerini **Clear()** metodu ile temizledik. Formun üzerindeki ilgili nesnenin TextBox tipine dönüşüme uygun olup olmadığını öğrendikten sonra bu nesneyi dönüştürüp referansını TextBox tipindeki değişkene aktardık.

```
private void button1_Click_1(object sender, EventArgs e)
{
 int sayi = this.Controls.Count;
 for (int i = 0; i < sayi; i++)
  {
   if (this.Controls[i] is TextBox)
   {
     TextBox edit = this.Controls[i] as TextBox;
     edit.Clear();
   }
  }
}
```

Bu kodu incelerken gördüklerimizi maddeler halinde sıralayacağız:

- Önce formun üzerindeki nesnelerin sayısı öğrenilmektedir. Bu amaçla Form sınıfının **Controls** koleksiyonun **Count** özelliğine bakılmaktadır.
- Count özelliğinin içeriğine bağlı olarak for döngüsü kurularak formun üzerindeki bütün kontroller tek tek incelenmektedir.
- Control koleksiyonundan okunan nesnenin TextBox tipine dönüşüme uygun olup olmadığı "is" operatörü ile araştırılmaktadır.
- if deyimine ait parantezin içinde "is" operatörü ile kontrol edilen nesne TextBox tipinde ise hemen orada TextBox tipinde bir değişken tanımlanıp Controls koleksiyonun o elemanı "as" operatörü ile TextBox tipine dönüştürülüp tanımlanan değişkene aktarılmaktadır.
- En son olarak TextBox sınıfının **Clear()** metodu ile ilgili TextBox nesnesinin içeriği temizlenmektedir.

Bit Düzeyinde İşlem Yapan Mantıksal Operatörler

Yukarıdaki sayfalarda kısaca anlatıldığı gibi && ve || gibi operatörler ile kolay bir şekilde birden fazla karşılaştırma birbirleri ile ilişkilendirilebilmektedir. Ancak bit düzeyinde işlem yapılmak istenildiğinde "&", "|" ve XOR operatörlerini kullanmak gerekmektedir.

Bit düzeyinde işlem yapan operatörler, sayının kendisi yerine sayıyı oluşturan bitler üzerinde işlem yapmaktadır. Herkes tarafından bilinmekle birlikte bu operatörlerle ilgili olarak aşağıda verilen tabloyu hazırladık.

&(ve)		
0	0	0
0	1	0
1	0	0

|(veya)		
0	0	0
0	1	1
1	0	1

^(XOr)		
0	0	0
0	1	1
1	0	1

1	1	1	1	1	1	1	1	0

Bu tabloda yer verdiğimiz 3 operatörden başka ayrıca ~(değil) operatörü bulunmaktadır. Tek parametreye sahip olan "~" operatörü ilgili bit üzerinde işlem yapmaktadır. Kendisine parametre olarak verilen sayının 1 olan bitlerini 0, 0 olanları ise 1 yapmaktadır. Bu konuda bilgi vermek için 10 tabanlı 32 sayısını aktardığımız byte tipindeki değişkeni bu operatöre parametre olarak vereceğiz. Önce bu sayının 2 tabanlı halini vereceğiz.

32 sayısı	0	0	1	0	0	0	0	0

Buna göre byte tipindeki değişkene 10 tabanlı 32 sayısı aktarıldığında sayı bellekte bu şekilde durmaktadır. Bu sayı **değil** operatörüne parametre olarak verildiğinde 0'lar 1, birler ise 0 yapılır ve geriye aşağıdaki gibi bir sonuç gönderilir

~32 sayısı	1	1	0	1	1	1	1	1

2 tabanlı bu sayının 10 tabanlı karşılığı 223'tür. ~ operatörünün nasıl kullanılabildiğini aşağıda görebilirsiniz. Bu örnekte ~ operatörüne byte bir değişkeni parametre olarak verdik ve çıkan sonucu başka bir byte değişkene aktardık. byte yerine **uint** değişken kullanmış olsaydık sonuç farklı olurdu.

```
byte sayi = 32;
byte sonuc = (byte)~sayi; // sonuc 223 olur.
```

Bu operatörlere daha çok sayılar parametre olarak verilmekle birlikte **bool** bilgiler parametre olarak verilebilir. Bu konuda bilgi vermek için aşağıda verilen satırları hazırladık. Bitsel operatörler bu şekilde kullanıldığında mantıksal işlem yapılıp geriye bool bilgi gönderilir.

```
bool sonuc = false & false; // sonuc False olur
sonuc = false | false; // sonuc False olur
sonuc = true | false; // sonuc True olur
```

Bu operatörlerin nasıl kullanıldığına örnek olması için aşağıda verilen kodu hazırladık. Bu kod sayesinde TextBox'a girilen sayının çift mi tek mi olduğunu öğreneceğiz.

```
ushort sayi = ushort.Parse(textBox1.Text);
if ((sayi & 1) == 0)
  textBox2.Text = "Bu sayı ÇİFT bir sayıdır";
else
  textBox2.Text = "Bu sayı TEK bir sayıdır";
```

Bu kodda ilk olarak kullanıcının çalışma anında formdaki TextBox'a girmiş olduğu sayıyı **Parse()** metodu ile **ushort** tipindeki değişkene aktardık. Ardından bu değişkenin ilk bitinin 0 olup olmadığını Ve(&) operatörü ile öğrendik. Bu bit 0 ise sayı çift, değilse tektir. Sayının tek sayı olup olmadığını öğrenmek istemiş olsaydık bu kodu aşağıdaki gibi değiştirip ilk bitin 1 olup olmadığını araştırırdık.

```
ushort sayi = ushort.Parse(textBox1.Text);
if((sayi & 1) == 1)
```

```
    textBox2.Text = "Bu sayı TEK bir sayıdır";
else
    textBox2.Text = "Bu sayı ÇİFT bir sayıdır";
```

Konunun anlaşılmasına katkı olması için forma 2 ListBox yerleştirip ilk ListBox'a tasarım anında çift sayılar aktardık. Kod yazıp ilk ListBox'taki çift sayıları tek sayıya dönüştürüp 2. ListBox'a aktaracağız. Burada yapılması gereken sayının ilk bitini sıfırlamaktır.

```
int sayi, cift_sayi;
for (int i = 0; i < listBox1.Items.Count; i++)
{
 sayi = Convert.ToInt16(listBox1.Items[i]);
 cift_sayi = (sayi & 0XFFFE);
 listBox2.Items.Add(cift_sayi.ToString());
}
```

Doğrudan bitler üzerinde işlem yapan operatörlerin nasıl işlev gördüğü konusunu netleştirmek için aşağıda verilen kodu hazırladık. Bu kodda 10 ve 30 sayılarını & operatörüne parametre olarak verdik.

```
int number1 = 10;
int number2 = 30;
int netice = number1 & number2; // netice 10;
```

Burada yapılan şudur: iki sayının 2 tabanlı halinin her bitinin Ve(&) operatörü ile karşılaştırılmasıdır. Bu kod işletildiğinde TextBox'a 10 yazılır. Bu sonucun nasıl bulunduğunu açıklamak için 10 ve 30 sayılarının 2 tabanlı halleri aşağıda verildi.

10 sayısı	0	0	0	0	1	0	1	0
30 sayısı	0	0	0	1	1	1	1	0

Bu 2 sayının bitleri karşılıklı olarak Ve(&) işlemine tabi tutulursa aşağıdaki gibi bir sonuç alınır. Örneğin her iki sayının ilk bitleri 0. Bu nedenle 0&0 işleminin sonucu 0 olur. Aynı şekilde her 2 sayının 2. bitleri 1'dir ve 1&1 işleminin sonucu 1'dir.

10 sayısı	0	0	0	0	1	0	1	0
30 sayısı	0	0	0	1	1	1	1	0
ve işlemi	0	0	0	0	1	0	1	0

Yukarıda verilen örnekte & yerine | (veya) operatörünü kullansaydık farklı bir sonuç çıkardı. 30 ve 10 sayıları veya operatörüne parametre olarak verildiğinde 30 elde edilir.

10 sayısı	0	0	0	0	1	0	1	0
30 sayısı	0	0	0	1	1	1	1	0
veya işlemi	0	0	0	1	1	1	1	0

Şimdiye kadar verilen bilgilere göre "&" ile "|" gibi operatörler kendilerine parametre olarak verilen sayının bütün bitlerini etkilemektedir. Şimdi örnek olması için 32 sayısının 0 olan ilk bitini 1 yapacağız. Bu amaçla ilk biti 1 olan 10 tabanlı 1 sayısından yararlanacağız. 32 sayısının 0 olan ilk biti 1 yapıldığında sayı 33 olur.

32 sayısı	0	0	1	1	0	0	1	0
1 sayısı	0	0	0	0	0	0	0	1
veya işlemi	0	0	1	1	0	0	1	1

Bu işlemı yapmak üzere hazırladığımız kod aşağıdadır. Bu kodda Or(|) yerine Ve(&) operatörünü kullansaydık sonuç farklı olurdu ve geriye 0 gönderilirdi.

```
byte sayi1 = 32;
byte sayi2 = 1;
int sonuc = (int)sayi1 | sayi2 ; // sonuc 33 olur.
```

Şimdi vereceğimiz örnekte çalışma anında kullanıcının TextBox'a yazdığı byte tipindeki sayının 1., 3. ve 5. bitlerinin mutlaka 1 olmasını sağlayacak kod yazacağız. Bu amaçla ilk olarak 1., 3. ve 5. bitleri 1, diğerleri 0 olan sayıyı öğrendik. Bu sayı 10 tabanlı 21'dir.

```
byte sayi1 = 21;
byte sayi2 = Convert.ToByte(textBox1.Text);
int sonuc = sayi1 | sayi2;
textBox2.Text = sonuc.ToString();
```

?: (Ternary) Operatörü

2 olasılığın söz konusu olduğu karşılaştırmalarda ternary (üçlü) olarak bilinen "?:" operatöründen yararlanabilirsiniz. 3 operanda gerek duyduğu için üçlü operatör olarak bilinir. "?" ve ":" işaretlerinin bir arada kullanılmasıyla işlevsel olan bu operatörün nasıl kullanıldığını anlatmadan önce if deyiminin kullanıldığı aşağıda verilen kodu hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 int borc = Convert.ToInt32(textBox1.Text);
 if (borc > 0)
 {
   MessageBox.Show("Borçlusunuz");
 }
 else
 {
   MessageBox.Show("Borçlu değilsiniz");
 }
}
```

2 ihtimalin söz konusu olduğu böyle karşılaştırmalar için "if" yerine aşağıda yapıldığı gibi "?:" operatörünü kullanabilirsiniz. Karşılaştırmadan sonra önce "?" işareti, ardından karşılaştırma doğru olduğunda yapılacak işlem veya işletilecek satır yazılır. Karşılaştırma yanlış ise ":" işaretinden sonra yazılan bilgi geriye gönderilir.

```
int borc = Convert.ToInt32(textBox1.Text);
string sonuc = borc > 0 ? "Borçlusunuz":"Borçlu değilsiniz";
MessageBox.Show(sonuc);
```

Bu "?:" kullanımında "borc" değişkeni 0'dan büyükse "Borçlusunuz" mesajı verilir. "borc" değişkeni 0'dan büyük değilse bu kez geriye "Borçlu değilsiniz" bilgisi gönderilir.

Bu ternary operatörü kullanımında karşılaştırmanın sonucuna göre seçenek olarak sunulan 2 bilgi string tipte olduğu için geriye string bilgi gönderildi. Örneğin "?" operatöründen sonra char tipinde bilgilere yer verilmiş olunsaydı geriye char tipinde bir sonuç gönderilirdi.

```
int borc = Convert.ToInt32(textBox1.Text);
char sonuc = borc > 0 ? 'B' : 'A';
MessageBox.Show(sonuc.ToString());
```

Verilen her iki ternary operatörü kullanım örneğinde fark edilmiş olacağı üzere operatörün solunda mantıksal sonuç üreten bir karşılaştırma var. Bu karşılaştırmayı yaparken ">" operatöründen yararlandık. Konu üzerinde düşünmenizi sağlamak için ternary operatörünün soluna bool sonuç döndüren bir metodun adını yazacağız. Tabi önce bu metodu hazırlamamız gerekiyor.

```
bool Ayni_yil(int yil)
{
 DateTime Tarih = DateTime.Today;
 int buyil = Tarih.Year;
 if (buyil == yil)
   return true;
 else
   return false;
}
```

"Ayni_yil()" adını verdiğimiz bu metot int tipinde bir parametreye sahiptir ve geçerli sistem tarihinin yıl bilgisi, kendisine parametre olarak verilen int bilgi ile karşılaştırıyor. Geçerli tarihin yıl bilgisi ile parametre olarak verilen yıl bilgisi birbirine eşitse geriye true, yoksa false göndermektedir. Bu metodu ternary operatörüne nasıl parametre olarak verdiğimizi aşağıda görebilirsimiz.

```
private void button1_Click(object sender, EventArgs e)
{
 int yil = Convert.ToInt32(textBox1.Text);
 string sonuc = Ayni_yil(yil) ? "Aynı Yıldayız" : "Farklı Bir Yıl";
 MessageBox.Show(sonuc);
}
```

Bu ternary operatörü kullanma örneğinde mantıksal sonuç üretmesi için bir metot hazırladık ve operatörün sol tarafına yazdık. Bu metot geriye true gönderirse, ternary operatörü tarafından geriye "Aynı Yıldayız" bilgisi gönderilir. Konu üzerinde düşünmeye devam etmek amacıyla şimdi ikinci bir metot hazırlayıp ternary operatörünün sağındaki ilk parametrenin yerine kullanacağız.

```
string Bugun()
{
 DateTime tarih = DateTime.Today;
 return tarih.ToShortDateString();
}
```

Yukarıda ternary operatörümüzün geriye string bilgi göndermesini sağladığımız ve bunda değişiklik yapmayı düşünmediğimiz için hazırlayıp Bugun() adını verdiğimiz metot geriye geçerli tarihin string halini göndermektedir. Bu hazırlıktan sonra ternary operatörümüzü kullandığımız satırı aşağıdaki gibi değiştirdik.

```
private void button1_Click(object sender, EventArgs e)
{
 int yil = Convert.ToInt32(textBox1.Text);
 string sonuc = Ayni_yil(yil) ? Bugun() : "Farklı Bir Yıl";
 MessageBox.Show(sonuc);
}
```

Benzer şekilde ternary operatörünün sağına yazılan 2. operand olarak sabit bilgi yerine yine bir metot hazırlayıp kullanabilirsiniz. Ternary operatörün bütün operandlarının metot olarak düzenlemeye örnek olması için ayrıca aşağıda verilen metodu hazırladık.

```
string Eskigun(int y)
{
 DateTime tarih = DateTime.Today;
 int gecerliyil = tarih.Year;
 int fark = gecerliyil - y;
 DateTime yeniTarih = tarih.AddYears(-fark);
 return yeniTarih.ToShortDateString();
}
```

Şimdi elimizde Ayni_yil(), Bugun() ve Eskigun() adında 3 metot var. Ayni_yil() adını verdiğimiz metot kendisine parametre olarak verilen yıl bilgisinin içinde bulunulan yılı temsil edip etmediğini göre geriye bool sonuç göndermektedir. Bugun() metodunun geçerli tarih bilgisinin kısa halini geriye gönderdiğini biliyorsunuz. Eskigun() metodu ise kendisine parametre olarak verilen yıl bilgisinin bugüne ait tarihini bulup geriye göndermektedir. Bu 3 metodu birlikte kullanmak için ternary operatörünü aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
 int yil = Convert.ToInt32(textBox1.Text);
 string sonuc = Ayni_yil(yil) ? Bugun() : Eskigun(yil);
 MessageBox.Show(sonuc);
}
```

() Operatörü

Casting adı verilen bu operatör C# kodu içinde bir tip bir başkasına dönüştürülürken kullanılmaktadır. Dönüştürme sırasında hedef tipin adı parantezlerin arasına yazılmaktadır. Bu operatörün nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 byte sayi1;
 int sayi2 = 200;
 sayi1 = (byte)sayi2;
}
```

Burada Casting operatörü kullanmasaydık hata meydana gelirdi. Burada int tipindeki "sayi2" değişkenin içeriği dönüştürüp byte tipindeki değişkene aktarılmaktadır. int tipindeki değişkenin içeriği 255'ten büyük olmadığı için bu dönüştürme sırasında bilgi kaybı olmaz.

Bu operatörün işlevi ve nasıl kullanıldığı bu kitabın değişkenler adlı bölümde ayrıntılı olarak işlenmişti. Dolayısıyla bu kitabın "Değişken Tanımlamak ve Kullanmak" adı bölümünü okuyanlar bu operatör hakkında yeteri bilgiye sahiptirler.

[] Operatörü

Bildiğiniz gibi dizi değişkenler gerçekte Array tipinde nesnelerdir. Array nesneleri dahili bir koleksiyon içermektedir. Bu koleksiyonda istenilen miktarda bilgi saklanabiliniyor. Örneğin aşağıda verilen satır işletildiğinde hazırlanacak array nesnesinin içinde 10 adet int bilgiyi bellekte yan yana tutabilecek bir yapı hazırlanır.

```
int[] dizi = new int[10];
```

"[]" operatörü bu yapıdaki elemanlara erişimde kullanılmaktadır. Aşağıda verilen kod işletildiğinde bu değişkenin ilk 2 elemanına önce bilgi aktarılır, sonra bu elemanların içerikleri okunup konsola yazılır.

```
public static void Main()
{
 int[] dizi = new int[10];
 dizi[0] = 2016;
 dizi[1] = 2017;
 Console.WriteLine(dizi[0]);
 Console.WriteLine(dizi[1]);
}
```

Verilen kodda görüleceği gibi dizi değişkenlerin elemanlarına bilgi aktarılırken veya mevcut içerikler okunurken "[]" operatöründen yararlanılmaktadır. Bu operatör hakkında bilgiyi "Dizi Değişkenler" ve "İndeksleyiciler" adlı bölümlerde bulabilirsiniz.

Null - Coalescing (??) Operatörü

Yukarıdaki sayfalarda ifade edildiği üzere 2 ihtimalin söz konusu olduğu karşılaştırma işlemleri için "if" yerine "?:" operatörü kullanılabilmektedir. Benzer durum **nullable** değişkenler için de geçerlidir. Bu operatörden "Değişken Tanımlamak ve Kullanmak" adlı bölümde Nullable değişkenler anlatılırken söz edilmişti. Bu nedenle burada bu operatörden kısaca söz edeceğiz. Bu amaçla aşağıda verilen kodu hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 int? sayi1, sayi2, sonuc;
 try
  {
   sayi1 = Int32.Parse(textBox1.Text);
  }
 catch (Exception)
  {
   sayi1 = null;
  }

 try
  {
   sayi2 = Int32.Parse(textBox2.Text);
  }
 catch (Exception)
  {
   sayi2 = null;
  }

 if (sayi1 == null)
    sayi1 = 1;
 if (sayi2 == null)
    sayi2 = 1;
 sonuc = sayi1 * sayi2;
 textBox3.Text = sonuc.ToString();
}
```

Bu kodu yazmadan önce üzerinde çalıştığımız Windows Forms uygulamasının formuna 3 TextBox ve bir Button nesnesi yerleştirdik. Button nesnesi tıklandığında işletilecek bu kodda önce 3 adet nullable değişken tanımladık. Amacımız ilk 2 nullable değişkenin içeriklerini birbiri ile çarpıp çıkan sonucu 3. değişkene aktarmaktır. Değişkenlerimiz null değer alabilme özelliğine sahip olduğu için, içerikleri null iken çarpma işlemini yapamayız.

Ardından try bloğu içinde ilk TextBox'ın içeriğini okuyup Int32 yapısının **Parse**() metodu ile int tipinde dönüştürüp "sayi1" adını verdiğimiz ilk nullable değişkene aktardık. Kullanıcı ilk TextBox'a int tipine dönüşmeyecek bir şey yazar veya boş bırakırsa Parse() metodu ile dönüştürme yapılırken hata meydana gelir ve kodun işletimi **catch** bloğuna geçer. Bu blokta değişkene "null" bilgisi aktarılıyor. Çünkü amacımız null bilgisi içeren nullable değişkenler üzerinde işlem yapılırken kullanılan "??" operatöründen söz etmektir. Benzer işlemi ikinci nullable değişken için de yaptık.

Formdaki ilk 2 TextBox'ın içeriğini int tipine çevirip burada "sayi1" ve "sayi2" adını verdiğimiz değişkenlere aktardıktan sonra 2 ayrı if bloğu hazırlayarak bu değişkenlerin null olup olmadıklarını araştırdık. Hangi değişken null ise ona 1 sabit değerini aktardık. Böylece bu değişkenlerin içeriklerini birbirleri çarpabilmenin şartlarını sağlamış olduk. Burada yaptığımız işlemin benzerini "??" operatörü ile yapmak daha pratiktir. Aynı işleve sahip kodun Null - Coalescing (??) operatörü kullanılmış şekli aşağıdadır.

```
private void button1_Click(object sender, EventArgs e)
{
 int? sayi1, sayi2, sonuc;
 try
  {
   sayi1 = Int32.Parse(textBox1.Text);
  }
 catch (Exception)
  {
   sayi1 = null;
  }
 try
  {
   sayi2 = Int32.Parse(textBox2.Text);
  }
 catch (Exception)
  {
   sayi2 = null;
  }
 sayi1 = sayi1 ?? 1;
 sayi2 = sayi2 ?? 1;
 sonuc = sayi1 * sayi2;
 textBox3.Text = sonuc.ToString();
}
```

Bu örnekte "sayi1 = sayi1 ?? 1" satırında??'dan önce yazılan değişken (burada sayi1) null değilse eşitliğin sol tarafına aktarılır, null ise eşitliğin sol tarafındaki değişkene ??'den sonra yazılan (burada 1) aktarılır.

9. BÖLÜM

BLOK KONTROL ve DÖNGÜLER

Bu bölümde C# kodu yazarken sıklıkla kullanılan blok kontrol ve döngü deyimlerinden söz edilecektir. C#'ta blok kontrol işlemleri diğer programlama dillerinden pek farklı değildir. Dolayısıyla daha önce C, C++ veya Java gibi programlama dilleriyle kod yazmış olanlar bu bölümde anlatılanlara yabancı olmayacaklardır. Bu bölümde anlatılanlar bağlamında C#'ın diğer programlama dilerinden ayrıldığı nokta **foreach** döngüleridir. Buna bir de C# 7.0 ile gelen ve **switch** deyiminin kullanım şeklini değiştiren pattern'leri ekleyebiliriz.

C# kodu yazarken bu bölümde sözü edilecek anahtar kelimeleri sıklıkla kullanacağınız için kullanma alışkanlığını kazanmanız şarttır. Bu bölümde kısaca sözü edilen foreach döngüleriyle ilgili ayrıntıları kavramak için koleksiyonlar hakkında bilgi sahibi olmayı gerektiğinden daha sonra ayrı bir bölüm olarak ele alınacaktır.

if Deyimi

if deyimi, programın akışını verilecek koşula bağlı olarak "{ }" işaretleri ile oluşturulan bloktaki satırların işletilmesini veya söz konusu bloğun işletilmeyip atlanmasını sağlar.

```
if (ifade)
 {
   .......
 }
```

if deyimine parametre olarak verilen "ifade" mantıksal doğru değerini içeriyorsa "{" ile "}" arasında yer alan satırlar işletilir. "ifade" **false** ise bu kez "{" ile "}" arasında yer alan satırlar atlanıp programın işletimi "}" işaretinden sonra gelen ilk satıra geçer. Parantezin arasına yazılan ifade bool bir sonuç üretmiyorsa hata meydana gelir. Örneğin "if" deyimi aşağıdaki gibi kullanılırsa hata meydana gelir.

```
int sayi = 100;
if (sayi = 100)  // hata meydana gelir.
{
}
```

'{' ve '}' ile sınırlanan blokta tek satır varsa blok başını ve sonunu belirtmeye gerek kalmaz. Örneğin aşağıda verilen kodda önce **DialogResult** tipinde bir değişken tanımlanıp kullanıcıdan programdan çıkılıp çıkılmayacağı konusunda onay alındı.

```
DialogResult onay = MessageBox.Show("Projeden Çıkılsın mı?",
        "Onay Kutusu", MessageBoxButtons.YesNo);
if (onay == DialogResult.Yes)
{
 Application.Exit();
}
```



Burada kullanılan Exit(), **Application** sınıfının statik bir metodudur ve C# uygulamalarının çalışmasını sona erdirmede kullanılmaktadır. Application sınıfı System.Windows.Forms'de bulunmaktadır. Bu sınıf, Windows Forms Application şablonu ile hazırlanan uygulamaların Windows'la ilişkisini kurmaktadır.

Show() metodu ile ekrana getirilen diyalog kutusundaki Evet düğmesi tıklanırsa "onay" değişkenine "DialogResult.Yes", Hayır düğmesi tıkladığında ise "DialogResult.No" aktarılır. "onay" değişkenine "DialogResult.Yes" aktarılmışsa **if**'e parametre olarak verilen karşılaştırmanın sonucu true olur ve "{}" ile bloklanan satırlar işletilir. if deyimi için parantezlerin içinde "==" operatörü ile eşitlik aramak yerine **Equal**() metodu aşağıdaki gibi kullanılabilir.

```
DialogResult onay = MessageBox.Show("Projeden Çıkılsın mı?",
          "Onay Kutusu", MessageBoxButtons.YesNo);
if (onay.Equals(DialogResult.Yes))
 {
  Application.Exit();
 }
```

Burada kullanılan "onay" değişkeni ""DialogResult.Yes" değerini içermiyorsa if ile bloklanan satır işletilmeyip programın işletimi if bloğundan sonra gelen ilk satıra geçer. Bazen if'e parametre olarak verilen karşılaştırmanın sonucuna bağlı olarak yapılmak istenen birden fazla işlem olabilir. Bu nedenle arka arkaya birden fazla if bloğu hazırlanabilir.

```
int Puan = Convert.ToInt32(textBox1.Text);
if (Puan < 50)
{
  MessageBox.Show("Bu öğrenci zayıf not aldı");
}
if (Puan >= 50)
{
  MessageBox.Show("Bu öğrenci geçer not aldı");
}
```

İçeriği araştırılan değişkenin alma olasılığı olan her değer için ayrı bir if bloğu hazırlamak pratik değildir. Şimdi yukarıda verilen ve 2 "if" deyimin kullanıldığı kodla aynı işleve sahip başka bir kod vereceğiz. Bu kodda 2. bir blok hazırlamak yerine **else** deyiminden yararlandık.

```
if (Puan < 50)
{
  MessageBox.Show("Bu öğrenci zayıf not aldı");
}
else
{
  MessageBox.Show("Bu öğrenci geçer not aldı");
}
```

Yapılan karşılaştırmada ikiden fazla olasılık varsa birden fazla **else** deyimi kullanılabilir. Aşağıda verilen if kalıbında toplam 3 olasılık için karşılaştırma ve karşılaştırmanın sonucuna göre işlem görecek 3 blok var.

```
if (ifade1)
 {
   .........
 }
else if (ifade2)
 {
 }
else if (ifade3)
 {
 }
```

Birinci karşılaştırma (ifade1) doğru değer üretiyorsa ise ilk "{" işaretinin bulunduğu satır ile ilk "}" işaretinin bulunduğu satıra kadar olan satırlar işletilir. Ardından programın işletimi blok sonunu işaret eden "}" işaretinden sonraki satıra geçer. Bu durumda bloktaki **else** deyimleri ile hazırlanan karşılaştırmaların sonucuna bakılmaz.

if deyimine parametre olarak verilen ifade1 yanlış değerine sahipse, bu kez ilk else'ye parametre olarak verilen ifade2'nin sonucuna bakılır. ifade2 doğru değere sahipse, ilk else ile 2. else deyimleri arasında kalan satırlar işletilir.

Farklı bir if örneği olsun diye aşağıda verilen kodu hazırladık. Bu kodda "C:" sürücüsünde "\Windows" klasörünün varlığı araştırılmaktadır. **DirectoryInfo** sınıfının **Exists** özelliği söz konusu klasör, işaret edilen yerde varsa true, yoksa false bilgisini içermektedir. Burada klasör olmadığında işlem yapmak istemediğimizden dolayı "!" değil operatöründen yararlandık.

```
string yol = "C:\\Windows";
DirectoryInfo dizin = new DirectoryInfo(yol);
if (!dizin.Exists )
{
 MessageBox.Show("Klasör bulunamadı");
}
```

Switch Case Deyimi

Switch case işlev bakımından **if** deyimine benzemektedir. Çok sayıda "if" bloğu kullanıldığı zaman programın okunurluğu azalmaktadır. Bu gibi durumlarda switch case deyiminden yararlanılması önerilmektedir. İsterseniz if deyimi ile switch case'i birlikte kullanabilirsiniz. **switch case** bloklarında **break** anahtar kelimesinin kullanılması zorunludur.

```
switch (Kontrol Değişkeni)
{
 case ifade 1 :
  {
   .........
   break;
  }
 case ifade 2 :
  {
   ..........
   break;
  }
}
```

case bloklarında aynı değere sahip birden fazla ifadenin olmasına derleyici izin vermemektedir. Switch case bloğunu hazırlarken Visual Studio'dan yararlanmak istiyorsanız **switch** yazıp 2 kez tab tuşuna basarsanız bir switch case kalıbı hazırlanır.

Blok başlatan **switch** deyiminden sonra karşılaştırmalarda kullanılan bir kontrol değişkeni bulunmaktadır. Bu değişkenin içeriği "ifade1" olarak verilen değerle aynı ise birinci ifadenin içeriğini araştıran **case** deyiminden sonraki "{" ve "}" işaretleri arasında kalan satırlar işletilir ve programın akışı case ile başlatılan bloğun sonunu işaret eden }'ni izleyen satıra geçer.

Kontrol değişkeni ifade2'ye eşitse bu kez ifade2'den "{" ve "}" deyimleri arasındaki satırlar işletilir. case deyimi ile döngü başlatıldıktan sonra sıra ile "ifade" bulunan bütün satırlar taranır. "ifade 2" olarak belirtilen bilgi ile kontrol değişkenin içeriği birbirine eşitse "ifade 2" bloğunda bulunan satırlar işletilir. Aşağıda standart bir switch-case örneği var.

```
int puan = Convert.ToInt32(textBox1.Text);
switch (puan)
{
 case 1: MessageBox.Show("Zayıf"); break;
 case 2: MessageBox.Show("Orta"); break;
 case 3: MessageBox.Show("İyi"); break;
 case 4: MessageBox.Show("Çok İyi"); break;
 default: MessageBox.Show("Yanlış seçim");
 break;
}
```

Bu kod üzerinde biraz konuşmak gerekecek: Programın işletimi burada switch anahtar kelimesine geldiğinde ilk olarak parantezin içine yazılan değişkenin (burada "puan") içeriğine bakılır. Devamında bu değişkenin içeriği ilk case'den sonra yazılan sabit değerle karşılaştırılır. Hazırladığımız switch case yapısında ilk case'in yanına 1 yazdığımız için puan değişkeni 1 sayısını içeriyorsa ilk case'den sonraki satırlar işletilir ve "break" anahtar kelimesi ile switch case yapısından çıkılır. Eğer "puan" değişkeni 1'den farklı bir değer içeriyorsa bu kez 2 bilgisini içerip içermediği araştırılır. "puan" değişkeni, 1, 2, 3 ve 4 bilgilerinden birisini içermiyorsa sıra **defautl** etiketine gelir.

Bu switch-case örneğinde her case için yazılan kodlar kısa olduğu için blok parantezleri "{}" kullanılmadı. İstenirse bu switch-case örneği aşağıdaki gibi düzenlenebilir.

```
int puan = Convert.ToInt32(textBox1.Text);
switch (puan)
{
 case 1:
 {
   MessageBox.Show("Zayıf");
   break;
 }
 case 2:
 {
   MessageBox.Show("Orta");
   break;
 }
```

```
        case 3:
        {
          MessageBox.Show("İyi");
          break;
        }
        case 4:
         {
          MessageBox.Show("Çok İyi");
          break;
         }
        default:
         {
          MessageBox.Show("Yanlış seçim");
          break;
        }
      }
```

Kontrol değişkeni **case** deyiminin yanına yazılanlardan farklı bir değer aldığında yapılmasını istediğiniz işlemler varsa **default** anahtar kelimesinden yararlanmalısınız. default anahtar kelimesine yer vermezseniz kontrol değişkeni hiçbir case ile eşleşmezse programın akışının switch case'e uğrayıp uğramadığını bilemeyebilirsiniz.

Break, switch bloğundan çıkışı sağlayıp programın akışını switch bloğundan sonraki satıra atlatmaktadır. **break** yerine **return** deyimini kullanabilirsiniz ama bu durumda metottan çıkılır ve switch bloğundan sonraki satırlar (varsa) işletilmez.

Yukarıda switch deyimine int değişkeni parametre olarak verdik. Kontrol değişkeni olarak string veya başka bir tipte değişkeni kullanabilirsiniz.

```
string secim = textBox1.Text;
switch (secim)
{
 case "Çıkış" :
  Application.Exit();
  break;
 case "Devam":
  break;
}
```

İstenilen case bloğundaki satırların işletilmesini, diğerlerinin göz ardı edilmesini sağlayan ifadeler tamsayı, string veya karakter sabiti olmalıdır. Default seçeneği genelde en son case olarak yazılsa bile istenilen sırada olabilir.

Bir diğer ayrıntı şudur: switch case yapısının içinde iken bir case'ten diğerine geçmek istenildiğinde **goto** anahtar kelimesi kullanılmaktadır. Örneğin yukarıda verdiğimiz switch-case yapısında TextBox'a ister 3, ister 4 girilsin her seferinde kullanıcıya "Çok İyi" mesajı verilmek isteniyorsa aşağıdaki gibi bir düzenleme yapılabilir.

```
int puan = Convert.ToInt32(textBox1.Text);
switch (puan)
{
 case 1:
 {
  MessageBox.Show("zayıf");
  break;
 }
 case 2:
 {
  MessageBox.Show("orta");
  break;
 }
 case 3:
 {
  goto case 4;
 }
 case 4:
 {
  MessageBox.Show("Çok İyi");
  break;
 }
}
```

Switch case'lerin bir özelliğinden söz etmek için yukarıda hazırladığımız switch case yapısını aşağıdaki gibi düzenledik. Dikkat ettiyseniz case 1'den sonra herhangi bir şey yazmadık. Bu durumda kontrol değişkeni 1 iken bloktan çıkılmayıp "case 2:" olarak etiketlenmiş satırlar işletilir. Kontrol değişkeni 2 ise yine "case 2:" etiketi adı altında gruplanan satırlar işletilir.

```
int puan = Convert.ToInt32(textBox1.Text);
switch (puan)
{
 case 1:
 case 2:
 {
   MessageBox.Show("orta");
   break;
 }
 case 3:
 {
  MessageBox.Show("İyi");
  break;
 }
}
```

while Döngüsü

while anahtar kelimesi, verilen şarta bağlı olarak bloklanan satırların bir döngü halinde tekrar tekrar işletilmesini sağlar. **while** ile tekrar tekrar işletilecek kod birden fazla satır ise işletilecek satırların başına "{", sonuna ise "}" işaretinin konulması gerekir. While yazıp 2 kez tab tuşuna basarsanız Visual Studio size bir while kalıbı hazırlar.

```
while(Koşul)
{
 break
  .......
}
```

break deyiminin kullanılması isteğe bağlıdır. while deyimine parametre olarak verilen "koşul" doğru değerine sahip olduğu sürece while ile işaret edilen satırlar tekrar tekrar işletilir. "koşul" yerine mantıksal bir karşılaştırma veya tek başına true-false kullanılabilir.

while deyiminin genel kullanış biçiminde bulunan ve kullanılması isteğe bağlı olan **break** deyimi, döngüden "koşul" ile belirtilen şarta bağlı kalmadan çıkmayı sağlar. Genel alışkanlıkla yalnızca break deyimi ile çıkılacak şekilde oluşturulan do-while döngülerinde "ifade" olarak mantıksal sabit bir değer kullanılır.

```
while (true)
{
 if (koşul)
   break;
}
```

Bu döngünün her turunda if ile "koşul" test edilir. Koşul yanlış değerini içeriyorsa, döngüden çıkılarak blok sonunu işaret eden "}" işaretini izleyen satıra geçilir. Döngü kontrol ifadesi olarak **true** kullanılması halinde döngüden **break** deyimi ile çıkılmadığı sürece döngüde kalınır. Bu nedenle yukarıdaki gibi döngülerde mutlaka döngüden çıkabilmek için belirli şartlara bağlı olarak break deyiminin işletilmesi sağlanmalıdır.

```
while (true)
{
 string takim = Microsoft.VisualBasic.Interaction.InputBox(
    "Takim adı girin", "Bilgi girişi", "", 20, 20);
 if (takim == "Beşiktaş")
   break;
}
```

Yukarıda verilen blokta **while** deyimi ile koşul döngünün başında kontrol edilmektedir. Bu şekilde kullanılan while deyiminden başka C#'ın koşulun döngünün sonunda test edildiği **do-while** adında bir deyimi daha bulunmaktadır. do-while bloklarında koşul döngünün sonunda test edildiği için söz konusu blok en az bir kez işletilir.

```
do
{
 string takim = Microsoft.VisualBasic.Interaction.InputBox(
          "Takim adı girin", "Bilgi girişi", "", 20, 20);
 if (takim == "Beşiktaş")
   break;
}
while (true);
```

while veya do-while deyimleri ile kurulan bloktaki satırların sıra ile işletilmesini istemiyorsanız **continue** ile kodun işletimini tekrar döngünün başına alabilirsiniz. Aşağıda verilen kodda continue sayesinde 1'den 100'e kadar olan sayıların yalnızca tek olanları toplanır.

```
int toplam = 0;
for (int i=1; i<=100; i++)
{
 if (i%2 !=1) continue;
  toplam = toplam + i;
}
textBox1.Text = Convert.ToString(toplam);
```

for Döngüleri

do–while döngülerinde döngünün kaç kez tekrarlanacağı önceden bilinmez. for döngülerinin ise kaç kez tekrarlanacağı genellikle bellidir. do–while döngüsünde olduğu gibi istenirse belli bir şart dahilinde **for** döngüsünden her hangi bir anda çıkılabilinir. Kod penceresinde **for** yazıp 2 kez Tab tuşuna basarsanız Visual Studio sizin için bir for döngüsü kalıbı hazırlar.

```
for (döngü değişkeni = ilk değer; koşul; artım)
{
 [Break]
}
```

for deyimine 2. parametre olarak verilen **ilk değer** döngü değişkenin alacağı ilk değerdir. "koşul" ifadesi ile döngüde kalma şartları belirtilir. **artım** ile döngünün her turunda değişkenin ne kadar artıp veya azalacağı belirtilir.

```
for (int X = 1; X<10; X++)
{
}
```



Burada dikkatinize sunmak istediğimiz bir nokta var: For döngüsüne ait parantezin içinde int tipinde ve X adında bir değişkeni tanımlanmış olmasıdır. Bu değişkenin yaşama alanı for döngüsüyle sınırlıdır. İsteseydik bu değişkeni for döngüsünün dışında metot içinde tanımlayabilirdik. Ancak bu pek tercih edilen bir seçim değildir. Çünkü döngüden çıkıldığında döngüyle ilgili değişkene ulaşılsın istenmez.

Bu döngü örneğinde "X" döngü değişkeni, '1' döngü değişkenin başlangıç değeri ve 10 ise son değeri olur. Bu döngüde for deyiminden sonraki blok içindeki satırlar 10 kez arka arkaya işletilir. **break** deyimi programın akışını for deyimi ile kontrol edilen bloğun sonuna aktarır.

```
int toplam = 0;
for (int i = 0; i <= 100; i++)
{
 toplam = toplam + i;
}
```

Yanda verilen kodda **for** ile kurulan döngüde 1'den 100'e kadar olan sayılar toplanmaktadır.

for döngüsü içinde bir başkası kurulabilir. İç-içe for döngülerinin nasıl kullanıldığını anlatmak için aşağıda verilen örneği hazırladık. Bu amaçla forma 2 ListBox yerleştirip ilk ListBox nesnesine tasarım anında **Items** özelliğinden yararlanıp bilgi aktardık.



Forma 2 ListBox yerleştirmekten söz ediyor olsak bile ilk ListBox'a aktarılan sayıları **Selection Sort** algoritması ile sıralayıp 2. ListBox'a aktarma fikrinin patentinin belki daha önceden alınmış olabileceği ihtimali karşısında 2. ListBox'ı silip yerine **RichTextBox** yerleştirdik. Ayrıca düğme adı benzerliği olmaması için Button nesnesinin adını değiştirdik.

"sort_et_kurban" adlı düğme tıkandığında işletilecek bu satırlarda ilk olarak bir dizi değişken tanımlandı. Dizi değişkenin eleman sayısı ListBox'ın elemanlarına göre belirlendi. Ardından ListBox'ın elemanlarını tanımladığımız bu dizi değişkene aktardık.

```
private void sort_et_kurban_Click(object sender, EventArgs e)
{
 int temp, item;
 item = listBox1.Items.Count;
 int[] dizi_dizi = new int[item];
 for (int i = 0; i <= item - 1; i++)
   dizi_dizi[i] = Convert.ToInt32(listBox1.Items[i].ToString());
 for (int i = 0; i <= item - 1; i++)
 {
  for (int j = 0; j <= item - 1; j++)
  {
   if (dizi_dizi[j] > dizi_dizi[i])
    {
    temp = dizi_dizi[i];
    dizi_dizi[i] = dizi_dizi[j];
    dizi_dizi[j] = temp;
   }
  }
 }
 for (int i = 0; i<= item - 1; i++)
  richTextBox1.Text = richTextBox1.Text +
              dizi_dizi[i].ToString() + "\n";
}
```

En az 30 yıldır az çok kod yazmış bütün programcılar tarafından "Selection Sort" olarak bilinen bu sıralama algoritmasında kullandığımız değişken adlarının başkalarına ait olabilme ihtimaline karşılık tedbir olarak "dizi" yerine "dizi_dizi" adını kullandık.

Bu kodda önce ListBox'ın mevcut elemanları for döngüsü ile okunup burada "dizi_dizi" adını verdiğimiz int tipindeki dizi değişkene aktarılmaktadır. Sonra iç-içe 2 for döngüsü hazırlanıp küçük ile büyüğü sürekli yer değiştirmek suretiyle elemanlar küçükten büyüğe doğru sıralanmaktadır. Sıralanmış dizi değişkenin içeriği ise RichTextBox'a aktarılmaktadır.

foreach Döngüsü

Koleksiyon veya dizi değişkenlerin elemanları üzerinde ileriye doğru hareket edilirken foreach döngülerinden yararlanmak programcı için kolaylıktır. **foreach** ile **IEnumerable** arayüzünü uygulamış koleksiyon nesneleri üzerinde ileri doğru hareket edilebilmektedir. Bu bölümde foreach döngüleri üzerinde teknik bilgi vermek yerine nasıl kullanıldığından söz edeceğiz.



foreach döngüsünün nasıl kullanıldığını anlatmak için üzerinde çalıştığımız projenin formuna **RichTextBox** ve **ListBox** nesnesi yerleştirdik. RichTextBox'a girilecek bilgiyi satır satır okuyup ListBox'a aktaracağız. Yani RichTextBox'ın her satırı ListBox'ta ayrı bir eleman olarak değerlendirecektir.

RichTextBox nesnesine çalışma anında girilen bilgiler nesnenin dahili **Lines** koleksiyonunda tutulmaktadır. Dolayısıyla RichTextBox'ın içeriğini foreach döngüsü ile satır satır elde etmek mümkündür. Foreach döngüleri anlatmak için RichTextBox nesnesi yerine dizi değişkenlerden yararlanabilirdik.

RichTextBox'a bilgi girilip "Naklet" başlığı verilen düğme tıklandığı zaman mevcut içeriğin ListBox'a aktarılmasını sağlamak için aşağıda verilen kodu hazırladık. Bu kodda **foreach** yerine **for** döngüsünü kullandık. Çünkü for ile foreach arasında karşılaştırma yapabilmenizi istedik.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int satir_sayi = richTextBox1.Lines.Length;
 for (int i = 0; i < satir_sayi; i++)
  {
   string str = richTextBox1.Lines[i];
   listBox1.Items.Add(str);
  }
}
```

RichTextBox'ın satır sayısı **Lines** koleksiyonun **Length** özelliğine bakılıp öğrenilmektedir. RichTextBox nesnesinin ilgilenilen satırın nosu Lines koleksiyonuna parametre olarak verilmektedir. Bu gibi işlemleri yaparken veya bir koleksiyonun içeriğini elde ederken for yerine **foreach** döngüsünü kullanarak yazmanız gereken satırları azaltabilirsiniz. Aşağıda verilen kod sayesinde RichTextBox nesnesinin bütün içeriği ListBox'a aktarılır.

```
private void Naklet_Click(object sender, EventArgs e)
{
 foreach(string str in richTextBox1.Lines)
  {
   listBox1.Items.Add(str);
  }
}
```

Burada asıl üzerinde durulması gereken foreach deyiminden sonra parantezlerin içine yazılanlardır. Söz konusu koleksiyon hangi tipte bilgi içeriyorsa önce o tipte bir değişken tanımlanmalıdır. RichTextBox sınıfının **Lines** koleksiyonun elemanları string tipte oldukları için "str" adında string bir değişken tanımlandık. Devamında "in" anahtar kelimesi ile belirtilen koleksiyonun sıradaki elemanı okunup parantezin içinde tanımlanan değişkene aktarılır. Bu değişkene aktarılacak eleman kalmadığında foreach döngüsünden çıkılır.

Yukarıda verilen foreach döngüsünün daha anlaşılır olmasını sağlamak için aşağıdaki gibi düzenledik. RichTextBox sınıfının Lines özelliği string tipte bir dizi değişken olduğu için derleyici bu yaptığımıza itiraz etmez.

```
private void naklet_Click_1(object sender, EventArgs e)
{
 string[] satirlar = richTextBox1.Lines;
 foreach (string str in satirlar)
 {
  listBox1.Items.Add(str);
 }
}
```

RichTex:Box sınıfının dahili koleksiyonu **Lines** özelliğine IEnumerable arayüzü uygulanmamış olsaydı satırları üzerinde **foreach** ile ileri yönde hareket etmek mümkün olmazdı. Bu kitabın ileriki sayfalarında IEnumerable arayüzü hakkında bilgi verilecektir. Aşağıda verilen foreach döngüsünde dizi değişkenin elemanları üzerinde ileri doğru hareket edildi.

```
string[] basket = new string[4];
basket[0] = "Kerem";
basket[1] = "Mirsat";
basket[2] = "Hidayet";
basket[3] = "Mehmet";
foreach (string oyuncu in basket)
  Console.WriteLine(oyuncu);
```

foreach deyimi 2 parametreye sahiptir. İlk parametrede koleksiyondan okunacak bilginin aktarılacağı değişkenin verilmektedir. Ardından içeriği okunacak koleksiyon 2. parametre olarak kullanılmaktadır. Bu 2 parametrenin arasına "in" yazılmaktadır.

Şimdi **for** ile **foreach** döngüleri arasındaki farkın anlaşılmasına katkı olması için başka bir örnek hazırlayacağız. Bu amaçla forma 2 ListBox yerleştirip ilk ListBox'a bazı ürünlerin adlarını, 2. ListBox'a ise fiyatlarını yazacağız. Sonra bu ürünlerin KDV'li fiyatlarını hesaplayıp 2. ListBox'a aktaracağız. Bu işlem için yazdığımız kodu aşağıda görebilirsiniz.

```
int sayi = listBox2.Items.Count;
for (int i = 0; i < sayi; i++)
{
 double fiyat = Convert.ToDouble(listBox2.Items[i]);
 double kdv = fiyat * 1.18;
 listBox2.Items[i] = kdv.ToString();
}
```

Gördüğünüz gibi for döngüsü içinde döngüye konu edilen veriler üzerinde işlem yapılmaktadır. For döngülerinde döngüye konu edilen dizi değişken veya koleksiyonun üzerinde ilerlerken elemanlara erişimde indeksleyiciden yararlanılıyor. Dolayısıyla söz konusu koleksiyonu döngünün içinde iken araya eleman eklenmediği sürece sorun yaşanmaz. Diğer yandan elemanları indekslenmemiş koleksiyonların üzerinde for döngüsü ile ilerlemek sorunludur. İndeksleyiciler hakkında ayrıntılı bilgiyi ilgili bölümde bulabilirsiniz.

Yukarıda for döngüsünün kullanıldığı kodun benzerini foreach döngüsü için hazırlamanız halinde hata meydana gelir. Çünkü foreach döngüsü içinde, döngüye konu edilen koleksiyonun (burada 2. ListBox'ın **Lines**) elemanlarında değişiklik yapılmasına izin verilmez.

```
int sayi = listBox2.Items.Count;
int i = 0;
foreach (string eleman in listBox2.Items)
{
 double fiyat=Convert.ToDouble(eleman);
 double kdv = fiyat * 1.18;
 listBox2.Items[i] = kdv.ToString();
 i = i + 1;
}
```

Çünkü foreach döngüsünde döngüye konu edilen dizi değişken veya koleksiyonla ilgili olarak geri planda birçok işlem yapılmaktadır. Bu nedenledir ki for döngüleri foreach döngülerine nazaran daha hızlı çalışmaktadır. Foreach döngülerinde indeksleyicilerden yararlanılmadığı için döngüye konu edilen nesnenin indeksleyiciye sahip olması şart değildir.

C# 3.0 ile birlikte programcılara **var** anahtar kelimesi ile değişken tanımlayabilme imkanı verildi. Söz konusu koleksiyonun hangi tipte bilgi içerdiğinden emin değilseniz "in" deyiminden önce değişken tanımlarken **var** anahtar kelimesinden yararlanabilirsiniz. Var anahtar kelimesiyle ilgili ayrıntılı bilgiyi bu kitabın ilgili bölümünde bulabilirsiniz.

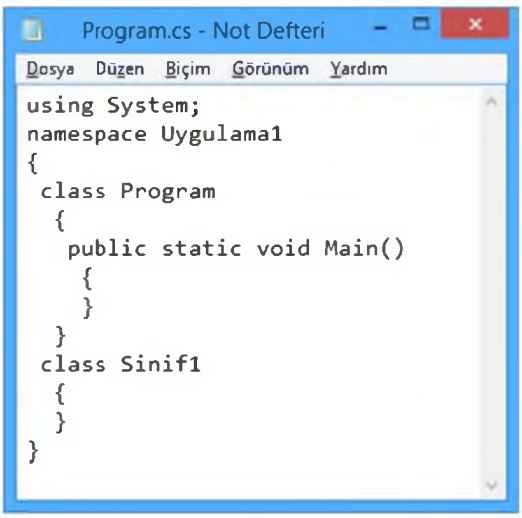
```
foreach (var eleman in koleksiyon)
 {
 }
```

Yukarıda foreach döngüleri hakkında yazılanları tekrarlamak gerekirse şunlar söylenebilir: foreach; **GetEnumerator()** metoduna sahip, başka bir deyişle **Enumerable** olan koleksiyonların elemanları üzerinde ilerlerken kullanılmaktadır.

10. BÖLÜM
CLASS'lar ve NESNEYE YÖNELİK PROGRAMCILIK

Bu kitabın ilk bölümünde ifade edildiği üzere C# uygulamalarında ayrı kod dosyaları olarak düzenlenebilinen struct'lar, interface'ler ve enum'lar olsalar bile uygulamaların temelini class'lar oluşturmaktadır. Ötesi, class'sız C# uygulaması söz konusu değildir. Bu bölümde class'ları mercek altına alınırken geri planda nesneye yönelik programcılıkla ilgili temel kavramlarından söz edilecektir. Bu kitabın daha önceki bölümlerinde class'lardan az çok söz etmiş olsak bile bu bölümde class'ları temelden anlatmayı tercih ettik.

Class'lar hakkında konuşmaya aşağıda verilen konsol uygulamasını hazırlamakla başladık. Bu uygulamada aynı kod dosyasında yer alan 2 sınıf bulunmaktadır. İsteseydik her sınıf için ayrı bir kod dosyası hazırlayabilirdik. Bu konsol uygulamasındaki 2. sınıfta henüz herhangi bir metot yoktur. Ekran görüntüsü olarak verilen bu uygulama hatasızca derlenir ama çalıştırıldığında herhangi bir işlem yapmaz. Ayrıca bu 2 class'ı bir namespace'e yerleştirmek üzere **namespace** anahtar kelimesine yer verdik. Bu C# uygulamasını Not Defteri ile hazırladık. Sizler bu uygulamayı hazırlarken Visual Studio'dan yararlanabilirsiniz.



C# uygulamasında birden fazla sınıfta Main() metoduna yer verildiği zaman konsolda derleme yapılırken "**/main**" parametresi ile hangi sınıftaki Main() metodunun uygulamanın başlangıç noktası olarak kullanılmak istendiğini belirtmek gerektiğini "Konsol Uygulamaları" adlı bölümden hatırlıyor olmalısınız.

Visual Studio ile hazırladığınız uygulamanızda birden fazla sınıfta Main() metodu varsa **Project Properties** penceresinde **Startup Object** liste kutusunda seçim yapmak gerekir.

Şimdi tekrar niyetine, yaklaşık 10 satırdan meydana gelen bu C# uygulamasına yakından bakıp gördüklerimizi maddeler halinde anlatacağız.

- Visual Studio'dan etkilenip 2 sınıf içeren bu kod dosyasına "Program.cs" adını verdik. Ayrıca uygulamadaki ilk sınıfa (Main metoduna sahip olan sınıf) Visual Studio ile hazırlanan konsol uygulamalarında olduğu gibi "Program" adını verdik.
- Bu kod dosyası dahilinde hazırlayacağımız sınıflarda, NET Framework ile gelen "System" adlı nasmespace'teki sınıflardan yararlanmayı düşündüğümüz için kod dosyasının en başında "using" anahtar kelimesine yer verdik.
- Class'lar hakkında temel bilgi vermek amacıyla hazırlamaya başladığımız bu uygulamada çok sayıda class ve kod dosyası olmayacaktır. Buma rağmen bu kod

dosyasında **namespace** anahtar kelimesine yer verip bu uygulamayla ilgili olarak hazırlayacağımız sınıfları burada "Uygulama1" adını verdiğimiz namespace'te toplayacağız. Mevcut duruma göre hem "Program" hem de "Sinif1" adını verdiğimiz class'lar aynı namespace'e üyedir.

- C# uygulamalarında Main() metodunun bulunması şart olduğu için ilk sınıfta Main() metoduna yer verdik. Bu uygulamada Main() metodu ilk sınıf yerine 2. sınıfta olsaydı değişen bir şey olmazdı. Çünkü C# derleyicisi Main() metodunu bulup kodu oradan itibaren çalıştırmaktadır.
- Uygulama için hazırladığımız Main() metodunun parametresiz halini kullandığımızı fark etmiş olmalısınız. Visual Studio ile hazırlanan uygulamalarda Main() metodu "args" adında ve string tipte bir parametreye sahip olduğu için C# programcılarının Main() metodunu parametreli kullanma alışkanlıkları vardır. Konuya dikkatinizi çekmek için burada parametresiz halini kullandık.
- Hazırladığımız uygulamadaki her iki class'ın adlarından önce herhangi bir erişim belirleyicisine yer vermedik. İsteseydik "class" anahtar kelimesinden önce public, private, protected veya internal yazarak burada hazırladığımız class'ların nerelerden erişebilir olacağı konusunda ayarlama yapabilirdik. Uygulamamızdaki 2. class için erişim belirleyicisi olarak private'yi seçmiş olsaydık ilk class (burada Program) dahilinde 2. class'tan (burada Sinif1) yararlanamazdık.
- Class anahtar kelimesinden önce herhangi bir erişim belirleyicisi yazılmazsa bunun **internal** olduğu varsayılır. Başka bir deyişle yukarıda hazırladığımız uygulamadaki 2 sınıfa uygulamamızdaki herhangi bir yerden erişim sağlanabilir.



Burada araya girip class'larla ilgili birkaç öneride bulunacağız. Bu öneriler kurumsal bir firmada C# kodu yazmaya başladığınızda işinize yarayacaktır.

1- Her ne kadar yukarıda aynı kod dosyası içinde 2 class'a yer vermiş olsak bile birden fazla kapsamlı class'ın aynı kod dosyasında kodlanması önerilmez.

2- Kod dosyalarına vereceğiniz ad ile, class'a vereceğiniz ad birebir aynı olmasa bile yakın olmasında fayda vardır.

3- Hazırladığınız class, sonradan aklınıza gelen işlemler, özellikler ve metotlardan dolayı çok büyümüş olabilir. Bir kod dosyasında binlerce satırın olması önerilmez. Bu nedenle sınıfınızı tasarlarken kod satırı sayısını baştan tahmin etmeye çalışmak ve ona göre yol almak iyi bir alışkanlık olacaktır.

4- Hazırladığınız sınıfın aşırı büyümesini istemiyorsanız sınıfınızın iyi düşünülmüş bir tek amacı olmalıdır. .NET Framework ile gelen sınıflardan örnek vermek gerekirse, Math ile Random sınıfları tek sınıfta birleştirilmiş olsaydı programcılar için iyi olmazdı. Bu nedenle sınıfınız üzerinde çalıştıkça, metotları kodladıkça uzayıp karmaşık olmaya başladıysa uygun parçalara ayırmayı denemelisiniz.

5- C# programcılarının sınıflara, metotlara, özelliklere verdikleri adların ilk harfini büyük yazmak gibi alışkanlıkları vardır. Yerel değişkenlere vereceğiniz adların ilk harflerinin küçük olmasını tercih edebilirsiniz. Bu kitap metni içinde hazırladığımız kodlarda bu önerilere harfiyen uyduğumuzu söyleyemeyiz.

Şimdi yukarıda verilen uygulamadaki 2. sınıfta bir metot hazırlayacağız. Bu metodu nesne hazırlamadan direk sınıf üzerinden kullanabilmek için başına **static** anahtar kelimesini

yazdık. C# programcılarının metot adlarının ilk harfini büyük olarak seçmek gibi bir alışkanlığı olduğunu hatırlamak ve metoda verilen adın "fiil" olması yerinde olacaktır.

```
using System;
namespace Uygulama1
{
 class Program
  {
   public static void Main()
    {
    }
  }
 class Sinif1
  {
  public static void Mesaj()
   {
    Console.Write("Devam etmek için Enter tuşuna basın");
    Console.ReadLine();
   }
 }
}
```

Gördüğünüz gibi 2. sınıfta geriye herhangi bir değer göndermeyen (**void**) ve herhangi bir parametreye sahip olmayan "Mesaj" adında bir metot bulunmaktadır. Bu metodu diğer sınıflarda (burada Program) kullanabilmek için başına erişim belirteci olarak **public** yazdık.

Burada Mesaj() adını vermiş olduğumuz metodun adından önce public anahtar kelimesine yer vermeyip, başka bir deyişle metot için herhangi bir erişim belirleyicisi kullanmamış olsaydık bu metodun **private** bir metot olduğu varsayılırdı. Dolayısıyla bu metot sadece üyesi olduğu sınıfta (burada Sinif1) kullanılabilirdi. Sinif1'de hazırladığımız Mesaj() metodunu "Program" adlı sınıfta kullanmak istediğimiz için erişim belirleyici olarak public'i seçtik. Public yerine **internal**'i seçebilirdik.



Erişim belirleyicileri konusunda kafanızın karışmaması için ek açıklama yapma gereğini duyduk:

* Hazırladığınız class için erişim belirleyici seçmezseniz derleyici bunun **internal** olduğunu varsaymaktadır. internal erişim belirleyicisine sahip class'lara, üzerinde çalışılan uygulamadaki diğer class'lardan erişilebilinmektedir.

* Yukarıda hazırladığımız "Sinif1" adlı class için erişim belirleyici kullanmadık. Dolayısıyla derleyici sanki class anahtar kelimesinden önce "internal" yazılmış gibi bu sınıfa "Program" sınıfı içinden erişim sağlamamıza izin verilir.

* Uygulamanızdaki herhangi bir class dahilinde hazırladığınız metot için erişim belirleyicisi kullanmazsanız derleyici bu metodun **private** olduğunu varsayar.

Şimdi Mesaj() adını verdiğimiz metodu ilk sınıfta yani Main() metoduna sahip sınıfta kullanacağız. Bu metot başka bir sınıfın içinde olmasaydı metodu işletmek için metot adını yazmak yeterli olurdu. "Mesaj()" adını verdiğimiz bu **static** metot başka bir sınıfta bulunduğu için metot adından önce üyesi olduğu sınıfın adını yazdık.

```
namespace Uygulama1
{
 class Program
  {
   public static void Main()
    {
      Sinif1.Mesaj();
    }
  }
 class Sinif1
  {
  public static void Mesaj()
   {
    Console.Write("Devam etmek için Enter tuşuna basın");
    Console.ReadLine();
   }
 }
}
```

Bu şekilde düzenlenen konsol uygulaması derlenip çalıştırılırsa programın işletimi Mesaj() metodu dahilinde kullanılan **ReadLine**() metodu ile askıya alınır. Programın işletimi ReadLine() metodu tarafından askıya alınmış halde iken Enter tuşuna basılırsa programın çalışması sona erer. Şimdi yukarıda verilen metodun başlangıç satırındaki **static** anahtar kelimesini sileceğiz.

```
class Sinif1
{
 public void Mesaj()
  {
   Console.Write("Devam etmek için bir tuşa basın");
   Console.ReadLine();
  }
}
```

Bu şekilde düzenlenen konsol uygulaması derlendiğinde hata meydana gelir. Çünkü static olmayan metotlar **"Sınıf adı.Metot adı"** şeklinde çağrılıp işletilemezler. **static** olmayan metotları işletebilmek için önce o sınıftan yola çıkıp nesne hazırlamak gerekiyor. Bu nedenle Main() metoduna aşağıdaki gibi ekleme yaparak "Sinif1" tipinde bir nesne hazırladık.



Bir metodu hangi durumlarda static, hangi durumlarda normal (instance) yapmak gerektiğinin net bir cevabı yoktur. Programcılar genellikle nesneyle direk ilgisi olmayan ve sıklıkla kullanılan metotları statik yapıyorlar. İleri bir tarihte bir yazılım firmasının bünyesine katıldığınızda statik metotların hangi şartlarda tercih edildiği firma politikası olarak önceden kurala bağlanmış olabilir.

```
class Program
{
 public static void Main()
  {
   Sinif1 obje;
   obje = new Sinif1();
  }
}
```

Yanda işaret edilen 2 satır ile önce "Sinif1" tipinde bir değişken tanımlanmakta ve sonra Sinif1'den yola çıkılarak new anahtar kelimesi sayesinde "obje" adı verilen nesne hazırlanmaktadır.

Daha önceki konulardan bildiğiniz gibi bu 2 satırı veya 2 işlemi aşağıda yapıldığı gibi tek satırda toplamak mümkündür. Sınıftan yola çıkarak nesne hazırlandıktan sonra metodun adının önüne nesnenin adını yazarak işletebilirsiniz. Bundan şu sonucu çıkarabilirsiniz: Statik olmayan metotlar nesne üzerinden işletilmektedir.

```
class Program
{
 public static void Main()
  {
    Sinif1 obje = new Sinif1(); // Nesne hazırlanır.
    obje.Mesaj(); // Metot işletilir.
  }
}
class Sinif1
{
 public void Mesaj()
  {
    Console.Write("Devam etmek için bir tuşa basın");
    Console.ReadLine();
  }
}
```

Bu kodda sınıftan yola çıkarken **new** anahtar kelimesi ile nesne hazırlanırken **Sinif1()** adında bir metot kullanılmaktadır. Bu yorumu şuna dayanarak yaptık: C# kodu dahilinde adına "()" eklenen kelimelerin metot olduğunu anlamak gerek. Yani yukarıda new anahtar kelimesinden sonra yazılan "Sinif1()" hazırladığımız sınıf olmayıp bu sınıfla ilgili bir metottur.

Bu tespitten ayrıca şu sonucu çıkarabilirsiniz. Hazırlanan her sınıf, kendisi ile aynı ada sahip ve nesne hazırlamaya yarayan bir metoda sahiptir. Sınıftan yola çıkılarak nesne hazırlanırken kullanılan bu metodu programcı yazmadıysa C# derleyicisi geri planda otomatik olarak hazırlamaktadır.

Sınıftan yola çıkılarak nesne hazırlanırken kullanılan ve sınıf ile aynı ada sahip (programcı hazırlamadıysa derleyici tarafından otomatik olarak hazırlanan) metoda yapıcı metot (**constructor**) adı verilmektedir. Başka bir deyişle constructor adı verilen metot, sınıfın örneği **new** anahtar kelimesi ile hazırlarken kullanılan metottur denilebilir.

Şimdi "Sinif1" adını vermiş olduğumuz bu sınıfa, sınıf ile aynı ad sahip ve işletilebilir herhangi bir satıra sahip olmayan bir metot ekleyeceğiz. Bu şartlarda üzerinde çalıştığımız konsol uygulamasındaki 2. sınıf 2 metoda sahip olacaktır.

```
class Sinif1
 {
  public void Mesaj()
   {
     Console.Write("Devam etmek için bir tuşa basın");
     Console.ReadKey();
   }
  public Sinif1() // yapıcı metot
   {
   }
}
```

Üzerinde çalıştığımız konsol uygulamasındaki 2. sınıftaki 2. metoda sınıfın adını vermiş olduğumuz için bu metot **constructor** metot görevini görür.

Bu metot geriye bir değer göndermemesine rağmen imzasında **void** deyimini kullanmadık. Bu metot, sınıf ile aynı ada sahip olmasaydı bu şartlarda hata meydana gelirdi. Konunun devamında **yapıcı**, diğer adıyla **constructor** metotlar hakkında bilgi verilecektir. İlgili sınıf tipinde nesne hazırlanırken new anahtar kelimesi ile birlikte yapıcı metodu kullanmak gerektiği için burada yapıcı metotlardan kısaca söz ettik.

Sınıfların Alanlara Sahip Olması

Yukarıda hazırladığımız sınıflarda yalnızca metotlar vardı. C# ve nesneye yönelik programcılıkta metotlar hakkında yazacaklarımız fazla olduğu için metotlar için ayrı bir bölüm hazırladık. Bu nedenle bu bölümde metotlardan kısaca söz ettikten sonra alanlara geçmeyi tercih ettik.

Sınıfların metotlardan sonra önemli üyesi alanlardır. Alanlar bilgi tutmak amacıyla sınıf dahilinde tanımlanan değişkenlerdir. Başka bir deyişle class bloğu içinde tanımlanan değişkenlere **alan** adı verilmektedir. Bu kavram karışıklığının nedeni, nesneye yönelik programlama dilleri anlatılırken klasik programlama dilleriyle ilgili alışkanlıkların sürdürülmek istenmesidir. Bu kitapta kurulan her cümlede terminolojiye bağlı olmayı gözetmek istesek de bazen C# programlama dilini C'yi anlatır gibi anlatmış olabiliriz.

Değişkenler adlı bölümde ifade edildiği gibi değişkenler hakkında bilgi verirken klasik programlama derslerinden gelen alışkanlığa uyup konuya nesneye yönelik programcılık penceresinden bakmamıştık. Söylenenler havada kalmasın diye "Değişken Tanımlamak ve Kullanmak" adlı bölümünde sözü edilen birkaç ayrıntıdan burada söz edeceğiz:

* C# derleyicisi metotların içinde public değişkenlerin tanımlanmasına izin vermez. Public değişkenler ancak class bloğu içinde tanımlanabilir.

* Class bloğu içinde public olarak tanımlanan değişkenlere başka bir class'tan erişim sağlanabilir.

* Metot içinde tanımlanan değişkenlere **local değişken** adı verilir ve metodun dışında yaşamazlar.

* Class bloğu içinde **private** olarak tanımlanan değişkenlere başka sınıflardan erişilemez ve bu değişkenler sınıfa özel olurlar.

* Bu bölümde alan olarak ifade ettiğimiz değişkenler class bloğu içinde tanımlanan değişkenlerdir

Sınıflar üzerinde konuşmaya devam etmek üzere her ay adresimize gelen cep telefonu ile ilgili faturayı temel alan basit bir sınıf hazırlayacağız. Bu sınıfımızda önce alanlar olacak. Devamında bu sınıfta metotlara yer vereceğiz. "Cep_Fatura" adını verdiğimiz sınıfın 4 alanı olacağı için sınıf içinde 4 **public** değişken tanımladık.

```
using System;
namespace Uygulama1
{
 class Program
 {
  public static void Main()
    {
    }
 }
```

Daha önce C# veya nesneye yönelik programcılık hakkında az çok bilgi edinmiş olanlar bu alanların public yapılmasına itiraz edebilirler. Konuyu adım adım anlatmayı tercih ettiğimiz için alanların başlangıçta public olmasını tercih ettik.

```
public class Cep_Fatura
{
 public string Sahip;
 public string Numara;
 public string Adres;
 public int Tutar;
}
}
```

Bu sınıftan yola çıkıp nesne hazırlayabilmek ve alanlara bilgi aktarıp içeriklerimi okurken yararlanmak üzere aynı kod dosyası içinde "Program" adında ve Main() metoduna sahip ikinci bir sınıf hazırladık.

Bu şekilde sınıfta bulunmasını istediğimiz her alan için class bloğunda değişken tanımladıktan sonra uygulamamızda Main() metoduna sahip diğer sınıfa geçtik. Bu sınıfta **new** anahtar kelimesinden yararlanıp "Cep_Fatura" tipinde bir nesne hazırladık. "f1" adını verdiğimiz bu nesnenin alanlarına herhangi bir bilgi aktarmadan içeriklerini ekrana yazdık.

```
namespace Uygulama1
{
 class Program
 {
  public static void Main()
   {
    Cep_Fatura f1 = new Cep_Fatura();
    Console.WriteLine(f1.Sahip);
    Console.WriteLine(f1.Numara);
    Console.WriteLine(f1.Adres);
    Console.WriteLine(f1.Tutar);
   }
 }
 public class Cep_Fatura
 {
  public string Sahip;
  public string Numara;
  public string Adres;
  public int Tutar;
 }
}
```

Bu konsol uygulaması derlenip çalıştırıldığında ekrana önce 3 kez boşluk ardından bir kez 0 yazılır. Çünkü **new** anahtar kelimesi ile Cep_Fatura sınıfından yola çıkıp nesne hazırlanırken varsayılan yapıcı metot (derleyici tarafından geri planda hazırlanan metot) sayesinde sınıfın **string** tipindeki alanların başlangıç içerikleri boşluk ve **int** tipindeki alanın başlangıç içeriği ise 0 olarak ayarlanır. Şimdi hazırladığımız nesnenin alanlarına kendimiz bilgi aktarıp ondan sonra içeriklerini konsola yazdıracağız.

```
public static void Main()
{
 Cep_Fatura f1 = new Cep_Fatura();
 f1.Sahip = "Seçkin Yayıncılık";
 f1.Numara = "0-533-770 50 99";
 f1.Adres = "Mustafa Kemal Mah 2158. Sok No:13";
 f1.Tutar = 155;
 Console.WriteLine(f1.Sahip);
 Console.WriteLine(f1.Numara);
 Console.WriteLine(f1.Adres);
 Console.WriteLine(f1.Tutar);
}
```

Main() metodu bu şekilde düzenlenen konsol uygulaması derlenip çalıştırıldığında "Cep_Fatura" sınıfından yola çıkılarak hazırlanan nesnenin alanlarına aktarılan bilgiler konsola yazılır. Cep_Fatura adını verdiğimiz sınıfın alanları public olmasaydı nesne üzerinden alanlara bu şekilde bilgi aktarıp okuyamayabilirdik.

Yukarıda "Cep_Fatura" sınıfından yola çıkarak "f1" adında bir nesne hazırladık ve bu nesnenin alanlarına bilgi aktarıp içeriklerini konsola yazdık. Bu sınıftan istediğiniz sayıda nesne oluşturabilirsiniz. Aşağıda verilen satır ile "f2" adında 2. bir nesne hazırlanır.

```
Cep_Fatura f2 = new Cep_Fatura();
```

Bu satırda geri planda gelişen olaylar önemli olduğu için tekrar niyetine maddeler halinde birkaç cümle yazacağız:

- Bu satırda ilk yazılan bir tiptir, bir sınıftır. Bu sınıf .NET Framework ile gelen bir sınıf olabileceği gibi programcı tarafından yazılmış da olabilir. Olmayan bir tipin adı yazıldığında derleyici hata rapor eder.
- C# uygulaması geliştirirken Visual Studio kullanacağımız için mevcut olmayan bir tipin veya sınıfın adı yazıldığında Visual Studio anında uyarıda bulunur.
- Tip adından sonra yazdığımız "f2" hazırlanacak nesnenin sembolik adıdır. Satırın devamında **new** anahtar kelimesine yer verildiği için satırın en başında yazılan (burada Cep_Fatura) sınıfın yapıcı metodunu kullanmak gerekecektir.
- Metot adlarına "()" parantezleri eklendiği için neyin metot neyin tip veya sınıf olduğu kolayca ayırt edilebilmektedir. Çalıştırılmasına new anahtar kelimesinin aracılık ettiği yapıcı metot, geriye sınıf tipinde bir nesnenin referansını göndermektedir. Burada bu nesnenin referansı "f2" adını verdiğimiz değişkene aktarılmaktadır. Dolayısıyla Cep_Fatura tipindeki nesnenin adresi f2'de tutulduğu için f2 demek bir bakıma Cep_Fatura tipinde bir nesne demek oluyor.

Normalde programcılar sabit bilgileri veritabanına yazıp gerek duyduklarında bu bilgileri veritabanından okuyup nesnenin alanlarına aktarırlar. Burada veritabanı hazırlayıp işlemleri ona göre yapma imkanımız olmadığına göre gerek duyduğumuz bilgileri ya doğrudan koda yazıyoruz ya da kullanıcıdan istiyoruz. Şimdi kullanıcıdan cep telefonu faturasının ayrıntılarını isteyeceğiz. Bu amaçla "Cep_Fatura" adını verdiğimiz sınıfta **Toplam**() adında bir metot hazırladık ve Main() metodu içinde bu metodu işlettik. Main() metoduna sonradan eklediğimiz ve Toplam() metodunu işlettiğimiz satıra dikkat ediniz.

```
using System;
namespace Uygulama1
{
 class Program
 {
  public static void Main()
  {
   Cep_Fatura f1 = new Cep_Fatura();
   f1.Sahip = "Seçkin Yayıncılık";
   f1.Numara = "0-533-770 50 99";
   f1.Adres = "Mustafa Kemal Mah 2158. Sok No:13";
```

```
        f1.Tutar = f1.Toplam();   // Toplam() metodu işletiliyor
        Console.WriteLine(f1.Sahip);
        Console.WriteLine(f1.Numara);
        Console.WriteLine(f1.Adres);
        Console.WriteLine(f1.Tutar);
      }
    }
    public class Cep_Fatura
    {
      public string Sahip;
      public string Numara;
      public string Adres;
      public int Tutar;
      public int Toplam()
      {
        Console.Write("Konuşma tutarı :");
        int konusma = Int32.Parse(Console.ReadLine());
        Console.Write("Mesaj tutarı :");
        int mesajlar = Int32.Parse(Console.ReadLine());
        Console.Write("İnternet tutarı :");
        int internet = Int32.Parse(Console.ReadLine());
        return konusma + mesajlar + internet;
      }
    }
  }
```

Dışarıdan bilgi almayan dolayısıyla parametreye sahip olmayan Toplam() metodunu public yaptık. Çünkü bu metodu uygulamadaki başka sınıflarda kullanmak istiyoruz. Bu metot geriye int tipinde bilgi göndereceği için adından önce int yazdık.

Bu metotta önce int tipinde 3 değişken tanımladık. Ardından bu değişkenlere aktarmak üzere kullanıcıdan istenen bilgileri konsola yazmasını istedik. Kullanıcın konsola yazdığı 3 bilgiyi toplayıp geriye gönderdik. Bu metot tarafından geriye gönderilen bilgi "Cep_Fatura" adını verdiğimiz sınıftan yola çıkılarak hazırlanan nesnenin "Tutar" alanına aktarılacaktır.

Bu kodda işlev bakımından baktığınızda sizin için ilgi çekici herhangi bir yanı olmayabilir. Böyle bir kodu hazırlamamızdaki amaç, sizleri nesneye yönelik programcılık kavramlarıyla tanıştırmaktır. Örneğin yukarıda, uygulamadaki 2. sınıfın önce alanlara sahip olmasını sağladık. Devamında Toplam() adını verdiğimiz bir metot hazırlayıp kullandık.

Şimdi sırada "Cep_Fatura" adını verdiğimiz sınıf için yapıcı metot hazırlamak var. Programcılar, yapıcı metotlarda genellikle hazırlayacakları nesnenin alanlarına ilk değerleri aktarırlar. Bu alışkanlığa bağlı kalıp "Cep_Fatura" sınıfımız için hazırlayacağımız yapıcı metot dahilinde kullanıcıdan telefon sahibinin adı, adresi ve numarası istenecek. Sınıfımızın yapıcı metot hazırlanmış hali aşağıdaki gibidir.

```
public class Cep_Fatura
{
  public string Sahip;
  public string Numara;
  public string Adres;
  public int Tutar;
```

```
public Cep_Fatura()
{
 Console.Write("Telefonun Sahibi :");
 Sahip = Console.ReadLine();
 Console.Write("Fatura adresi :");
 Adres = Console.ReadLine();
 Console.Write("Telefon no :");
 Numara = Console.ReadLine();
}
public int Toplam()
{
 int konusma, mesajlar, internet;
 Console.Write("Konuşma tutari :");
 konusma = Int32.Parse(Console.ReadLine());
 Console.Write("Mesaj tutari :");
 mesajlar = Int32.Parse(Console.ReadLine());
 Console.Write("İnternet tutarı :");
 internet = Int32.Parse(Console.ReadLine());
 return konusma + mesajlar + internet;
}
}
```



İşe yarar programlarda sınıfın alanlarına ilk değer aktarma işleminin bu şekilde yapılmayacağını bilmenizi isteriz. Doğrusu, bu yapıcı metodun parametrelere sahip olması ve sınıfın alanlarına aktarılacak bilgilerin sınıfı kullanan sınıf dahilinde (burada Program) hazırlanması gerekirdi. Buradaki amacımız deneysel olduğundan bu gibi ayrıntılara bazen dikkat etmiyoruz.

Bu metot, sınıf ile aynı ada sahip olduğu için **new** anahtar kelimesiyle nesne hazırlandığı zaman otomatik olarak işletilerek kullanıcıdan bu bilgileri girmesi istenir. Bu şekilde yapıcı metodu hazırladıktan sonra uygulamamızdaki Main() metodunun olduğu diğer sınıfı aşağıdaki gibi düzenledik.

```
class Program
{
 public static void Main()
  {
   Cep_Fatura f1 = new Cep_Fatura();
   f1.Tutar = f1.Toplam();
   Console.WriteLine(f1.Sahip);
   Console.WriteLine(f1.Numara);
   Console.WriteLine(f1.Adres);
   Console.WriteLine(f1.Tutar);
 }
}
```

Cep_Fatura sınıfımızın örneğini bu şekilde alıp alanların içeriklerini konsola yazdıktan sonra şimdi sor. durum hakkında biraz konuşacağız.

- Yukarıda "Program" adını verdiğimiz sınıfta Main() metodunda Cep_Fatura sınıfı tipinde nesne hazırladık ve bu nesnenin üzerinden Toplam() metodunu işlettik. Başka bir deyişle "f1" adını verdiğimiz nesne sadece Main() metodu içinde yaşar.

- "f1" nesnesinin alanlarının içeriklerini yine Main() metodu içinde konsola yazdırdığımız için herhangi bir sorunla karşılaşmadık. Nesneye yönelik programcılık ilkeleri bize yazdırma işleminin sınıf içinde gerçekleştirilmesinin daha iyi olacağını söylemektir.
- Bu durumda yaşam alanı Main() metodunun içiyle sınırlı bir nesneye başka bir yerde erişip kullanmak mümkün olmadığında göre "Cep_Fatura" adını verdiğimiz sınıfta aşağıdaki gibi bir metot hazırlamaya çalışırsak hata meydana gelir.

```
public void Yaz()
{
 Console.WriteLine(f1.Sahip);
 Console.WriteLine(f1.Numara);
 Console.WriteLine(f1.Adres);
 Console.WriteLine(f1.Tutar);
}
```

Bu metotta hataya neden "f1" yerine içinde bulunulan sınıf anlamında "this" yazılırsa hatalı durum düzeltilmiş olur. Uygulamamızın son hali aşağıdaki gibi oldu. Bir sonraki paragrafta "this" anahtar kelimesi hakkında biraz konuşacağız. Bu kodlar size basit gelse bile dikkatle incelemenizi öneririz. Çünkü nesneye yönelik programcılıkla ilgili çok sayıda yapı kendisine bu kodda yer buldu.

```
namespace Uygulama1
{
 class Program
 {
  public static void Main()
  {
   Cep_Fatura f1 = new Cep_Fatura();
   f1.Tutar = f1.Toplam();
   f1.Yaz();
  }
 }
 public class Cep_Fatura
 {
  public string Sahip;
  public string Numara;
  public string Adres;
  public int Tutar;
  public Cep_Fatura()
  {
   Console.Write("Telefonun Sahibi :");
   Sahip = Console.ReadLine();
   Console.Write("Fatura adresi :");
   Adres = Console.ReadLine();
   Console.Write("Telefon no :");
   Numara = Console.ReadLine();
  }
  public int Toplam()
  {
   Console.Write("Konuşma tutari :");
   int konusma = Int32.Parse(Console.ReadLine());
```

```
        Console.Write("Mesaj tutari :");
        int mesajlar = Int32.Parse(Console.ReadLine());
        Console.Write("İnternet tutarı :");
        int internet = Int32.Parse(Console.ReadLine());
        return konusma + mesajlar + internet;
      }
      public void Yaz()
      {
        Console.WriteLine(this.Sahip);
        Console.WriteLine(this.Numara);
        Console.WriteLine(this.Adres);
        Console.WriteLine(this.Tutar);
      }
    }
}
```

Bu sınıfın bütün alanları public olduğu için herhangi bir sınıftan bu alanlara erişim sağlanıp içerikleri istenildiği gibi değiştirilebilir. Örneğin normalde alana artı bir değer aktarılması gerekirken kullanıcı eksi bir değer girilebilir. Bu tutarsızlıklara neden olabilir. C# programlama dilinde public alanlarla ilgili muhtemel sorunların çözümü için özelliklerden (property) yararlanma yoluna gidilmektedir. Bu bölümde özellikler ayrıntılı bir şekilde incelenecektir.

This Anahtar Kelimesi

C# konulu kaynaklarda this anahtar kelimesi genellikle içinde bulunulan veya geçerli sınıf olarak anılmaktadır. This, gerçekte nesneye yönelik programcılıkta özünün özü bir kavramdır. Dolayısıyla henüz yolun başında olan birisi bir cümlelik açıklamadan şey anlamayabilir. Anlamaz ama birden fazla kaynakta this'in nasıl kullanıldığını gören programcı adayının this'le fazla bir sorunu olmaz.

Madem yukarıda this'i kullanmak durumunda kaldık, Cep_Fatura sınıfı üzerinde anlatmak daha anlamlı olabilir. Maddeler halinde mevcut durumu anlatmayı deneyeceğiz:

- Cep_Fatura adında bir sınıfımız var. Bu sınıf, birisi yapıcı olmak üzere toplam 3 metoda sahiptir. Bu metotların hepsi instance, yani static değildir. Dolayısıyla yapıcı metot hariç bu metotlar nesne üzerinden işletilecektir.
- Demek ki "Cep_Fatura" sınıfımızın Yaz() ve Toplam() metotlarını işletebilmek için nesneye ihtiyacımız vardır. Bu metotlar static olsaydı nesne üzerinden değil sınıf üzerinden işletilirdi. Tabi konu nesne üzerinden işletilen metot olunca yüzlerce nesne söz konusu olabilir. Programcı **new** anahtar kelimesini kaç kez kullanıp sınıfının yapıcı metodunun işletilmesini sağladıysa o kadar nesne söz konusudur.
- Tam bu noktada yukarıda verdiğimiz "Program" sınıfının Main() metodunun son halini biraz değiştirip buraya aldık.

```
public static void Main()
{
 Cep_Fatura f1 = new Cep_Fatura();
 f1.Tutar = f1.Toplam();
 f1.Yaz();
```

```
        Cep_Fatura f2 = new Cep_Fatura();
        f2.Tutar = f2.Toplam();
        f2.Yaz();
      }
```

Main() metodunda Cep_Fatura tipinde 2 nesne hazırlanmakta ve her iki nesnede de "this" anahtar kelimesini içeren Yaz() metodu işletilmektedir. Yaz() metodu, buradaki adıyla söylemek gerekirse, "f1" üzerinden işletildiğinde this, f1'i temsil eder, Yaz() metodu f2 nesnesi üzerinden işletildiğinde ise this anahtar kelimesi f2'yi temsil eder.

Şimdi bu bilgilerin ışığında Cep_Fatura sınıfımızın Yaz() metoduna tekrar bakacağız.

```
      public void Yaz()
      {
        Console.WriteLine(this.Sahip);
        Console.WriteLine(this.Numara);
        Console.WriteLine(this.Adres);
        Console.WriteLine(this.Tutar);
      }
```

Burada kullanılan this'in anlamı bizce şudur: Muhtemelen ileri biz zamanda bu sınıftan yola çıkılarak nesne hazırlanacaktır. İşte buradaki this her hangi bir zamanda hazırlanacak "Cep_Fatura" tipindeki nesnelerden herhangi birisidir.

Private Alanlara Dışarıdan Erişim

Daha önceki konularda işaret edildiği üzere C# uygulamaları dahilinde hazırlanan class'larda metot ve alanlardan başka ayrıca özellikler bulunmaktadır. Nesneye yönelik programcılıkta özellikler, private alanların içerikleri okunup yazılırken kullanılmaktadır. Yukarıdaki sayfalarda hazırlayıp üzerinde konuştuğumuz "Cep_Fatura" sınıfımızdaki bütün alanlar public'ti. Dolayısıyla özellik (property) hazırlamaya gerek duymadık.

Konunun arka planını kavramanız için başlangıçta C# dilinde tıpkı Java'da olduğu gibi özelliklerin olmadığını varsayıp ona göre private alanlar üzerinde işlem yapacağız. Böyle yapmakla konuyu uzatmış olacağız ama konu üzerinde düşünmüş ve bazı ayrıntıları kavramış olacaksınız.

Yukarıdaki sayfalarda hazırladığımız class'ın bütün alanlarını public yapmıştık. Alanlar public oldukları için her yerden bu alanlara erişip içeriklerini okuyup yazmıştık. Alanların public yapılmasının önünde herhangi bir engel olmasa bile nesneye yönelik programcılığın ilkelerine aykırıdır. Dolaysız söylemek gerekirse, sınıfın bazı alanlarına dışarıdan doğrudan erişime izin verilmesi sorunlara neden olabilir. Bu sorunun çözümü, ilgili alanı **private** yapıp alanın içeriğini okurken ve/veya değiştirirken kullanmak üzere 2 özel metot hazırlamaktır. Bu işleme nesneye yönelik programcılıkta kapsülleme (**encapsulation**) denilmektedir. Tabi kapsülleme ilkesi sadece burada hazırlayacağımız özel metotlarla sağlanmıyor; başrolde erişim belirleyicileri var.



Buna günlük hayattan örnek vermek gerekirse şunlar söylenebilir: Bir kurumun veya sitenin kapısına giriş çıkışları kontrol edecek görevli koymakla burada sözünü edeceğimiz özel metotlarla yapılan işlemler çok benzerdir.

Nesneye yönelik programcılığın kapsülleme ilkesinden söz etmek için yukarıda hazırladığımız "Cep_Fatura" adlı sınıfın kısıtlı bir versiyonunu hazırladık. Private alanlar hakkında konuşmak amacıyla (şimdilik C#'ta özelliklerinin olmadığı varsayımıyla hareket ettiğimizi unutmayalım) hazırlayacağımız uygulamada yine "Program" ve "Cep_Fatura" adında 2 sınıf olacaktır. Aşağıda verilen "Cep_Fatura" sınıfının yeni haline yakından bakıldığında sınıfta "Sahip" ve "Numara" adında ve 2 public alanın olduğu görülebilir.

```
public class Cep_Fatura
{
 public String Sahip;
 public String Numara;
 public void Yaz()
 {
  Console.WriteLine(this.Sahip);
  Console.WriteLine(this.Numara);
 }
}
```

"Cep_Fatura" adını verdiğimiz sınıfın "Sahip" ve "Numara" alanları **public** oldukları için başka sınıflardan bu alanlara bilgi aktarılabilinir. Bu alanlara aktarılmak üzere kullanıcıdan istenen bilgiyi diğer sınıfın Main() metodu (veya başka bir metodun) içinde kontrol edip uygun olup olmadığı araştırabilinir. Ancak bu işlemi başka bir sınıfın içinde yapmak pratik değildir. Yani buradaki sınıf adlarıyla ifade etmek gerekirse, "Cep_Fatura" sınıfının "Sahip" ve "Numara" alanlarına aktarılacak bilgileri "Program" sınıfı içinde kontrol etmek pratik değildir. Bu nedenle başka bir sınıftan doğrudan erişilmesine izin verilmek istenmeyen alanlar **private** yapılır.

Yukarıda verilen uygulamada "Cep_Fatura" sınıfının "Sahip" ve "Numara" alanlarının önündeki **public** silinip yerine **private** yazsaydık hata meydana gelirdi. Çünkü başka sınıflara kapatılan bu alanların içeriğini 'Program' sınıfının Main() metodunda okunmaya çalışmıştık.

Buradaki amacımız bu alanlara başka sınıflardan erişimi tümden kapatmak değildir. Amacımız, bu alanlara aktarılacak bilgiyi kontrol edip öyle aktarmaktır. Örneğin telefon numarasının belli sayıda rakama sahip olup olmadığını kontrol edebilmektir. Alana aktarılmak istenen telefon numarası eksikse yapılmak istenen işlemi engellemektir.

Sınıfın hangi alanı kontrol edilmek isteniyorsa o alan için 2 özel metot hazırlanmalıdır. Şimdi "Cep_Fatura" adını vermiş olduğumuz sınıfın "Numara" alanı için, bu alana bilgi aktarırken ve mevcut içerik okunurken kullanmak üzere 2 metot hazırlayacağız. Bu amaçla ilk olarak alanın önündeki **public** anahtar kelimesini silip yerine **private** yazdık.

```
public class Cep_Fatura
{
 public String Sahip;
 private String Numara;
 public void Yaz()
 {
  Console.WriteLine(this.Sahip);
  Console.WriteLine(this.Numara);
 }
}
```

Bu şartlarda "Cep_Fatura" sınıfının "Numara" alanına başka sınıflardan erişip içeriğini okumak veya mevcut içeriği değiştirmek mümkün değildir. Çünkü bu alan artık sınıfa özel bir alandır. Private alanlara başka sınıflardan erişebilmek için söz konusu alan için özel işlevi olan metotlar hazırlamak gerekmektedir. Bu amaçla aşağıdaki gibi 2 metot hazırladık. Bu metotlara **setter** ve **getter** adı verilmektedir.

```
public class Cep_Fatura
{
  public String Sahip;
  private String Numara;
  public void setNumara(String n)
  {
   this.Numara = n;
  }
  public String getNumara()
  {
   return this.Numara;
  }
  public void Yaz()
  {
   Console.WriteLine(this.Sahip);
   Console.WriteLine(this.getNumara());
  }
}
```

Adında "set" eki olan metotta sınıfın private alanın içeriği değiştirilmekte, adında "get" eki olan metotta ise alanın mevcut içeriği okunmaktadır. Sınıfın alanlarını okumayı bu sınıfın Yaz() metodunda yaptığımız için Yaz() metodunda değişiklik yapmamız gerekti.

Buradaki amacımız bu metotların nasıl hazırlanıp kullanıldığını anlatmak olduğu için metot dahilinde herhangi bir işlem yapmadık. C# programlama dilinde özellikler olmasaydı veya bu kod Java ile yazılıyor olsaydı, getter ve setter metotlarda herhangi bir işlem yapmıyor olsanız bile hazırlayın diye öneride bulunurduk. Çünkü ileri bir zamanda getter veya setter metotlara bir şeyler yazma gereğini duyduğunuzda elinizde hazır yapı olurdu.

Sınıfımızın son halinde özellikle Yaz() metodunu incelemenizi istiyoruz. Yaz() metodunda "Sahip" alanının içeriği direk okunup konsola yazılırken "Numara" alanı, getNumara() adını verdiğimiz metot aracılığıyla okunup konsola yazılmaktadır. Burada kullandığımız "getNumara" bir metot olduğu için adına parantezleri ekledik.

Bu şekilde sınıfın private alanına erişimde kullanılacak 2 public metot hazırladıktan sonra bu sınıfı kullandığımız "Program" adlı sınıfta Main() metodunu aşağıdaki gibi düzenledik. "Numara" alanına bilgi aktarma şeklinin "Sahip" alanına olan farkını görmenizi istiyoruz.

```
class Program
{
 public static void Main()
 {
   Cep_Fatura f1 = new Cep_Fatura();
   Console.Write("Telefonun sahibi:");
   f1.Sahip = Console.ReadLine();
```

```
            Console.Write("Telefonun no:");
            f1.setNumara(Console.ReadLine());
            f1.Yaz();
            Console.ReadKey();
        }
    }
```

Şimdi yukarıda yaptığımız işlemleri özetleyeceğiz:

- Burada "Cep_Fatura" adını verdiğimiz sınıftaki private alana dışarıdan erişebilmek için adında set ve get eki olan 2 public metot hazırladık.
- Programcı bu sınıfın private alanın mevcut içeriğini okumak veya değiştirmek istediğinde bu metotlardan yararlanacaktır.
- Bu alana dışarıdan bilgi aktarılmak istendiğinde metotlarda istenilen kontrol işlemleri yapılıp bilgi tutarsızlıklarının önüne geçilebilinecektir.
- Sınıfımızda 2 private alan olmasına rağmen kod uzamasın diye sadece bir alan için setter ve getter metotları hazırladık. Bu metotlarda herhangi bir kontrol işlemi yapmadık.
- "Numara" alanına bilgi aktarmak amacıyla hazırladığımız setter metot alanla aynı tipte bir parametreye sahip void bir metot iken, getter metodumuz geriye string bilgi göndermektedir.

Yukarıda anlatılan şekilde getter ve setter metotlarını yazıp sınıfın alanlarını private yapma imkanı elde etmekle gerçekte nesneye yönelik programlamanın üç temel ilkesinden biri olan sarmalamaya (encapsulation) bağlı kalmış oluyoruz. Yukarıda ifade edildiği üzere sınıfların private alanlarına erişimde kendiniz getter ve setter metotlar hazırlayıp kullanabileceğiniz gibi bir sonraki başlık altında hakkında bilgi verilen özelliklerden yararlanabilirsiniz.

C# dilinde özelliklere (property) yer verilmesinin nedeni programcının daha az kod yazmasını sağlamaktır. Çünkü sınıftaki çok sayıda alan (field) için getter ve setter metotlarının yazılması koddaki satırların sayısını ister istemez arttıracaktır.

Özellikler – Properties

Yukarıda private alanlara erişim için 2 public metot hazırladık. C# programlama dilinde private alanlara erişimde genellikle özellikler kullanılmaktadır. Java tecrübesi olup Java'daki alışkanlıklarını devam ettirmek istiyorum diyenler özelliklere kayıtsız kalabilirler.



Yukarıdaki sayfalarda anlatılanlara göre sınıftaki alanların hepsi public ise özellik tanımlamaya gerek kalmaz. Böyle bir tercih, nesneye yönelik programcılık ilkelerini uymamak olur. Hazırladığınız class deneysel değilse bütün alanlarını private yapmanızı ve her alan için özellik hazırlamanızı öneririz.
Konu private alanlar olduğunda şöyle bir pencereden bakılabilir: Nesneye yönelik programcılığın ilkeleri boşuna konulmuş değildir; her birisi bir sorunun önüne geçmek veya bir sorunu çözmek amacıyla uzun araştırmalardan sonra ilke olarak benimsenmiştir. Dolayısıyla bu ilkelere bağlı olmakta fayda vardır.

Sınıfımızdaki bir alanın içeriği kontrol edilmeden direk değiştirildiğinde sorunlar yaşanabilir. Bunun önüne geçmenin yolu, kullanıcının seçimini değerlendirip ona göre işlem yapmaktır. Örneğin veritabanında kayıtlı bir stokun miktarının direk değiştirilmesine izin verilmez. Stok miktarı giriş ve çıkış işlemleri sonucu değişsin istenir. Başka bir anlatımla, sınıfın bazı üyelerine başka sınıflardan doğrudan erişime izin verilmesi sorunlara neden olabilmektedir.

Daha önceki konulardan bildiğiniz gibi Toolbox'tan yararlanılarak forma yerleştirilen nesnelerin birçok özelliğini **Properties** penceresinde değiştirmek mümkündür. Ancak bazı özellikleri kod yazarak çalışma anında değiştirmek mümkün değildir. Bazı özelliklere ise aktarılabilinen değerlerin belli sınırlar içinde olması istenir. Örneğin **ProgressBar** sınıfının **Value** özelliğine **Maximum** özelliğinin içeriğinden daha büyük ve **Minimum** özelliğinin içeriğinden daha küçük değer aktarılamaz.

Benzer durum kendi hazırladığınız sınıflarda da karşınıza çıkar ve sınıftaki her alan için bir özellikle hazırlamanız beklenir. Şimdi yukarıda verdiğimiz "Cep_Fatura" sınıfını bu kez özellikleri anlatmak için kullanacağız.

"Cep_Fatura" sınıfımızda yine "Sahip" ve "Numara" adında 2 alan olacaktır. C# konulu kaynaklarda sınıfın alanlarına bazen yanlışlıkla özellik denilmektedir. Çünkü özellikler genellikle alanlara bağlı olarak işlev görmektedir. Sınıfımızın son hali aşağıdaki gibidir.

```
public class Program
{
 static void Main(string[] args)
 {
  Cep_Fatura f1 = new Cep_Fatura();
 }
}
public class Cep_Fatura
{
 public string Sahip;
 public string Numara;
 public void Yaz()
 {
  Console.WriteLine(this.Sahip);
  Console.WriteLine(this.Numara);
 }
}
```

Tekrar etmek gerekirse, class bloğunda tanımlanan değişkenlere alan, metotların içinde tanımlananlara ise yerel değişken denilmektedir.

Yukarıda ifade edildiği gibi sınıfımız bu hali ile nesneye yönelik programlama ilkelerinden sarmalamayı karşılamıyor. Çünkü sınıfımızda 2 public alan bulunmaktadır. Yani sınıfımızın 2 alanı dışarıya açık duruyor. Bu alanların özellikler ile sarmalanıp dışarıya kapatılması gerek.

Sınıfın hangi public alanı dışarıya kapatılıp sarmalanmak isteniyorsa o alan **private** yapılıp o alan için özellik hazırlanmalıdır. Yukarıda da ifade edildiği dışarıya kapatılan alanlara erişimde özellikleri kullanmak tek yöntem değildir. C# programlama dilinin programcılara sunduğu bir imkân olan özellikler yerine kendiniz setter ve getter metotlar hazırlayabilirsiniz.

Özelliklerin **get** ve **set** blokları olmaktadır. Şimdi "Cep_Fatura" adını verdiğimiz sınıfın "Numara" alanı için bir özellik tanımlayacağız. Bu amaçla alanın önündeki public deyimini silip yerine **private** yazdık.

```
public class Cep_Fatura
{
 public string Sahip;
 private string Numara;
 public void Yaz()
 {
  Console.WriteLine(this.Sahip);
  Console.WriteLine(this.Numara);
 }
}
```

Bu şartlarda "Cep_Fatura" sınıfının "Numara" alanına başka sınıflardan erişip içeriğini okumak veya değiştirmek mümkün değildir. Çünkü bu alan artık sınıfa özel bir alandır. Şimdi bu alana erişimde kullanmak üzere bir özellik hazırlayacağız. Özelliğe istediğiniz adı verebilirsiniz. Ancak özelliğe ilgili alanı çağrıştıran bir ad vermeniz önerilir. Bu özelliğe "numara_property" adını verdik. Property tanımlarken Visual Studio'dan yardım almak istiyorsanız "prop" yazıp 2 kez tab tuşuna basabilirsiniz.

```
public class Cep_Fatura
{
 public string Sahip;
 private string Numara;
 public string numara_property
 {
  get
   {
   }
  set
   {
   }
 }
}
```

Sınıfın private alanına hizmet edecek özelliğin **public** olması gerekir. Yoksa özelliği kullanmak mümkün olmaz. Ayrıca tanımladığınız özellik hangi tipteki alana hizmet edecekse alanın tipini belirten tip özellik adından önce yazılmalıdır. İlgili alanın ve özelliğin farklı tipte olmasına C# derleyicisi izin vermemektedir.

set bloğunda ilgili alana değer aktarılmakta ve **get** bloğunda ise alanın mevcut içeriği okunmaktadır. Sınıfımızda "numara_property" adını verdiğimiz özelliğin set ve get bloklarını nasıl ayarladığımızı aşağıda görebilirsiniz. Get bloğunda, özellik aracılığıyla alanın içeriği okunurken **return**, set bloğunda alanın içeriği özellik sayesinde değiştirilirken **value** anahtar kelimesinden yararlanılmaktadır.

```
public class Cep_Fatura
{
 public string Sahip;
 private string Numara;
 public string numara_property
 {
  get
   {
    return Numara;
   }
```

Alan için özellik tanımlandıktan sonra nesne hazırlanıp nesnenin bu elemanına bilgi aktarılmak veya elemanın içeriği okunmak istendiğinde alan yerine alanı temsil eden özellik kullanılmalıdır. Özellikler **public** oldukları için her sınıfta kullanılabilir.

```
  set
   {
    Numara = value;
   }
 }
 public void Yaz()
 {
  Console.WriteLine(this.Sahip);
  Console.WriteLine(this.numara_property);
 }
}
```

Özelliğin set ve get bloklarını ayarladıktan sonra "Yaz()" metodunda değişiklik yaptık. Çünkü Yaz() metodunun ilk halinde "Numara" alanına direk erişiliyordu. Şimdi ise property aracığıyla erişiliyor. Yaz() metodunu incelerseniz, kullanım şekli bakımından özelliklerin alanlardan farklı olmadığı görülür.



Özellikler aslında programcı ile derleyici arasında yapılmış bir anlaşmadır. Derleyici diyor ki, böyle get-get blokları olan, return ve value anahtar kelimelerinin kullanıldığı bir yapı hazırlarsan, senin get ve set bloklarında adı geçen alan için getter ve setter metotları hazırlamak istediğini anlarım, senin için derleme sırasında geri planda 2 metot hazırlarım. Böylece yazman gereken kod kısalmış olur. C#'ın ilk 2 sürümünde bu kod kısaltma büyütülecek kadar değildi. Derken C# 3.0 ile birlikte programcıların getter ve setter metotlar için yazmaları gereken satırlarda azalma oldu.

Anlatılan şekilde sınıfımızdaki private alan için özellik hazırladıktan sonra şimdi bu sınıfın örneğini hazırlayıp bu özellikten yararlanmayı deneyeceğiz. Bu amaçla üzerinde çalıştığımız uygulamadaki diğer sınıfta Main() metodunu aşağıdaki gibi düzenledik.

```
public class Program
{
 static void Main(string[] args)
 {
   Cep_Fatura f1 = new Cep_Fatura();
   Console.Write("Telefonun sahibi:");
   f1.Sahip = Console.ReadLine();
   Console.Write("Telefon No:");
   f1.numara_property = Console.ReadLine();
   f1.Yaz();
  }
}
```

Tam bu noktada "numara_property" adını verdiğimiz alanın set ve get blokları üzerinde biraz konuşmak gerekecek. Yukarıdaki sayfalarda işaret edildiği gibi set ve set blokları gerçekte 2 özel metot gibi işlev görmektedir. Get bloğundaki **return** anahtar kelimesi sayesinde ilgili alanın (burada Numara) mevcut içeriği okunup geriye gönderilmektedir. Set bloğunda ise **value** anahtar kelimesi aracılığıyla istenilen bilgi ilgili alana aktarılmaktadır. Bu söylediğimizi aşağıda verilen satırda net olarak görebilirsiniz.

```
f1.numara_property = Console.ReadLine();
```

Burada konsoldan okunup "numara_property" adını verdiğimiz özelliğe aktardığımız bilgi geri planda set bloğundaki "value" anahtar kelimesi sayesinde ilgili alana aktarılmaktadır. Get bloğunda yapılan okuma işlemini net olarak gösterebilmek için Yaz() metodunu kullanmak yerine Main() metoduna 2 satır ekledik.

```
class Program
{
 static void Main(string[] args)
 {
  Cep_Fatura f1 = new Cep_Fatura();
  Console.Write("Telefonun sahibi:");
  f1.Sahip = Console.ReadLine();
  Console.Write("Telefon No:");
  f1.numara_property = Console.ReadLine();
  Console.WriteLine(f1.Sahip);
  Console.WriteLine(f1.numara_property);
 }
}
```

Buraya kadar anlatılanların açıklaması şudur: "Numara" alanı **private** olduğu için başka sınıflardan erişemiyorduk. Bu engeli aşmak için özellik tanımladık. Bu hali ile set ve get bloklarını kullanmanın bir yararı oldu mu? Şimdilik hayır. Şimdi kullanıcının gireceği telefon numarasının uzunluğunu kontrol edeceğiz. Bu amaçla **set** bloğunu aşağıdaki gibi düzenledik.

```
public string numara_property
{
 get
  {
   return Numara;
  }
 set
  {
   string str = value;
   int uzunluk = str.Length;
   if (uzunluk < 11)
     Console.WriteLine("Telefon numarası eksik");
   else
     Numara = value;
  }
}
```

set bloğuna yazılan bu satırlar sayesinde nesne hazırlanıp özelliğe aktarılan telefon numarasının rakam sayısı (string bilginin uzunluğu) öğrenilmektedir. Bilginin uzunluğu 11'den küçükse mesaj verilip girilen bilgi nesnenin "Numara" alanına aktarılmamaktadır. Bu yapılırken **value** anahtar kelimesinden yararlanılıyor.

Yani programcı herhangi bir metodun içinde özelliğe bilgi aktarmak istediğinde bu bilgi geri planda value anahtar kelimesi aracılığıyla özelliğe aktarılmaktadır. Söz konusu özelliğe ait set bloğunda value anahtar kelimesinin içeriği ilgili alana aktarılmalıdır. Property'lerin programcılara sağladığı kolaylıkların asıl kaynağı **value** anahtar kelimesidir.

Konunun iyice anlaşılmasını sağlamak için "Cep_Fatura" sınıfındaki "Sahip" adlı 2. alan için de özellik tanımlayacağız. Bu özellik sayesinde kullanıcının girdiği ad bilgisi String sınıfının **ToUpper()** metodu ile büyük harflere çevrilecektir. Özellik aracılığıyla "Sahip" alanından okunacak bilgi üzerinde herhangi bir işlem yapmak istemediğimiz için bu özelliğin get bloğunda ek bir işlem yapmadık.

```
public class Cep_Fatura
{
 private string Sahip;
 private string Numara;
 public string sahip_property
 {
  get
   {
    return Sahip;
   }
  set
  {
   Sahip = value.ToUpper();
  }
 }
 public string numara_property
 {
  get
   {
    return Numara;
   }
  set
   {
    string N = value;
    int uzunluk = N.Length;
    if (uzunluk < 11)
      Console.WriteLine("Telefon Numarası eksik");
    else
     Numara = value;
  }
}
```

Bazen alanların içeriğinin okunmasına izin verilir ama değiştirilmesi istenmez. Hazırladığınız sınıfta, içeriğinin okunmasına izin verip değiştirilmesini engellemek istediğiniz alanlar varsa söz konusu alan için hazırladığınız özelliğin **set** bloğunu yazmamanız gerekir. Bu ayarlamanın nasıl yapıldığını anlatmak için "Cep_Fatura" adını verdiğimiz sınıfa "Kod" adında 3. bir alan tanımladık ve bu alana ilk değer aktardık.

```
public class Cep_Fatura
{
 private string Sahip;
 private string Numara;
 private string Kod = "0-312";
```

Bu alan **private** olduğu için içeriğine başka sınıflardan erişebilmek için bu alan için özellik veya erişimci metot hazırlamak gerekir. Amacınız yalnızca alanın içeriğini okumaksa özellik için set kısmını yazmayıp get kısmını aşağıdaki gibi yazmalısınız.

```
public string Kod_property
{
 get
  {
   return Kod;
  }
}
```

Burada "Kod" alanını tanımlarken ilk değeri aktardık. Bu alana sınıfın içinde değer aktarabilirsiniz. Bu özelliğin **set** kısmı olmadığı için başka sınıflarda iken sınıftan nesne hazırlandığı zaman nesnenin "Kod "adı verilen alanın içeriği değiştirilemez.

Yukarıda özellikler hakkında yazılanları ve yapılan işlemleri özetlemekte fayda vardır:

- Bu bölümde C# ve nesneye yönelik programcılığın temel yapı taşı olan class'lardan söz etmek istediğimiz için class'lar hakkında biraz konuştuktan sonra metotların üzerinde biraz durduk. Biraz diyoruz, çünkü metotlarla ilgili ayrıntıları ayrı bir bölüm olarak ele alacağız.
- Metotlar hakkında biraz konuştuktan sonra sınıfların önemli üyesi alanların (field) üzerinde durduk. Nesneye Yönelik Programcılık ilkeleri gereği alanlar private yapıldığında alanlara erişimde kullanılan setter ve getter metotları anlattık.
- Son olarak private alanlara erişimde kullanılan ve C#'a özel bir yapı olan Property'lerde söz ettik.

Automatic Properties

Yukarıda anlatılanlara göre programcılar dışarıdan görülmesinde ve içeriğinin direk değiştirilmesinde sakınca görmedikleri alanları **public** yapabiliyorlar. Ancak söz konusu alanın içeriği başka sınıflardan erişilerek değiştirilirken programcı bu değiştirme işlemini kontrol altına almak istiyorsa alanı **private** yapıp o alan için **özellik** hazırlamaktadır. C# 3.0 öncesinde özellikler herhangi bir alanla ilişkilendirilmediği zaman hata meydana geliyordu.

3.0 versiyonundan sonra derleyici her özelliğin bir alanla ilişkilendirilmesini şart koşmuyor. Çünkü derleyici, bir alanla ilişkilendirilmeyen özellikler için geri planda otomatik olarak alan tanımlıyor. Bu söylenenleri örneklemek için yukarıda verilen sınıfın bir kısmını aşağıya aldık.

```
public class Cep_Fatura
{
 private string Numara;
 public string numara_property
 {
  get
   {
    return Numara;
   }
  set
   {
    Numara = value;
   }
 }
}
```

Dikkat ederseniz bu sınıfta "numara_property" adını verdiğimiz özellik "Numara" adlı private alanın içeriğinin dışarıdan okunmasına ve gerekirse değiştirilmesine aracılık etmekten öte bir işleve sahip değildir. Yani bu özellik gelene buyur, gitmek isteyen geçebilirsin diyor. Bu özelliğin **set** bloğunda "Numara" adını verdiğimiz alana aktarılacak bilginin geçerliliğini kontrol ediyor olsaydık bu özellik o zaman işlevsel olurdu.

Bu hali ile özelliği silip "Numara" adını verdiğimiz alanı public yaparsak değişen bir şey olmaz. C# 3.0'dan itibaren property'de herhangi bir kontrol yapılmıyorsa veya private alana erişimde aracılık yapmaktan öte işlevi olmayan özellikler söz konusu olduğunda alan hazırlamaya gerek kalmadan direk özellik hazırlamak mümkündür. Bu durumda C# derleyicisi ilgili alanı geri planda otomatik olarak hazırlamaktadır. Bu nedenle yukarıda verilen sınıftaki alan ve özellik için aşağıdaki gibi kod yazıldığında aynı sonuç alınır.

```
public class Cep_Fatura
{
 public string numara_property
  {
   get;set;
  }
}
```

Bu şekilde property hazırlayan programcı ileri bir zamanda sınıfında bir alana yer verip ve bu alanı bu özellikle ilişkilendirmek istediğinde zaman kaybetmeyecektir. Çünkü özelliği zaten hazırdır, yapması gereken, özelliğin set-get bloklarına istediklerini yazmasıdır.

Bu şekilde özellikler hazırlarken Visual Studio'dan yardım alabilirsiniz. "prop" yazılıp 2 kez Tab tuşuna basıldığında aşağıdaki gibi bir özellik hazırlanmaktadır. Programcıya düşen, Visual Studio tarafından özelliğe verilen adı değiştirmektir.

```
public int MyProperty { get; set; }
```

C# 6.0 ile birlikte Property'lerle ilgili **Auto-Property Initializers** adı verilen bir yenilik geldi. Bu yenilik sayesinde özelliklere ilk değer verilebilinmektedir. Dolayısıyla ilgili sınıftan yola çıkıp nesne hazırlandığında ilk değer verilen özelliğin bir içeriği olmaktadır.

Bu yeniliği bahane edip yukarıda sözü edilen ve C# 3.0 ile gelen Automatic Properties'lerden söz etmek amacıyla basit bir örnek hazırlayacağız. Çünkü Automatic Properties ve 6.0 ile gelen Auto-Property Initializers yenilikleri konunun karıştırılmasına neden olmaktadır. Konu üzerinde konuşurken adım adım ilerlemeyi tercih ettiğimiz için sınıfımızın sadece bir alan ve özelliğe sahip olmasını sağladık.

```
public class Takimlar
{
 private string takim;
 public string takimOzellik
 {
  get
   {
    return takim;
   }
```

```
    set
     {
      takim = value;
     }
    }
    public Takimlar(string par1)
    {
     this.takimOzellik = par1;
    }
}
```

Bu koda dikkat ederseniz özelliğin set ve get bloklarında herhangi bir işlem yapılmadığını görürsünüz. Bu sınıfı C# 1.0 veya 2.0 günlerinde yazsaydık özelliği böyle düzenlemek bir zorunluluktu. Bu kodu hazırlarken C# 3.0'dan itibaren gelen Automatic Properties yeniliğinden yararlanmak istemiş olsaydık aşağıdaki gibi düzenlerdik.

```
public class Takimlar
{
 public string takimOzellik
  {
    get; set;
  }
 public Takimlar(string par1)
 {
   this.takimOzellik = par1;
  }
}
```

Bu durumda burada "takimOzellik" adını verdiğimiz property'nin bir alanla ilişkilendirilmesi işlemini derleyici geri planda derleme sırasında otomatik olarak yapar. Bu şekilde özellik tanımlayıp derleyicinin geri olanda field hazırlamasının bize bir katkısı var mı? Bu haliyle Automatic Properties yeniliğinin programcıya bir katkısı yoktur. Bu durumda public özellik yerine direk field hazırlanabilir. Ancak ileri bir zamanda programcı özelliğe, dolayısıyla geri planda tanımlanan alana aktarılmak istenen bilgi üzerinde hakimiyet kurmak istediğinde kendisi kolayca field tanımlayıp, Automatic Property'nin set ve get bloklarında ayarlama yapabilir. Burada önemli olan, bu özelliğin kullanıldığı yerlerde değişiklik yapılmaya gerek olmamasıdır. Programcı Automatic Property yerine direk alan tanımlasaydı ve sonradan bu alan için özellik hazırlamaya gerek duysaydı, bu sınıfın kullanıldığı yerlerde de değişiklik yapması gerekecekti ki bu hiç istenmez. Bize sorarsanız, yazdığınız uygulama deneysel değilse field yerine her seferinde automatic property hazırlayın deriz.

Konuyu C# 6.0 ile gelen Auto-Property Initializers yeniliğine getirmeden önce yukarıda hazırladığımız sınıfta 2. bir özelliğe yer verip sonra örneğini alıp kullandık.

```
public class Takimlar
{
 public string takimOzellik
  {
   get; set;
  }
 public int kurulusOzellik
  {
   get; set;
  }
```

```
 public Takimlar(string par1, int par2)
 {
  this.takimOzellik = par1;
  this.kurulusOzellik = par2;
 }
}
class Program
{
 static void Main(string[] args)
 {
  Takimlar nesne = new Takimlar("Fenerbahçe", 1907);
  Console.WriteLine(nesne.takimOzellik);
  Console.WriteLine(nesne.kurulusOzellik);
 }
}
```

Programcının "Takimlar" adını verdiği sınıftaki "takimOzellik" adlı property için sonradan field hazırlamak istediğinizde, başka bir deyişle automatic properties imkanından yararlanmak istemediğinde sınıfın kullanma şekliyle ilgilenmeden aşağıda yapıldığı gibi field hazırlayıp elindeki mevcut property ile ilişkilendirebilir.

```
public class Takimlar
{
  private string takim;
  public string takimOzellik
  {
   get
    {
      return takim;
    }
    set
    {
      takim = value.ToUpper();
    }
   }
   public int kurulusOzellik { get; set; }
   public Takimlar(string par1, int par2)
   {
    this.takimOzellik = par1;
    this.kurulusOzellik = par2;
   }
 }
```

Sanırız önce automatic properties imkanından yararlanan programcı sonradan söz konusu özellik için field tanımlamak istediğinde neler yapması gerektiği ve C# 3.0 ile gelen automatic properties imkanın ana fikri anlaşılmış oldu. Şimdi C# 6.0 ile gelen ve automatic property'lere ilk değer verilirken yararlanılan yeniliğe örnek olması için yukarıda hazırladığımız sınıfta 3. özelliğe yer vereceğiz. Sınıfta yaptığımız değişiklikleri aşağıda görebilirsiniz.

```
public class Takimlar
{
 private string takim;
 public string takimOzellik
 {
```

```
    get
     {
      return takim;
     }
    set
    {
     takim = value.ToUpper();
    }
   }
   public int kurulusOzellik { get; set; }
   public string direktorOzellik { get; set; } = "Fatih Terim";
   public Takimlar(string par1, int par2)
    {
     this.takimOzellik = par1;
     this.kurulusOzellik = par2;
    }
}
```

Bu sınıfla ilgili olarak ilk dikkatle bakmanızı istediğimiz "direktorOzellik" adını verdiğimiz automatic property'nin tanımlandığı satırdır. Bu satıra "=" operatörünü ve sonra ilk değer olarak özelliğe vermek istediğimiz metni yazdığımızı fark etmiş olmalısınız.

Dikkat etmenizi istediğimiz 2. nokta, sınıfın yapıcı metodunda bu özellikle ilgili herhangi bir değişiklik yapmamış olmamızdır. Çünkü Auto-Property Initializers yeniliğinin işlevini göstermek istiyoruz. Sınıfımızda bu şekilde Auto-Property Initializers yeniliğinden yararlandıktan sonra aşağıda yapıldığı gibi nesne hazırlayıp özelliklerin içeriklerini konsola yazdık.

```
class Program
{
 static void Main(string[] args)
 {
  Takimlar nesne = new Takimlar("Galatasaray", 1905);
  Console.WriteLine(nesne.takimOzellik);
  Console.WriteLine(nesne.kurulusOzellik);
  Console.WriteLine(nesne.direktorOzellik);
 }
}
```

Sınıfımızın yapıcı metodunun 2 parametreye sahip olduğunu ve yapıcı metoda parametre olarak verilen bilgilerin ilk 2 özelliğe aktarıldığını biliyorsunuz. Yani burada "direktorOzellik" adını verdiğimiz özellikle ilgili herhangi bir işlem yapmamış olmamıza rağmen bu kod işletilirse aşağıdaki gibi bir sonuç alınır.



Bu sonucun alınmasındaki neden, Auto-Property Initializers yeniliği sayesinde bu özelliğe ilk değeri vermiş olmamızdır. Yukarıda yaptığımız gibi özelliğe ilk değer verdiğimizde sanki setter bir metot yazmışız gibi işlev görür. Tabi ilgili sınıftan yola çıkıp nesne hazırladıktan sonra ilk

değer ataması yapılan özelliğin içeriği değiştirilebilir. Daha sonra nesne hazırlanıp özelliğin içeriği okunmak istendiğinde ise get bloğu devreye girer. Bu işlemler için derleyicinin geri planda setter ve getter metotlar hazırladığını tahmin edebilirsiniz.

Bu noktada akla şöyle bir soru gelebilir: Auto-Property Initializers yeniliğiyle ilk değer ataması yaptığımız özellik yalnız okunabilir, yani set bloğu olmayan bir özellik ise ne olur? Bu durumda özelliğin içeriği yapıcı metot aracılığıyla ayarlanmazsa sürekli ilk değer olarak atanan bilgiyi içerir.

Object Inializers – Nesne Hazırlayıcılar

Yukarıda anlatıldığı üzere sınıflarının örnekleri alınırken otomatik olarak işletilen yapıcı metotlar sayesinde hazırlanan nesnelerin özellikleri ayarlanmaktadır. Bu konuyu hatırlatmak için aşağıda verilen sınıfı hazırladık

```
class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
 public Tarih()
 {
  DateTime Bugun = DateTime.Today;
  Gun = Bugun.Day;
  Ay = Bugun.Month;
  Yil = Bugun.Year;
 }
}
```

Bu sınıfın yapıcı metodunda alanlara sistem tarihi gün, ay, yıl olarak aktarılmaktadır. Bu sınıf için yapıcı metot hazırlamasaydık bu 3 alan **int** tipinde olduğu için hepsine 0 aktarılırdı.

C# 3.0 ile gelen yenilik sayesinde sınıfların örnekleri alınırken aynı zamanda istenen özellikler ayarlanabilmektedir. Bu işlemin nasıl yapıldığını anlatmak için yukarıda verilen sınıfın yapıcı metodunu silip örneğini aşağıdaki gibi aldık.

```
namespace WindowsApp1
{
 public partial class Form1 : Form
  {
   public Form1()
   {
    InitializeComponent();
   }
   private void Form1_Click(object sender, EventArgs e)
   {
    Tarih tarih;
    tarih = new Tarih {Gun = 12, Ay = 01, Yil = 2017};
   }
  }
}
```

"Tarih" adını verdiğimiz bu sınıfta yalnızca public alanlara yer verdik. Alanları **private** yapıp aşağıdaki gibi özellikler hazırlasaydık değişen bir şey olmazdı.

```
class Tarih
{
 private int Gun;
 private int Ay;
 private int Yil;
 public int gun_ozellik
 {
  get {return Gun;}
  set { Gun = value;}
 }
 public int ay_ozellik
 {
  get { return Ay;}
  set { Ay = value;}
 }
 public int yil_ozellik
 {
  get { return Yil;}
  set {Yil = value;}
 }
}

namespace WindowsApp1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
  private void Form1_Click(object sender, EventArgs e)
  {
   Tarih_class tarih;
   tarih = new Tarih {gun_ozellik=12, ay_ozellik=01, yil_ozellik=2017};
  }
}
```

Bu sınıfın örneğini alırken veya nesne hazırlarken sınıftaki her 3 özelliği de ayarladık ki sizler buna mecbur değilsiniz. Çünkü C# derleyicisi başlangıç içeriği belirtilmeyen özelliğe varsayılan değeri aktarmaktadır.

C# derleyicisi 3.0 sürümünden itibaren herhangi bir class'ın örneği alınırken veya nesne hazırlanırken süslü parantezlerin içine yazılanların özellik olduğunu varsayıp bu özellikleri sınıfta aramaktadır. Süslü parantezin içine yazılan özellik class'ta varsa başlangıç içeriğini ayarlamaktadır.

Süslü parantezlerin içine örneği alınan class'ta olmayan bir özelliğin adı yazılırsa veya özelliğe farklı tipte bilgi aktarılmak istenirse hata meydana gelir. Bu nedenle yukarıda verilen "Tarih" adlı sınıfın örneği aşağıdaki gibi alınırsa hata meydana gelir.

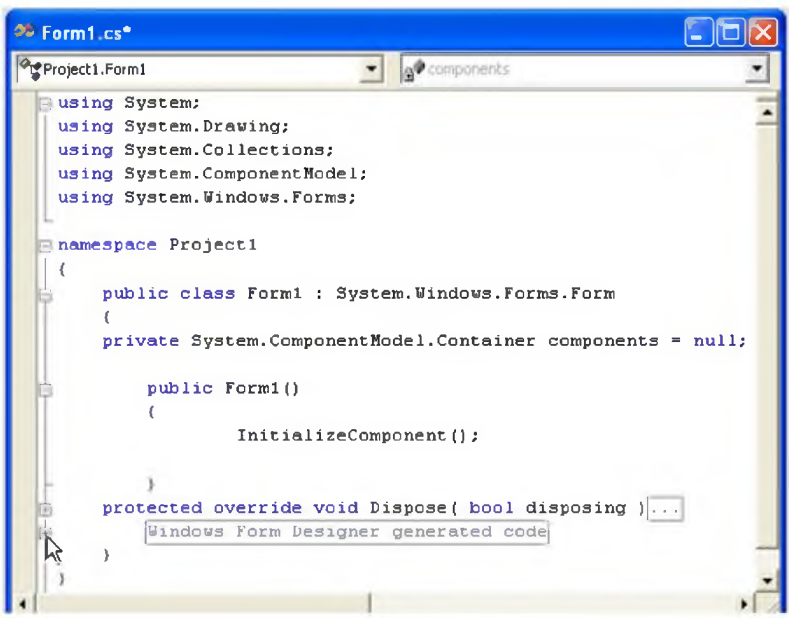
```
private void Form1_Click(object sender, EventArgs e)
{
 Tarih tarih = new Tarih
  {
    gun_ozellik="12", ay_ozellik=01, yil_ozellik=2017
  };
}
```

Partial (Parçalı) Types

C# derleyicisi sınıfların, arayüzlerin ve struct'ların birden fazla parça olarak düzenlenmesine izin vermektedir. **Partial Types** adı verilen bu imkan C# 2.0 ile programcılara sunuldu. Partial Types desteği bir sınıfı birden fazla parça olarak bölmenize, yapıcı metotları, değişkenleri ve özellikleri ayrı dosyalarda kodlamanızın önünü açmaktadır.

Visual Studio ile hazırlanan uygulamalarda formları oluşturan kodlar Partial Class imkanından yararlanılıp 2 dosya arasında paylaştırılmaktadır. Örneğin "Form1" adında bir form hazırlandığında programcının yazdığı kodlar "Form1.cs" dosyasında, formun görsel yapısıyla ilgili olarak Visual Studio tarafından geri planda hazırlanan satırlar ise "Form1.Designer.cs" dosyasında tutulmaktadır.

C# 2.0 ile gelen Partial Types desteğinin programcılara en önemli faydası, programcının yazdıkları kodlar ile Visual Studio'nun geri planda yazdığı kodların bir birinden ayrılmış olmasıdır. Partial Types'lar hakkında konuşmak amacıyla sizi C# 1.0 ve .NET uyumlu ilk Visual Studio günlerine yani 2003 yılına götüreceğiz. Aşağıda ekran görüntüsü olarak verilen kodlar Form1 adlı form için Visual Studio tarafından C# 1.0 uyumlu olarak düzenlenmiştir.

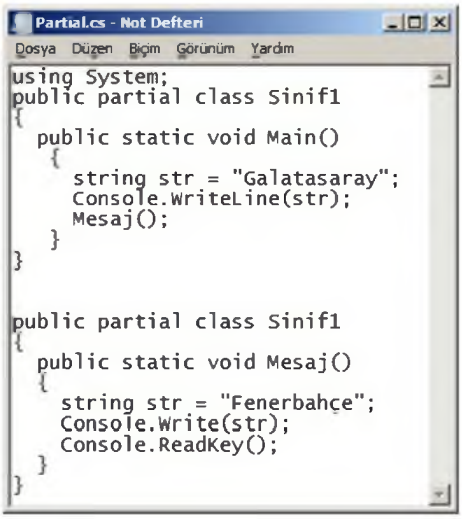


Bu sırada "Windows Form Designer generated code" bölgesine yerleştirilen kodlar gizlenmiş olsa bile ortam oldukça karmaşıktır. Hele hele formun üzerinde 5-10 tane nesne yerleştirdiyseniz Visual Studio'nun geri planda yazdığı satırların sayısı az olmayacaktır.

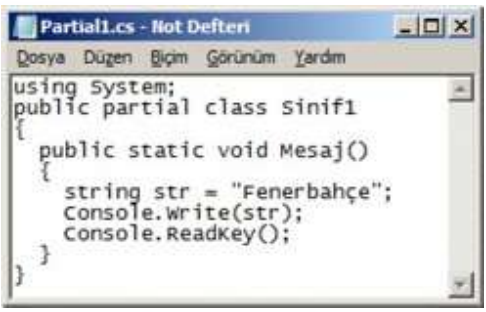


Java programlama dilinin partial types desteği olmadığı için Java programcıları tarafından sıklıkla kullanılan Eclipse adlı IDE, kendi hazırladığı kodlara programcının müdahale etmesine izin vermekte ve bu izin sıklıkla sorunlara neden olmaktadır.

Partial sınıfların nasıl hazırlandığını göstermek için aşağıda ekran görüntüsü olarak verilen kod dosyası içinde aynı ada sahip 2 sınıf hazırladık. Bu şartlarda bu uygulama derlenirse hata mesajı alınmaz ve derleme sırasında bu 2 sınıf tek sınıf olarak değerlendirilir.



Benzer şekilde aynı kod dosyası içinde aynı ada sahip 3. bir sınıfı hazırlamak mümkündür. Bu uygulamadaki sınıfları bir namespace'e yerleştirmediğimizi fark etmiş olmalısınız. Şimdi yukarıda hazırlanan uygulamadaki 2. sınıfı başka bir CS dosyasına taşıdık.

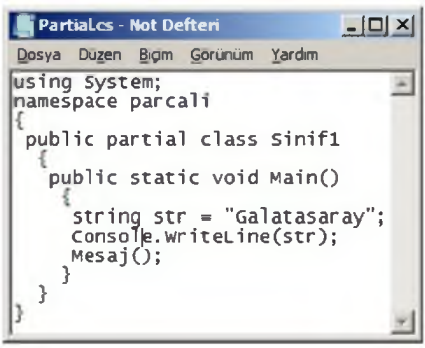


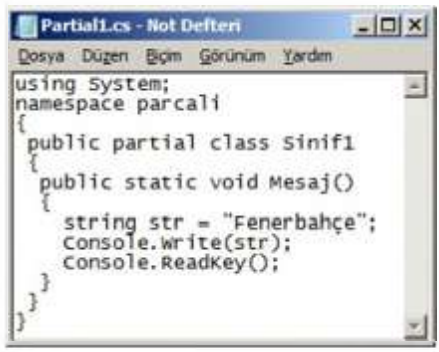
Derleme sırasında 2 sınıf birleştirildiği için 2. sınıftaki "Mesaj()" adlı metodu herhangi bir hazırlık yapmadan ilk sınıfta kullandık. Bu şekilde 2 CS dosyasından meydana gelen konsol uygulaması derlenirken komut satırı aşağıdaki gibi düzenlenmelidir.



Şimdi yukarıda yaptığımız işlemleri sıralayalım:

- Önce aynı kod dosyası içinde aynı ada sahip 2 class hazırladık.
- İlk class'taki Main() metodu içinde 2. class'taki statik metot işletilmektedir.
- Aynı ada sahip bu 2 class aynı kod dosyası içinde iken ikisinin de yararlandığı **Console** sınıfının yer aldığı "System" adlı namespace'i using deyimi ile bir kez işaret etmek yeterli olurken class'lar ayrı kod dosyalarına alındığında her 2. kod dosyasında da using anahtar kelimesine yer vermek gerekti.
- Yukarıda işaret edildiği üzere ayrı kod dosyalarına olan bu sınıfları bir namespace'e yerleştirmediğimiz için derleyici geri planda her iki sınıfın aynı namespace'e üye olduğunu varsayar. Bu namespace konusu önemlidir. Çünkü partial olarak işaret edilen birden fazla kod dosyasına bölünen class veya struct'ların aynı namespace'e üye olmaları gerekiyor. Bu nedenle her 2 kod dosyasındaki sınıfları aşağıdaki gibi aynı namespace'e üye yaptık.





Nested Class'lar

Kısaca tanımlamak gerekirse, sınıf içinde hazırlanan sınıflara **Nested Class** (dahili veya iç sınıf) adı verilmektedir. Bazı kaynaklarda bu sınıflardan gömülü sınıf diye söz edilmektedir. Nested Class'ı içeren sınıfa ise genellikle "kapsayıcı sınıf" adı verilmektedir. Bu başlık altında yazılanlar C# kodu geliştirirken işinize yaramayabilir, ileri bir tarihte birlikte uygulama geliştireceğiniz ekip nested class hazırlamaya sıcak bakmıyor olabilir. Diğer yandan C# derleyicisinin isimsiz tip desteği nested class ihtiyacını azaltmış olabilir. Buna rağmen henüz yolun başında bulunan bir yazılımcı olarak kendini donatma adına burada 3-5 sayfa boyunca hakkında konuşulan nested class'larla ilgilenmenizi öneriyoruz. Tabi hemen ilerlemek istiyorsanız nested class'lara şimdilik kayıtsız kalabilirsiniz. Çünkü nested class'lar öncelikle öğrenilmesi gereken bir konu değildir.

Daha önceki konularda işaret edildiği üzere C#'ın çoklu kalıtım desteği yoktur. Bu eksiklik nested class'lar, abstract class'lar ve interface'ler aracılığıyla giderilmeye çalışılmaktadır. Tabi nested class kullanmanın yegane amacı çoklu kalıtıma destek değildir. Örneğin benzer amaçlar için hazırlanmış birden fazla sınıfı nested class olarak bir arada tutmayı tercih edebilir veya daha iyi bir kapsülleme için nested class hazırlamak isteyebilirsiniz.

Nested class'lar üzerinde konuşmaya devam etmek üzere üzerinde çalıştığımız konsol uygulamasında yeni bir kod dosyası, bu kod dosyasında ise aşağıdaki gibi kısıtlı özelliklere

sahip bir class hazırladık. Bu kısımda örnekleri class'lar üzerinde vermiş olsak bile, C# derleyicisi bir strcut'ın içinde struct hazırlanmasını desteklemektedir.

```
namespace ConsoleApp1
{
 public class Adana
  {
  }
}
```

Sizin için herhangi bir yenilik içermeyen bu class'ın sınırları içinde şimdi ikinci bir class hazırlayacağız. C# derleyicisi bir sınıfın içinde başka bir sınıfın olmasına izin verdiği için burada yaptığımız düzenleme hataya neden olmaz.

```
namespace ConsoleApp1
{
 public class Adana
 {
  class Seyhan
   {
   }
 }
}
```

Bu kodda nested class için erişim belirleyici kullanmadık. Erişim belirleyici olarak public, protected veya private'yi seçebilirdik. Private yapılan nested class'lara dışarıdan erişim sağlanamayacağını tahmin edebilirsiniz. Nested class, private yapılsa bile kapsayıcı sınıfın private olmayan metotlarına erişebilir. Konuyu netleştirmek için "Seyhan" adını verdiğimiz nested class'ı private yaptık.

```
public class Adana
{
 private class Seyhan
  {
  }
}
```

Bu şartlarda uygulama derlenirse hata meydana gelmez. Bundan şu sonucu çıkarabiliriz, nested class private olsa bile kapsayıcı sınıftan kendisine erişim sağlanabiliyor. İlerideki sayfalarda görüleceği üzere bu şekilde hazırlanan nested class, başka bir sınıfın içinde olmanın dışında normal sınıflardan pek farklı değildir.

Şimdi "Seyhan" adını verdiğimiz bu nested class'ın bir metoda sahip olmasını sağlayacağız. Burada "Seyhan" sınıfında alan ve özelliklere yer verebilirdik. Konunun kolay takibi için sadece metot hazırladık.

```
public class Adana
{
 private class Seyhan
 {
  void Hakkinda()
  {
   System.Console.WriteLine("Adana'nın en büyük ilçesi");
  }
 }
}
```

Nested class'lar kalıtımı destekledikleri için bulundukları yer hariç normal class'lardan pek farklı değillerdir. Nested class için bu şekilde bir metot hazırladıktan sonra şimdi kapsayıcı sınıfın yapıcı metoda sahip olmasını sağlayacağız.

```
public class Adana
{
 public Adana()
  {
  }
 class Seyhan
 {
  private void Hakkinda()
   {
     System.Console.WriteLine("Adana'nın en büyük ilçesi");
   }
  }
}
```

Bu şartlarda kapsayıcı sınıfımızın (Adana) yapıcı metodu içinde nested sınıfımız tipinde nesne hazırlayabiliriz. Bundaki amacımız kapsayıcı sınıf dahilinde nested sınıfa erişimin nasıl olduğunu ortaya koymaktır. Aşağıda verilen kodda kapsayıcı sınıf dahilinde private olan nested sınıfın örneğini almış olmakla birlikte **Hakkinda** adını verdiğimiz metodu işletemeyiz.

```
public class Adana
{
 public Adana()
  {
   Seyhan nesne = new Seyhan();
  }
 private class Seyhan
 {
  void Hakkinda()
   {
    System.Console.WriteLine("Adana'nın en büyük ilçesi");
   }
 }
}
```

Daha önceki konulardan bildiğiniz gibi class'lar için erişim belirleyici anahtar kelimelerden herhangi birisi kullanılmadığı zaman o class'ın internal olduğu varsayılıp üzerinde çalışılan uygulamadaki herhangi bir sınıftan erişilebilmektedir. Burada nested class'ımızı private yapmamıza rağmen kapsayıcı class'ta örneğini hazırlayabildik. Bu şartlarda uygulamamızda Main() metoduna sahip diğer sınıfa geçip kapsayıcı sınıf tipinde nesne hazırlayabiliriz.

```
class Program
{
 static void Main(string[] args)
  {
   Adana a1 = new Adana();
  }
}
```

Kapsayıcı sınıfımızın yapıcı metodunda nested class'ımız tipinde nesne hazırlamamıza rağmen Main() metodu içinde nested class'tan yararlanmak bu şartlarda mümkün değildir.

Başka bir deyişle, kapsayıcı sınıftan başka bir sınıfta, mevcut hali ile nested class'ımızın örneğini alamayız. Çünkü private'dir.

Şimdi tekrar kapsayıcı sınıfımıza geçip nested class'ın metodunu işletmeyi deneyeceğiz. Aşağıda verilen kodu incelerseniz nested class'ımızdaki metot için erişim belirleyici anahtar kelimelerden herhangi birisini kullanmadık. Dolayısıyla bu metot private'dir.

```
public class Adana
{
 public Adana()
  {
   Seyhan nesne = new Seyhan();
   nesne.Hakkinda() // hata meydana gelir.
  }
 private class Seyhan
 {
  void Hakkinda()
  {
    System.Console.WriteLine("Adana'nın en büyük ilçesi");
  }
 }
}
```

Bu şartlarda nested class'ın metodunu işletmek istersek hata meydana gelir. Bu hatalı durumu şöyle yorumlamak mümkündür:

- C# derleyicisi private olan nested class'ların örneklerini kapsayıcı sınıfın içinde alınmasına izin verirken private metotlara erişilmesine izin vermemektedir.
- Nested class'ımız private iken başka class'lardan erişim sağlanamayacağına göre nested class'ın private olan metotlarının internal veya public olmasının bir sakıncası olmaz.

Bu tespitten sonra nested class'ımızın metodunu internal yaptık ve kapsayıcı sınıf içinde bu metodu kullandık. Bu şekilde düzenlenen kapsayıcı sınıfın örneği başka bir sınıfta alınırsa nested sınıftaki metot işletilir. Çünkü nested sınıfın metodu kapsayıcı sınıfın yapıcı metodunda işletilmektedir.

```
public class Adana
{
 public Adana()
 {
  Seyhan nesne = new Seyhan();
  nesne.Hakkinda();
 }
 private class Seyhan
 {
  internal void Hakkinda()
   {
    System.Console.WriteLine("Adana'nın en büyük ilçesi");
   }
 }
}
```

Buraya kadar anlatılanları 1. senaryo olarak özetleyelim:

- Uygulamanızda üzerinde çalıştığınız sınıfta yararlanmak üzere başka bir sınıf hazırlamak istiyorsanız ve bu sınıfı başka sınıflarda kullanmayı düşünmüyorsanız, yukarıda yaptığımız gibi bu sınıfı nested class olarak düzenleyebilirsiniz.
- Böyle bir amaç için hazırlamak istediğiniz nested class'ı private yapabilirsiniz. Çünkü nested class private olsa bile kapsayıcı class'tan erişim sağlanabiliyor. Yani nested class, sadece kapsayıcı class'a özel bir class olmuş oluyor.
- Bu sayede yani nested class olarak düzenlediğiniz class'ınız kullanılacağı yerde durmuş oluyor.

Şimdi gelelim 2. senaryoya:
Ya nested class olarak düzenlediğiniz sınıfı kapsayıcı sınıfın dışında başka sınıflarda da kullanmak isterseniz? Böyle bir ihtiyaç karşısında nested class'ı internal veya public yapmanız gerekir. Bu düşünce ile yukarıda verdiğimiz kapsayıcı class'ımızı ve içerdiği nested class'ı aşağıdaki gibi düzenledik.

```
public class Adana
{
  public class Seyhan
  {
   public void Hakkinda()
   {
    System.Console.WriteLine("Adana'nın en büyük ilçesi");
   }
  }
}
```

Hem kapsayıcı hem de nested class'ımız public oldukları için başka class'ların içinde örnekleri alınabilir. Ancak nested class tipinde nesne aşağıda yapıldığı gibi hazırlanamaz.

```
class Program
{
 public static void Main()
 {
  Adana n1 = new Adana();
  Seyhan s1 = new Seyhan(); // hata meydana gelir.
 }
}
```

Çünkü nested class, "Adana" adını verdiğimiz kapsayıcı sınıftan bağımsız değildir. Başka bir deyişle bu nested class'ın varlığı "Adana" adını verdiğimiz kapsayıcı sınıfa bağlıdır. Bu nedenle nesne hazırlarken nested class'tan önce kapsayıcı yani dış sınıfın adını yazmak gerekiyor. Nested class tipinde nasıl nesne hazırladığımızı aşağıda görebilirsiniz.

```
class Program
{
 public static void Main()
  {
   Adana.Seyhan s1 = new Adana.Seyhan();
   s1.Hakkinda();
  }
}
```

Şimdiye kadar nested class'lar hakkında bilgi vermek üzere hazırladığımız uygulamada bir kapsayıcı sınıf ve sınıfın içinde bir nested sınıf hazırladık. Gerek duyulursa nested class'ın içinde başka bir nested class hazırlanabilir.

Kalıtım - Inheritance Kavramı

Nesneye yönelik programcılıkta kalıtım konusu önemlidir. İlk bakışta şunları bilmenin yeterli olacağı öngörülür: Önce kalıtım işlemine kaynaklık edecek bir taban sınıf (base class) hazırlanır. Sonra bu taban sınıftan yola çıkılarak yeni sınıflar hazırlanır (türetilir). Taban sınıfta yapılan hazırlıklar türetilen sınıflara aktarıldığı için aynı işlemler için tekrar kod yazmak zorunda kalınmaz.

Gerçekte kalıtım işleminin özeti böyledir. Ancak C# programcısı kalıtım hakkında daha fazlasını bilmek durumundadır. Çünkü C# kodu yazarken neredeyse yapılan her işlem kalıtımdan bir şekilde etkileniyor. Örneğin şimdiye kadar defalarca kullandığımız **new** anahtar kelimesi kalıtım kavramının merkezi gibidir.

Kalıtım, nesneye yönelik programcılıkta geniş ve değişik etkileri olan bir konudur. Bu kısımda C# ile uygulama geliştirenlerin kalıtım konusunda gerek duyabilecekleri konular üzerinde konuşulup bazı ayrıntılardan söz edilecektir.

Kalıtım nedir sorusuna bir cümleyle cevap vermek gerekirse, bir sınıfın bütün metotlarını, özelliklerini, alanlarını ve olaylarını başka bir sınıfa aktarmaktır denilebilir. Böylece programcı sınıflarını her seferinde sıfırdan hazırlamak zorunda kalmıyor.



Konunun hemen başında önemli bir noktadan söz etmekte fayda vardır: C# programlama dili, nesneye yönelik programcılıkta önemli bir dil olan C++'dan farklı olarak çoklu kalıtımı desteklemiyor. C#'ın bu eksikliği interface'ler, abstract class'lar ve nested class'lar sayesinde belli ölçüde giderilmektedir. Bu nedenle konu C# ve kalıtım olduğunda interface'ler ve abstract sınıflar önemlidir. Yani, abstract class'lar ve interface'ler hakkında yeteri bilgiye sahip olmadığınız sürece kalıtım konusundaki bilgileriniz eksik kalmış olacaktır. Bu bölümün ileriki sayfalarında abstract class'lardan söz edilecektir. Interface'ler ise ayrı bir bölüm olarak ele alınacaktır.

C#'ın kalıtım imkanlarından bu kitabın 3. bölümünden itibaren yararlanmaya başladık. Örneğin daha önceki konulardan bildiğiniz gibi Visual Studio ile gelen "Windows Forms Application" şablonu yardımıyla bir C# projesi hazırladığınızda bu proje dahilinde aşağıdaki gibi satırlara sahip "Form1" adında bir sınıf hazırlanmaktadır. Her ne kadar Visual Studio tarafından bu sınıf için hazırlanan kodların bir kısmı "Form1.Designer.cs" dosyasında olsa bile programcının ilgileneceği kodlar bunlardır. "Form1.Designer.cs" dosyasındaki kodlar Visual Studio yardımıyla görsel tasarım yapıldığında işlevsel olmaktadır.

```
public partial class Form1 : Form
{
 public Form1()
  {
   InitializeComponent();
  }
}
```

Visual Studio tarafından hazırlanan "Form1" adlı sınıfın deklare edildiği ilk satırda, sınıf adından sonra özellikleri, olayları, metotları vs. her şeyi miras alınıp yeni sınıfa aktarılmak istenen sınıfın (base class) adı yazılmaktadır. Taban sınıf (base class, burada Form) ile türeyen (derived class, burada Form1) sınıf ":" ile birbirinden ayrılmaktadır.

"System.Windows.Forms" adlı namespace'te bulunan **Form** sınıfı çok sayıda metoda, olaya ve özelliğe sahip olduğu için yukarıda 3 satırda hazırlanan "Form1" sınıfı Windows uyumlu pencerelerin gerek duyduğu bütün metot, özellik ve alanlara sahip olur.

Visual Studio tarafından hazırlanan bu 3 satırın (blok başlatan parantezler hariç) üzerinde biraz durmakta fayda vardır. Çünkü Visual Studio ile C# uygulaması geliştirenler bu 3 satırla sıklıkla yüz yüze gelmektedir. Bu sınıfla ilgili olarak söyleyeceğimiz, herhangi bir parametreye sahip olmayan yapıcı metoda sahip olduğudur. Bu bölümde "Form1" sınıfı hakkında konuşmayı düşünmeyip, konuya sadece kalıtım penceresinden bakmak istediğimiz için burada kullanılan **InitializeComponent**() metoduyla ilgilenmeyeceğiz.

Bu şekilde Form sınıfının mirasçısı bir sınıf hazırlandığımızda **Form** sınıfından gelen bütün metot, olay ve özelliklerden yararlanabiliyoruz. Çünkü .NET Framework ile gelen Form sınıfının orijinal halinde sınıfın üyelerinin çoğunluğu public, bazıları ise protected'dır. Bu sayede .NET Framework ile gelen Form sınıfının bütün üyeleri türeyen sınıfa aktarılmaktadır. Form sınıfının orijinal yapısının bir kısmı aşağıda verildi.

```
namespace System.Windows.Forms
{
    public class Form : ContainerControl
    {
        public Form();

        public static Form ActiveForm { get; }
        public bool AllowTransparency { get; set; }
        public override Size MaximumSize { get; set; }
        public Point Location { get; set; }
        public bool KeyPreview { get; set; }
        public bool IsRestrictedWindow { get; }
        public bool IsMdiContainer { get; set; }
        public bool IsMdiChild { get; }
        public Icon Icon { get; set; }
        public bool HelpButton { get; set; }
        public DialogResult DialogResult { get; set; }
```

Microsoft firmasının programcıları "Form" sınıfından türeyen sınıfa (burada Form1) aktarılmasını istemedikleri özellikler, metotlar veya olaylar olsaydı bunları public veya protected yapmayıp **private** yaparlardı. Erişim belirleyicisi olarak public ile protected farkı önemlidir. Base class'ın public üyeleri herkese açık iken protected üyelere ancak türeyen sınıftan (derived) erişilebilir. Dolayısıyla erişim belirleyici olarak **protected** anahtar kelimesi ancak kalıtım söz konusu olduğunda işlevsel olmaktadır.

Kalıtım kavramı hakkında biraz konuştuktan sonra konuyu anlatmayı kod yazarak ve yazdığımız kodu anlatarak devam etmek istiyoruz. Bu amaçla Visual Studio ile hazırladığımız konsol uygulamasında yeni bir sınıf hazırladık ve Main() metoduna sahip "Program" adlı sınıfı aşağıdaki gibi düzenledik.

```
class Aritmetik
{
 public static int Toplama(int i, int j)
  {
   return i + j;
  }
}
class Program : Aritmetik
{
 public static void Main()
  {
   int sonuc = Toplama(100, 200);
   Console.WriteLine(sonuc);
  }
}
```

Bu kodda önce bir statik metoda sahip "Aritmetik" adında bir sınıf hazırladık. Ardından Visual Studio'nun hazırladığı "Program" sınıfının ilk satırında önce "Program" sonra "Aritmetik" sınıfının adını yazdık. Bu sayede "Aritmetik" sınıfının metotları ve özellikleri "Program" sınıfında kullanılabilir duruma geldi. Bu hazırlıktan sonra bu uygulama derlenip çalıştırıldığında hata vermez ve beklenen sonuç alınır.

"Aritmetik" adını verdiğimiz sınıftaki metot statik olduğu için türeyen (derived) sınıfta bunları kullanmak için nesne hazırlamamız gerekmedi. Diğer yandan burada "Program" adını verdiğimiz sınıfı "Aritmetik" sınıfının mirasçısı yapmakla elimize geçen tek şey Main() metodu içinde "Aritmetik" sınıfı kaynaklı metotları işletirken metot adından önce sınıf yazmaktan kurtulmuş olmamızdır.

Doğaları gereği statik metotların kalıtıma fazla katkıları yoktur. Kalıtımla ilgili ayrıntıları ortaya çıkarmak için yukarıda verdiğimiz uygulamada base class durumundaki "Aritmetik" sınıfındaki metodun statik olma özelliğini kaldırdık. Ayrıca konunun kolay izlenebilmesi için konsol uygulamamızda 3. bir sınıf hazırladık.

```
class Aritmetik
{
 public int Toplama(int i, int j)
  {
   return i + j;
  }
}
class Islemler : Aritmetik
{
}
class Program
{
 public static void Main()
 {
  Islemler nesne = new Islemler();
  int sonuc = nesne.Toplama(100, 200);
  Console.WriteLine(sonuc);
 }
}
```

Bu kodu yakından incelediğinizde kalıtımla ilgili en temel işlemin gerçekleştirildiğini görebilirsiniz. "Aritmetik" adını verdiğimiz bir base class'ımız ve bu class'ın metotlarını olduğu gibi alıp kullanan türeyen (Islemler) sınıfımız var. Burada hazırladığımız base class'ta sadece metotlara yer verdik. İsteseydik alan ve özelliklere de yer verebilirdik.

Bu örnek kodda "Aritmetik" adlı sınıfta hazırladığımız metodun ilk halini koruyup derived sınıfta (burada Islemler) kullandık. Ancak bazen programcılar base class'tan alınan metotları türeyen sınıfta revize etmek isterler. Şimdi base class'taki Toplama() metodunu türeyen sınıfta yeniden yazacağız.

```
class Aritmetik
{
 public int Toplama(int i, int j)
  {
   return i + j;
  }
}
class Islemler : Aritmetik
{
 public int Toplama(int i, int j)
  {
   return (i * i) + (j * j);
  }
}
class Program
{
 public static void Main()
  {
   Islemler nesne = new Islemler();
   int sonuc = nesne.Toplama(100, 200);
   Console.WriteLine(sonuc);
  }
}
```

Bu şekilde düzenlenen kod derlenmek istenirse derleyici uyarı mesajı verir ama exe dosyayı hazırlar. Bu kodu Visual Studio ile hazırlarsanız burada Toplama() adı verilen metodun altı uyarı amaçlı olarak çizilir. Buna rağmen Visual Studio, kodu derleyip çalıştırır ve beklenen sonuç alınır. Derleme sırasında uyarı mesajının verilmesinin nedeni, programcının türeyen sınıfta aynı metot adını dalgınlıkla kullandığının varsayılmasıdır.

Hem base hem de derived sınıfta aynı ada sahip olan 2 metodun olmasından dolayı alınan uyarı mesajını kaldırmak istiyorsanız türeyen sınıftaki metodun imzasına aşağıda yapıldığı gibi **new** anahtar kelimesini eklemeniz gerekir. Bu işleme üye gölgeleme adı verilmektedir. Bu konu bu bölümün ileriki sayfalarında tekrar ele alınıp ayrıntılardan söz edilecektir.

```
class Islemler : Aritmetik
{
 public new int Toplama(int i, int j)
 {
  return (i * i) + (j * j);
 }
}
```

Bu şartlarda "Islemler" sınıfı bağlamında elimizde Toplama() adında ve farklı işlemler yapan 2 metot bulunmaktadır. Yukarıda türeyen sınıf tipinde bir nesne hazırlayıp bu sınıftaki metodu işlettik. Base class'taki (Aritmetik) metodu işletmek istediğinizde ise ilk akla gelen Base Class tipinde nesne hazırlamaktır. İlk aklımıza gelene uyup Main() metodunun olduğu sınıfımızı aşağıdaki gibi düzenledik.

```
class Program
{
 public static void Main()
 {
  Islemler nesne1 = new Islemler();
  Aritmetik nesne2 = new Aritmetik();
  int sonuc = nesne2.Toplama(100, 200);
  Console.WriteLine(sonuc);
 }
}
```

Bu şartlarda ister base class'taki ister türeyen class'taki metodu işletebiliriz. Base Class'ta bulunan ve türeyen sınıfta new anahtar kelimesiyle gölgelenen metodu işletmek için genel alışkanlıkla base class tipindeki değişkene türeyen tipteki nesnenin referansı aktarılıp öyle işletilmektedir. Ne demek istediğimizi aşağıda görebilirsiniz.

```
class Program
{
 public static void Main()
 {
  Aritmetik nesne = new Islemler();
  int sonuc = nesne.Toplama(100, 200);
  Console.WriteLine(sonuc);
 }
}
```

Aynı sonucu almak, başka bir deyişle base class'taki metodu işletmek için türeyen class tipindeki nesne "as" operatörü ile base class'a dönüştürülebilir. Dolayısıyla uygulamanın Main() metodu aşağıda gibi düzenlendiğinde base class'taki metot işletilir.

```
class Program
{
 public static void Main()
 {
  Islemler nesne = new Islemler();
  int sonuc = (nesne as Aritmetik).Toplama(100, 200);
  Console.WriteLine(sonuc);
 }
}
```

Hazırladığınız sınıfın mirasının alınmasını yani base class olarak kullanılmasını engellemek istiyorsanız o sınıfı **sealed** sınıf yapmalısınız. Bütün metotları **static** olan sınıfların örneklerini hazırlamak anlamlı olmadığı için sealed sınıf olarak ayarlanabilir. Tabi, C# 2.0 ile birlikte gelen static class desteği sayesinde bütün üyeleri statik olan sınıfların kalıtıma kaynaklık etmesini engellemek için sealed anahtar kelimesini kullanma ihtiyacı ortadan kalkmıştır.

Sealed olarak işaret edilen sınıf herhangi bir sınıfın ana sınıfı yapılmak istendiğinde hata meydana gelir.

```
sealed class Aritmetik
{
 public static int Toplama(int i, int j)
  {
   return i +j;
  }
}
class Program : Aritmetik // Hata meydana gelir.
{
 public static void Main()
  {
   int sonuc = Toplama(100, 200);
   Console.WriteLine(sonuc);
  }
}
```

.NET Framework'te bütün sınıflar **Object** sınıfından türetilmektedir. Bu nedenle aşağıda verilen 2 sınıf hazırlama örneği birbirinden farklı değildir.

```
class Islemler : Object
{
}
```

```
Class Islemler
{
}
```

Bu bölümün ilk sayfalarında önce sınıflar hakkında genel bilgi edindiniz. Devamında sınıflarda bulunabilen özellikler ve metotlardan (biraz) söz edildi. Çünkü metotlarla ilgili ayrıntılar ayrı bir bölüm olarak ele alınacaktır. Şimdi sırada abstract class'lar var.

Abstract (Özet) Sınıflar

Abstract class'lara içeriden yani .NET Framework gelen sınıflardan örnek vermek gerekirse **stream** sınıfları uygun gibi durmaktadır. Şimdi ayrıntılara girmeden en basit şekli ile stream sınıflara "abstract class" penceresinden bakacağız. Muhtemelen Microsoft firmasının C# dilini geliştiren ekibindekiler aralarında toplandılar ve C# uygulamasına hangi kaynaklardan akış gelebileceği konusunu masaya yatırdılar. .NET Framework ile Stream adı altında gruplanan sınıflara bakılınca bu kaynaklar aşağıdaki gibi listelenebilir:

- Yerel bilgisayardaki dosyalar,
- Network'ten bir yerden gelen veya gönderilen bilgiler,
- Belleğe akıtılıp geçici olarak toplanan bilgiler,
- Değişik kaynaktan alınırken veya gönderilirken arabelleklere(buffer) uğratılıp biriktirilen bilgiler,
- Önemli oldukları için şifrelenip gönderilen veya şifreli halde gelmeleri nedeniyle şifrelerini çözmenin gerektiği bilgiler.

Bu listeden anlamamız gereken şudur: C# uygulaması dahilinde dışarıdan alınması muhtemel bilgiler hem farklı ortamlarda bulunuyorlar hem de işlenmeleri farklılık arz ediyor. Programcı olarak hem yerel dosya üzerinde işlem yaparken hem de network üzerinden bilgi

gönderir veya alırken kullandığımız sınıfların birçok metodunun aynı ada sahip olmasını isteriz. İşte C# dilini geliştiren ekip söz konusu bilgi alış verişi nereden ve nasıl yapılırsa yapılsın ortak özellik ve metotlarını içeren özet bir sınıf hazırlamışlar. Bu sınıfın adı **Stream**'dir. C# kodu yazarken kullandığımız stream sınıfları ise bu özet sınıftan yola çıkılarak hazırlanan sınıflardır.

Hazırlanmak istenen class'ın sadece base class olarak kullanılması ve örneğinin alınmasına izin verilmek istenmiyorsa **abstract** anahtar kelimesinden yararlanılıp özet sınıf hazırlanabilir. Tek cümle ile ifade etmek gerekirse, abstract class, benzer işlev ve özelliklere sahip class'lara temel olsun diye hazırlanır. Bu kitapta daha sonra hakkında bilgi verilecek interfacelerin benzer işleve sahip olduğunu en başta ifade etmeliyiz.

Bazen programcıların verimli olabilmeleri için ön çalışma yapılıp hazırlayacakları sınıflarda hangi alanların, hangi metotların bulunması konusunda bilgi içeren ve programcılara rehber olabilen sınıflar hazırlanır ki bunlara **abstract** sınıf denilir. Bazı kaynaklarda abstract sınıflardan soyut sınıf diye söz edilmektedir. Konu abstract sınıflar olduğunda ortada soyutlama veya soyut bir şey olmadığı için "özet" olarak adlandırmak gerekir.

Özet sınıf hazırlamadaki önemli amaçlardan birisi büyük projeleri karmaşadan uzak tutmaktır: Örneğin bir metot değişik verilere ihtiyaç duyup aynı işi yapıyorsa, bu metodun abstract bir sınıfın içerisinde tanımlanması önerilmektedir. Tabi abstact class'larda nelerin olması nelerin interface'lere konulması gerektiği tartışmaya açık bir konudur.

Bir sınıfa abstract özelliği verilmek istendiğinde "abstract" anahtar kelimesinden yararlanılmaktadır. Abstract sınıflardan yola çıkılarak **new** anahtar kelimesi ile nesne hazırlanamaz. Bu sınıflar metot ve alanlarını başka sınıflara miras yoluyla verebilirler.

Şimdi bu söylediklerimizi basit bir uygulama ile örnekleyeceğiz. Bu amaçla aşağıda verilen 4 sınıfın olduğu uygulamayı hazırladık. "Ozet" adını verdiğimiz sınıfın **abstract** olduğunu belirttik. 4 sınıfın bulunduğu bu uygulamada herhangi bir işlem yapılmamaktadır. "Futbol" ve "Basket" sınıfları "Ozet" adlı **abstract** sınıfın mirasçısıdır. Bu şartlarda bu uygulama hatasız derlenir.

```
namespace Uygulama1
{
 abstract class Ozet
  {
  }
 class Futbol : Ozet
  {
  }
 class Basket : Ozet
  {
  }
 class Program
  {
   public static void Main()
    {
    }
  }
}
```

Abstract class'lardan söz etmeye devam etmek üzere abstract sınıfın içinde abstract bir metot deklare edeceğiz. Bu cümleden şunu anlamış olmalısınız: Abstract sınıflardan başka bir de abstract metotlar vardır ve bu metotlar ancak abstract sınıflarda bulunabilir. Abstract sınıflarda abstract metotların nasıl kodlandığını aşağıda görebilirsiniz.

```
abstract class Ozet
{
 abstract public void Adres(string str);
}
```

Dikkat ederseniz abstract class'larda yer alabilen abstract metotlar yalnızca deklare edilmektedir. Başka bir deyişle metodun ne yaptığı belli değildir. Metodun geriye gönderdiği bilginin tipi, parametrelerinin sayısı ve tipi işaret edilmekle yetinilmektedir ki bu bilgilerin hepsine metodun imzası adı verilmektedir.



Konunun karışmasının önüne geçmek için ayrı bir yerde açıklama yapma gereğini duyduk: Daha birinci cümleden abstract sınıf ve metotların özelliklerini ve işlevlerini kavramanızı beklemiyoruz. Bu nedenle araya girip ek bilgiler vermeyi tercih ettik.

- Abstract sınıflarda istenirse normal metotlara yer verilebilir. Başlangıçta abstract class ve abstract metotlara odaklanmak istediğimiz için abstract class'ta normal metotlara yer vermeyeceğiz.
- Abstract olmayan sınıflarda abstract metotların olmasına derleyici izin vermez. Demek ki şimdiye kadar gördüğümüz abstract olmayan sınıflarda abstact metotlara yer vermek mümkün değilmiş.

Uygulamamızdaki "Ozet" adlı abstract class'ta bu şekilde bir metodun imzasına yer vermiş olmamızdan şu sunucu çıkarmak gerekir: Bu şartlarda "Futbol" ve "Basket" sınıfları geriye değer göndermeyen ve string tipte parametreli metotlara sahip olmak zorundadır. Çünkü "Basket" ve "Futbol" adını verdiğimiz sınıflar kendi hazırladığımız abstract sınıfın mirasçısıdır. Bu uygulamanın hatasızca derlenebilmesi için "Futbol" ve "Basket" sınıflarında string tipte parametrelere sahip metotlar hazırladık.

```
class Futbol : Ozet
{
 public override void Adres(string adr)
  {
   Console.WriteLine(adr);
  }
}
class Basket : Ozet
{
 public override void Adres(string str)
  {
   string s = str.ToUpper();
   Console.WriteLine(s);
  }
}
```

Burada abstract sınıfın mirasçısı durumunda olan "Basket" ve "Futbol" sınıflarında abstract metotların **override** anahtar kelimesi kullanılarak yazılması gerekiyor. Başka bir deyişle bu metotların yapacakları işlemler için kod yazılması şarttır. Aşağıda abstract class ve metotlarla ilgili olarak şimdiye kadar yazdığımız kodun tamamı var.

```
using System;
namespace Uygulama1
{
 abstract class Ozet
 {
  abstract public void Adres(string str);
 }
 class Futbol : Ozet
 {
  public override void Adres(string str)
   {
    Console.Write(str);
   }
 }
 class Basket : Ozet
 {
  public override void Adres(string str)
   {
    string s = str.ToUpper();
    Console.Write(s);
   }
 }
 class Program
 {
  public static void Main()
   {
    Basket b = new Basket();
    b.Adres("Adana");
   }
 }
}
```

Bu uygulama derlenip çalıştırılırsa önce "Basket" sınıfının örneği alınır. Basket sınıfı "Ozet" adını verdiğimiz abstract sınıfın mirasçısı olduğu ve bu sınıfta "Adres" adında abstract bir metoda yer verildiği için bu metoda sahip olacaktır. Adres() adını verdiğimiz metot "abstract class" kaynaklı bir metot olduğu için "Basket" sınıfında bu metodu kodlarken **override** anahtar kelimesini kullanmamız gerekti.

Bu paragraf bir bakıma abstract sınıfların özelliklerini özetlemektedir. Bu nedenle ilk okumada size karışık gelebilir. Bu paragrafta yazılanların anlaşılması önemli olduğu için bu paragrafta söylenenleri bir de maddeler halinde sıralayacağız:

- Uygulamamızda "Ozet" adında bir abstract class'ımız var.
- Bu abstract class'ta, "Adres" adında geriye değer göndermeyen string tipte bir parametreye sahip abstract bir metot bulunmaktadır.
- Abstract metotlar sadece abstract sınıflarda yer alabilirler ve abstract metotlar abstract sınıflarda kodlanırken yalnızca imzaları olur.
- Normal bir sınıf, abstact sınıfı base class olarak kullanırsa yani onun mirasçısı olursa base class işlevi gören abstract class'taki bütün abstract metotlar **override** anahtar kelimesi kullanılarak hazırlanmalıdır.

- Uygulamamızda "Ozet" adlı class'ı base class olarak kullanan 2 sınıf olduğu (Basket ve Futbol) için her iki sınıfta da abstract metodu override anahtar kelimesini kullanarak hazırladık.

Abstract sınıflarda abstract olmayan metotlara yer verebilirsiniz. Hatta sadece abstract olmayan metotlara sahip bir abstract class hazırlayabilirsiniz. Eğer abstract sınıfınızdaki bütün metotlar normal ise, abstract anahtar kelimesinin o sınıfa tek katkısı örneğinin alınmasına engel olmak olur. Diğer yandan hazırladığınız abstract sınıfta sadece abstract metotlar varsa, bu sınıf işlev bakımından interface'lere yakın olmuş olur.

Şimdi konunun diğer ayrıntılarını ortaya çıkarmak için yukarıda hazırladığımız abstract sınıfta normal bir metoda yer vereceğiz. Bu şartlarda abstract sınıfımız birisi normal diğeri abstract olmak üzere 2 metoda sahip olmuş olacak.

```
abstract class Ozet
{
 abstract public void Adres(string str);
 public void Mesaj(string msj)
  {
   Console.WriteLine(msj);
  }
}
```

Bir projeyi geliştirmek üzere bir araya gelmiş ekipte söz sahibi bir programcının işe koyulduğunu ve base class olarak kullanacakları birkaç abstract class hazırladığını varsayalım. Bizce proje yöneticisi abstract class'ta bulunacak normal metotlar üzerinde kafa yormalıdır ve mirasçı sınıflarda ihtiyaç duyulması ihtimali yüksek olan her işlem için bir normal metot hazırlamalıdır. Örneğin .NET Framework ile gelen Object sınıfının ToString() metodu böyle bir metottur. Programcı ToString() metodunu mirasçı sınıfta ister yeniden yazar isterse de ilk haliyle olduğu gibi bırakır.

Konuya, abstract sınıftaki normal metotlar penceresinden bakılırsa, abstract sınıftaki metotlar türeyen sınıfta ilk yazıldığı haliyle kullanıma hazır halde dururlar. Bu değişiklikten sonra bu abstract sınıfın mirasçısı "Futbol" ve "Basket" adını verdiğimiz sınıfların örneği alındığında burada Mesaj() adını verdiğimiz metot ilk yazıldığı haliyle kullanılabilir.

```
class Program
{
 public static void Main()
  {
   Basket b = new Basket();
   b.Mesaj("Abstract olmayan metot işletildi");
  }
}
```

Programcı abstract sınıftan gelen normal metodu olduğu gibi bırakmayıp yeniden yazabilir. Türeyen sınıfta abstract sınıf kaynaklı metot yeniden yazılırken **new** anahtar kelimesinden yararlanmak gerekmektedir. new anahtar kelimesinden yararlanarak "Basket" adını verdiğimiz sınıfta abstract sınıf kaynaklı metodu nasıl yeniden yazdığımızı aşağıda görebilirsiniz.

```
class Basket : Ozet
{
 public override void Adres(string str)
  {
   string s = str.ToUpper();
   Console.Write(s);
  }
 public new void Mesaj(string msj)
  {
   string s = msj.ToLower();
   Console.WriteLine(s);
  }
}
```

Bu şartlarda uygulamamızdaki Main() metodunda bu sınıfın örneği alınıp burada Mesaj() adını verdiğimiz metot işletilirse metodun yeniden yazılmış hali işletilir.

```
class Program
{
 public static void Main()
 {
  Basket b = new Basket();
  b.Mesaj("Abstract Olmayan Metodun Yeni Hali İşletildi");
 }
}
```

"Basket" adını verdiğimiz sınıf tipinde nesne hazırlayıp hem metodun yeni haline hem de abstract sınıftaki ilk haline gerek duyuyorsanız nesneye yönelik programcılığın çok biçimlilik silahına başvurabilirsiniz. Yani "Basket" sınıfının örneğinin referansını abstract sınıfımız tipindeki değişkene aktarmanız gerekir.

Abstract sınıflarla ilgili olarak şimdiye kadar anlatılan işlemleri net olarak görebilmeniz için kodun tamamını bir arada verdik. Yazdığımız uygulamanın Main() metodunu dikkatlice incelerseniz polimorfizm ilkesinin izlerini görebilirsiniz.

```
using System;
namespace Uygulama1
{
 abstract class Ozet
 {
  abstract public void Adres(string str);
  public void Mesaj(string msj)
   {
    Console.WriteLine(msj);
   }
 }
 class Futbol  : Ozet
 {
  public override void Adres(string str)
   {
    Console.Write(str);
   }
 }
```

```
class Basket : Ozet
{
 public override void Adres(string str)
  {
   string s = str.ToUpper();
   Console.Write(s);
  }
 public new void Mesaj(string msj)
  {
   string s = msj.ToLower();
   Console.WriteLine(s);
  }
}
class Program
{
 public static void Main()
  {
   Basket b1 = new Basket();
   b1.Mesaj("Abstract Olmayan Metodun Yeni Hali İşletildi");
   Ozet b2 = new Basket();
   b2.Mesaj("Abstract Olmayan Metodun İlk Hali İşletildi");
  }
 }
}
```

Yukarıda hazırladığımız abstract sınıfta yalnızca metotlar vardı. Abstract sınıflar, metotlardan başka alanlara sahip olabilmektedir. Başka bir deyişle abstract sınıfın mirasçısı olacak sınıflarda bulunması istenen alanları abstract sınıfta belirtebiliriz. Şimdi abstract sınıfın ayrıca bir alana sahip olmasını sağlayacağız.

```
abstract class Ozet
{
 public string subeBaskan;
 abstract public void Adres(string str);
}
```

Abstract sınıfta tanımladığınız alana ilk değer aktarmazsanız derleme sırasında uyarı mesajı alırsınız. Bu değişiklikten sonra bu sınıfın mirasçısı sınıfların ayrıca string tipinde bir alana sahip olması gerekir. Abstract sınıfta bu değişikliği yaptıktan sonra bu sınıfın mirasçısı durumundaki "Futbol" sınıfını aşağıdaki gibi düzenledik. "subeBaskan" adını verdiğimiz alan abstract sınıfta yer aldığı için **new** anahtar kelimesi ile yeniden oluşturulmalıdır.

```
class Futbol : Ozet
{
 public new string subeBaskan;
 public override void Adres(string adr)
  {
   subeBaskan = "Selim Soydan";
   Console.WriteLine(adr);
  }
}
```

Abstract sınıflardan yola çıkılarak nesneler hazırlanamaz. Buna rağmen abstract sınıfların yapıcı metotları olabilir. Abstract sınıflarda en azından bir adet abstract metot olmalıdır.

Abstract sınıftaki metotlar türeyen sınıflarda mutlaka override edilmelidir, abstract sınıflardan yararlanarak başka abstract sınıfı türetmek mümkündür.

abstract class'ta hazırlanan abstract metotlar **private** olamazlar. abstract class'larda abstract özellikler olabilir. Abstract class abstract özellik içeriyorsa abstract class'ın mirasçısı class'ta bu özellik **override** edilmeli ve get ile set blokları olmalıdır. Herhangi bir metodu abstract olan class'ın mutlaka abstract olması gerekir.

```
abstract class Ozet
{
 abstract public void Adres(string str);
 abstract public int toplam
 get;
 set;
}
```

Interface'ler ve Özet Sınıflar

Interface'ler ile abstract sınıflar benzer işleve sahiptir. Yukarıda ifade edildiği üzere özet sınıflar normal metotlara sahip olabilirken interface'ler sahip olamıyorlar. Bazen içinde bulunulan şartlara göre mevcut interface'ler hazırlanacak abstract sınıfa uygulanmak istenebilir. Bu konuda bilgi vermek üzere aşağıdaki gibi herhangi bir metoda sahip olmayan bir abstract sınıf hazırladık ve bu sınıfa elimizdeki interface'lerden birisini uyguladık.

```
public interface Islemler1
{
 int Kare(int X);
}
public abstract class Ozet1 : Islemler1
{
 public int Kare(int Sayi)
  {
    return Sayi * Sayi;
  }
}
```

C# derleyicisi burada yaptığımız işleme itiraz etmez. Başka bir deyişle bu andan itibaren "Ozet1" adını verdiğimiz sınıf interface'den gelen metot ve alanlara sahiptir. Bu özet sınıf hangi sınıfa uygulanırsa o sınıfta interface'den gelen metotların olduğu gibi bırakılıp kullanılabilir. Aşağıda verilen kodda bu özet sınıfı normal bir sınıfa nasıl uyguladığımızı görebilirsiniz.

```
public class Sinif1 : Ozet1
{
}
```

Abstract sınıfların özelliklerini maddeler halinde özetlemek gerekirse;

- Abstract sınıflardan yola çıkıp new anahtar kelimesi ile nesne hazırlanamaz.
- Abstract sınıftaki abstract metotlar türeyen sınıflarda mutlaka override edilmelidir.
- Bir sınıfın abstract olarak değerlendirilebilmesi için en azından abstract bir metoda sahip olmasını sağlamanız önerilir.

- Abstract sınıflar başka abstract sınıfın mirasçısı olabilmektedir.
- Abstract sınıflar yapıları gereği private olamazlar.
- Abstract bir sınıf hem abstract hem de abstract olmayan üyelere sahip olabilmektedir.
- Abstract anahtar kelimesi **hem metotlar** için **hem de sınıflar** için kullanılabilmektedir.

Yukarıda işaret edildiği üzere hazırladığınız sınıflardan birisinden yola çıkılarak başka bir sınıfın hazırlanmasını engellemek, başka bir deyişle bu sınıfın base class olarak kullanılmasını istemiyorsanız **sealed** anahtar kelimesinden yararlanmak gerekiyor.

Statik Sınıflar

Bu kitabın ilk bölümünde statik metotlardan kısaca söz edilmişti. Sonra Değişkenler adlı bölümde statik değişkenler hakkında bilgi vermek amacıyla statik metot ve alanlardan söz etme gereği duyulmuştu. Şimdi sırada statik sınıflar var. Statik sınıflardan söz edilirken ister istemez statik alan ve metotlardan da söz edilecektir. Burada sözü edilecek statik metot ve sınıflardan başka bir de statik yapıcı metotlar vardır.

C# programlama dilinin ilk sürümü için konuşmak gerekirse, sınıflarda statik metot ve alanlara yer verilebiliniyordu. Sınıfı statik yapma imkanı olmadığı için sınıflar, statik üyelere sahip olsalar bile statik olmayan üyeleri de içerip örneğini hazırlamak mümkündü. C# 2.0 ile birlikte statik sınıf desteği geldi. Yani sınıfın kendisini statik yapma imkanı programcılara sunuldu. Statik olmayan sınıflarda hem statik hem de statik olmayan üyelere yer verilebilinirken statik sınıfların bütün üyeleri statik olmak zorundadır. Bir sınıfı statik olarak ayarlamakta amaç, tüm üyeleri statik olmaya zorlamak olabilir.

Önce 2 cümlelik genel bilgi: Şimdiye kadar herhangi bir sınıftaki alan ve metoda gerek duyulduğunda önce sınıf tipinde bir nesne oluşturup ardından alan veya metoda nesne üzerinden ulaşıp kullanıyorduk. Ancak bazı durumlarda nesnelerden bağımsız olarak metotlara veya alanlara erişme ihtiyacı duyulabilir. Bu ihtiyaç karşısında **static** anahtar kelimesine başvurulur.

Statik alanlar, metotlar ve sınıflar hakkında adım adım bilgi vermek için Visual Studio ile aşağıdaki gibi standart bir konsol uygulaması hazırladık. Bu şartlarda uygulamamızdaki sınıf (Program) static bir metoda sahiptir. Bu statik metotta Console sınıfının statik **ReadKey**() metodu çağrılmıştır.

```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
   {
    Console.ReadKey();
   }
  }
}
```

Konu ile ilgili bazı ayrıntıları ortaya çıkarmak için Main() metoduna sahip "Program" sınıfını static yaptık ve Main() metodu içinde.NET Framework ile gelen ArrayList sınıfı tipinde nesne hazırlayıp elemanlar ekledik.

```
static class Program // statik sınıf
{
 static void Main(string[] args)
 {
  ArrayList liste = new ArrayList();
  liste.Add("Adana");
  liste.Add("Ankara");
 }
}
```

Sınıfımız statik olmasına rağmen statik olmayan ArrayList sınıfının örneğini alıp yine statik olmayan **Add**() metodunu kullandık. Derleyicinin bu yaptığımıza bir itirazı olmadı. Çünkü derleyicinin nazarın burada "liste" adını verdiğimiz ArrayList tipindeki nesne yerel değişken muamelesi görmektedir. Daha doğrusu, "liste" nesnesinin yaşam alanı Main() metodu ile sınırlıdır. Statik alan ve metotlar doğaları gereği global oldukları için yerel değişkenlerden etkilenmiyorlar. "liste" adını verdiğimiz nesnenin burada "Program" adı verilen sınıfın her yerinde yaşaması için aşağıdaki gibi düzenleme yapmış olsaydık derleyici hata rapor ederdi.

```
static class Program
{
 ArrayList liste; // hata meydana gelir.
 static void Main(string[] args)
 {
  liste = new ArrayList();
  liste.Add("Adana");
  liste.Add("Ankara");
 }
}
```

Çünkü class bloğu içinde bu şekilde ArrayList tipinde değişken tanımlamakla gerçekte "Program" sınıfının statik olmayan bir alana sahip olmasını sağlamaya çalışmış oluyoruz. Yukarıda işaret ettiğimiz üzere C# derleyicisi statik sınıflarda statik olmayan üyelerin olmasına izin vermemektedir. Şimdi ise statik olmasını sağladığımız "Program" sınıfında normal bir metot hazırlayacağız.

```
static class Program
{
 static void Main(string[] args)
  {
   Console.ReadKey();
  }
  void Mesaj(string str) // hata meydana gelir.
   {
    Console.WriteLine(str);
   }
}
```

Derleyici bu yaptığımıza da itiraz eder. Çünkü yine statik sınıf içinde statik olmayan üyeye (metot) yer vermeye çalıştık. Bu hatayı gidermenin bir yolu, burada Mesaj() adını verdiğimiz metodu statik yapmaktır.

Static üye elemanları oluştururken "static" ile erişim belirleyici anahtar kelimesinin sırasının önemi yoktur. Yani "static public void Mesaj()" ile "public static void Mesaj()" kullanımları aynı kabul edilir.

```
static class Program
{
 static void Main(string[] args)
  {
    Mesaj("Fenerbahçe");
  }
 static void Mesaj(string str)
  {
   Console.WriteLine(str);
  }
}
```

Daha önceki konulardan bildiğiniz gibi burada Mesaj() adını verdiğimiz statik metot başka bir sınıfta olsaydı metot adından önce sınıf adını yazardık. Aynı şekilde bu metot başka bir namespace'te olsaydı sınıf adından önce ayrıca namespace adını yazardık.

Statik sınıflar, statik metotlar ve statik alanlar hakkında konuşmaya devam etmek üzere üzerinde çalıştığımız konsol uygulamasındaki "Program" adlı sınıfın statik olma halini iptal edip normal sınıf yaptık.

```
class Program // statik olmayan sınıf
{
 static void Main(string[] args)
  {
   Mesaj("Fenerbahçe");
   Console.ReadKey();
  }
  static void Mesaj(string str)
  {
    Console.WriteLine(str);
  }
}
```

Bu şartlarda değişen bir şey olmaz. Çünkü uygulamanızdaki statik Main() metodu içinde sınıfın yine statik olan diğer metodu işletmekle yetindik. Burada Mesaj() adını verdiğimiz metot statik olmasaydı ve bu metodu Main() metodu içinde işletmek isteseydik derleyicinin tepkisi ne olurdu.

```
class Program
{
 static void Main(string[] args)
  {
   Mesaj("Fenerbahçe"); // hata meydana gelir.
   Console.ReadKey();
  }
```

```
    void Mesaj(string str)
     {
      Console.WriteLine(str);
     }
}
```

Daha önceki konulardan bildiğiniz gibi Main() metodu statik bir metot olduğu için aynı sınıftaki statik olmayan metotlar Main() metodu içinde direk metot adı yazılarak işletilemezler. Çünkü burada Mesaj() adını verdiğimiz instance bir metottur ve ancak nesne üzerinden işletilebilir. Dolayısıyla bu metodu işletmek için üyesi olduğu sınıf tipinde bir nesneye ihtiyaç vardır. Mesaj() metodu "Program" sınıfının üyesi olduğu için "Program" sınıfı içinde "Program" tipinde bir nesne hazırladık.

```
class Program
{
 static void Main(string[] args)
  {
   Program p = new Program();
   p.Mesaj("Fenerbahçe");
  }
 void Mesaj(string str)
  {
    Console.WriteLine(str);
  }
}
```

Bu kitabın daha önceki bölümlerinde ifade edildiği gibi burada ilk bakışta garip bir durum varmış gibi görünüyor; "Program" sınıfı içinde Program tipinde nesne hazırlıyoruz. Konunun özüne inildiğinde burada yapılan gayet normaldir. Her ne kadar buradaki adlarıyla Main() ve Mesaj() metotları aynı sınıfın üyeleri olsalar bile uygulama derlenip çalıştırıldığında bellekte Main() metodu ile, new anahtar kelimesi ile hazırladığımız nesne ayrı yerlerde olurlar. Dolayısıyla yukarıda verilen koda nesneye yönelik programcılık penceresinden bakıldığında her şey normal görünüyor.



Hangi durumlarda statik sınıf hazırlanması, hangi amaçla kullanılacak üyelerin statik yapılacağı sorusunun net bir cevabı yoktur. Bu konuda genellikle programcının veya firmanın tercihleri ön plana çıkıyor. Yine de şöyle bir öneride bulunabilinir: Nesnelere bağlı olmadan yapılan işlemler söz konusu olduğunda statik sınıflar, metotlar veya alanlar hazırlanabilir

Böylece statik sınıf ve metotlar hakkında ön bilgiye sahip olduktan sonra uygulamamızda ikinci bir sınıfa yer vereceğiz. Statik sınıf, metot ve alanlarla ilgili diğer ayrıntıları anlatırken bu sınıftan yararlanacağız.

```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
   {
   }
 }
```

```
 public class Islemler
  {
  }
}
```

Bu şartlarda "Islemler" adını verdiğimiz sınıf normal yani statik olmayan bir sınıf olduğu için hem statık hem de statik olmayan üyelere (metot ve alan) sahip olabilir. Bu sınıfta ilk olarak 2 statik alana yer verdik. Ardından uygulamadaki Main() metodu içinde bu alanların içeriklerini konsola yazmaya çalıştık.

```
class Program
{
 static void Main(string[] args)
 {
  Console.WriteLine(Islemler.Sehir);
  Console.WriteLine(Islemler.Yil);
 }
}
public class Islemler
{
 public static string Sehir;
 public static int Yil;
}
```

Dikkat ettiyseniz "Sehir" ve "Yil" adını verdiğimiz alanları public yaptığımızı fark ederseniz. Bu 2 alanı public yapmasaydık diğer sınıftan (burada Program) erişilmez olurlardı. 2 sınıfın olduğu bu uygulama derlenip çalıştırılırsa statik alanlar otomatik olarak bellekte kendilerine yer bulurlar ve diğer sınıflar bunlara erişebilirler. Başka bir anlatımla statik alanlar, diğer programlama dillerindeki global değişkenler gibi işlev görmektedir. Çünkü statik alanlar, ilgili sınıfın bütün kopyaları tarafından ortak olarak kullanılan alanlardır.

Bu şekilde düzenlenen uygulama derlenip çalıştırılırsa herhangi bir hata ile karşılaşılmadan önce konsola "Sehir" alanın içeriği olarak boşluk, sonra "Yil" alanın içeriği olarak 0 yazılır. Daha önce değişkenler adlı bölümde ifade edildiği üzere statik olmayan değişkenlere ilk değer aktarılmadan içeriğine erişmeye çalışmak hataya neden olurken statik değişkenlere ilk değerler otomatik olarak aktarılmaktadır. Statik alanlara ilk değerleri aşağıda yapıldığı gibi tanımlama satırında aktarabilirsiniz.

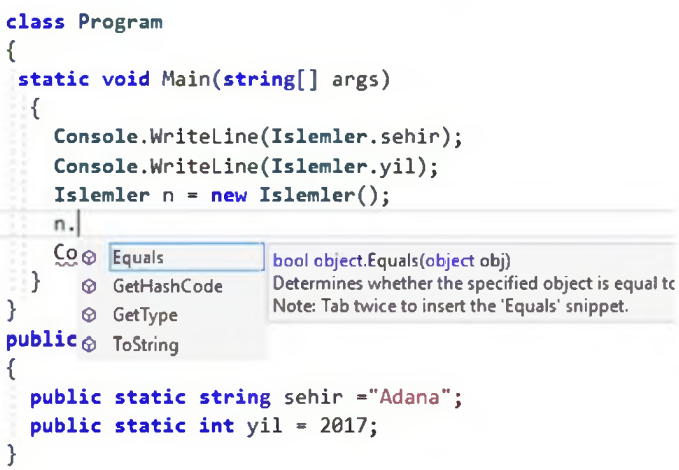
```
public class Islemler
{
 public static string Sehir ="Adana";
 public static int Yil = 2017;
}
```

Bu şekilde düzenlenen "Islemler" sınıfının statik alanların içeriği herhangi bir metodun içinde aynı şekilde değiştirilmediği sürece içeriklerini korurlar. Örneğin bu sınıfın örneği alındığında statik alanlar bundan etkilenmez. Konu üzerinde konuşup bazı noktaları açığa çıkarmak için yukarıda verilen koda ekleme yapıp "Islemler" tipinde bir nesne hazırladık.

```
class Program
{
 static void Main(string[] args)
  {
```

```
        Console.WriteLine(Islemler.Sehir);
        Console.WriteLine(Islemler.Yil);
        Islemler n = new Islemler();
    }
}
```

Burada örneğini aldığımız "Islemler" sınıfı sadece 2 statik alana sahip olduğu için hazırladığımız nesneyi herhangi bir amaçla kullanmak mümkün değildir. Çünkü bu nesne kullanılarak statik alanlara erişim sağlanamaz. Bu son cümlemiz konusunda ikna olmak istiyorsanız Visual Studio'dan yararlanabilirsiniz. Hazırladığınız nesnenin adını yazıp nokta karakterini eklerseniz Visual Studio, nesne üzerinden erişilebilinen üyeleri listelemektedir.



Listelenen bu 4 üye de Object sınıfından miras yoluyla alınanlardır. Bu durumu şöyle yorumlamak mümkündür: Statik metot, alan ve özellik içeren sınıfların örnekleri alınabilmektedir. Ancak bu metotların ve alanların sınıftan yol çıkılarak hazırlanan nesne üzerinden kullanılması mümkün değildir.

Yukarıda üzerinde çalıştığımız statik olmayan sınıfımızda sadece statik alanlara yer verdik. Şimdi, "Islemler" adını verdiğimiz sınıfta bir de statik metoda yer yereceğiz. Statik metotların nesne üzerinden işletilemediğini, yani statik metotların sınıfa özel olduğunu biliyorsunuz.

```
public class Islemler
{
 public static string Sehir ="Adana";
 public static int Yil = 2017;
 public static int Topla(int i, int j)
 {
   return i + j;
 }
}
```

Bu sınıfta "Islemler" adlı sınıfta Topla() adını verdiğimiz **static** bir metot var. Bu metot statik dolduğundan kullanmak için nesne hazırlamaya gerek yoktur. Bu sınıftan yola çıkılıp nesne hazırlanabilir ama statik metotlar aşağıdaki gibi işletilemezler.

```
class Program
{
 public static void Main(string[] args)
 {
  Islemler n = new Islemler();
  int sonuc = n.Topla(10, 20); //hata meydana gelir.
 }
}
```

Statik metotlar üyesi oldukları sınıfın statik üyelerine erişim sağlayabiliyorlar. Bunu size göstermek için üzerinde çalıştığımız uygulamadaki ikinci sınıfı aşağıdaki gibi düzenledik. "Islemler" adlı sınıfımızda statik bir alan ve bir metot bulunmaktadır. Dolayısıyla statik metot dahilinde statik alana erişim sağlanabiliyor.

```
public class Islemler
{
 public static string Sehir ="Adana";
 public static void Islet()
  {
   Console.WriteLine(Sehir);
  }
}
```

Burada "Sehir" adını verdiğimiz alan statik olmasaydı bu statik metot içinden bu alana erişmek mümkün olmazdı. Çünkü bu durumda alanın hangi nesneye ait olduğu bilinemezdi. Söz konusu sınıftan yola çıkılarak yüzlerce nesne hazırlanmış olabilir.

Statik metot dahilinde hangi nesnenin alanına erişim sağlanmak istendiği biliniyorsa derleyicinin önümüze koyduğu bu engel aşılabilir. En ideali statik metodun ilgili nesne tipinde bir parametreye sahip olmasını sağlayarak bu sorunu çözmektir. Başka bir deyişle uygulama aşağıdaki gibi düzenlenirse burada "Islemler" sınıfının statik olmayan alanına "Islet" adını verdiğimiz statik metodu dahilinde erişim sağlanabilir. Tabi nesne hazırlamayı burada Islet() adını verdiğimiz metodun içinde yapmak mümkündür.

```
class Program
{
 public static void Main(string[] args)
  {
   Islemler n = new Islemler();
   Islemler.Islet(n);
  }
}
public class Islemler
{
 public string Sehir ="Adana";
 public static void Islet(Islemler par1)
  {
   Console.WriteLine(par1.Sehir);
  }
}
```

Burada statik metot dahilinde sınıfın normal alanlarına erişemediğimiz için hile yapıp erişim sağladık. Benzer engel, metotlar için de vardır. Yani normal sınıfın statik metodu dahilinde

normal metodu işletmek mümkün değildir. Son cümlede anlatılmak isteneni aşağıda verilen sınıfta görebilirsiniz.

```
public class Islemler
{
 public string Sehir ="Adana";
 public static void Islet()
  {
  }
  public void Mesaj()
  {
    Console.WriteLine("Statik olmayan metot işletildi");
  }
}
```

Sınıfımızda birisi statik 2 metot bulunmaktadır. Statik metodun içinden statik olmayan metodu işletmek istersek derleyici buna izin vermez. Çünkü statik olmayan metodu nesne üzerinde işletebiliyoruz. Statik olmayan metot mutlaka işletilmek isteniyorsa statik metodun içinde nesne hazırlanabilir veya aşağıda yapıldığı statik metoda bir nesne parametre olarak gönderilebilir.

```
class Program
{
 public static void Main(string[] args)
 {
  Islemler n = new Islemler();
  Islemler.Islet(n);
 }
}
public class Islemler
{
 public string Sehir ="Adana";
 public static void Islet(Islemler par1)
 {
  par1.Mesaj();
 }
 public void Mesaj()
 {
  Console.WriteLine("Statik olmayan metot işletildi");
 }
}
```

Şimdi sınıfın statik metotları dahilinde statik olmayan üyelerine erişilemiyor olması ve yukarıdaki yapıldığı gibi hile yapılarak erişilmesi üzerinde biraz duracağız. Sınıfımızda birisi statik olmak üzere 2 metot ve statik olmayan bir alan var. Yukarıdaki sayfalarda ifade edildiği gibi herhangi bir nesneyle bağlantısı olmayan işlemler için statik alan ve metotlara başvurulması öneriliyor. Burada biz ne mi yaptık? Statik olmayan metodumuzu statik metot dahilinde çağırmak için statik metoda nesne örneği gönderdik. Bu önerilen bir yöntem değildir.



Yukarıda işaret edildiği gibi statik sınıf desteği C# 2.0 ile geldi. C# 1.0 sürümüyle kod yazıp bir sınıfı statik alan ve metotlarla dolduranlar doğal olarak bu sınıf tipinde nesneye ihtiyaç duymazlar. Sadece statik üyelerden meydana gelen sınıfların örneğinin

	alınmasını engellemek isteyen programcılar varsayılan yapıcı metodu (default constructor) **private** yapıyorlardı. Sonra statik sınıf desteği gelince bu yönteme başvurma ihtiyacı ortadan kalktı.

Şimdiye kadar statik alan ve metotlardan söz ettik. Şimdi sırada statik sınıflar var. Hazırlamak istediğiniz sınıfın alanlarının, metotlarının, özelliklerinin ve olaylarının static olmasını istiyorsanız sınıfın **static** olmasını sağlayabilirsiniz. Bu sayede sınıfınızın başka amaçlarla (örneğini almak, base class olarak kullanmak) kullanılmasının önüne geçmiş olursunuz.

Uygulamanız içerisinde sıklıkla başvurulacağını öngördüğünüz alan ve metotları statik bir sınıfta toplamanız iyi bir programcılık alışkanlığıdır. Bu alışkanlık, statik olmayan sınıflarda hazırlanacak statik alan ve metotların sayısının azalmasına neden olacaktır. Statik sınıflar hakkında konuşmak amacıyla aşağıda verilen kodu hazırladık.

```
class Program
{
 public static void Main(string[] args)
 {
  Islemler n = new Islemler();  // hata meydana gelir.
 }
}
public static class Islemler
{
 public static string Sehir = "Adana";
 public static void Islet()
 {
  Console.WriteLine("Statik metot işletildi");
 }
}
```

Açık bir şekilde yazmak gerekirse, statik sınıfların nesneye yönelik programcılığın en önemli ilkesi olan kalıtımla ilgileri yoktur. Bundan şu sonucu çıkarabiliriz: Standart ve başka bir işlemle direk ilgisi olmayan işlemler için statik class'lar hazırlanır. Diğer class'lar ise bu statik class'lardan gerektikçe yararlanırlar. Tıpkı bizim **Console** sınıfından yararlandığımız gibi.

"Islemler" adını verdiğimiz sınıf **static** olduğu için **new** anahtar kelimesi ile örneği hazırlanamaz. Benzer şekilde static sınıflarda static olmayan alanların ve metotların olmasına izin verilmez. Bu nedenle bu static sınıf aşağıdaki gibi düzenlendiğinde hata meydana gelir.

```
public static class Islemler
{
 public float kdv = 0.18f; // Hata meydana gelir
 public static string sehir = "Adana";
 public static void islet()
 {
  Console.WriteLine("Statik metot işletildi");
 }
}
```

new anahtar kelimesi ile static sınıfların örnekleri alınıp nesne hazırlanamadığı için yapıcı metot içeremezler. Bu nedenle aşağıda verilen kod hataya neden olur.

```
public static class Islemler
{
 public static string sehir = "Adana";
 public static void islet()
 {
  Console.WriteLine("Statik metot işletildi");
 }
 public static Islemler() // hata meydana gelir.
  {
  }
}
```

Ayrıca statik sınıflar başka sınıfların temel sınıfı veya başka bir sınıfın mirasçısı olamazlar. Başka bir deyişle statik sınıflardan başka sınıflar türetilemez. Bu durumda sadece static üyeler içermesini istediğiniz, örneklenmesini ve türetilmesini istemediğimiz sınıfları static anahtar kelimesiyle statik yapmalısınız.

Şimdiye kadar statik sınıf, alan ve metotlarla ilgili olarak anlatılanları özetlemek gerekirse..

- Statik sınıfları, nesnelerle ilişkisi olmayan, nesne bazlı işlemlerinin yapılmadığı durumlarda tercih edilmesi önerilmektedir.
- Statik sınıfların bütün üyeleri statik olmak zorundadır. C# derleyicisi statik sınıfta statik olmayan metot veya alanların olmasına izin vermez.

Using Static Ayarlaması

Şimdiye kadar herhangi bir statik bir sınıfın metotlarını işletirken her seferinde metot adından önce ilgili sınıfın adını yazdık. C# programlama dilinin 6.0 sürümüyle birlikte using bloğunda statik sınıfların adlarını yazabiliyoruz. Bu kitapta metotları en çok kullanılan statik sınıf **Console** olduğu için using static ayarlamasını Console sınıfı üzerinden yapacağız.

Console sınıfı System'de yer aldığı için ReadLine() ve WriteLine() gibi metotları kullanırken sınıf adından önce namespace adını yazmamak için hazırladığımız her kod dosyasının using bloğunda şu satıra yer verdiğimizi biliyorsunuz:

```
using System;
```

Diğer yandan Console sınıfı statik bir sınıf olduğu için örneği alınamıyor, dolayısıyla Console sınıfının bütün metotları statiktir. Bu metotları kullanmak için şimdiye kadar metot adından önce hep sınıf adını yani Console yazdık.

```
using System;
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
   {
    Console.WriteLine("Adana Demirspor");
   }
 }
}
```

Mevcut bir statik sınıfın metotlarını işletirken sınıf adını yazmak istemiyorsanız, aşağıda yapıldığı gibi using bloğunda "using static" ayarlaması yapabilirsiniz.

```
using System;
using static System.Console;
namespace ConsoleApp3
{
 class Program
 {
  static void Main(string[] args)
  {
   WriteLine("Adana Demirspor");
  }
 }
}
```

Statik sınıflar için using tanımlaması yaparken metot adı çakışmalarına dikkat etmek gerekir. Bu konuya dikkatinizi çekmek için System.IO'daki **File** ve **Directory** sınıflarının metotlarını kullanmayı deneyeceğiz. Bu amaçla hazırladığımız kod aşağıdadır.

```
using System;
using System.IO;
namespace ConsoleApp3
{
 class Program
 {
  static void Main(string[] args)
  {
    string dosya1 = "C:\\Dosyalar1";
    string dosya2 = "C:\\Dosyalar\\remove.png";
    Directory.Delete(dosya1);
    File.Delete(dosya2);
  }
 }
}
```

Her ikisi de statik olan bu sınıfların statik metotları kullanılarak dosya ve klasörler üzerinde birçok işlemi yapmak mümkündür. Bu sınıfların Delete() metotları yukarıdaki gibi kullanılarak istenen dosya ve klasör kolayca silinebilir. Şimdi bu kodda değişiklik yapıp bu statik sınıflar için "using static" ayarlaması yapacağız. Bu sayede metot adından önce sınıf adını yazmak zorunda kalmayacağız.

```
using System;
using static System.IO.File;
using static System.IO.Directory;
namespace ConsoleApp3
{
 class Program
 {
  static void Main(string[] args)
  {
   string dosya1 = "C:\\Dosyalar1";
```

```
        string dosya2 = "C:\\Dosyalar\\remove.png";
        Delete(dosya1);
        Delete(dosya2);
      }
     }
    }
```

Ancak bu kod hataya neden olur. Çünkü hem Directory hem de File sınıfının Delete() adında bir metodu vardır. Dolayısıyla using static ayarlaması bu şekilde kullanılamaz. Bu sınıflar için using bloğunda using static ayarlaması yapılmış olunsa bile metot adından önce sınıf adını yazmak gerekir.

```
using System;
using static System.IO.File;
using static System.IO.Directory;
namespace ConsoleApp3
{
 class Program
 {
  static void Main(string[] args)
   {
    string dosya1 = "C:\\Dosyalar1";
    string dosya2 = "C:\\Dosyalar\\remove.png";
    Directory.Delete(dosya1);
    File.Delete(dosya2);
   }
  }
}
```

Ad Gizleme – Name Hidding

Bazen base class'ta bulunan bir metodun, alanın veya özelliğin "adı" derived sınıfta da farkında olarak veya olmayarak kullanılmış olabilir. Bu durumda base class'taki ilgili metot veya alandan yeni hazırlanan class'ta yararlanılamaz. Bu konu üzerinde konuşmak için aşağıda verilen kodu hazırladık. Bu kodda hem base class pozisyonundaki "Baskan" hem de "Oyuncu" sınıfı "Ad" ve "Soyad" alanlarına sahiptir.

```
class Baskan
{
 public string Ad = "Aziz";
 public string Soyad = "Yıldırım";
}
class Oyuncu : Baskan
{
 public string Ad;
 public string Soyad;
 public Oyuncu(string Str1, string Str2)
 {
  this.Ad = Str1;
  this.Soyad = Str2;
 }
}
```

```
class Program
{
 public static void Main()
 {
  Oyuncu Nesne = new Oyuncu("Burak", "Yılmaz");
  Console.WriteLine(Nesne.Ad);
  Console.WriteLine(Nesne.Soyad);
 }
}
```

Bu uygulama bu hali ile derlenip işletilirse beklendiği gibi ekrana "Burak Yılmaz" yazılır. Yani derleyici önceliği derived class'taki alanlara verir. Ancak derleme sırasında 2 uyarı mesajı alınır. Çünkü derleyici programcının her iki sınıfta da aynı alan adlarını farkında olmadan kullanmış olabileceğini varsayar. Yani derleyici diyor ki, hem "Baskan" sınıfını base olarak kullanıyorsun hem de derived class'ta, base class'ta bulunan alana yokmuş gibi davranıyorsun.



Tam bu noktada ana metnin dışına çıkıp ekleme yapacağız; bunu yaparken kendimizi derleyicinin yerine koyacağız. Burada isim benzerliği olan bir alandır. Benzer isim benzerliği metotlarda da olabilir.

* Ey programcı bu isim benzerliğini bilmeden yaptıysan ya base class'taki ya da derived class'taki alana farklı bir ad verebilirsin.

* Base class'taki alan adını derived class'ta bilerek kullandıysan ileride bu kodu inceleyecek olanları ikilemde bırakmamak için "new" anahtar kelimesini kullanarak bu seçimin hakkında not bırakmalısın.

* Tabi bu durumda base class'taki alanın derived class bağlamında gölgede kalacağını, başka bir deyişle yok hükmünde olacağını unutmamalısın.

* Sana base class'taki alanı silmeni öneremiyorum. Çünkü bu class'ı belki sen yazmadın belki de başka bir derived class'larda böyle bir isim benzerliği sorunu yoktur.

Derleme sırasında uyarı mesajı almak istemiyorsanız ve programı okunur yapmak istiyorsanız türeyen sınıfta **new** anahtar kelimesine yer verip söz konusu alanları gölgelemeniz gerekir. Bazen isim gizleme olarak anılan bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
class Oyuncu : Baskan
{
 public new string Ad;
 public new string Soyad;
 public Oyuncu(string Str1, string Str2)
 {
  this.Ad = Str1;
  this.Soyad = Str2;
 }
}
```

New anahtar kelimesini kullanarak base class'tan gelen Ad ve Soyad alanlarını geri plana aldık. Bu şartlarda base sınıftaki "Ad" ve "Soyad" alanlarının içeriklerine gerek duyuyorsanız **base** anahtar kelimesinde yararlanabilirsiniz. base anahtar kelimesi aşağıda verildiği gibi kullanıldığında "Baskan" sınıfının elemanlarının içerikleri ekrana yazılır.

```
class Oyuncu : Baskan
{
 public new string Ad;
 public new string Soyad;
 public Oyuncu(string Str1, string Str2)
  {
   this.Ad = Str1;
   this.Soyad = Str2;
   Console.WriteLine(base.Ad);
   Console.WriteLine(base.Soyad);
  }
}
```

Burada dikkatlerden kaçmasını istemediğimiz ince bir nokta var: Base anahtar kelimesini ancak derived class içinde kullanabiliriz. Yoksa derived class tipinde nesnenin hazırlandığı ve Main() metoduna sahip diğer sınıfta base anahtar kelimesini bu şekilde kullanamayız.

Programcı burada inat edip "new" anahtar kelimesini kullanmayacağım derse ne mi olur? Maddeler halinde sıralayalım:

- Derleyiciden uyarı mesajı alınır.
- Yazılan kod okunurluk bağlamında eksi puan alır. Kodu ileri bir tarihte inceleyen birisi "derived class'taki ilgili alana verilecek isimlere kıran mı girmiş" diyebilir.
- Base class'taki alana erişebilmek için "base" anahtar kelimesini kullanmak bir zorunluluk olur.

Her ne kadar gerçek hayatta böyle isim benzerliği karşısında programcılar inat etmeyip derived class'taki ilgili alana farklı isim verme yoluna gidiyor olsalar bile şimdi benzer örneği metotlar için yapacağız. Yani hem base hem de derived class'ta aynı ada sahip birer metot hazırlayacağız. Bu amaçla **Renkler**() adında birer metoda sahip 2 sınıf hazırladık. Devamında Main() metodunun olduğu diğer sınıfta "Besiktas" adını verdiğimiz sınıf tipinde nesne hazırlayıp bu sınıfın Renkler() metodunu işlettik.

```
class Takim
{
 public void Renkler()
  {
   Console.WriteLine("Sarı Lacivert");
  }
}
class Besiktas : Takim
{
 public void Renkler()
 {
   Console.WriteLine("Siyah Beyaz");
 }
}
class Program
{
 public static void Main()
 {
  Besiktas Takim = new Besiktas();
  Takim.Renkler();
 }
}
```

Aynı ada sahip 2 metodun olduğu bu uygulama derlendiğinde derleyici aynı ada sahip 2 metodun olmasını programcının dalgınlığına yorar ve uyarı mesajları verir. Aynı metot adını bilerek hem base hem de derived class'ta kullandıysanız tıpkı yukarıda alanlar için yaptığımız gibi "new" anahtar kelimesinden yararlanabiliniz.

```
class Takim
{
 public void Renkler()
 {
   Console.WriteLine("Sarı Lacivert");
 }
}
class Besiktas : Takim
{
 public new void Renkler()
 {
  Console.WriteLine("Siyah Beyaz");
 }
}
```

Tekrar etmek gerekirse, derived class'ta metot imzasında new anahtar kelimesine yer vermekle programcı derleyiciye demiş oluyor ki, metoda ad verirken aynı ada sahip bir metodun base class'ta olduğunu biliyorum. Tabi bu durumda burada "Besiktas" adını verdiğimiz sınıfın örneği alınıp Renkler() metodu işletilmek istenirse base class'taki aynı adlı metot gölgede kalmış olur ve derived class'taki metot işletilir.



Bu şartlarda programcı base class'taki metoda gerek duyarsa işletmek için **base** anahtar kelimesinden yararlanabilir. Örneğin base class'taki Renkler() adını verdiğimiz metot aşağıdaki gibi işletilebilir.

```
class Besiktas : Takim
{
 public new void Renkler()
 {
   base.Renkler();
   Console.WriteLine("Siyah Beyaz");
 }
}
```

Base class'taki metodu bu şekilde işletmenin anlamlı olmadığını unutmamak gerekir. Derived class'taki Renkler() metodu bu şekilde değiştirilip sonra Main() metodunun olduğu diğer sınıfta "Besiktas" adını verdiğimiz sınıfın örneği alınıp işletilirse aşağıdaki gibi bir sonuç alınır.



Programcı, base class'taki metodun adıyla bir sorunum yok, ancak mevcut hali işimi tam olarak görmüyor derse, kendisine ne önereceğiz? Yani sorunun kökeni, base class'taki metoda derived class'ta yanlış isim verilmiş olmasından kaynaklanmıyorsa programcıya çözüm önermeden önce şöyle bir soru yöneltmek gerek: Base class'taki metoda ihtiyacın olacak mı olmayacak mı?

Programcı bu soruya, "base class'taki aynı isimli metodu bu class bağlamında hiç kullanmayacağım" diye cevap verirse ona çözüm olarak diyeceğimiz şu olabilir: Derived class'ta kendi yazdığın metodu **new** diye işaretle ve base class'taki aynı isimli metodu unut gitsin.

Programcı sorduğumuz soruya, base class'taki metottan tümden vazgeçmek istemiyorum diye cevap verip ve base anahtar kelimesini kullanmaya karşı çıkarsa ona vereceğimiz cevapta nesneye yönelik programcılıkla ilgili başka kavramlar olabilir. Bu nedenle isim gizleme konusunu burada bitiriyoruz.

Metot Override İşlemleri

Daha önceki konulardan bildiğiniz gibi C# programlama dilinde bütün sınıflar geri planda .NET Framework ile gelen **Object** sınıfından türetilmektedir. Bu sınıf hakkında ayrıntılı bilgiyi ilgili bölümde bulabilirsiniz. Konu ezbere olmasın diye bu sınıfın orijinal halinin bir kısmını aşağıda verdik. Çünkü override işlemini bu sınıf üzerinden anlatmayı deneyeceğiz.

```
namespace System
{
public class Object
{
  protected virtual void Finalize()
    {
    }
  public virtual int GetHashCode()
    {
    }
  public virtual bool Equals(System.Object obj)
    {
    }
  public virtual string ToString()
    {
    }
  public static bool Equals(System.Object objA, System.Object objB)
    {
    }
  public static bool ReferenceEquals(System.Object objA,System.Object objB)
    {
    }
  public Type GetType()
    {
    }
```

```
        protected System.Object MemberwiseClone()
          {
          }
        public Object()
          {
          }
      }
  }
```



Nesneye yönelik programcılıkla ilgili olarak şimdiye kadar öğrendiklerinizi gözden geçirmek ve bazı konuların yerli yerine oturmasını istiyorsanız, bu sınıfı neden ikinci kez anlatmayı denediğimizi ve satır satır incelediğimize lütfen itiraz etmeyin. Daha önce herhangi bir kaynaktan C# programlama dilinin kalıtımla ilgili özelliklerini öğrenmeye çalışırken neden zorlandığınızı anlamak için bu başlık altında yazılanları lütfen sonuna kadar okuyup öyle değerlendirme yapın.

Yukarıda bu sınıfın orijinal yapısının tümü yerine bir kısmını buraya aldığımızı işaret etmiştik. Sınıfın buraya almadığımız üyeleri **private** olanlardır. Private üyelere kendi yazdığımız kod içinden erişemediğimiz için ilgilenmedik. Object sınıfın yapısına yakından baktığımızda metotların çoğunluğu için erişim belirleyicisi olarak public kullanılmış iken iki tanesi için **protected** olarak seçilmiş. Bu sınıfa ait metotların public veya protected seçilmiş olması bizim bu sınıfla burada ilgilenme nedenimiz üzerinde bir etkisi olmadığından metotlar için seçilen erişim belirleyici anahtar kelimeleri görmezden geliyoruz.

Bu sınıftaki metotların 2 tanesi **static** olarak ayarlanmış. Bundan, **ReferenceEquals** ve **Equals** metotlarını istediğimiz gibi kullanabileceğimizi anlayabiliriz. Daha önceki konulardan hatırlayacağınız gibi programcı için statik metot, işlevini öğrenip gerektiğinde kullanacağı metotlardır. Dolayısıyla Object sınıfının statik metotlarıyla ilgilenmiyoruz.

Bu şekilde elediğimiz (statik ve private) metotların dışında kalan metotların hepsi (yapıcı metot hariç) **virtual** olarak ayarlanmıştır. Derleyici virtual metotlarla ilgili olarak bize şunu söylemektedir:

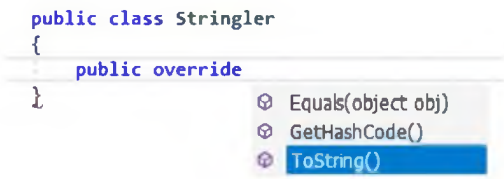
- Ey programcı bu virtual olarak işaret edilen metotların adının önünde **virtual** anahtar kelimesi kullanılmış olsa bile burada sanal yani virtual olan bir şey yoktur.
- Zamanında C# programlama dilini geliştiren Microsoft uzmanları Visual Basic programlama dilini geliştirenleri örnek almak yerine nedense C# dilini öğrenmek isteyenlerin kafalarını karıştırmayı seçmişlerdir. Visual Basic'te olduğu gibi Virtual anahtar kelimesi yerine **Overridable** anahtar kelimesini kullansaydılar bu konu son derece kolay anlaşılırdı, kimsenin kafası karışmazdı.
- Ey programcı yukarıda ifade edilen nedenlerden dolayı ne zaman C# kodu içinde bir metodun imzasında virtual anahtar kelimesini görürsen sen bunun Overridable yani ezilebilir olduğunu düşün.

Bu bilgilerin ışığında şu söylenebilir: Programcı, Object sınıfının ToString(), Finalize(), Equals() ve GetHashCode() metotlarını istediği zaman istediği sınıf içinde yeniden yazıp .NET Framework ile gelen halini ezebilir. Çünkü saydığımız bu metotların imzalarında overridable anlamında **virtual** anahtar kelimesine yer verilmiştir.

Bu konunun geri planını gözler önüne sermek ve kafanızdaki soru işaretlerini dağıtmak için Visual Studio ile hazırladığımız konsol uygulamasında aşağıdaki gibi herhangi bir metodu olmayan kısıtlı bir sınıf hazırladık.

```
public class Stringler
{
}
```

Bu sınıfı hazırlarken herhangi bir sınıfı base class olarak seçmediğimiz için Object sınıfının mirasçısı olur. Dolayısıyla Object sınıfı kaynaklı olup yukarıda sözünü ettiğimiz metotları yeniden yazabiliriz. Sınıfın içinde iken override anahtar kelimesini yazarsanız, Visual Studio yeniden yazıp ezebileceğiniz metotları aşağıdaki gibi listeler.



Bu sırada listelenen metotlardan hangisi override edilmek isteniyorsa o metot tıklanırsa Visual Studio standart bir metot kalıbı hazırlar. Bu listeden ToString() metodunu tıklayınca Visual Studio sınıfımızda bu metodu aşağıdaki gibi override etti.

```
public class Stringler
{
 public override string ToString()
  {
    return base.ToString();
  }
}
```

Şimdi bu class'ta kendimiz bir metot yazıp virtual olarak işaret edeceğiz. Başka bir deyişle bu sınıfta hazırladığımız metodun yeniden yazılması gerektiğini işaret edeceğiz.

```
public class Stringler
{
 public virtual string Buyut(string str)
  {
    return str.ToUpper();
  }
}
```

Bu şekilde **virtual** olarak işaretlenmiş bir metoda sahip sınıfı hazırladıktan sonra başka bir sınıf hazırladık. Burada "Stringler" adını verdiğimiz sınıfı ise base class olarak kullandık.

```
public class Islemler : Stringler
 {
 }
```

Base class'ta virtual olarak işaret ettiğiniz metodu derived class'ta override etmezseniz bunu derleyici geri planda aşağıda yapıldığı gibi yapmaktadır. Başka bir deyişle, virtual metotları override anahtar kelimesini kullanarak ezmezseniz derleyici hata rapor etmez.

```
public class Stringler
{
  public virtual string Buyut(string str)
  {
    return str.ToUpper();
  }
}
public class Islemler : Stringler
{
  public override string Buyut(string str)
  {
    return base.Buyut(str);
  }
}
```

Burada geri planda yapılan şudur: Kod içinde Buyut() adını verdiğimiz ve "Stringler" sınıfı kaynaklı virtual metot, buradaki adıyla "Islemler" sınıfında "base" anahtar kelimesinden yararlanılıp işletilmektedir.

Bize sorarsanız kodu okunur yapmak için base class'taki virtual olarak işaret edilmiş metotları bu şekilde düzenlemeniz iyi bir alışkanlık olacaktır. Konu üzerinde konuşmaya devam etmek amacıyla bu kez aşağıdaki gibi 2 class hazırladık.

```
class Takim
{
 public virtual void Renkler()
  {
   Console.WriteLine("Sarı lacivert");
  }
}
class Besiktas : Takim
{
 public override void Renkler()
 {
   Console.WriteLine("Siyah beyaz");
  }
}
```

Burada verilen 2 class'tan base class durumunda olanında **virtual** bir metot var. Derived olarak düzenlediğimiz diğerinde ise virtual metot **override** anahtar kelimesi kullanılarak ezilmektedir. Virtual ve override anahtar kelimelerinde neden gerek duyulur ve hangi şartlarda kullanılır diye soracak olursanız şöyle bir açıklama yapmak anlamlı olabilir:

- Base Class'ı hazırlayan programcı yazdığı bir metodun derived class'larda ihtiyaçları karşılamayacağını öngörüyorsa bu metodun imzasında virtual anahtar kelimesine yer verir. Tıpkı Object sınıfının ToString() metodu gibi.

- Başka programcılar tarafından yazılan bir sınıfı base class olarak kullanan programcı kendi sınıfının içinde iken yeni bir metot hazırlamak üzere override yazıp boşluk tuşuna bastığında Visual Studio base class'taki virtual metotları listeler. Bu sayede programcı override edebileceği metotları görmüş olur.

- Base Class'ı yazan programcı class dahilinde hazırladığı metodun derived class'ta yeterli olacağını düşünüp metodun imzasında virtual anahtar kelimesine yer vermemiş olabilir. Bu durumda base class'ı kullanan programcı aynı fikirde değilse metodun imzasında **new** anahtar kelimesine yer verip aynı ada sahip yeni bir metot hazırlayabilir.

Base class'taki metotlardan birisi override edilmek istendiğinde derleyici söz konusu sınıfta override edilmek istenen metodu aramaktadır. Metot yoksa yani aynı imzaya sahip böyle bir metot yoksa derleyici hata mesajı vermektedir. Bu nedenle aşağıda verilen kod derlenemez.

```
class Fener
{
 public virtual void Renkler(string str)
  {
   Console.WriteLine(str);
  }
}
class Besiktas : Fener
{
 public override void Renkler(string str1, string str2)
 {
  Console.WriteLine(str1);
  Console.WriteLine(str2);
 }
}
class Program
{
 public static void Main()
 {
  Besiktas Takim = new Besiktas();
  Takim.Renkler("Siyah", "Beyaz");
 }
}
```

Çünkü burada mevcut "Fener" sınıfında string tipte 2 parametreye sahip Renkler() adında bir metot yoktur. "Fener" sınıfındaki Renkler() metodu 1 parametreye sahiptir. Bu hatayı gidermek için ya derived class durumundaki sınıftaki Renkler() metodu base class'taki aynı metoda uygun imzaya sahip olması sağlanmalı ya da **override** anahtar kelimesi burada Renkler adını verdiğimiz metodun imzasında kullanılmamalıdır.

Nesneye Yönelik Programcılıkta Polimorfizm

Bu bölümde şimdiye kadar Nesneye Yönelik Programcılığın ilkelerinden kalıtım ve kapsüllemeden söz edildi. Şimdi sırada çok biçimlilik var. Çok biçimlilik yani polimorfizm, tek cümle ile anlatmak gerekirse, bir nesnenin kullanım şekline bağlı olarak farklı bir nesne gibi davranabilmesidir.

Bu kısımda C# programlama dilinin Nesneye Yönelik Programlama ilkelerinden birisi olan çok biçimliliği anlatırken henüz sözünü etmediğimiz herhangi bir anahtar kelimeyi kullanmayacağız. Çünkü yukarıdaki sayfalarda kalıtım, isim gizleme, metot ezme gibi temel işlemlerden söz ettik. Bu kısımda elimizdeki mevcut teknikleri kullanarak nesnelerin şartlara göre farklı davranmasını sağlayacağız. Yani nesnelere çok biçimlilik özelliği kazandıracağız. Bu konunun kafaları karıştırma potansiyeli yüksek olduğu için adım adım ilerleyeceğiz.

Bu konu üzerinde durmaya .NET Framework ile gelen generic destekli koleksiyon sınıflarından **Dictionary** ve **SortedDictionary** sınıflarından söz ederek başlayacağız. Bu sayede bu 2 koleksiyon sınıfını farklı bir gözle incelemiş olacağız.

Bu koleksiyon sınıfları elde anahtar-değer şeklinde düzenlenmiş bilgiler varken işlevsel olmaktadırlar. Sözlüklerde genelde bir kelimenin 2 farklı dildeki karşılıkları saklandığı için anahtar-değer şeklinde düzenlenen bilgiler üzerinde yapılacak işlemler için hazırlanan koleksiyonlara **Dictionary** adı verilmiştir. Dictionary'den farklı olarak SortedDictionary tipindeki nesnelerde tutulan bilgi çiftleri geri planda otomatik olarak sıralanmaktadır.

İşe Dictionary ve SortedIDictionary nesneleri hazırlamakla başlayacağız. Ancak bu nesnelerin referanslarını **IDictionary** tipindeki değişkenlere aktaracağız. Bu koleksiyonların her ikisi de bu interface'i uyguladığı için aşağıda verilen 2 satır hataya neden olmaz.

```
IDictionary<string, string> dic1 = new Dictionary<string, string>();
IDictionary<string, string> dic2 = new SortedDictionary<string,string>();
```

Şimdi bu 2 satır üzerinde biraz konuşalım: Önce Dictionary tipinde bir nesne hazırlayıp referansını **IDictionary** tipindeki değişkene aktardık. Bu şartlarda burada "dic1" adı verilen değişken sayesinde generic Dictionary nesnesinin IDictionary arayüzü kaynaklı metot ve özelliklerinden yararlanabiliriz. Bunu bir de şöyle anlatabiliriz: Belleğin heap denilen kısmında Dictionary tipinde bir nesnemiz var. Bu nesneye erişimde kendimize IDictionary tipindeki değişkeni rehber ettiğimiz için ancak bu rehberin sahip olduğu metot ve özellikleri görüp kullanabiliyoruz. Benzer şeyler yukarıda hazırladığımız generic SortedDictionary nesnesi için de söylenebilir.



Bu kitabı baştan sona okunduğunda, kitapta sözü edilen hiçbir konunun, hiçbir kavramın, hiçbir işlemin anlaşılmaz olarak kalmaması için aynı konu veya kavramı değişik şekillerde tekrar anlatıyoruz. Örneğin yukarıda Dictionary sınıfından söz etmekle, nesneye yönelik programcılıkta çok önemli olan sınıf-interface ilişkisini dikkatinize sunuyoruz.

Konuya devam etmek yani Dictionary<string, string> tipindeki nesneye IDictionary<string, string> aracılığıyla anahtar-değer şeklinde düzenlenmiş bilgileri aktarmak için IDictionary kaynaklı **Add**() metodunu aşağıdaki gibi kullandık. Her iki IDictionary değişkenine aynı bilgileri aktardık.

```
class Program
{
 public static void Main()
 {
 IDictionary<string, string> dic1 = new Dictionary<string, string>();
```

```
 IDictionary<string, string> dic2 =new SortedDictionary<string, string>();
  dic1.Add("Screen", "Ekran");
  dic1.Add("Printer", "Yazıcı");
  dic1.Add("Cpu", "işlemci");
  dic1.Add("Ram", "Bellek");

  dic2.Add("Screen", "Ekran");
  dic2.Add("Printer", "Yazıcı");
  dic2.Add("Cpu", "işlemci");
  dic2.Add("Ram", "Bellek");
 }
}
```

Bu kod için ilk bakışta şu söylenebilir: Madem hem "dic1" hem de "dic2" değişkenleri IDictionary tipindedir ve her ikisine de Add() metodu ile aynı bilgiler aktarıldı bu değişkenlerin birebir aynı olması gerekir. Ancak bu 2 değişken aynı bilgileri içeriyor (gerçekte içerme yerine 'refere ediyor' demek daha doğru olur) olsalar bile aynı değiller. Aşağıda verilen kod işletildiğinde listelenen bilgilerin sıralanışının farklı olduğu görülür.

```
class Program
{
 static void Main(string[] args)
 {
 IDictionary<string, string> dic1 = new Dictionary<string, string>();
 IDictionary<string,string> dic2 = new SortedDictionary<string, string>();
 dic1.Add("Screen", "Ekran");
 dic1.Add("Printer", "Yazıcı");
 dic1.Add("Cpu", "işlemci");
 dic1.Add("Ram", "Bellek");
 dic2.Add("Screen", "Ekran");
 dic2.Add("Printer", "Yazıcı");
 dic2.Add("Cpu", "işlemci");
 dic2.Add("Ram", "Bellek");
 ICollection<string> anahtar1 = dic1.Keys;
 ICollection<string> anahtar2 = dic2.Keys;
 foreach (string ing in anahtar1)
 {
   Console.WriteLine(ing);
  }
  Console.WriteLine();
  foreach (string tr in anahtar2)
  {
   Console.WriteLine(tr);
  }
 }
}
```

Başka bir ifade ile burada "dic1" ve "dic2" adını verdiğimiz değişkenler çok biçimlilik ilkesine örnektir. Polimorfizm kavramını üzerinde konuşmaya kendi hazırlayacağımız basit uygulama üzerinde devam edeceğiz.

Varsayalım ki bir otomobil bakım firması için uygulama geliştireceğiz. Tamir ve bakım için gelen araçlar için farklı işlemler yapılabildiği için kalıtımdan yararlanıp ortak özelliklere

sahip bir base class, sonra bu base class'tan yola çıkıp çok sayıda derived class hazırlamak durumunda kalabiliriz. İlk sınıfı aşağıdaki gibi düzenledik.

```
public class Arac
{
 public void Hakkinda()
  {
    Console.WriteLine("Araç Sınıfı");
  }
}
```

Nesneye yönelik programcılığın polymorphism ilkesinden söz edebilmek amacıyla hazırladığımız bu class'ta sadece konsola bilgi yazan bir metot vardır. Çünkü buradaki amacımız derived class'larda hangi şartlarda hangi metodun işletileceğini araştırmaktır. Devamında bu sınıfın mirasçısı 2 ayrı sınıf hazırladık. "Arac" sınıfından miras yoluyla gelen "Hakkinda()" adlı metodu bu 2 sınıfta da yeniden yazdığımıza dikkatinizi çekmek istiyoruz.

```
public class Binek : Arac
{
 public void Hakkinda()
  {
   Console.WriteLine("Binek Araç Sınıfı");
  }
}
public class Ticari : Arac
{
 public void Hakkinda()
  {
   Console.WriteLine("Ticari Araç Sınıfı");
  }
}
```

Bu şekilde "Arac" sınıfının mirasçısı 2 sınıf hazırladıktan sonra Main() metoduna sahip başka bir sınıf hazırladık. Bu sayede yukarıda hazırladığımız sınıfları test edebileceğiz. Bu sınıfımız, base class durumundaki "Arac" sınıfı tipinde bir parametreye sahip "BilgiVer()" adında static bir metot bulunmaktadır.

```
class Program
{
 public static void BilgiVer(Arac nesne)
 {
  nesne.Hakkinda();
 }
 public static void Main()
 {
  Arac a = new Arac();
  Binek b = new Binek();
  Ticari t = new Ticari();
  BilgiVer(a);
 }
}
```

Buraya kadar sizin için yeni bir şey yoktur. Asıl anlatmak istediğimiz **BilgiVer()** adını verdiğimiz metot "Arac" tipinde bir parametreye sahip olmasına rağmen bu metoda "Arac"

sınıfından türeyen sınıflardan yola çıkılarak hazırlanan nesneleri parametre olarak vermenin mümkün olduğudur. Başka bir deyişle bu metoda "Binek" veya "Ticari" sınıfından yola çıkılarak hazırlanan nesneleri aşağıda yapıldığı gibi parametre olarak vermek mümkündür.

```
class Program
{
 public static void BilgiVer(Arac nesne)
 {
  nesne.Hakkinda();
 }
 public static void Main()
 {
  Arac a = new Arac();
  Binek b = new Binek();
  Ticari t = new Ticari();
  BilgiVer(a);
  BilgiVer(b); // burada upcasting (yukarı çevrim) işlemi var.
  BilgiVer(t); // burada upcasting işlemi var.
 }
}
```

Bu kodda Main() metoduna sonradan eklediğimiz 2 satırda kullandığımız metot "Arac" tipinde nesne beklerken "Binek" ve "Ticari" tipindeki nesneleri gönderdik. Nesneye yönelik programcılığın imkanlarından polimorfizmin burada devreye girmesini ve BilgiVer() adını verdiğimiz metoda hangi tipte nesne parametre olarak verilirse o nesneyle ilgili "Hakkinda" metodunun işletilmesini bekliyoruz. Ancak bu kodu derleyip çalıştırdığımızda BilgiVer() metodunu her işlettiğimizde base class'ın (burada Arac) Hakkında() metodu işletilir ve aşağıdaki gibi bir sonuç alınır.



Beklediğimiz sonucu alamama nedenimiz şudur: Yukarıda hazırlayıp base class olarak kullandığımız "Arac" adlı sınıfta Hakkinda() adını verdiğimiz parametresiz ve void bir metot bulunuyor. "Arac" sınıfının mirasçısı "Binek" ve "Ticari" sınıflarımızda da aynı imzaya sahip Hakkinda() metodu bulunuyordu. Dolayısıyla derleyiciden uyarı mesajı almıştık. Bu uyarıya aldırış etmeden yolumuza devam etmiştik. Yukarıda işaret ettiğimiz Upcasting işleminin yapılabilinmesi için base class'taki metodun imzasında **virtual**, derived class'lardaki aynı metodun imzasında **override** anahtar kelimesine yer verilmesi gerekir. Bu nedenle kodumuzu aşağıdaki gibi düzenledik.

```
public class Arac
{
 public virtual void Hakkinda()
  {
   Console.WriteLine("Araç Sınıfı");
  }
}
```

```
public class Binek : Arac
{
 public override void Hakkinda()
 {
  Console.WriteLine("Binek Araç Sınıfı");
 }
}
public class Ticari : Arac
{
 public override void Hakkinda()
 {
   Console.WriteLine("Ticari Araç Sınıfı");
 }
}
```

Bu değişiklikten sonra kodumuz beklediğimiz sonucu verir. Konu üzerinde düşünmenizi sağlamak için "Binek" adını verdiğimiz sınıfta, "Arac" sınıfından miras yoluyla gelen Hakkinda() metodunu yeniden yazmaktan vazgeçtik. Başka bir deyişle "Binek" sınıfını aşağıdaki gibi düzenledik.

```
public class Binek : Arac
{
}
```

Bu şartlarda yukarıda verdiğimiz uygulamayı derleyip çalıştırırsak "Binek" sınıfının Hakkinda() metodu yeniden yazılmadığı için "Arac" sınıfının Hakkında() metodu işletilir ve aşağıdaki gibi bir sonuç alınır. Başka bir deyişle bu sırada **upcasting** işlemi olmaz.



Burada gerçekleşen olayla ilgili şunlar söylenebilir: Hangi nesnenin ilgili metodunun çağırılıp işletileceği **çalışma anında** belirleniyor. Çağrılacak metodun çalışma anında belirlenmesine **geç bağlama** adı verilmektedir. Çağrılıp işletilecek metodun derleme anında belirlenmesine **erken bağlama** adı verilmektedir.

Yapıcı Metotlar

Hazırladığınız sınıfın bütün üyeleri statik değilse ondan yararlanabilmeniz için **new** anahtar kelimesi ile sınıftan yola çıkıp nesne hazırlamanız gerektiğini biliyorsunuz. Sınıflardan yola çıkılarak nesne hazırlanırken kullanılması gereken metotlara **yapıcı metot** (constructor) adı verilmektedir. Kendi hazırladığınız sınıfınız için C# derleyicisinin otomatik olarak hazırlayabildiği standart yapıcı metodu kullanmak yerine kendiniz hazırlayabilirsiniz.

Yapıcı metotlar ile yapılabilinecek birçok işlem olsa bile genellikle ilgili sınıf tipinde nesne hazırlarken veya nesneler hazırlandığında ilk sahip olması gereken özellikleri ayarlamak

amacıyla kullanılıyor. Yapıcı metotlar hazırlanıp kullanılmadığında nesnelerin özelliklerini sonradan ayarlamak gerekebilir.

Sınıfınız için yapıcı metot hazırlamak istiyorsanız bu metot, sınıfla aynı ada sahip olmalıdır. Yapıcı metotlar geriye değer göndermemelerine rağmen **void** olarak işaret edilmezler. Yapıcı metotların geriye değer göndermemelerine rağmen void anahtar kelimesinin kullanılmama nedeni üzerinde düşünürsek akla şunlar gelebilir: Yapıcı metotlar sınıftan yola çıkılarak nesne hazırlanırken kullanılan metotlar olduğuna göre geriye hazırladıkları nesneyi zaten gönderiyorlar. Dolayısıyla metodun imzasında void kullanılmaması normaldir.

Yapıcı metotlar genellikle **public**'tir. Çünkü bu metotlar doğaları gereği olarak başka sınıflardan çağrılırlar. Tabi yapıcı metodu private yaparsanız derleyicinin buna bir itirazı olmaz. Bu durumda o sınıf tipinde nesne hazırlamak bilinen yöntemlerle mümkün olmaz.

Yapıcı metotlar üzerine konuşabilmek amacıyla hazırlayıp aşağıda verdiğimiz uygulamada 2 sınıf bulunmaktadır. "Tarih" sınıfımız 3 public alana sahiptir. Main() metoduna sahip "Program" sınıfının Main() metodunda ise "Tarih" sınıfının örneği alınmaktadır. Başka bir deyişle "Tarih" tipinde bir nesne hazırlanmaktadır.

```
public class Tarih
{
 public int gun;
 public int ay;
 public int yil;
}

public class Program
{
 public static void Main()
 {
  Tarih t = new Tarih();
 }
}
```

Bu uygulama derlenip çalıştırıldığında Main() metodundaki satır sayesinde "Tarih" tipinde bir nesne hazırlanır. Bu satırda **new** anahtar kelimesinden sonra yazılan ve sonuna "()" eklenenin bir metot olduğunu biliyorsunuz. "Tarih" adını verdiğimiz sınıfta yapıcı metot hazırlamadığımız için derleyici kendiliğinden "varsayılan yapıcı metot" adı verilen metodu hazırlamaktadır. Bu metotta sınıfın sayısal tipli alanlarına 0, referans tipli alanlarına null ve bool alanlara ise false aktarılmaktadır. Bu nedenle Main() metoduna ekleme yapıp "t" adını verdiğimiz nesnenin alanlarının içeriklerini konsola yazarsak hepsinin 0 olduğu görülür.

```
public class Program
{
 public static void Main(string[] args)
  {
   Tarih t = new Tarih();
   Console.WriteLine(t.Gun);
   Console.WriteLine(t.Ay);
   Console.WriteLine(t.Yil);
  }
}
```

Henüz konunun başında olmamıza rağmen maddeler halinde kısa özet geçeceğiz. Çünkü konunun özü tam olarak kavranmadan ilerlersek anlattıklarımız zor anlaşılabilir:

- 3 public alana sahip bir sınıf hazırladık. Amacımız yazdığımız kodu kısa tutarak yapıcı metotları anlatmak olduğu için alanları public yaptık. Yoksa alanların public yapılmasını önermiyoruz.
- **new** anahtar kelimesi ile sınıftan yola çıkıp varsayılan yapıcı metodu kullanarak burada "t" adını verdiğimiz bir nesne hazırladık. Dolayısıyla elimizdeki nesnenin alanlarına henüz uygun veriler aktarılmış değildir.
- Bu durumu görmek için uygulamamızdaki Main() metodunda nesneye ait alanların içeriklerini konsola yazdık. Aslında yapmamız gereken nesneyi hazırladıktan sonra alanlarına uygun bilgileri aktarmaktır.

Bu özetten sonra şimdi Main() metodu içinde varsayılan yapıcı metodu kullanarak hazırladığımız nesnenin alanlarına uygun bilgiler aktarıp içeriklerini ondan sonra konsola yazacağız.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
}
public class Program
{
 public static void Main()
 {
   Tarih t = new Tarih();
   t.Gun = 20;
   t.Ay = 7;
   t.Yil = 2017;
   Console.WriteLine(t.Gun);
   Console.WriteLine(t.Ay);
   Console.WriteLine(t.Yil);
 }
}
```



Sınıfınızın varsayılan yapıcı metodunu kendiniz hazırlayabilirsiniz. Ancak bu konuda önemli bir ayrıntıyı ortaya çıkarmak için burada "Tarih" adını verdiğimiz sınıfımız için varsayılan yapıcı metot hazırlamayı biraz erteleyeceğiz. Bir diğer konu şudur: Sınıfınız için yapıcı metot hazırlarken Visual Studio'dan yardım almak istiyorsanız "ctor" yazıp 2 kez tab tuşuna basarsanız varsayılan yapıcı metot olarak kullanabileceğiniz bir metot kalıbı hazırlanır.

Bu şekilde yapıcı metot hazırlamadan, başka bir deyişle varsayılan yapıcı metottan yararlanarak nesne hazırladıktan sonra şimdi kendimiz bir yapıcı metot hazırlayacağız. Yukarıda varsayılan yapıcı metot ile nesne hazırladıktan sonra alanlarına uygun bilgiler aktardık. Bu tercih edilen bir yöntem değildir. Madem yapıcı metotlar, ilgili sınıf tipinde nesne hazırlarken nesnenin sahip olması gereken başlangıç özellikleri ayarlamak amacıyla sıklıkla kullanılıyor, bizim yukarıda Main() metodunda yaptığımız işlemi sınıf içinde yapıcı metot dahilinde yapmamız daha anlamlı olacaktır. Bu amaçla "Tarih" adını verdiğimiz sınıfımızı aşağıdaki gibi düzenledik.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
 public Tarih(int g, int a, int y)
 {
  Gun = g;
  Ay = a;
  Yil = y;
 }
}
```

Sınıfımızın 3 public alana sahip olması ve sınıftan yola çıkılarak hazırlanacak nesnelerin özelliklerini yapıcı metot dahilinde ayarlamak istediğimiz için yapıcı metodun 3 parametreye sahip olmasını sağladık. Metodun içinde ise parametre olarak gönderilen bilgileri alanlara aktardık. Yapıcı metotların sınıfla aynı ada sahip olması gerektiğini ve imzasında **void** anahtar kelimesine yer verilmediğini biliyorsunuz.

Bu hazırlıktan sonra üzerinde çalıştığımız uygulamadaki Main() metoduna sahip diğer sınıfa geçip aşağıdaki gibi "Tarih" tipinde bir nesne hazırlayıp nesnenin alanlarının içeriklerini konsola yazdık.

```
public class Program
{
 public static void Main()
  {
   Tarih t = new Tarih(20, 7, 2017);
   Console.WriteLine(t.Gun);
   Console.WriteLine(t.Ay);
   Console.WriteLine(t.Yil);
  }
}
```

Şimdi yukarıda yapıcı metotlarla ilgili olarak yaptığımız işlemleri maddeler halinde özetleyeceğiz:

- Yapıcı metotları anlatmak amacıyla hazırladığımız sınıfımız 3 public alana sahiptir ve bu sınıf tipinde nesne hazırlarken varsayılan yapıcı metodu kullanmak istemedik.
- Varsayılan yapıcı metot yerine 3 parametresi olan ve sınıfla aynı ada sahip bir yapıcı metot hazırladık. Metodun içinde, parametre olarak gönderilecek 3 bilgiyi hazırlanacak nesnenin public alanlarına aktardık. Tabi bu alanlar public olduğu için sonradan nesnenin alanlarına müdahale etmek mümkündür.
- Devamında hazırladığımız yapıcı metodu kullanarak burada "t" adını verdiğimiz "Tarih" tipinde bir nesne hazırladık.

Yukarıda ifade edildiği üzere programcı sınıfı için herhangi bir yapıcı metot hazırlamadığı zaman derleyici geri planda varsayılan yapıcı metodu hazırlamaktadır. Bizim yukarıda yaptığımız gibi programcı bir şekilde yapıcı metot hazırladığında derleyici varsayılan yapıcı metot hazırlama desteğini çekmektedir. Bu nedenle 3 parametreli yapıcı metodumuz varken aşağıdaki gibi parametresiz (yani varsayılan) yapıcı metodu kullanarak nesne hazırlamak mümkün değildir.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
 public Tarih(int g, int a, int y)
 {
   Gun = g;
   Ay = a;
   Yil = y;
 }
}
public class Program
{
 public static void Main()
  {
   Tarih t = new Tarih(); // hata meydana gelir.
  }
}
```

Sınıfınız için kendiniz yapıcı metot hazırlamış iken varsayılan yapıcı metoda ihtiyaç duyuyorsanız yani parametresiz yapıcı metodu kullanarak nesne hazırlamak istiyorsanız bunu kendiniz hazırlamanız gerekir. Aşağıda verilen kodu varsayılan yapıcı metodu hazırladıktan sonra buraya aldık.

```
public class Tarih
{
  public int Gun;
  public int Ay;
  public int Yil;
  public Tarih(int g, int a, int y)
  {
   Gun = g;
   Ay = a;
   Yil = y;
  }
  public Tarih()
  {
   DateTime Bugun = DateTime.Today;
   Gun = Bugun.Day;
   Ay = Bugun.Month;
   Yil = Bugun.Year;
  }
}
public class Program
{
 public static void Main()
 {
  Tarih t = new Tarih();
  Console.WriteLine(t.Gun);
  Console.WriteLine(t.Ay);
  Console.WriteLine(t.Yil);
 }
}
```

Sonradan hazırladığımız yapıcı metotta geçerli tarihin gün, ay ve yıl bilgileri nesnenin alanlarına aktarılmaktadır. Bu şartlarda "Tarih" sınıfımız 2 yapıcı metoda sahiptir. Nesne hazırlanırken yapıcı metot 3 parametre ile kullanılırsa derleyici 3 parametreli sürümü işletir. "Tarih" tipinde nesne hazırlanırken yapıcı metot parametresiz olarak kullanılırsa derleyici parametresiz sürümü işletir.

Şimdi ise aynı sınıf için 3. bir yapıcı metot hazırlayacağız. C# derleyicisi yapıcı metotların **overload** versiyonlarını desteklediği için aynı sınıf için değişik sayı ve tipte parametreye sahip birden fazla yapıcı metot hazırlayabilirsiniz.

```
public class Tarih
{
  public int Gun;
  public int Ay;
  public int Yil;
  public Tarih(int g, int a, int y)
  {
   Gun = g;
   Ay = a;
   Yil = y;
  }
  public Tarih()
  {
   DateTime Bugun = DateTime.Today;
   Gun = Bugun.Day;
   Ay = Bugun.Month;
   Yil = Bugun.Year;
  }
  public Tarih(int g)
  {
    DateTime Bugun = DateTime.Today;
    Gun = g;
    Ay = Bugun.Month;
    Yil = Bugun.Year;
  }
}
```

Bu hazırlıktan sonra bu sınıftan yola çıkıp nesne hazırlanırken sınıfın yapıcı metoduna int tipinde bir değer parametre olarak verilirse bir parametreye sahip yapıcı metot işletilir. "Tarih" sınıfından yararlanılıp aşağıdaki gibi bir nesne hazırlanırsa ekrana önce 25, sonra geçerli tarihin ay ve yıl bilgileri yazılır.

```
public class Program
{
 public static void Main()
 {
  Tarih t = new Tarih(25);
  Console.WriteLine(t.Gun);
  Console.WriteLine(t.Ay);
  Console.WriteLine(t.Yil);
 }
}
```

Bizim burada hazırladığımız varsayılan yani herhangi bir parametreye sahip olmayan yapıcı metot, derleyici tarafından gerektiğinde geri planda hazırlanandan farklıdır. Derleyici tarafından hazırlanan varsayılan yapıcı metotta, sınıfın sayısal tipli alanlarına 0, bool alanlarına false ve referans tipli alanlara null aktarırken biz sınıfımızın int tipindeki 3 alanına geçerli tarihin gün, ay ve yıl bilgilerini aktardık.

Hazırladığımız varsayılan yapıcı metodun derleyici tarafından hazırlanan ile birebir aynı olmasını istemiş olsaydık "Tarih" sınıfımız için aşağıdaki gibi bir metot hazırlardık. Tabi bu durumda "Tarih" tipinde nesne hazırladığımızda alanlara bilgileri kendimiz aktarmak durumunda kalırdık.

```
public Tarih()
{
}
```

Kalıtım ve Yapıcı Metotlar

Bu bölüm dahilinde yukarıdaki sayfalarda kalıtım kavramı hakkında bilgi verilmişti. Üzerinde çalıştığımız sınıfınızda "base" olarak kullanılan sınıfın yapıcı metodu varsa, **new** anahtar kelimesi kullanılıp nesne hazırlanmak istendiğinde ayrıca base sınıfın da yapıcı metodu işletilir. Derived yani türeyen sınıfın örneği hazırlandığında yapıcı metotların nasıl bir işleve sahip olduğunu anlatmak için aşağıda verilen kodu hazırladık.

```
public class Islemler
{
 public Islemler()
  {
   Console.WriteLine("Islemler tipinde nesne hazırlandı");
  }
}
class Program : Islemler
{
 public static void Main(string[] args)
 {
  Islemler nesne = new Islemler();
 }
}
```

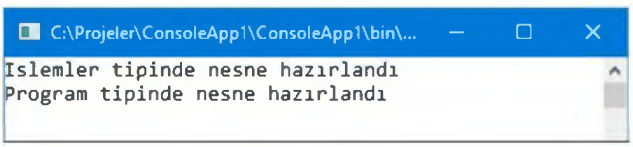
Bu kodda önce "Islemler" adında bir sınıf hazırladık. "Program" adını verdiğimiz diğer sınıf ise "Islemler" sınıfının mirasçısıdır. Bu şartlarda uygulama çalıştırılırsa Main() metodunda "Islemler" tipinde nesne hazırlandığı sırada yapıcı metot işletilir.

Main() metoduna sahip "Program" sınıfının da yapıcı metodu olsaydı ne olurdu? Bu soruya cevap verebilmek için yukarıda verilen "Program" adlı sınıfı aşağıdaki gibi değiştirdik.

```
public class Islemler
{
 public Islemler()
 {
  Console.WriteLine("Islemler tipinde nesne hazırlandı");
 }
}
```

```
class Program : Islemler
{
 public Program()
 {
  Console.WriteLine("Program tipinde nesne hazırlandı");
 }
 public static void Main(string[] args)
 {
  Program nesne = new Program();
 }
}
```

Bu şartlarda uygulama çalıştırılırsa önce "base" sınıfın yani "Islemler" sınıfının yapıcı metodu, ardından mirasçı yani "Program" sınıfının yapıcı metodu işletilir. Aşağıda verilen ekran görüntüsünü 2 sınıftan meydana gelen bu uygulamayı derleyip çalıştırdıktan sonra aldık. 2 sınıfın bulunduğu bu uygulamada sadece base class'ın (Islemler) yapıcı metodu olsaydı sadece o metot işletilirdi.



Böylece **new** anahtar kelimesi aracılığıyla yapıcı metot kullanılarak bir sınıftan nesne hazırladığımızda geri planda base class'ın yapıcı metodunun işletildiğini görmüş olduk. Ancak bu konu, yani base class'ın da yapıcı metoda sahip olduğunda bu kadar basit değildir. Yapıcı metotlar ve miras kavramı önemli olduğundan birkaç ayrıntıdan söz etmek için aşağıda verilen kodu hazırladık. Bu kodda 3 sınıf bulunmaktadır. "Tarih" ve "Zaman" adını verdiğimiz ilk 2 sınıfın ikişer alanı ve yapıcı metotları bulunmaktadır. Burada hazırladığımız "Tarih" adlı class, yukarıda hazırladığımızdan biraz farklıdır.

```
public class Tarih
{
 public int gun;
 public int ay;
 public Tarih(int i, int j)
 {
  gun = i;
  ay = j;
 }
}
public class Zaman
{
 public int saat;
 public int dakika;
 public Zaman(int x, int y)
 {
  saat = x;
  dakika = y;
 }
}
```

```
public class Program
{
 public static void Main(string[] args)
  {
   Zaman zaman = new Zaman(12, 44);
   Console.WriteLine(zaman.saat);
   Console.WriteLine(zaman.dakika);
  }
}
```

Bu şartlarda bu uygulama derlenip çalıştırılırsa "Zaman" adını verdiğimiz sınıfın örneği hazırlanır yani bu sınıftan yola çıkıp **new** anahtar kelimesi ile nesne hazırlanırken yapıcı metoda parametre olarak verilen 12 ve 44 sayıları ekrana yazılır.

Buraya kadar sizin için yeni bir şey yoktur. Ancak "Zaman" sınıfı "Tarih" sınıfının mirasçısı yapılırsa ne olur? Her 2 sınıfının yapıcı metotları parametrelere sahip olduğu için hata meydana gelir. Bunun önüne geçmek için türeyen (derived) sınıfın yapıcı metodunun gerek duyacağı parametreleri vermek gerekir. Bu işlemin nasıl yapıldığını anlatmadan önce yukarıdaki durumla ilgili kısa bir açıklama yapacağız.

- Uygulamanın Main() metodunun bulunduğu sınıftan başka 2 sınıfımız var ve bu sınıfların yapıcı metotları ikişer parametreye sahiptir.
- "Zaman" adını verdiğimiz sınıf "Tarih" sınıfının mirasçısı durumunda iken Main() metodunun olduğu sınıfta new anahtar kelimesiyle "Zaman" tipinde nesne hazırlanmak istendiğinde hata meydana geleceğini biliyorsunuz.
- Bu hatanın nedeni, bu şartlarda "Zaman" tipinde nesneyi oluşturacak üyelerin birden fazla sınıftan gelmesidir. Madem "Zaman" sınıfı Tarih sınıfının mirasçısıdır, Zaman tipindeki nesnede hem Zaman hem de Tarih sınıfının alanları olacaktır.
- Başka bir deyişle bu şartlarda hazırlanacak "Zaman" tipinde nesnenin yarısı "Tarih" sınıfının yapıcı metoduyla, diğer yarısı "Zaman" sınıfının yapıcı metoduyla oluşturulup "Zaman" tipindeki nesnede bir araya getirilecektir.
- Bu anlatılanlardan şunu çıkarmak mümkündür: "Zaman" tipinde nesne hazırlanırken kullanılacak yapıcı metot toplam 4 parametreye sahip olmalıdır.

Bu bilgilerin ışığında yukarıda verdiğimiz uygulamamızı aşağıdaki gibi düzenledik.

```
public class Tarih
{
  public int gun;
  public int ay;
  public Tarih(int i, int j)
  {
   gun = i;
   ay = j;
  }
}
public class Zaman : Tarih
{
 public int saat;
 public int dakika;
```

```
 public Zaman(int x, int y, int z, int w)
 {
  saat = x;
  dakika = y;
 }
}
public class Program
 {
  public static void Main(string[] args)
  {
   Zaman zaman = new Zaman(12, 44, 20, 7);
   Console.WriteLine(zaman.saat);
   Console.WriteLine(zaman.dakika);
  }
}
```

Uygulamamızın son halinde "Zaman" sınıfının yapıcı metodunun parametre sayısını 4 olarak değiştirdik. Gerçekte bu 4 bilginin ikisi "Tarih" sınıfı diğer ikisi ise "Zaman" sınıfı içindir. Ancak hangi parametrenin hangi sınıfın yapıcı metodu tarafından kullanılacağı belli olmadığı için uygulamamız bu hali ile derlenemez.

Hatayı gidermek için "Zaman" sınıfının yapıcı metoduna parametre olarak verilen 4 bilginin hangilerinin base, hangilerinin derived sınıfa ait olduğu işaret edilmelidir. Bu ayarlama yapılırken **base** anahtar kelimesinden yararlanılır. "Zaman" ve "Program" sınıfında yaptığımız değişiklikleri aşağıda görebilirsiniz.

```
class Zaman : Tarih
{
  public int saat;
  public int dakika;
  public Zaman(int x, int y, int z, int w) :base(z, w)
  {
   saat = x;
   dakika = y;
  }
}
class Program
{
 public static void Main(string[] args)
  {
   Zaman zaman = new Zaman(12, 44, 20, 7);
   Console.WriteLine(zaman.saat);
   Console.WriteLine(zaman.dakika);
   Console.WriteLine(zaman.gun);
   Console.WriteLine(zaman.ay);
  }
}
```

Burada yapılan 2 işlem var: Birincisi "Zaman" adlı sınıfın yapıcı metodunun parametre sayısını 4'e çıkardık. Çünkü hem bu sınıfın hem de mirasçısı olduğu "Tarih" sınıfının ikişer parametresi bulunmaktadır. Derleyici bu 4 parametrenin hangi sınıfa ait olduğunu bilemez. Bu nedenle **base** deyiminden yararlanıp hangi parametrelerin base sınıfa ait olduğunu işaret ettik.

Base Class'ın Yapıcı Metodu

Yapıcı metotları anlatmak üzere yukarıda hazırlayıp "Tarih" adını verdiğimiz sınıfta 2 public alanın olduğunu biliyorsunuz. En ideali bu sınıfta alanlardan başka özelliklerin ve metotların da olmasıydı. Kodu kısa tutmak için sadece public alanlara yer verdik.

Diğer yandan "base" anahtar kelimesinden yararlanıp base class'ın yapıcı metodunu işletmek mümkündür. Üzerinde çalıştığınız sınıfın mevcut yapıcı metotları dahilinde başka bir yapıcı metodu işletmek istiyorsanız "this" anahtar kelimesinden yararlanabilirsiniz.

Yukarıda hazırladığımız "Tarih" sınıfı "this" anahtar kelimesinin işlevini göstermeye, nasıl kullanıldığını anlatmaya uygun olmadığı için aşağıdaki gibi başka bir sınıf hazırladık. "Oyuncu" adını verdiğimiz bu sınıfta 3 public alan ve toplam 4 yapıcı metot bulunmaktadır.

```
public class Oyuncu
{
 public string adi;
 public string takimi;
 public double boyu;
 public Oyuncu()
 {
  adi = "atanmadı";
  takimi = "belli değil";
  boyu = 2.05;
 }
 public Oyuncu(string ad)
 {
  adi = ad;
  takimi = "belli değil";
  boyu = 2.05;
 }
 public Oyuncu(string ad, string takim)
 {
  adi = ad;
  takimi = takim;
  boyu = 2.05;
 }
 public Oyuncu(string ad, string takim, float boy)
 {
  adi = ad;
  takimi = takim;
  boyu = boy;
 }
}
```

Bu sınıfın örneği parametre kullanılmadan hazırlanmak istendiğinde "adi" ve "takimi" alanlarına belirtilen string sabitler, "boyu" alanına ise 0 aktarılmaktadır. Şimdi hedefimiz sayısı 4 olan bu yapıcı metotları "this" anahtar kelimesini kullanarak yazılan kodu azaltmaktır. Bu işlemi adım adım anlatmak için yukarıda verilen kodun 3 parametreli yapıcı metot hariç diğerlerini sildik.

```
public class Oyuncu
{
 public string adi;
 public string takimi;
 public double boyu;
 public Oyuncu(string ad, string takim, float boy)
 {
  adi = ad;
  takimi = takim;
  boyu = boy;
 }
}
```

Şimdi sırada 2 parametreli yapıcı metodu hazırlamak var. Amacımızın bu yapıcı metoda ait bloğa bir şey yazmamak olduğu için base anahtar kelimesinden yararlanacağız. 2 parametreli yapıcı metot kullanılarak nesne hazırlandığında nesnenin boyu alanına 0 aktarmak istediğimizi biliyorsunuz. Dolayısıyla bu koda aşağıdaki gibi bir yapıcı metot eklersek 2 parametreli yapıcı metot işlevini görür.

```
public class Oyuncu
{
 public string adi;
 public string takimi;
 public double boyu;
 public Oyuncu(string ad, string takim, float boy)
 {
  adi = ad;
  takimi = takim;
  boyu = boy;
 }
public Oyuncu(string ad, string takim) : this(ad, takim, 0)
 {
 }
}
```

Bu metot sayesinde bu sınıfta hazırladığımız 3 parametreli yapıcı metot işletilir. Programcı nesne hazırlarken yapıcı metoda 2 parametre verdiği için bu düzenleme sayesinde 3. parametrenin 0 olduğu varsayılır. Konunun iyice netleşmesi için bir de tek parametreli yapıcı metot hazırlayacağız. Amacımız yine bu metoda işletilen herhangi bir satır yazmamak.

```
public class Oyuncu
{
 public string adi;
 public string takimi;
 public double boyu;
 public Oyuncu(string ad, string takim, float boy)
 {
  adi = ad;
  takimi = takim;
  boyu = boy;
 }
public Oyuncu(string ad, string takim) : this(ad, takim, 0)
 {
 }
```

```
public Oyuncu(string ad) :this (ad, "belli değil", 0)
 {
 }
}
```

Programcı, "Oyuncu" tipinde nesne hazırlarken yapıcı metoda "adi" alanına aktarılmak üzere bir tek parametre kullanırsa bu son yazdığımız yapıcı metot işletilir. "takimi" alanına "belli değil" ve boyu alanına 0 aktarılır. Sanırız konu anlaşıldı. Aşağıda verilen kodda parametresiz yapıcı metodun yeni hali var.

```
public class Oyuncu
{
 public string adi;
 public string takimi;
 public double boyu;
 public Oyuncu() : this ("atanmadı", "belli değil", 0)
 {
 }
 public Oyuncu(string ad) :this (ad, "belli değil", 0)
 {
 }
 public Oyuncu(string ad, string takim) : this(ad, takim, 0)
 {
 }
 public Oyuncu(string ad, string takim, float boy)
 {
  adi = ad;
  takimi = takim;
  boyu = boy;
 }
}
```

Static Yapıcı Metotlar

Yukarıda anlatılan yapıcı metotlardan başka **Statik Yapıcı** metotlar bulunmaktadır. Static yapıcı metotlar genellikle örneği alınmak istenen sınıfın statik üyelerine ilk değer aktarmak veya değiştirmek amacıyla kullanılmaktadır. Statik olmayan metotlarda statik alanları değiştirmek mümkün olmadığı için statik yapıcı metotlara başvurulmaktadır. Bu konuyu adım adım anlatmak için Visual Studio ile aşağıda verilen sınıfları hazırladık. İlk sınıftaki yapıcı metodun statik olmasını sağladık.

```
public class Sinifim
{
 static Sinifim()
 {
   System.Console.WriteLine("Statik yapıcı metot işletildi");
  }
}
class Program
 {
  public static void Main()
  {
   Sinifim nesne1 = new Sinifim();
  }
}
```

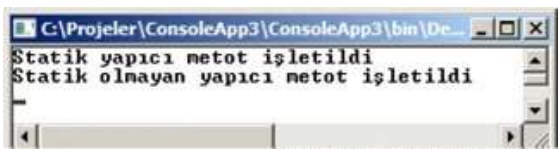
Bu şartlarda bu uygulama derlenip çalıştırılırsa Main() metodu içinde ilk sınıfın örneği hazırlandığı için yapıcı metot statik olmasına rağmen işletilir ve konsola statik yapıcı metodun işletildiği mesajı yazılır.



Buraya kadar her şey beklendiği gibidir, **new** anahtar kelimesi ile sınıfın örneği alındığında bu sınıfın yapıcı metodu statik olsa bile işletilmektedir. Static metotlarda **public** veya **private** gibi erişim belirticileri kullanılmaz. Ayrıca static yapıcı metotlar parametreye sahip olamazlar. Şimdi ilk sınıf için ayrıca statik olmayan bir yapıcı metot hazırlayacağız.

```
class Sinifim
{
 static Sinifim()
  {
   Console.WriteLine("Statik yapıcı metot işletildi");
  }
 public Sinifim()
  {
   Console.WriteLine("Statik olmayan yapıcı metot işletildi");
  }
}
class Program
{
 public static void Main()
  {
   Sinifim nesne1 = new Sinifim();
  }
}
```

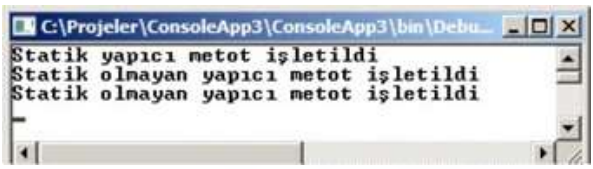
Bu şekilde değiştirilen uygulama derlenip işletildiğinde Main() metodunun bulunduğu sınıfta ilk sınıfın örneği hazırlandığında önce statik, ardından statik olmayan yapıcı metot işletilir. Aşağıda verilen ekran görüntüsü bu uygulama derlenip çalıştırıldıktan sonra alındı.



Statik yapıcı metotların bir diğer özelliğinden söz etmek için Main() metodunda ilk sınıfın ikinci bir örneğini hazırladık.

```
public static void Main()
{
 Sinifim nesne1 = new Sinifim();
 Sinifim nesne2 = new Sinifim();
}
```

2. sınıfta bulunan Main() metodu bu şekilde değiştirilip uygulama çalıştırıldığında aşağıdaki gibi bir sonuç alınır. Buna göre statik yapıcı metotlar o sınıftan yola çıkılıp ilk kez nesne hazırlanırken işlevsel oluyor. Sınıfın örneği 2. kez hazırlanırken sadece statik olmayan yapıcı metot işletilmektedir.



Bildiğiniz gibi statik alanlara nesne üzerinden değil sınıf üzerinden erişilir. Daha önce statik sınıflar ayrıntılı bir şekilde incelenmiş olunsa bile burada ister istemez statik sınıfların bazı özelliklerinden söz edeceğiz. Konunun geri planını ortaya çıkarmak için yukarıda verdiğimiz ve kendisi için statik yapıcı metot hazırladığımız sınıfta 2 statik alana yer verdik.

```
public class Sinifim
{
 public static int Yil = 2016;
 public static string Sehir = "Adana";
 static Sinifim()
 {
  System.Console.WriteLine("Statik yapıcı metot");
 }
 public Sinifim()
 {
  Console.WriteLine("Statik olmayan yapıcı metot");
 }
}
```

Bu şekilde düzenlenen sınıfın örneği alınmadan statik alanların içeriklerine her yerden erişilebilir. Bu statik alanları tanımlarken aynı zamanda ilk değer atamamış olsaydık, bu alanlara herhangi bir metotta değer aktarıp kullanabilirdik. Aşağıda bu işlemi nasıl yaptığımızı görebilirsiniz.

```
class Program
{
 static void Main(string[] args)
  {
     Sinifim.Yil = 2017;
     Sinifim.Sehir = "Adana";
  }
}
```

Ancak statik alanların bu şekilde kullanılması önerilmez. Yapılması gereken, sınıfın statik alanlarına statik yapıcı metot dahilinde ilk değerleri vermektir. Tabi programcı gerek duyduğunda statik yapıcı metot aracılığıyla ilk değerlerini verdiği statik alanların içeriğinde sonradan değişiklik yapabilir.

```
class Program
{
 static void Main(string[] args)
 {
```

```
        Console.WriteLine(Sinifim.Yil);
        Console.WriteLine(Sinifim.Sehir);
      }
    }
    public class Sinifim
    {
      public static int Yil;
      public static string Sehir;
      static Sinifim()
      {
       Yil = 2017;
       Sehir = "Adana";
      }
    }
```

Statik yapıcı metotlarla ilgili 3 önemli ayrıntı vardır:

- Parametreye sahip olamazlar. Dolayısıyla bir sınıfın birden fazla static yapıcı metodu olamaz.
- Statik yapıcı metot içinde sınıfın static olmayan elemanlarıyla ilgili işlem yapılamaz.
- Erişim belirleyiciler ile yaşam alanları işaret edilemez.

Destructor – Yok Edici Metotlar

Yukarıdaki sayfalarda işaret edildiği gibi **new** anahtar kelimesi ile sınıfın örneği alınan sınıfın **constructor** adı verilen metodu varsa işletiliyor. Sınıfın yapıcı metodu yoksa derleyici tarafından otomatik olarak hazırlanan varsayılan yapıcı metot işletilmektedir. Benzer şekilde hazırlanan nesne Garbage Collector tarafından bellekten silinirken sınıfın **destructor** metodu varsa işletilmektedir. Bu konuda bilgi vermek için aşağıda verilen kodu hazırladık.

```
class Mesaj
{
 public Mesaj()
  {
   Console.WriteLine("Mesaj Sınıfının örneği Hazırlandı");
  }
 ~Mesaj()
  {
   Console.WriteLine("Nesne bellekten Silinecek");
  }
}
class Program
{
 public static void Main()
 {
  Mesaj Nesne = new Mesaj();
  Console.WriteLine("Çıkmak için bir tuşa basın");
  Console.ReadLine();
 }
}
```

Bu kodda sizin için yeni olan "~" ile başlayan ve class ile aynı ada sahip olan metottur. Eğer söz konusu sınıf için **Destructor** yani yok edici metot hazırlamak istiyorsanız metot sınıfla

aynı ada sahip olmalı ve adının önüne tilda("~") karakteri yazılmalıdır. Destructor metotların parametreleri yoktur ve geriye bilgi göndermezler.

Bu uygulama derlenip çalıştırıldığında new anahtar kelimesi ile "Mesaj" adlı class'ın örneği hazırlandığında bu class'ın yapıcı metodu işletilir. Devamında sıra uygulamanın çalışmasını durdurmak için yazılan **Console.ReadLine();** satıra gelir. Aşağıda verilen ekran görüntüsü bu konsol uygulamasının işletimi bu satırda iken alındı.



Bu sırada bir tuşa basılıp uygulamanın işletimi sona erdirildiğinde "Mesaj" adını verdiğimiz ilk sınıfın yok edici metodu işletilir.

ExpandoObject Sınıfı İle Dinamik Tipler Hazırlamak

Bu kitabın isimsiz tipler hakkında bilgi verilen bölümde ifade edildiği üzere C# 3.0'la birlikte metot içinde kullanmak üzere isimsiz tipler hazırlamak mümkündür. Bu sayede sadece bir kez kullanılacak tipler için ayrı sınıf hazırlama zorunluluğu ortadan kaldırılmıştı.

C# programlama dilinin 4.0 sürümü ile gelen **Dynamic Language Runtime** alt yapısından yararlanan **ExpandoObject** sınıfı sayesinde dinamik tipler hazırlayıp kullanmak mümkün hale geldi. ExpandoObject sınıfı Sealed bir sınıftır ve çok sayıda interface'i uygulamaktadır. Bu interface'ler sayesinde bu sınıfı değişik amaçlar için kullanmak mümkündür. ExpandoObject sınıfının orijinal yapısını aşağıda görebilirsiniz.

```
public sealed class ExpandoObject :
   IDynamicMetaObjectProvider,
     IDictionary<string ,object>,
       ICollection<KeyValuePair<string ,object>>,
         IEnumerable<KeyValuePair<string ,object>>,
           IEnumerable, INotifyPropertyChanged
```

Bu sınıfın nasıl kullanıldığını adım adım anlatmak için üzerinde çalıştığımız uygulamada Main() metodunun içinde aşağıdaki gibi bir ExpandoObject nesnesi hazırladık. ExpandoObject sınıfı System.Dynamic'de yer almaktadır. Her seferinde sınıf adından önce namespace adını yazmamak için **using** bloğunda ayarlama yapılabilir.

```
public static void Main()
{
 System.Dynamic.ExpandoObject Takim = new
              System.Dynamic.ExpandoObject();
}
```

ExpandoObject sınıfının asıl kullanım amacı çalışma anında yeni tipler oluşturup kullanmak olduğu için yapıcı metodun geriye gönderdiği nesneyi dinamik bir tipe aktarmak gerekiyor.

```
public static void Main()
{
 dynamic Takim = new ExpandoObject();
}
```

ExpandoObject sınıfının yapıcı metodunun geriye gönderdiği nesne dinamik bir değişkene aktarıldığında Visual Studio'dan IntelliSense desteğinin alınamadığını biliyorsunuz. ExpandoObject nesnesi hazırlama amacımız çalışma anında yeni bir tip hazırlamak olduğundan koda ekleme yapıp burada "Takim" adını verdiğimiz nesnenin 2 alana sahip olmasını sağladık. Alanlara mutlaka değer aktarılması gerekiyor.

```
public static void Main()
{
 dynamic Takim = new System.Dynamic.ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
}
```

Bu hazırlıktan sonra elimizde 2 alana sahip bir nesne bulunmaktadır. Bu alanlardan ilki System.String iken diğeri System.In32 tipindedir. Alanların tipleri çalışma anında aktarılan bilginin tipine göre derleyici tarafından seçiliyor. Elbette 2 alana sahip bir nesneyi isimsiz tip olarak hazırlamak son derece kolaydır. Ancak ExpandoObject sınıfıyla yapılabilinecek birçok işlem vardır. ExpandoObject nesnesi hazırlayıp istenilen sayıda alana sahip olmasını sağlamak ne kadar kolaysa bu alanların içeriklerine erişip kullanmak ta o kadar kolaydır. Bu alanların içeriklerini nasıl okuduğumuzu aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 dynamic Takim = new System.Dynamic.ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 Console.WriteLine("Takımın adı : {0}", Takim.Adi);
 Console.WriteLine("Kuruluş yılı: {0}", Takim.Kurulus.ToString());
}
```

Şimdiye kadar anlatılanlardan fark etmiş olacağınız gibi referansı dinamik değişkene aktarılan ExpandoObject nesnesi sayesinde çalışma anında kolayca alan tanımlayıp istenen tipteki bilgi aktarılabiliniyor. Üstelik mevcut alanların içerikleri sonradan değiştirebiliniyor. Bu nedenle bu kod aşağıdaki gibi değiştirilebilir.

```
static void Main(string[] args)
{
 dynamic Takim = new System.Dynamic.ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 Console.WriteLine("Takımın adı  : {0}", Takim.Adi);
 Console.WriteLine("Kuruluş yılı : {0}", Takim.Kurulus.ToString());
 Takim.Adi = "Adana Demirspor";
 Takim.Kurulus = 1940;
 Console.WriteLine("Takımın adı : {0}", Takim.Adi);
 Console.WriteLine("Kuruluş yılı: {0}", Takim.Kurulus.ToString());
 Console.ReadKey();
}
```

Konunun en ilginç yanlarından birisi ExpandoObject nesnesinin mevcut alanlarından birisine başka bir ExpandoObject nesnesinin aktarılabiliniyor olmasıdır. Bu sayede çalışma anında istenen veri yapıları hazırlanabiliyor.

```
static void Main(string[] args)
{
 dynamic Takim = new ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 Takim.Futbol = new ExpandoObject();
 Takim.Futbol.Manager = "Aykut Kocaman";
 Takim.Futbol.Kaptan = "Mehmet Topal";
}
```

Yukarıda ifade edildiği üzere ExpandoObject sınıfı generic yapıdaki IDictionary<string, object> arayüzünü uygulamaktadır. Bu arayüzden gelen metotlar sayesinde birçok işlemi yapmak mümkündür. Bu konuda bilgi vermek için mevcut ExpandoObject nesnemizi as operatörü ile dönüştürüp IDictionary nesnesine aktardık.

```
static void Main(string[] args)
{
 dynamic Takim = new ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 IDictionary<string, object> liste =
              Takim as IDictionary<string,object>;
}
```

Bu andan itibaren IDictionary nesnesi üzerinde yapılan her işlemden ExpandoObject nesnesi etkilenir. Çünkü yukarıda ExpandoObject nesnesi için hazırlanan alanlar gerçekte ExpandoObject nesnesinin dahili nesnesi olan IDictionary nesnesinde tutulmaktadır. Bunu size göstermek için IDictionary arayüzünün Add() metodu ile IDictionary nesnesine bir anahtar-değer çifti ekledik.

```
static void Main(string[] args)
{
 dynamic Takim = new ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 IDictionary<string, object> liste =
              Takim as IDictionary<string, object>;
 liste.Add("Manager", "Aykut Kocaman");
}
```

Bu andan itibaren ExpandoObject nesnemizde artık 3 alan bulunmaktadır. Sonradan IDictionary nesnesi aracılığıyla hazırladığımız alanın içeriğini yukarıda yaptığımız gibi okuyabiliriz.

```
static void Main(string[] args)
{
 dynamic Takim = new ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
```

```
 IDictionary<string, object> liste = Takim as IDictionary<string, object>;
 liste.Add("Manager", "Aykut Kocaman");
 Console.WriteLine("Takımın adı      : {0}", Takim.Adi);
 Console.WriteLine("Kuruluş yılı     : {0}", Takim.Kurulus.ToString());
 Console.WriteLine("Teknik Direktör : {0}", Takim.Manager);
 Console.ReadKey();
}
```

IDictionary arabirimi kaynaklı metotlar yardımıyla ExpandoObject nesnesi üzerinde yapılabilinen işlemlere örnek olmak için üzerinde çalıştığımız kodu aşağıdaki gibi düzenledik. Bu kodda önce mevcut hazırladığımız ExpandoObject nesnesini cast ederek IDictionary nesnesi olarak değerlendirdik. Ardından foreach döngüsü ile IDictionary nesnesinin elemanlarının anahtar ve değer bilgilerini konsola yazdık.

```
public static void Main()
{
 dynamic Takim = new ExpandoObject();
 Takim.Adi = "Fenerbahçe";
 Takim.Kurulus = 1097;
 Takim.Manager = "Aykut Kocaman";
 var liste = (IDictionary<string, object>) Takim;
 foreach (var alan in liste)
 {
   Console.WriteLine(alan.Key + "  :  " + alan.Value);
  }
  Console.ReadKey();
 }
```

11. BÖLÜM

METOT HAZIRLAMAK ve KULLANMAK

Daha önceki konulardan bildiğiniz gibi en küçük C# uygulaması bile class şeklinde düzenlenmektedir. Class'lar ise metot, alan, özellik, interface ve delagate'leri üye olarak kabul edebilmektedir. Ayrıca en kısıtlı C# programı bile Main() adında bir metoda sahip olmak zorundadır. Özetle, C# programcısı metotlarla ilgili bütün ayrıntıları bilmelidir.

Bir önceki kuşak programlama dillerinde sadece metotların olduğu düşünülürse metotların önemi fark edilir. C# kodunda her şey dönüp dolaşıp sonunda metoda gelir. Örneğin delegate'ler metotlar için varlar, olaylar metotları işletmek için durup beklerler. Interface'ler, ilgili sınıfta hazırlanması gereken metotları listelerler. Java'da özelliklerin olmadığını ve bizim C#'ta özellikler aracılığıyla yaptığımız işlemlerin metotlarla yapıldığını söylersek özelliklerin de bir nevi metot olduğunu düşüneceksiniz.

Metotların nasıl hazırlanıp kullanıldığı üzerinde çalışmaya başlamadan her C# programcısının haberdar olmasını istediğimiz bazı önemli noktalar var. Maddeler halinde burada sıralayıp önemli saydığımız bu noktaları daha sonra okumak üzere atlayabilirsiniz. Çünkü burada verilen önerilerle ilgili bilgiler henüz verilmedi.

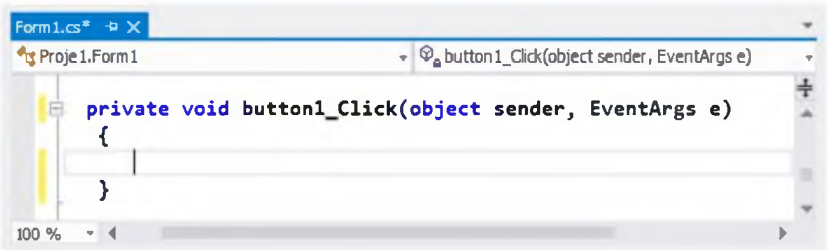
- Nasıl ki herhangi bir konuyu anlatırken kolay anlaşılmak için bir cümlede birden fazla kavramdan, işlemden vs söz etmemek gerekiyorsa, hazırlamak istediğiniz metotta mümkünse birden fazla işlem yapılmasın.
- Sınıfınızda yapmak istediğiniz kaç işlem varsa o kadar metodunuz olsun. Düşünsenize, .NET Framework ile gelen String sınıfının çok sayıda metodu vardır ve her metot sadece bir işlemi yapmaktadır.
- Metotlarda satır sayısının fazla olması önerilmez. Bizce bir metot 30-40'tan fazla satır içermemelidir. Hazırlanan metodun satır sayısı arttığında durup kodu incelemek ve gerekirse söz konusu metotta yapılan işlemleri birden fazla metot arasında paylaştırmak gerekir diyoruz.
- Metoda ad seçerken gereken özeni göstermek, akla ilk geleni metot adı olarak kullanmamak gerekir. Metoda verilen ad, yaptığı işlemi açıklayıcı olmasında fayda vardır. Metot adlarının çok kısa olması ileride kod incelenirken sorunlara neden olabilir.
- Metotlarınızda kullandığınız değişkenlerin class bloğunda mı yoksa metot içinde mi tanımlı olduğuna dikkat etmelisiniz. Bize sorarsanız metotların içinde mümkün mertebe kullandığınız bütün değişkenlerin yaşam alanı metotla sınırlı olmasında fayda vardır. Bu durumda class bloğunda yer alacak bir alan veya özelliğe gerek duyduğunuzda bunun metoda parametre olarak gönderilmesini öneriyoruz. Bu sayede sonradan incelediğiniz bir metot her şeyi ile karşınızda durur, metot dahilinde direk kullandığınız bir alan veya özellik olmadığı için kodun başka yerine geçip neyin ne olduğunu inceleme derdinden kurtulursunuz.
- Metotların parametre sayılarına dikkat etmek gerekir. Çünkü çok sayıda parametreye sahip metotları incelemek, sonradan değiştirmek zahmetlidir. Metoda çok

sayıda bilgiyi parametre olarak göndermeniz gerekiyorsa bu bilgileri bir nesnede toplayıp metoda öyle gönderebilirsiniz.

- Metot dahilinde yapılan işlem, metoda gönderilen bilginin tipine göre değişiklik gösteriyorsa if gibi deyimlerden yararlanmak yerine söz konusu metodu aşırı yüklemeyi tercih edebilirsiniz.
- Metotlara bilgileri parametre olarak gönderirken tür dönüşümü kurallarına uyulmalıdır, uygun tipler kullanılmalıdır. "return" anahtar kelimesi ile geriye gönderilen bilginin uygun tipte olmasına dikkat edilmelidir.

Sınıflar hakkında temel bilgi verilen bir önceki bölümde ister istemez metotlardan biraz söz edildi. Bu bölümde metotlar ayrıntılı bir şekilde incelenecektir. Olay ve metot kavramını hatırlatmak, ayrıca bazı ayrıntılar hakkında konuşabilmek için Visual Studio ile hazırladığımız projedeki formlardan ilkine bir **Button** nesnesi yerleştirdik.

Button nesnesini forma yerleştirip tasarım anında çift tıklarsanız Visual Studio, button nesnesinin Click olayı için metot hazırlamak istediğinizi varsayar. Bu varsayımdan yola çıkıp kod yazılabilinen pencereyi açar ve çalışma anında düğme tıklandığı zaman işletilmek üzere bir metot kalıbı hazırlar.



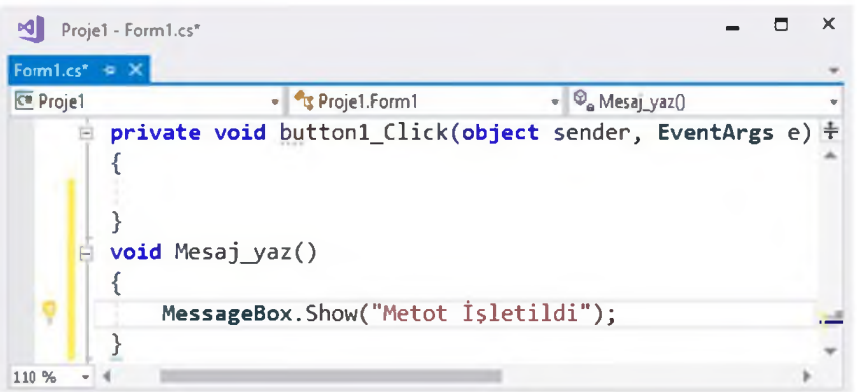
Visual Studio tarafından hazırlanan bu metodu dikkatlice incelerseniz "sender" ve "e" adında 2 parametreye sahip olduğunu görürsünüz. "sender" adındaki ilk parametre **System.Object** tipinde iken "e" adı verilen parametre **System.EventArgs** tipindedir. EventArgs, sıklıkla karşımıza çıkacak bir sınıftır. Bu metodun işletilmesine hangi nesne neden olursa o nesnenin referansı "sender" adı verilen parametre aracılığıyla metoda gönderilmektedir.

Burada metodun işletilmesine neden olacak nesneyle ilgilenmediğimiz için "sender" parametresi hakkında az konuşmayı tercih edip bu metodun işletilmesine neden olan Button nesnesinin başlık metnini elde etmekle yetineceğiz. Bu amaçla button1_click() metodunu aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
 System.Windows.Forms.Button dugme;
 dugme = (Button)sender;
 MessageBox.Show(dugme.Text);
}
```

Metoda yazdığımız ilk satırda Button tipinde bir değişken tanımadık. Çünkü bu metodun işletilmesine neden olan nesneyi elde etmeliyiz ki Button sınıfıyla ilgili metotları kullanabilelim ve bu nesnenin özelliklerine erişebilelim. Burada Button sınıfının yerini göstermek için sınıf adından önce namespace adlarını yazdık. 2. satırda ise object tipindeki "sender" parametresini açık olarak **Button** tipine dönüştürüp tanımladığımız değişkene aktardık. En son olarak bu metodun işletilmesine neden olan Button nesnesinin başlığını yani Text özelliğini okuyup MessageBox sınıfının Show() metodu ile ekrana getirdik.

Form veya diğer nesnelerle ilgili olarak Visual Studio'nun yardımıyla hazırlanan metotların dışında yapmak istediğiniz işlemler için kendiniz metot hazırlamak durumunda kalacaksınız. Şimdi üzerinde çalıştığımız projedeki formun kodlandığı "Form1" sınıfı içinde "Mesaj_yaz" adında geriye değer göndermeyen bir metot hazırlayacağız.



Bu metodun başına erişim belirleyici olarak **public** veya **private** yazılmadığı için metodun private yani sınıfa özel olduğuna karar verilir. Bu metodun form tıklandığında işletilmesini istediğimiz için "Form1" sınıfının Click olayını temsil eden metoduna **Mesaj_yaz()** metodunu işletecek bir satır yazacağız.



Visual Studio ile gelen Windows Forms Application şablonu ile hazırlanan projelerde en azından System.Windows.Forms'de bulunan "Form" sınıfının mirasçısı bir sınıf bulunmaktadır. Hazırlayacağınız metot, ilgili Form sınıfı ve üyeleri üzerinde işlem yapmaya yönelikse metodu Form sınıfına ait kod dosyası içinde hazırlamak anlamlı olabilir. Hazırlamak istediğiniz metodun direk form sınıfı ve üyeleriyle ilgili değilse ayrı bir kod dosyasında hazırlamak daha mantıklıdır. Buradaki amacımız metotlar hakkında genel bilgi vermek olduğu için metodun hazırlandığı yerle pek ilgilenmedik.

Aşağıda verilen Form1_Click() metodundaki "Form1" sınıf adını, **Click** ise olayı işaret etmektedir. Visual Studio'dan yardım alarak hazırladığımız bu metoda istenen ad verilebilir. Bize sorarsanız Visual Studio'nun seçtiği adları olduğu gibi bırakmakta fayda vardır. Tabi formlar ve formlara yerleştirdiğiniz nesneler için Visual Studio'dan yararlanarak metot hazırlamadan önce nesnelere uygun adları vermiş olmanız gerekir.

```
void Mesaj_yaz()
{
 MessageBox.Show("Metot işletildi");
}
```

```
private void Form1_Click(object sender, EventArgs e)
{
 Mesaj_yaz();
}
```

Bu şekilde metot hazırlanıp çalışma anında form tıklanırsa "Form1" sınıfıyla ilgili olarak **Click** olayı meydana gelir ve bu olayla ilişkilendirilen Form1_Click() metodu işletilir. Bu metotta kendi hazırladığımız **Mesaj_yaz()** metodunu çağıran bir satır olduğundan bir diyalog kutusu içinde ekrana mesaj getirilir. Hazırlayıp Mesaj_yaz() adını verdiğimiz metot hakkında maddeler halinde birkaç cümle yazacağız:

- Metot için erişim belirleyici anahtar kelimelerden birisini (private, public) kullanmadığımız için bu metodun sınıfa özel, yani **private** olduğu varsayılır.
- Metodumuz herhangi bir parametreye sahip değildir. Bu nedenle metodun adına eklenen parantezlerin içine bir şey yazılmış değildir.
- Metot adından önce **void** anahtar kelimesine yer verdiğimiz için geriye bilgi göndermeyecektir.
- Bu metot void olmasaydı metot içinde **return** anahtar kelimesine yer vermek şart olurdu.
- Metoda ad seçilirken özenli davranılmamış. Çünkü C# kodu içinde hazırlanan metotlara verilen ad birden fazla kelimeden meydana geliyorsa alt tire(_) kullanılma alışkanlığı pek yoktur. Alt tire sadece Formlara yerleştirilen nesnelerin olaylarını temsil eden metotlarda kullanılmaktadır. Tabi bu bir kural değildir.



Bazı arkadaşlar bu kadar basit metot üzerinde konuşmaya değer mi diye sorabilirler. Böyle bir sorunun gelebileceğini öngördüğümüz için hazırlıklıyız. Bütün programcılık kitaplarında verilen örnek metotlar bir işlemi, bir kavramı anlatmaya, işlevini göstermeye yöneliktir. Buna rağmen birçok kaynakta ne yapıldığı bilmece gibi olan metotlara yer veriliyor. Yani okur önce kafa yorup bilmeceyi çözecek sonra asıl konuyu anlamaya çalışacaktır. Bizde ise verilen kodların basit olması ve kodun verilme amacından uzaklaşmamak esastır. Bu kitapta verilen kod ister uzun ister kısa olsun ilk bakışta ne yapıldığının anlaşılmasını hedefledik.

Metotlar İçin Parametre Tanımlamak

Metotların parametrelerinden söz etmeden önce sizi, bu kitabın "Değişken Tanımlamak ve Kullanmak" adlı bölümde yazılanları hatırlamaya davet etmek istiyoruz. Bu daveti yaptıktan sonra şu soruyu yöneltiyoruz: Değişkenler nerelerde tanımlanıyordu? Bu soruya verdiğimiz cevap aşağıda maddeler halinde sunulmuştur

- Değişkenleri en çok metotların içinde tanımladık. Örneğin şimdiye kadar konsol uygulamalarında Main() metodu içinde sıklıkla değişken tanımladık. Metotların içinde tanımlanan değişkenlere **yerel değişken** adı veriliyordu. Çünkü metot içinde tanımlanan değişkenler metodun dışında yaşamıyorlardı.
- Değişkenleri bazen if bloklarının içinde tanımladık. Bu değişkenler tanımlandıkları bloktan çıkıldığında tanımsız duruma düşüyorlardı.
- Değişkenleri tanımladığımız bir başka yer for ve foreach döngülerine ait parantezlerin içiydi. Bu değişkenler sadece döngü içinde yaşadıkları için döngüden

çıkıldığında tanımsız duruma düşüyorlardı. Kötü programcılığın işaretlerinden birisi, artık kullanılmayan değişkenlerin yaşamaya devam etmesidir. Bu nedenle sadece döngüde kullanılacak değişkenlerin döngüye ait parantezin içinde tanımlanması önemlidir.

- Değişkenleri tanımladığımız bir başka yer ise class bloğuydu. Class bloğu içinde tanımlanan değişkenlere "alan" deniliyordu ve class'taki bütün metotlarda yaşıyorlardı.
- Değişken tanımlayabileceğimiz en son yer metotlara ait parantezlerin içidir. Metotlara ait parantezlerin içinde tanımlanan değişkenlere **parametre** adı verilmekte ve bu değişkenlerin yaşam alanı ilgili metotla sınırlıdır.

Daha önceki konulardan bildiğiniz gibi .NET Framework ile gelen sınıfların (kontrollerin) olaylarını temsil etmek üzere hazırlanan metotlarda parametreler mutlaka kullanılıyor. Örneğin daha önce **TextBox** sınıfının **KeyPress** olayından söz edilmişti. KeyPress olayı çalışma anında kullanıcı klavyenin bir tuşuna bastığında meydana gelmekte ve söz konusu nesnenin bu olayını temsil etmesi amacıyla aşağıdaki gibi bir metot hazırlandıysa işletilmektedir.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
}
```

KeyPress olayını temsil eden metot adından sonra parantezlerin içinde 2 parametre tanımlıdır. "sender" adlı **object** tipindeki ilk parametre hakkında yukarıda Button sınıfının Click olayını temsil etmesi için hazırladığımız metot hakkında bilgi verirken biraz söz etmiştik. Bu metodun içinde metodun işletilmesine neden olan nesneyi (burada textBox1) elde edip üzerinde işlem yapmak istiyorsanız "sender" adlı parametreden yararlanabilirsiniz.

Bu metodun 2. parametresinin adı "**e**" iken tipi **KeyPressEventArgs**'dır. KeyPress olayı meydana geldiğinde hangi tuşa basılmışsa basılan tuşla ilgili bilgi "e" parametresi üzerinden metoda gönderilir. Burada vurgulamak istediğimiz nokta şudur: Metotların parametreleri int, long ve string gibi standart tipte olabileceği gibi **KeyPressEventArgs** gibi çok sayıda özelliğe sahip bir sınıf veya yapı olabilir.

Yukarıda hazırladığımız metot hem geriye değer göndermiyordu hem de parametreye sahip değildi. Şimdi "Mesaj_yaz()" metodunun bir parametreye sahip olmasını sağlayacağız. Metot dışarıdan bilgi alacaksa veya parametrelere sahip olacaksa bu parametreler değişken tanımlanıyor gibi parantezin içine yazılmalıdır. Metotların parametreleri metot içinde yaşayan yerel değişkenler gibidir.

```
void Mesaj_yaz(string txt)
 {
  MessageBox.Show(txt);
 }
```

Mesaj_yaz() metodu dışarıdan **string** tipte bilgi alacağı için parantezin içinde string bir değişken tanımladık. Yaşam alanı metotla sınırlı olan parametreler için erişim belirleyici anahtar kelimelerden herhangi birisi kullanıldığında hata meydana gelir. Çünkü buna gerek

yoktur, yaşam alanı zaten bellidir. Söz konusu metot 2. veya 3. kez işletildiğinde parametreler ve metot içindeki değişkenler yeniden tanımlanır, metodun sonuna gelinip metottan çıkıldığında hepsi bellekten silinir.

"Mesaj_yaz()" metodunu bu şekilde değiştirip bir parametreye sahip olmasını sağladıktan sonra işletmek üzere formun Click olayını temsil eden metodu aşağıdaki gibi düzenledik. Bu metodun çalışma anında form tıklandığında işletildiğini biliyorsunuz.

```
private void Form1_Click(object sender, EventArgs e)
{
 string str = textBox1.Text;
 Mesaj_yaz(str);
}
```

Buradan bakınca Mesaj_yaz() adını verdiğimiz metodu kodlarken ve işletilirken yabancısı olduğunuz bir şey görünmüyor. Bundan yararlanıp, yani anlaşılmayan veya henüz anlatılmayan bir şey yokken parametrelerle ilgili olarak geri planda gelişen bir işlemden söz edeceğiz. Araya girip yapacağımız bu ek açıklama bu bölümün ileriki sayfalarında anlatılacak bazı işlemlerin kolay anlaşılmasına yardımcı olacaktır. Konuyu iyice odaklamanız için yukarıda ayrı ayrı verdiğimiz Mesaj_yaz() metoduyla ilgili satırları burada bir arada vereceğiz.

```
void Mesaj_yaz(string txt)
{
 MessageBox.Show(txt);
}
private void Form1_Click(object sender, EventArgs e)
{
 string str = textBox1.Text;
 Mesaj_yaz(str);
}
```

Çalışma anında uygulamanın formu tıklandığında işletilen Form_Click() metodunda önce "textBox1" nesnesinin içeriğini okuyup string bir değişkene aktardık. Devamında bu string değişkeni Mesaj_yaz() metoduna parametre olarak verip metodu işlettik. Bu son cümlemizin dalgınlık sonucu buraya tekrar yazdığımızı düşünmüş olabilirsiniz. Gerçekte değil, bu cümleyi bilerek kurduk. Bundan sonrasının kolay takip edilmesini sağlamak için maddeler halinde yazacağız:

- Burada tanımlayıp TextBox'ın içeriğini aktardığımız "str" adlı değişken yerel bir değişkendir ve Form1_Click() metodundan çıkıldığı an ömrünü tamamlar.
- Programın işletimi Mesaj_yaz(str); yazılı satıra geldiğinde Form1_Click() metodundan çıkılıp Mesaj_yaz() metoduna geçilir. Ancak bu metoda geçilirken burada "str" adı verilen değişkenin bir kopyası alınıp götürülür. Bu kopya, Mesaj_yaz() metodunun burada "text" adını verdiğimiz parametresine yani yerel değişkene aktarılır.
- Şu anda Mesaj_yaz() metodunun içindeyiz, Form1_Click() metodundan gelen bilgi "text" adını verdiğimiz yerel değişkende duruyor. Bu değişkenin içeriğinde değişiklik yapılsa bile bundan Form_Click() metodundaki değişken etkilenmez. Tabi diyeceksiniz ki, Form_Click() metodundaki değişken zaten yerel bir değişkendi ve

ömrünü çoktan tamamladı. Form_Click() metodundaki değişken yerel bir değişken olmamış olsaydı bile değişen bir şey olmazdı. Bu konuda sizi ikna etmek için yukarıda verdiğimiz 2 metodu aşağıdaki gibi düzenledik.

```
void Mesaj_yaz(string txt)
{
 MessageBox.Show(txt);
}
string str;
private void button1_Click(object sender, EventArgs e)
{
 str = textBox1.Text;
 Mesaj_yaz(str);
}
```

Yaptığımız değişiklikte "str" adlı değişkeni class bloğuna alıp burada adı "Form1" olan sınıfın her yerinde yaşamasını sağladık. Bu değişiklikten sonra programın işletimi Mesaj_yaz() metodunun içinde iken "str" adlı değişkenle "txt" adlı değişken arasında hiçbir ilişki olmaz. Çünkü Form_Click() metodunun içinde iken "str" değişkenin adı şu şekilde kullanılsa bile;

```
Mesaj_yaz(str);
```

metoda "str" değişkenin kendisi değil bir kopyası gönderilir. Dolayısıyla Mesaj_yaz() metodu içinde "txt" değişkeninde yapılacak değişiklikten "str" değişkeni etkilenmez. Tabi, "str" değişkenini class bloğunda tanımladığımız için herhangi bir metotta doğrudan değiştirmek mümkündür.

Yukarıda hazırladığımız metot aynı tipte birden fazla parametreye sahip olsaydı bu parametreleri aralarına virgül koyarak tek tek tanımlamamız gerekirdi Yani aşağıdaki gibi bir seferde birden fazla parametreyi tanımlamak mümkün değildir.

```
void Mesaj_yaz(string txt1, txt2) // hata meydana gelir
 {
 }
```

Metoda ait parametrelerinin adını metodun gövdesinde değişken adı olarak kullanamayız. Çünkü parametreler, derleyici tarafından metot içinde tanımlanmış yerel değişkenler olarak görülmektedir. Dolayısıyla bir değişken adının metodun içinde ikinci kez kullanılmasına izin verilmez.

```
void Mesaj_yaz(string txt1, string txt2)
{
 string txt1;  // hata meydana gelir.
}
```

Mevcut bir metodu aynı sınıfın içinden çağırmak için metot adını yazmak yeterlidir. İşletilecek metot parametreye sahipse metodun çağrıldığı satırda metot adından sonra parantezlerin içine içerikleri metoda aktarılmak istenen değişkenler veya bilgiler yazılır.

Aynı sınıfın içindeki metotları işletme konusunda bir ayrıntı var: Aynı sınıfın içinde olsalar bile normal bir metodu static metodun içinde metot adıyla çağırıp işletmek mümkün değildir. Bu nedenle aşağıda verilen kod hataya neden olacaktır. Çünkü içinde metot işletmek istediğimiz Main() metodu **static** bir metottur.

```
class Program
{
 public static void Main()
 {
  Mesaj_yaz("Şampiyon Beşiktaş"); // hata meydana gelir.
  Console.ReadKey();
 }
 void Mesaj_yaz(string str)
 {
   Console.WriteLine(str);
 }
}
```

Burada Mesaj_yaz() metodunun çağrıldığı metot statik olmasaydı bu kod hata üretmezdi. Bu konuda muhtemel soru işaretleri gidermek için bu konsol uygulamasını aşağıdaki gibi değiştirdik. Bundan şu sonucu çıkarabiliriz: Statik metotların içinde işletmek istediğiniz metot statik değilse o metodu ancak nesne üzerinden işletebiliriz.

```
public class Program
{
 public static void Main()
 {
  Program nesne = new Program();
  nesne.Mesaj_yaz("Şampiyon Beşiktaş");
  Console.ReadLine();
 }
 void Mesaj_yaz(string str)
 {
   Console.WriteLine(str);
 }
}
```

Metotların Geriye Bilgi Göndermeleri

Yukarıda hazırladığımız metot geriye bilgi göndermiyordu. Bu nedenle metot adından önce **void** anahtar kelimesini kullandık. Şimdi hazırlayacağımız metodun kendisine parametre olarak gönderilen int tipindeki 2 sayıyı toplayıp sonucu int olarak geriye göndermesini sağlayacağız. Bu amaçla metot adının önüne **int** yazdık.

```
int Topla(int number1, int number2)
{
 int sonuc = number1 + number2;
 return sonuc;
}
```

Metot geriye ne tür bilgi gönderecekse o tipin metot adının önüne yazılması gerekmektedir. Bu metot geriye int tipinde bilgi göndereceği için metot adının önüne **int** yazdık. Metotlarda geriye bilgi gönderme **return** deyimi ile yapılmaktadır.

Burada Topla() adını verdiğimiz metodu bu şekilde hazırladıktan sonra işletmek üzere üzerinde çalıştığımız projedeki formun Form1_Click() metodunu aşağıdaki gibi düzenledik.

```
private void Form1_Click(object sender, EventArgs e)
{
 int i, j, yekun;
 i = 1250;
 j = 525;
 yekun = Topla(i, j);
}
```

Daha önce "Değişkenler" adlı bölümde ifade edildiği üzere C# programlama dilinde sayısal bilgiler üzerinde işlem yapılırken int ve double varsayılan tiplerdir. Benzer durum return anahtar kelimesiyle geriye gönderilen bilgilerde karşımıza çıkar. Bu konuyu açığa çıkarmak için yukarıda verdiğimiz Topla() metodunun int olan parametrelerinin tipini byte olarak değiştirdik. Ayrıca geri dönüş tipini long yaptık.

```
long Topla(byte number1, byte number2)
{
 int sonuc = number1 + number2;
 return sonuc;
}
```

Metodun yeni halini dikkatlice incelerseniz parametrelerin byte, üzerinde işlem yapılan bilgilerin int, return anahtar kelimesi ile geriye gönderilen bilgi ise long tipindedir. Dolayısıyla bu metot dahilinde kapalı dönüştürme işlemleri yapılacaktır. byte tipindeki 2 bilgi toplandığında sonucun int olacağını biliyorsunuz. Sonra int tipindeki "sonuc" değişkeni kapalı olarak long'a dönüştürülüp geriye gönderilir. Bu nedenle metodu aşağıdaki gibi değiştirirsek değişen bir şey olmaz.

```
long Topla(byte number1, byte number2)
{
 int sonuc = number1 + number2;
 return (long)sonuc;
}
```

Benzer kapalı dönüştürme işlemi metodun parametreleri olarak sabit sayılar kullanıldığında karşımıza çıkar. Bu ayrıntıyı dikkatinize sunmak için Topla() metodunu ve metodun çağrıldığı diğer metodu aşağıdaki gibi düzenledik.

```
long Topla(int number1, byte number2)
{
 int sonuc = number1 + number2;
 return (long)sonuc;
}
private void button1_Click(object sender, EventArgs e)
{
 long yekun = Topla(100, 50);
 textBox1.Text = yekun.ToString();
}
```

Topla() metoduna ve çağrıldığı satıra yakından baktığınızda metodun geriye long bilgi gönderdiği ve 2 parametresinden birisinin int, diğerinin byte tipinde olduğunu görürsünüz. Bu hali ile Topla() metodu hata üretmeden işletilebilir. Ne zaman ki 2. parametre olarak byte değişkenlerin kapasitesini aşan sabit bir değerle Topla() metodu işletilmek istenirse Visual Studio anında hata rapor edip kodu derlemez.

```
private void button1_Click(object sender, EventArgs e)
{
 long yekun = Topla(100, 350); // hata meydana gelir.
 textBox1.Text = yekun.ToString();
}
```

Return Anahtar Kelimesi

void olmayan metotlarda **return** anahtar kelimesine yer verilmediği zaman hata meydana gelmektedir. Örneğin aşağıdaki gibi bir metot hazırlarsanız derleyici hata rapor eder.

```
public static void Main()
{
 Program nesne = new Program();
 nesne.Kareler(10, 20);
}
long Kareler(int x, int y) // hata meydana gelir.
{
 long sonuc = (x * x) + (y * y);
 Console.WriteLine(sonuc);
}
```

Konunun iyice anlaşılmasını sağlamak için şimdi başka bir metot hazırlayacağız. Konsol uygulaması dahilinde hazırlayacağımız bu metoda **int** tipinde bir bilgiyi parametre olarak vereceğimiz için metot adından sonra parantezlerin arasına "int x" yazdık. Bu metot geriye **double** tipinde bilgi göndereceği için metot adından önce double yazdık. Bu metodu, dahil ettiğimiz sınıfın örneğini almadan kullanmak istediğimiz için ayrıca **static** olmasını sağladık.

```
static double Karekok(int x)
{
 double kok = Math.Sqrt(x);
 return kok;
}
```

Burada kullanılan **Sqrt**() metodu .NET Framework ile gelen **Math** sınıfında tanımlıdır. Bu nedenle metot adından önce bu sınıfın adını yazdık. Bu metodu aşağıda verilen konsol uygulamasında kullandık.

```
class Program
{
 static void Main(string[] args)
 {
  Console.Write("Karekökü alınacak sayıyı girin :");
  string str = Console.ReadLine();
  int sayi = Convert.ToInt32(str);
  double sonuc = Karekok(sayi);
  Console.WriteLine(sonuc);
 }
```

```
 static double Karekok(int x)
 {
  double kok = Math.Sqrt(x);
  return kok;
 }
}
```

Bir class'a sahip bu uygulama derlenip çalıştırıldığında kullanıcıdan karekökü alınacak sayı istenir. Kesirli bir sayının karekökü alınmak istenirse hata meydana gelir. Çünkü bu metodu **int** tipinde bilgi almak üzere düzenledik. Meydana gelmesi muhtemel bu hatanın önüne geçmek için girilen sayının int tipine dönüştürülmesi veya bu örnekte kullanılan **Karekok**() metodunun geriye **double** bilgi göndermesi sağlanabilir.

Diğer yandan eksi bir sayının karekökünü almak mümkün değildir. Bu şekilde düzenlenen uygulama çalıştırılıp eksi bir sayı girildiğinde programın çalışması kırılmaz ama kullanıcının bu konuda bilgilendirilmesi gerekir. Bu amaçla hazırlamış olduğumuz metodu aşağıdaki gibi düzenledik. Buradaki amacımız return anahtar kelimesi hakkında bilgi vermek olduğu için yapılan işleme odaklanmadık.

```
static double Karekok(int X)
{
 double kok;
 if (Math.Sign(X) < 1)
  {
   return 0;
  }
 else
 {
  kok = Math.Sqrt(X);
  return kok;
 }
}
```

Sayısal bilginin negatif olup olmadığını öğrenirken **Math** sınıfının **Sign**() metodundan yararlandık. Sign() metodu parametre olarak aldığı bilgi negatif ise Math sınıfının Sqrt() metodu kullanılmadan geriye 0 gönderilmektedir. Metodun yeni halinde ilk return anahtar kelimesi ile geriye 0 gönderildikten sonra metodun geri kalan satırları işletilmez. Bunun anlamı şudur: **return** anahtar kelimesinin ikinci işlevi metottan çıkışı sağlamaktır.

Karekok() adını verdiğimiz metodu bu şekilde düzenledikten sonra bu metodu işlettiğimiz diğer sınıftaki Main() metodunu aşağıdaki gibi düzenledik. Karekok() metodundan geriye 0 gönderilmişse uygulamanın çalışmasını sona erdirmek üzere **Environment** sınıfın **Exit**() metodunu kullandık.

```
static void Main(string[] args)
{
  Console.Write("Karekökü alınacak sayıyı girin :");
  string str = Console.ReadLine();
  int sayi = Convert.ToInt32(str);
  double sonuc = Karekok(sayi);
  if (sonuc == 0)
  {
```

```
        Console.WriteLine("girdiğiniz sayi negatif");
        Console.ReadKey();
        System.Environment.Exit(0);
       }
       Console.WriteLine(sonuc);
}
```

Bu bölümde şimdiye kadar **return** anahtar kelimesiyle ilgili olarak söylenenleri özetlemek gerekirse şunlar söylenebilir:

- Hazırladığınız metodun geri dönüş tipini void olarak seçmediyseniz metodun içinde **return** anahtar kelimesine yer vermek zorundasınız.
- Metot içinde birden fazla yerde return'e yer verebilirsiniz. Bunlardan birisinin işletilmesinin garanti olması gerek. Yukarıda kullandığımız 2 return'den birisi şarta bağlanmış iken diğeri şarta bağlanmış değildi.
- Metodun işletimi return anahtar kelimesine geldiğinde metottan çıkılıp geri kalan satırlar işletilmemektir.
- Geri dönüş tipi void olmayan metotlarda return anahtar kelimesini kullanmak şart olmasının yanında geri dönüş tipi void olan metotlarda return anahtar kelimesine yer verilebilir. Tabi bu durumda geriye gönderilecek bilgi olmadığı için return anahtar kelimesinin yanına bir şey yazılmaz.

Dizi Değişkenleri Parametre Olarak Kullanmak

Metoda göndermek istediğiniz bilgileri bir dizi değişkende toplayıp bu dizi değişkeni metoda parametre olarak verebilirsiniz. Bu işlemin nasıl yapıldığını basit bir örnek üzerinde anlatacağız. Bu amaçla aşağıda verilen konsol uygulamasında string tipte 5 elemanlı bir dizi değişken tanımlayıp elemanlarına bilgi aktardık.

```
public static void Main()
{
 string[] basket = new string[4];
 basket[0] = "Fenerbahçe";
 basket[1] = "Anadolu Efes";
 basket[2] = "Türk Telekom";
 basket[3] = "Karşıyaka";
 Takim_yaz(basket);
}
static void Takim_yaz(string[] takimlar)
{
  foreach(string takim in takimlar)
  {
   Console.WriteLine(takim);
  }
}
```

Metoda dizi değişken aracılığıyla bilgi gönderileceği için parantezin içinde parametre olarak kullanılacak dizi değişkeni tanımlamak gerekir. "Takim_yaz" adını verdiğimiz metot bu şekli ile dizi değişkenin içeriğini **WriteLine**() ile konsola yazacaktır. Metotlara dizi değişkenler parametre olarak verilebildiği gibi metotlar geriye dizi değişken gönderebilir. Bu konuda bilgi vermek için aşağıdaki gibi bir metot hazırladık.

```
private string[] Liste()
{
 string[] dizi = new string[3];
 dizi[0] = "Fenerbahçe";
 dizi[1] = "Galatasaray";
 dizi[2] = "Beşiktaş";
 return dizi;
}
```

Geriye string tipte dizi değişken gönderen bu metotta dizi değişken tanımlayıp bilgi aktardık. En son olarak hazırladığımız dizi değişkeni return anahtar kelimesi ile geriye gönderdik.

Dizi değişkenleri parametre olarak kullanılmasıyla ilgili olarak dikkatinize sunmak istediğimiz bir ayrıntı ve bir önerimiz var:

- Dizi değişkenlerin referans tipler olduğunu ve bir metoda parametre olarak dizi değişken verildiğinde metoda dizinin bellekteki adresinin gönderildiğini biliyorsunuz. Bu nedenle metot içinde, parametre olarak gelen dizi değişkende yapılacak değişikliklerden dizi değişkenin kendisi de etkilenir.
- Hazırlamak istenen metot aynı tipte çok sayıda parametreye sahipse, metodu hazırlarken çok sayıda parametre yerine dizi değişken parametre olarak kullanılabilir. Bu metoda parametre olarak gönderilmek istenen bütün bilgiler bir dizi değişkende toplanıp öyle metoda gönderilmesi önerilmektedir.

Metotların Geriye Birden Fazla Bilgi Göndermesi

Bazen metodun geriye birden fazla değer göndermesi istenir. C#'ın 4.0 sürümünden itibaren bu konuda programcılara kolaylık sağlayan gelişmeler olsa bile konu üzerinde düşünmenizi sağlamak için başlangıçta 4.0 sürümüyle gelen yeniliklerden haberdar değilmişiz gibi davranacağız. Bu amaçla aşağıdaki gibi bir metot hazırladık. Bu metot kendisine parametre olarak verilecek sayının hem karesini hem de küpünü geriye gönderecektir.

```
static int Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
}
```

C# 4.0 öncesi için konuşmak gerekirse, bu metot dahilinde yapılan işlem sonucu elde edilen 2 bilgiyi geriye göndermek için başvurulabilinecek birden fazla yöntem vardır. İlk aklımıza gelen, int tipinde 2 alana sahip bir sınıf hazırlayıp metodun bu sınıf tipinde nesneyi geriye göndermesini sağlamaktır. Bu amaçla aşağıdaki gibi bir sınıf hazırladık.

```
public class Araci
{
 public int alan1;
 public int alan2;
}
```

Metodumuz geriye 3 bilgi gönderecek olsaydı bu sınıftaki alanların sayısını 3 yapardık. Metodumuzun geriye birden fazla bilgi göndermesini sağlarken yararlanacağımız sınıfı bu şekilde hazırladıktan sonra metodumuzda değişikliğe gittik. Metodumuz geriye "Araci" tipinde bir nesne gönderecektir.

```
static Araci Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 Araci nesne = new Araci();
 nesne.alan1 = kare;
 nesne.alan2 = kup;
 return nesne;
}
```

Metodun geriye göndereceği 2 bilgiyi "Araci" adını verdiğimiz tipteki nesnenin alanlarına aktardık. Şimdi sırada uygulamanın Main() metodu içinde bu metodumuzu işletmek var.

```
class Program
{
 static void Main(string[] args)
 {
  Araci A = Karekup(4);
  int karesi = A.alan1;
  int kupu = A.alan2;
  Console.WriteLine("Sayının Karesi : " + karesi);
  Console.WriteLine("Sayının Küpü  : " + kupu);
 }
 static Araci Karekup(int X)
 {
  int kare = X * X;
  int kup = X * X * X;
  Araci nesne = new Araci();
  nesne.alan1 = kare;
  nesne.alan2 = kup;
  return nesne;
 }
}
```

Main() metodu içinde önce "Araci tipinde bir değişken tanımladık ve Karekup() adını verdiğimiz metodun geriye gönderdiği nesneyi bu değişkene aktardık. Devamında bu nesnenin alanlarına erişim sağladık. Böylece Karekup() metodumuz geriye 2 bilgi göndermiş oldu. Bu metottan geriye birden fazla değer gönderirken class yerine Struct hazırlayabilirdik.

Şimdi benzer işlemi yani metodun geriye birden fazla değer gönderebilmesini sağlarken dizi değişkenlerden yararlanacağız. Bu amaçla yukarıda hazırlayıp kullandığımız Karekup() metodumuzu aşağıdaki gibi düzenledik.

```
static int[] Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 int[] dizi = new int[2];
 dizi[0] = kare;
 dizi[1] = kup;
 return dizi;
}
```

Bu metotta önce metodun parametresinin karesini ve küpünü bulup int tipindeki değişkenlere aktardık. Ardından 2 elemanlı int tipinde bir dizi değişken tanımladık. Çünkü metodumuzun geriye 2 bilgi göndersin istiyoruz. Devamında "kare" ve "kup" değişkenlerin içeriklerini bu dizi değişkenin elemanlarına aktarıp return anahtar kelimesi ile geriye gönderdik. Bu metodun yeni halinden nasıl yaralandığımızı aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 int[] pars = Karekup(10);
 int karesi = pars[0];
 int kupu = pars[1];
 Console.WriteLine("Sayının Karesi : " + karesi);
 Console.WriteLine("Sayının Küpü  : " + kupu);
}
```

Metodun geriye birden fazla değer göndermesini sağlarken ref ve out anahtar kelimelerinden yararlanabiliriz. Bu 2 anahtar kelime hakkında henüz bilgi verilmediği için bu konuda bilgiyi bu bölümün ileriki sayfalarında bulabilirsiniz.

İsteğe Bağlı ve İsim Verilmiş Parametreler

C#'ın ilk üç sürümünde isteğe bağlı parametre desteği yoktu. C# 4.0'dan itibaren bu destek verilmeye başlandı. İsteğe bağlı parametrelerin nasıl kullanıldığını aşağıda görebilirsiniz. Kullanılması isteğe bağlı parametreye genellikle ilgili tipin varsayılan değeri aktarılmaktadır.

```
private int Topla(int number1, int number2=0, int number3=0)
{
 return number1 + number1 + number3;
}
```

Bu şartlarda bu metot 1 veya 2 parametre ile çağrılırsa eksik parametrenin 0 olduğu varsayılır. Bu metodun 1. parametresi isteğe bağlı olmadığı için bu metot parametresiz çağrılamaz. Örneğin bu metot aşağıdaki gibi işletildiğinde 3. parametrenin 0 olduğu varsayılır.

```
private void button1_Click(object sender, EventArgs e)
{
 int n1 = 100;
 int n2 = 200;
 int sonuc = Topla(n1, n2);
}
```

Bu örnekte "n2" adını verdiğimiz değişkenin 2. veya 3. parametrenin yerini tutmasının önemi yoktur. Başka bir deyişle eksik parametrenin hangisi olduğunun burada önemi yoktur. Ancak bazen metot çağrılırken kullanılan veya kullanılmayan parametrelerin hangileri olduğu önemlidir. Bu ayrıntıdan söz etmek için aşağıdaki gibi 3 parametreli bir metot hazırladık.

```
private string Birlestir(string ad, string tlf, string adres)
{
 return "Adı :" + ad + " Telefonu:" + tlf + " Adresi:" + adres;
}
```

Bu metot kendisine parametre olarak verilen bilgileri string sabitlerle birleştirip geriye göndermektedir. Bu metot aşağıdaki gibi 3 parametre ile çağrılırsa sorunsuz çalışır.

```
private void button1_Click(object sender, EventArgs e)
{
 string ad1 = textBox1.Text;
 string tel1 = textBox2.Text;
 string adres1 = textBox3.Text;
 string sonuc = Birlestir(ad1, tel1, adres1);
 MessageBox.Show(sonuc.ToString());
}
```

Şimdi "Birlestir()" adını verdiğimiz bu metodun 2. ve 3. parametrelerinin isteğe bağlı olmasını sağlayacağız. Başka bir deyişle bu metot tek parametre ile işletilmek istendiğinde 2. ve 3. parametrenin boşluk olduğu varsayılacaktır.

```
private string Birlestir(string ad, string tlf="",
                string adres="")
{
 return "Ad :" + ad + " Telefon:" + tlf + " Adres:" + adres;
}
```

Bu şartlarda bu metot 2 parametre ile işletilirse sorunlar yaşanabilir. Örneğin formdaki 1. ve 3. TextBox'a bilgi girilip bu metot aşağıdaki gibi işletilirse adres bilgisinin başına "Telefon" sabit bilgisi yazılır.

```
private void button1_Click(object sender, EventArgs e)
{
 string ad1 = textBox1.Text;
 string tel1 = textBox2.Text;
 string adres1 = textBox3.Text;
 string sonuc = Birlestir(ad1, adres1);
}
```

Bu sorunun çözümü, metodu çağırırken parametreleri isimlendirmektir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Bu ayarlamadan sonra Birlestir() metodu işletilirken 2. parametre olarak kullanılan bilgi "adres" adlı 3. parametre ile eşleştirilir.

```
private void button1_Click(object sender, EventArgs e)
{
 string ad1 = textBox1.Text;
 string tel1 = textBox2.Text;
 string adres1 = textBox3.Text;
 string sonuc = Birlestir(ad1, adres:adres1);
 MessageBox.Show(sonuc.ToString());
}
```

Bu şekilde parametrelere isim verilmesine **Named Arguments** denilmektedir. Parametre isimlendirmede, ilgili metot işletilirken parametrenin içeriğinden önce parametrenin adı yazılmaktadır. Bu kullanımda "adres", metot hazırlanırken parametreye verilen ad iken, "adres1" parametrenin içeriğiyle ilgili değişkendir. Tabi Named Arguments'lerden yararlanıldığında işletilmek istenen metodun parametrelerinin adları önemli olmaya başlar. Metodun parametre adlarıyla, metot çağrıldığında kullanılan adın aynı olması gerekir.

Yukarıda **Named Arguments** imkanından mecbur kaldığımız için yararlandık. Bize sorarsanız yazdığınız kodun okunurluğuna katkı yapmak istiyorsanız parametrelere ad vermekte fayda vardır.

Değer ve Referans Parametreleri - Ref ve Out Anahtar Kelimeleri

.NET dillerinde değer tipleri (int, double, byte vs.) metotlara kopyalanarak geçirilirler. Yani metodun içinde değişkenin birebir yeni bir kopyası oluşturulur. Ancak bazı durumlarda değer tiplerini de referans yoluyla metotlara geçirmek isteyebiliriz. C ve C++ dillerinde değer tiplerini referans yolu ile geçirmek için pointer'lardan yararlanılıyordu. Yani metotlara değişkenlerin bellekteki adresleri geçiriliyordu. Dolayısıyla metot içinde ilgili değişken üzerinde her türlü işlemi yapmak mümkün hale geliyordu.

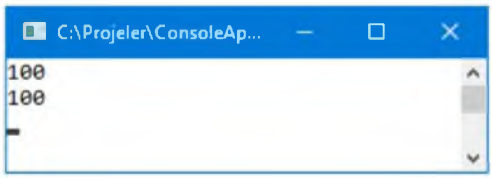
C# programlama dilinde bu işlem yapılırken **ref** anahtar kelimesi kullanılıyor. ref anahtar kelimesi değer tiplerinin metotlara referans yolu ile geçirilmesini sağlıyor. Yani bir parametrenin önüne ref anahtar kelimesi yazıldığında metoda o parametrenin içeriği değil bellekteki adresi gönderilir. Metot içinde bu adresten yararlanılıp söz konusu parametrenin işaret ettiği bellek bölgesi okunup yazılabilir.

Şimdiye kadar metotlarla ile ilgili olarak verilen örneklere göre int, char, long vs gibi belleğin **stack** adı verilen kısmında saklanan bilgiler metotlara **değer** olarak gönderilmektedir. Başka bir anlatımla parametre olarak herhangi bir metodun içine bir bilgi (değişken) gönderilmek istendiğinde o değişkenin bir kopyası çıkarılıp öyle metoda gönderilir. Metot dahilinde o kopyada yapılacak değişiklikten asıl kopya etkilenmez. Ayrıca metodun çalışması bitince kopya olarak metoda gönderilen değişkenler yerel oldukları için yok olurlar. Şimdi bu cümleleri anlaşılır kılmak için bir örnek üzerinde konuşacağız.

Referans parametreler hakkında bilgi vermek üzere hazırladığımız konsol uygulamasında Main() metodu içinde önce "sayi" adında **int** tipinde bir değişken tanımladık. Ardından bu değişkene değer aktarıp içeriğini **WriteLine**() metodu ile konsola yazdık. Devamında bu değişkenin bir kopyasını uygulamadaki "Procedure1" adını verdiğimiz metoda gönderip orada içeriğini değiştiriyoruz.

```
public static void Main()
{
 int sayi = 100;
 Console.WriteLine(sayi);
 Procedure1(sayi);
 Console.WriteLine(sayi);
}
static void Procedure1(int X)
{
 X = 200;
}
```

Bu uygulamada Main() metodu sınırları içinde tanımlanıp 100 değerini içeren "X" adlı yerel değişkenin içeriği "Procedure1" adını vermiş olduğumuz metotta 200 olarak değiştirilmektedir. Bu şartlarda uygulama derlenip çalıştırıldığında ekrana 2 kez 100 yazılır.



Bu olaya günlük hayattan karşılık bulmak istersek şöyle bir örnek verebiliriz: Elinizdeki bir evrakın fotokopisini çekip birisine veriyorsunuz. O kişi o kopyada sonradan değişiklik yapıyor. Bu şartlarda evrakın sizdeki asıl kopyası yapılan değişiklikten etkilenmez. Yukarıda verilen örnekte yapılan budur; 100 sayısı metoda gönderiliyor. Metotta bu değer 200 ile değiştiriliyor ancak elinizdeki sayı bundan etkilenmiyor.

Burada metoda parametre olarak gönderdiğimiz değişken referans tip olsaydı durum değişirdi. Bu konuyu netleştirmek için üzerinde çalıştığımız uygulamada aşağıdaki gibi sadece 2 public alana sahip bir sınıf hazırladık.

```
public class Takimlar
{
 public string takim;
 public int kurulus;
 public Takimlar(string ad, int yil)
 {
  takim = ad;
  kurulus = yil;
 }
}
```

Sınıfımızı bu şekilde hazırladıktan sonra uygulamanın Main() metodunda bu sınıfın örneğini hazırlayıp alanların içeriklerini konsola yazdırdık.

```
class Program
{
 public static void Main()
  {
   Takimlar t1 = new Takimlar("Beşiktaş", 1903);
   Console.WriteLine(t1.takim);
   Console.WriteLine(t1.kurulus);
  }
}
public class Takimlar
{
 public string takim;
 public int kurulus;
 public Takimlar(string ad, int yil)
  {
   takim = ad;
   kurulus = yil;
 }
}
```

Buraya kadar yazılanlarda sizin için yeni bir şey yoktur. Konu metotların parametreleri olduğunda değer tipler ile referans tiplerde karşılaşılan farklılığı göstermek için yukarıda

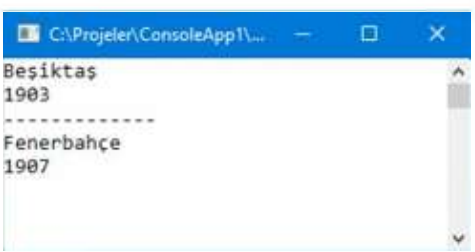
verdiğimiz ilk sınıfta "Degistir" adında bir metot hazırladık. Bu metot işlev bakımından yukarıda int tipli parametre kullanırken hazırladığımız Procedure1'den farklı değildir.

```
static void Degistir(Takimlar t)
{
 t.takim = "Fenerbahçe";
 t.kurulus = 1907;
}
```

Bu metot, kendisine parametre olarak gönderilen "Takimlar" tipindeki nesnenin alanlarını değiştirmekle yetiniyor. Şimdi yukarıda verdiğimiz Main() metoduna ekleme yapıp önce Degistir() metodunu, ardından Main() metodu içinde hazırlayıp "t1" adını verdiğimiz "Takimlar" tipindeki nesnenin alanlarını tekrar konsola yazacağız.

```
public static void Main()
{
 Takimlar t1 = new Takimlar("Beşiktaş", 1903);
 Console.WriteLine(t1.takim);
 Console.WriteLine(t1.kurulus);
 nsole.WriteLine("-------------");
 Degistir(t1);
 Console.WriteLine(t1.takim);
 Console.WriteLine(t1.kurulus);
}
```

Yukarıda bu işlemin benzerini int tipindeki parametre için yaptığımızda Main() metodunda yaptığımız her iki yazma işleminde aynı sonucu almıştık. Burada ise metoda parametre olarak gönderdiğimiz "Takimlar" tipindeki nesne referans bir tip olduğu için metoda bu nesnenin kopyası değil adresi gönderilir. Dolayısıyla burada Degistir() metodunda yapılan değişiklikten Main() metodundaki "t1" nesnesi etkilenir. Aşağıda verilen ekran görüntüsünü bu uygulamayı çalıştırdıktan sonra aldık.



Bazen çağrılan metoda içeriği gönderilen değer tipli değişkenin metotta yapılan işlemlerden etkilenmesi istenir. Bu gibi durumlarda değer tipli parametrenin başına **ref** anahtar kelimesinin eklenmesi gerekir. ref deyiminin nasıl kullanıldığını aşağıda görebilirsiniz.

```
public static void Main()
{
 int sayi = 100;
 Console.WriteLine(sayi);
 Procedure1(ref sayi);
 Console.WriteLine(sayi);
}
```

```
static void Procedure1(ref int X)
{
 X = 200;
}
```

Bu şekilde düzenlenen uygulama ile ilgili olarak şu yorum yapabiliriz:

- Procedure1() adında ve int tipinde bir parametreye sahip bir metodumuz var.
- Değer tipli bu parametre aracılığıyla üzerinde işlem yapılmak istenen bilginin kopyası yerine referansının beklenmesini sağlamak için metotta parametrenin önüne **ref** anahtar kelimesi yazıldı.
- Referans özellikli parametre içeren metotlar işletilirken metoda değişkenin kopyası yerine adresinin gönderilmesini sağlamak için ref anahtar kelimesi kullanıldı. Çünkü metot içerik değil referans beklemektedir.
- ref anahtar kelimesi hem ilgili metotta parametrenin hem de metodun çağrılıp işletildiği satırda parametrenin önüne yazılmalıdır.

ref anahtar kelimesi sayesinde değişkenlerin içerdikleri bilgiler metotlara içerik olarak değil yani kopyası hazırlanmayıp sadece **referansı** gönderilir. Başına ref yazılan parametrelere "referans parametreleri" adı verilmektedir. Referans derken "sayi" değişkenin bellekteki adresinden söz etmekteyiz.

Bu şartlarda bu konsol uygulaması derlenip çalıştırıldığında ekrana önce 100 ardından 200 yazılır. Çünkü "sayi" değişkeni metoda referans olarak gönderildi ve "Procedure1" adını vermiş olduğumuz metotta bu bellek adresinde tutulan bilgi değiştirildi. Dolayısıyla aynı değişkenin içeriğine tekrar bakıldığında değişmiş olduğu görülür.

Yukarıda ref anahtar kelimesini anlatmak amacıyla hazırladığımız uygulamada int tipindeki "sayi" değişkenine ilk değer vermeden metodu işletmiş olsaydık hata meydana gelirdi. Bu özelliği hatırlatmak için yukarıda verilen konsol uygulamasını aşağıdaki gibi değiştirdik.

```
public static void Main()
{
 int sayi;
 Procedure1(ref sayi);
 Console.WriteLine(sayi);
}
static void Procedure1(ref int X)
{
 X = 200;
}
```

Bu kodu düz mantıkla incelediğinizde hata meydana gelmemesi gerekir diye düşünebilirsiniz. Çünkü referans olarak gönderilen değişkene metotta zaten bilgi aktarılıyor ve bilgi aktarıldıktan sonra bu bellek adresinin içeriği ekrana yazılmaya çalışılıyor. Buna rağmen bu uygulama çalıştırıldığında hata meydana gelir. Bu tip hataların önüne geçmek için **ref** yerine **out** anahtar kelimesini kullanabilirsiniz. Out anahtar kelimesinin nasıl kullanıldığını aşağıda görebilirsiniz.

```
public static void Main()
{
 int sayi;
 Procedure1(out sayi);
 Console.WriteLine(sayi);
}
static void Procedure1(out int X)
{
 X = 200;
}
```

C# 7.0'da ref ve out Yenilikleri

Yukarıda ref anahtar kelimesi hakkında bilgi verirken bir metoda referansı gönderilmek istenen değişkenin daha önce tanımlanıp ilk değerin aktarılması gerektiğini söyledik. Devamında söz konusu değişkenin referansı bir metoda gönderilirken **out** anahtar kelimesi kullanıldığında değişkeni tanımlamanın yeterli olduğunu, ilk değeri aktarmanın şart olmadığını yazdık.

C# 7.0 versiyonu ile birlikte out anahtar kelimesinin kullanın şekli esnetildi ve değişken tanımının metodun çağrıldığı satırda yapılmasının önü açıldı. Bu nedenle yukarıda out anahtar kelimesiyle ilgili olarak verilen kod aşağıdaki düzenlenirse C# 7.0 derleyicisi bir itirazda bulunmaz.

```
public class Program
{
 static void Main(string[] args)
 {
  Procedure1(out int sayi);
  Console.WriteLine(sayi);
  Console.ReadLine();
 }
 static void Procedure1(out int X)
  {
    X = 200;
  }
}
```

Bu değişiklik sonucu referans tipli parametreye sahip olan metoda ait parantezin içi ile metodun çağrılma satırı birbirinin aynı olmuş oldu. Başka bir deyişle out parametresi olarak kullanılacak değişkenler gerek duyulan yerde tanımlanabilinecek. Bu değişiklik TryParse() ve TryDequeue() kullanımlarında da işe yarayacaktır.

C# programlama dilinin 7.0 versiyonunda değişen sadece out anahtar kelimesinin işleyişi değildir. Ayrıca ref anahtar kelimesine yeni bir kullanım şekli getirildi. Şimdi yukarıda ref anahtar kelimesiyle ilgili olarak söylediklerimizi biraz hatırlayalım:

İşletmek istediğimiz metoda, elimizdeki yerel değişkenin kopyası yerine referansını göndererek metot dahilinde yerel değişkene referansı ile erişim imkanı ediyorduk. C# 7.0'da işletilmek istenen metoda referansı gönderilen sadece değişkenler değildir; istenirse ref anahtar kelimesi geriye gönderilen tipin önüne yazılabiliniyor. Bu karışık durumu açıklamak için yukarıda ref anahtar kelimesini açıklarken hazırladığımız kodu tekrar buraya aldık.

```
public class Program
{
 static void Main(string[] args)
 {
  int sayi = 100;
  Procedure1(ref sayi);
  Console.WriteLine(sayi);
  Console.ReadLine();
 }
 static void Procedure1(ref int X)
  {
    X = 200;
  }
}
```

Yukarıdaki sayfalarda burada yapılan işlemlerden söz edilmiş olsa bile tekrar üzerinde durmakta bizce fayda vardır. Burada "sayi" adını verdiğimiz int tipindeki değişkenin kopyasını değil referansını metoda gönderiyoruz. Dolayısıyla burada "Procedure1" adını verdiğimiz metotta kendisine parametre olarak referansı gönderilen değişken üzerinde yapılan işlemler bir bakıma Main() metodunda "sayi" adını verdiğimiz yerel değişken üzerinde işlem yapmakla aynı anlama gelmektedir.

Bu konunun iyice netleşmesi için yukarıdaki sayfalarda dizi değişkenler aracılığıyla metotların geriye birden fazla değeri nasıl gönderdiğini anlatırken verdiğimiz örneği biraz değiştirip tekrar vereceğiz. Önce birden fazla değeri geriye göndermesini istediğimiz metodu veriyoruz.

```
static void Karekup(int[] pars, int X)
{
 int kare = X * X;
 int kup = X * X * X;
 pars[0] = kare;
 pars[1] = kup;
}
```

Bu metodu incelerseniz geriye değer göndermediğini ve birisi int tipinde dizi değişken olmak üzere 2 parametreye sahip olduğunu, metoda 2. parametre olarak verilen sayının karesi ve küpünün bulunup dizi değişkenin elemanlarına aktarıldığını görürsünüz. Dizi değişkenler referans tipler olduklarından bu metot aşağıdaki gibi işletildiğinde herhangi bir sorun çıkmaz ve geriye birden fazla değer gönderme işlemi başarılmış olunur.

```
static void Main(string[] args)
{
 int[] pars = new int[2];
 Karekup(pars, 10);
 int karesi = pars[0];
 int kupu = pars[1];
}
```

Daha önceki konulardan bildiğiniz gibi ref ve out anahtar kelimelerine en çok referans tipli olmayan değişkenler söz konusu olduklarında gerek duyulmaktadır. Bütün bunlardan söz etmemizin nedeni, ref anahtar kelimesine C# 7.0 ile kazandırılan yeni işlevlerdir. Bu yeni işlevler biraz zor anlaşıldığı için sizi hazırlamak ve eski bildiklerinizi hatırlatmak istedik. Yeni

ref'ten söz etmek amacıyla ilk olarak yukarıda verdiğimiz "Procedure1" adlı metodun geriye "int" tipinde bilgi göndermesini sağladık. Metottaki void anahtar kelimesini silip yerine int yazınca **return** anahtar kelimesine yer vermek gerekti.

```
static int Procedure1(ref int X)
{
 return X = 200;
}
```

Bu durumda buradaki adıyla "X" parametresinin önüne ref yazmanın bir anlamı olmasa gerek. Çünkü zaten parametrenin son içeriği geriye gönderiliyor. Ancak buradaki amacımız başka. Çünkü metot geriye ne gönderirse göndersin metodun içinde burada "X" adını verdiğimiz değişkenin referansı elimizde olsun istiyoruz. Metoda gönderdiğimiz parametrenin başına ref yazmakla yetinmeyip metodun geriye göndereceği bilginin de başına ref yazacağız.

```
static ref int Procedure1(ref int X)
{
 return X = 200;
}
```

Bunu yapmakla diyoruz ki, nasıl ki parametre olarak gönderilenin bilgi olmayıp ilgili değişkenin referansıdır, bu kez geriye gönderilen herhangi bir bilgi olmayıp bir değişkenin referansı olacaktır. Özetle, bu metoda bir referans gelecek, gelen referansta tutulan bilgi üzerinde istenilen işlemler yapılacak, sonra geriye yine referans gönderilecek. Bu nedenle yani geriye referans gönderebilmek için **return**'den sonra yine ref anahtar kelimesine yer vermek gerekir. Bu amaçla Procedure1() adlı metodumuzu aşağıdaki gibi düzenledik.

```
static ref int Procedure1(ref int X)
{
 X = 200;
 return ref X;
}
```

Şimdi sırada bu metodu işletmek var. Tahmin edeceğiniz gibi işletmek istediğiniz metodun neresinde ref anahtar kelimesi kullanıldıysa işletilirken de aynı yerlerde ref'e yer vermek gerekecektir. Bu metodu nasıl işlettiğimizi aşağıda görebilirsiniz.

```
class Program
{
 static void Main(string[] args)
  {
   int sayi = 100;
   ref int sonuc = ref Procedure1(ref sayi);
   Console.WriteLine(sonuc);
   Console.ReadKey();
  }
  static ref int Procedure1(ref int X)
  {
   X = 200;
   return ref X;
  }
}
```

Dikkat ederseniz hazırladığımız metotta nerede ref kullanıldıysa işletirken de aynı yerlerde ref'e yer verdik. Ref anahtar kelimesinin yeni kullanım şekliyle ilgili olarak yaptıklarımız ve söylediklerimizi maddeler halinde özetleyeceğiz:

- "Procedure1" adını verdiğimiz metodun int tipindeki parametresinin önüne ref yazdığımız için bu metot işletildiğinde parametre olarak bir referans gönderilir.
- Metodumuz şimdiye kadar alıştığımızdan farklı olarak bir değişkenin kopyasını veya bilgiyi göndermek yerine söz konusu bilginin bulunduğu bellek lokasyonun adresi geriye gönderilecektir. Çünkü metodun geriye göndereceği bilginin tipini işaret eden int anahtar kelimesinden önce ref'e yer verdik.
- Metodun geri dönüş tipinin önüne ref yazdığımız için return anahtar kelimesinden sonra, geriye gönderilecek bilginin önüne ref yazmak gerekecektir. Bu sayede geriye bilgi değil ilgilenilen değişkenin referansı gönderilecektir.
- Bu özelliklere sahip metodu işletirken parametre olarak gönderilen bilgi veya değişkenin önünü ref yazmak gerektiği için ref yazdık. Ayrıca metodun geriye bilgi yerine referans göndereceği için geriye gönderilen referansı yakalarken ref anahtar kelimesinden yararlandık.

Şimdi konu üzerinde düşünüp yapılan işlemleri iyice kavramanız için ikinci bir örnek hazırlayacağız. Konu kolay izlensin diye hazırlayacağımız metodun önce ref'sız halini vereceğiz.

```
static string Buyut(string[] dizi, int index)
{
 dizi[index] = dizi[index].ToUpper();
 return dizi[index];
}
```

Bu metodu incelerseniz sizin için anlaşılmaz ve yeni bir yanının olmadığını hemen fark edersiniz. Metoda parametre olarak string bir dizi değişken ve int bir bilgi gönderiliyor. Sonra bu dizi değişkenin 2. parametrede işaret edilen elemanın içeriği büyük harfe çevrilip geriye gönderiliyor.

Buradaki amacımız ref anahtar kelimesini anlatmak olduğu için metodun geriye gönderdiği string bilginin, dolayısıyla return anahtar kelimesi kullanılarak geriye gönderilen değişkenin adının önüne ref yazdık.

```
static ref string Buyut(string[] dizi, int index)
{
 dizi[index] = dizi[index].ToUpper();
 return ref dizi[index];
}
```

Bu şartlarda bu metodu işletirken hem geriye gönderilen string bilginin hem de geriye gönderilen bilginin yani eşitliğin sol tarafına ref yazmak gerektiğini düşünerek bu metodumuzu aşağıdaki gibi işlettik.

```
static void Main(string[] args)
{
 string[] takimlar = new string[3];
 takimlar[0] = "Fenerbahçe";
 takimlar[1] = "Beşiktaş";
 takimlar[2] = "Galatasaray";
 ref string takim = ref Buyut(takimlar, 1);
 Console.WriteLine(takim);
}
```

Burada yani metodun çağrıldığı satırdaki her iki ref anahtar kelimesini silsek bile derleyici hata rapor etmez. Çünkü burada metoda gönderip üzerinde işlem yaptığımız dizi değişken zaten bir referans tiptir. Dizi değişkenlerin referans tipler olduklarıyla ilgilenmeyip Buyut() adını verdiğimiz metodun dizi değişken olan ilk parametresinin önünü de ref anahtar kelimesini yazdık.

```
static ref string Buyut(ref string[] dizi, int index)
{
 dizi[index] = dizi[index].ToUpper();
 return ref dizi[index];
}
```

Bu değişiklikten sonra bu metot çağrılırken dizi değişken olan ilk parametrenin önüne ref yazmak şart olur. Yukarıda dediğimiz gibi sanki dizi değişkenlerin referans tipler olduğundan habersizmiş gibi davranıp bu metodumuzu aşağıdaki gibi işlettik.

```
class Program
{
 static void Main(string[] args)
 {
  string[] takimlar = new string[3];
  takimlar[0] = "Fenerbahçe";
  takimlar[1] = "Beşiktaş";
  takimlar[2] = "Galatasaray";
  ref string takim = ref Buyut(ref takimlar, 1);
  Console.WriteLine(takim);
 }
 static ref string Buyut(ref string[] dizi, int index)
 {
   dizi[index] = dizi[index].ToUpper();
   return ref dizi[index];
 }
}
```

Yukarıda işaret edildiği üzere burada üzerinde işlem yaptığımız dizi değişken zaten referans bir tip olduğu için referansını buradaki adıyla Buyut() metoduna göndermek için ref anahtar kelimesine ihtiyacımız yoktur. Bu ayrıntıyı bir kenara bırakırsak bu metot hakkında şunları söyleyebiliriz:

- Sadece Main() metodu içinde yaşayabilen "takimlar" adını verdiğimiz dizi değişkenin referansını burada Buyut() adını verdiğimiz metoda gönderdik.

- Dolayısıyla Buyut() metodu içinde iken Main() metodunda yerel bir değişken olan dizi değişkenin referansı elimizdedir artık.
- Bu metotta dizi değişkenin tümü yerine sadece bir elemanın içeriğini değiştirdik.
- Metodumuzun geri dönüş tipi string olduğu için dizi değişken yerine bu dizi değişkenin içeriği değişen elemanın referansını geriye göndermekle yetindik.

Ref anahtar kelimesinin 7.0 ile gelen yeni halinin nasıl kullanıldığını böylece anlattıktan sonra gelelim yorumumuza: Yukarıda ref hakkında yazılanları okuyanlar daha önce C++ ile çalışmış olanların aklına sürekli pointerlar gelmiş olabilir. C#'ta pointer kullanmak yasak olmasa bile önerilmemesinin sonucu kaybedilen performans bu şekilde geri kazanılmış olunacaktır. Çünkü işlemler direk bellekteki kopyalar üzerinde yapılmaktadır. Bu performans artışı yazılan uygulamanın bellekle alış verişi çok fazla ise önemsenebilir.

Ref Anahtar Kelimesi Yardımıyla Geriye Bilgi Gönderilmesi

Yukarıdaki sayfalarda metotların geriye birden fazla bilgi gönderebilmesi için başvurulabilinecek 2 yöntemden söz etmiştik. Bir metodun geriye birden fazla bilgi gönderme ihtiyacı karşısında programcıların ilk aklına gelen yöntem genellikle referans tipli parametrelerdir.

Ref ve out anahtar kelimeleri kullanılarak bir metodun birden fazla bilgiyi nasıl geriye gönderdiğini anlatmak için yukarıdaki sayfalarda kullandığımız Karekup() metodunu aşağıdaki gibi düzenledik.

```
static void Karekup(int X, ref int kare, ref int kup)
{
 kare = X * X;
 kup = X * X * X;
}
```

Geriye değer göndermeyen metodumuz int tipinde 3 parametreye sahiptir. İlk parametre normal iken diğer ikisi referans tiplidir. Metotta 1. parametrenin karesi ve küpü bulunup metoda referansları gönderilen değişkenlere aktarıldı. Bu metodu nasıl işlettiğimizi aşağıda görebilirsiniz.

```
public static void Main()
{
 int sayi = 4;
 int kare = 0;
 int kup = 0;
 Karekup(sayi, ref kare, ref kup);
 Console.WriteLine(kare);
 Console.WriteLine(kup);
}
```

Referans tipler yardımıyla metodun geriye birden fazla değer göndermesini sağlarken "ref" yerine "out" anahtar kelimesini kullanabilirdik. Bu durumda burada referansları metoda gönderilen 2 değişkene ilk değer ataması yapmamız gerekmezdi. Out anahtar kelimesinden yararlanarak geriye birden fazla değer göndermesini istediğiniz metodun **async** bir metot olmaması gerekiyor.

Diğer yandan dizi değişkenler zaten referans tipler oldukları için **ref** deyimini kullanmaya gerek yoktur. Bu cümleyi somutlaştırmak için üzerinde çalıştığımız konsol uygulamasının yukarıda verdiğimiz Karekup() ve Main() metotlarını aşağıdaki gibi düzenledik. Bu kod işlev bakımından yukarıda ref anahtar kelimesinden yararlandığımızdan farklı değildir.

```
public static void Main()
{
 int sayi = 4;
 int[] dizi = new int[2];
 dizi = Karekup(sayi, dizi);
 Console.WriteLine(dizi[0]);
 Console.WriteLine(dizi[1]);
}
static int[] Karekup(int X, int[] dizi)
{
 dizi[0] = X * X;
 dizi[1] = X * X * X;
 return dizi;
}
```

Bu koddan anlaşılacağına göre dizi değişkenlerin referanslarını metotlara gönderirken ref anahtar kelimesini kullanmaya gerek yoktur. Ancak programcı yine de ref anahtar kelimesini kullanacağım derse derleyicinin buna bir itirazı olmaz. Bu nedenle bu kod aşağıdaki gibi düzenlenirse değişen bir şey olmaz.

```
public static void Main()
{
 int sayi = 4;
 int[] dizi = new int[2];
 dizi = Karekup(sayi, ref dizi);
 Console.WriteLine(dizi[0]);
 Console.WriteLine(dizi[1]);
}
static int[] Karekup(int X, ref int[] dizi)
{
 dizi[0] = X * X;
 dizi[1] = X * X * X;
 return dizi;
}
```

Hemen belirtelim, C# derleyicisinin referans tipli parametrelerin başına boş yere ref anahtar kelimesinin konulmasına itirazı yoktur ama dizi değişkenlerin referansı metoda gönderilirken out anahtar kelimesinin kullanılmasına izin vermiyor.

Burada ref ve out anahtar kelimeleri bağlamında dizi değişkenlerle ilgili olarak söylenenleri maddeler halinde özetleyeceğiz:

- Dizi değişkenler referans tipler (ister string, ister int dizisi olsun) oldukları için metoda parametreler aracılığıyla gönderildiklerinde dizi değişkenin kendisi yerine bellekteki adresi yani referansı gönderilir.
- Metot içinde parametre olarak gönderilen dizi değişken üzerinde işlem yapıldığında yapılan işlemden dizi değişkenin kendisi etkilenir.

- Yukarıda dizi değişkenimizi Main() metodu içinde hazırladık. Sonra bu dizi değişkenin referansını burada Karekup() adını verdiğimiz metoda gönderdik. Bu sayede metotta dizi değişkenin elemanlarına aktardığımız bilgileri Main() metodu içinde kullanma şansı bulduk.

Bu özetten sonra önemli bir ayrıntıdan söz edeceğiz: String değişkenler tıpkı dizi değişkenler gibi referans tipli olmasında rağmen metotlara ref ve out anahtar kelimeleri kullanılmadığı zaman int değişkenler gibi parametreler aracılığıyla metoda değer olarak gönderilirler. Çünkü C# programlama dilinde String değişkenler bu bağlamda istisnai bir durumdalar. Yani string değişkenler referans tipler olmalarına metotlara referansları değil kopyaları gönderilmektedir.

Tuple Sınıfı İle Metotların Geriye Birden Fazla değer Göndermesi

Yukarıdaki sayfalarda metotların geriye birden fazla değer gönderirken yararlanabileceğiniz 3 farklı teknikten söz ettik. C# 4.0 ile gelen Tuple sınıfıyla bu işlemler son derece kolaylaşmaktadır. C# 7.0'da Tuple sınıfının becerilerinde artış sağlandı. Ancak başlangıçta 7.0 ile gelen yeni özellikler yokmuş gibi Tuple sınıfından söz edeceğiz.

Yukarıdaki sayfalarda metotların geriye birden fazla değer gönderebilmelerini özel class ile sağlarken bu class'ı metodun geriye göndereceği bilgilerin sayı ve tiplerine bakarak hazırlamıştık. Uygulamamızda geriye birden fazla değer gönderen başka bir metot olsaydı ikinci bir sınıfı daha hazırlamamız gerekecekti. İşte Tuple sınıfı bu işlemler için programcıyı özel sınıf hazırlamak zorunda bırakmamak üzere C# 4.0 ile birlikte kullanıma sunulmuştur.

Tuple sınıfı generic bir sınıf olmasına rağmen bu bölümde anlatmayı tercih ettik. Çünkü Tuple sınıfı direk metotlarla ilgili bir sınıftır. Generic class konusu kafanızı meşgul ediyorsa bu kitabın Generic Class'larla ilgili bölümüne göz gezdirip ondan sonra Tuple sınıfının anlatıldığı bu sayfalara dönebilirsiniz.

Tuple sınıfı yardımıyla metodunuzun geriye birden fazla değer göndermesine imkan sağlamak istiyorsanız ilk yapmanız gereken işlem, metodun geri dönüş tipini **Tuple** olarak seçmenizdir. Bu amaçla yukarıdaki sayfalarda hazırladığımız metodumuzu aşağıdaki gibi düzenleyip geri dönüş tipi olarak Tuple'yi seçtik.

```
static Tuple Karekup(int X)
 {
 }
```

Tuple sınıfı generic bir sınıf olduğu için sınıf adından sonra diamand operatörlerine "<>" yer yerip sınıfın üzerinde işlem yapacağı, başka bir deyişle metodun geriye göndereceği veri tiplerini belirtmeniz gerekiyor. Burada "Karekup()" adını verdiğimiz metodumuzdan geriye int tipinde 2 bilgi göndermek istediğimiz için metodun geri dönüş tipi olan Tuple'yi aşağıdaki gibi işaret ettik.

```
static Tuple<int, int> Karekup(int X)
 {
 }
```

Metot dahilinde kaç farklı veri geriye gönderilmek isteniyorsa "<>" operatörlerin arasına o kadar veri tipi yazmak gerekir. Bu metodumuz dahilinde geriye gönderilecek bilgilerin birisi string olsaydı "<>" operatörlerin arasına ayrıca string yazardık. Tuple sınıfıyla ilgili olarak "<>"operatörlerinin arasına en fazla 8 adet veri tipi yazılabilmektedir. Tabi 8 olan veri tiplerinden birisini Tuple yaparak bu sayıyı katlamak mümkündür. Bu açıklamadan şunu çıkarabiliriz: Tuple sınıfı 8 generic parametreye sahip 8 yapıcı metodu bulunmaktadır.

Hazırlamak istediğiniz metot geriye sadece bir bilgi göndermesine rağmen hiç gerek yokken yine de Tuple sınıfını kullanmak isterseniz C# derleyicisinin buna bir itirazı olmuyor. Bu şekilde geri dönüş tiplerini tespit ettikten sonra metodun içinde yapacağımız işlemlerle ilgili satırları yazdık.

```
static Tuple<int, int> Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
}
```

Şimdi yapılması gereken, Tuple sınıfından yola çıkıp nesne hazırlamak ve bu 2 bilgiyi Tuple nesnesinin mevcut alanlarına aktarıp geriye göndermektir. Tuple tipinde nesneyi nasıl hazırladığımızı aşağıda görebilirsiniz.

```
static Tuple<int, int> Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 Tuple<int, int> deste = new Tuple<int, int>(kare, kup);
 return deste;
}
```

Metoda eklediğimiz satırdan fark edileceği gibi Tuple nesnesi hazırlanırken metottan geriye gönderilmek istenen bilgiler Tuple sınıfının yapıcı metoda parametre olarak verilmektedir. Geriye gönderilmek istenen bilgilerin sayısı ve tipleri metodun imzasında işaret edilmiş olanlarla uyumlu olmalıdır. Metodun imzasında geri dönüş tipi olarak "Tuple<int, int>" kullandığımız için Tuple nesnesi hazırlarken yapıcı metoda int tipinde 2 bilgiyi parametre olarak vermemiz gerekiyor.

Bu hazırlıktan sonra artık metot dahilinde elde edip geriye göndermek istediğimiz 2 int bilgiyi içeren Tuple nesnesini return anahtar kelimesiyle geriye gönderebiliriz. Tuple nesnesi yardımıyla int tipinde 2 bilgiyi geriye gönderen bu metodu üzerinde çalıştığımız uygulamanın Main() metodu içinde nasıl kullandığımızı aşağıda görebilirsiniz.

```
class Program
{
 static void Main(string[] args)
  {
   Tuple<int, int> t = karekup(4);
   Console.WriteLine("Sayının Karesi : " + t.Item1);
   Console.WriteLine("Sayının Küpü   : " + t.Item2);
  }
```

```
 static Tuple<int, int> karekup(int X)
 {
  int kare = X * X;
  int kup = X * X * X;
  Tuple<int, int> deste = new Tuple<int, int>(kare, kup);
  return deste;
 }
}
```

Burada Karekup() adını verdiğimiz metot geriye Tuple nesnesi gönderdiği için önce "Tuple<int, int>" tipinde bir değişken tanımladık. Ardından metodumuzu işletip geriye gönderdiği Tuple nesnesini burada "t" adını verdiğimiz değişkene aktardık.

Burada hazırladığımız Tuple nesnesi 2 bilgi içerdiği için ilk özelliğin adı "Item1", ikincisinin ise Item2'dir. Eğer tuple nesnemiz 3 bilgi içeriyor olsaydı 3 özelliğe sahip olurdu ve bu özelliklerin adları Item1, Item2 ve Item3 olurdu. Bizde düşen, "Item" ön adlı bu özelliklerin içeriklerini elde edip gerektiği yerde kullanmaktır.

Yukarıda Tuple nesnesi hazırlarken new anahtar kelimesinden yararlandık. Tuple nesnesi hazırlarken Tuple sınıfının **Create**() adlı statik metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını göstermek için yukarıda verdiğimiz kodu aşağıdaki gibi düzenledik.

```
static Tuple<int, int> karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 Tuple<int, int> deste = Tuple.Create(kare, kup);
 return deste;
}
```

Verilen koddan fark edileceği üzere Create() metodu kullanıldığında satır biraz kısalmaktadır. Eşitliğin sol tarafında **var** anahtar kelimesi kullanılırsa satır biraz daha kısalır ama o zaman Create() metoduna verilen parametreler tipi ilk bakışta fark edilemeyebilir.

```
static Tuple<int, int> karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 var deste = Tuple.Create(kare, kup);
 return deste;
}
```

C# 7.0 İle Gelen Tuple Yenilikleri – ValueTuple Yapısı

Tuple sınıfının nasıl kullanıldığını bu şekilde gözler önüne serdikten sonra Tuple sınıfıyla ilgili bir tanımlama yapabiliriz: **Tuple**, .NET Framework 4.0 ile gelen ve birden fazla değer tutabilen bir veri yapısıdır.

Microsoft firması C# 7.0'da metotların birden fazla değeri geriye gönderme tekniğinde değişikliğe gitti. C# diline kazandırılan yeni özellikler sayesinde Tuple sınıfına alternatif olarak .NET Framework ile birlikte verilmeye başlanan **ValueTuple** yapısı sayesinde yukarıda anlatılan işlemler biraz kolaylaştı.

Visual Studio, ValueTuple yapısını kullanmanıza itiraz ediyorsa **Tools** menüsünden "NuGet Package Manager > Package Manager Console" komutu verildiği zaman açılan konsola aşağıda verilen satırı yazıp işletmeniz gerekebilir.

```
PM> Install-Package System.ValueTuple -Version 4.3.1
```

ValueTuple yapısının (struct) C#'ın 7.0 sürümüyle gelen şekliyle nasıl kullanıldığını anlatmak için yukarıda hazırlayıp "Karekup()" adını verdiğimiz metodu aşağıdaki gibi düzenledik.

```
static (int, int) Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
}
```

Bu metot ile geriye int tipinde 2 bilgi gönderileceği için metot adından önce normal parantezlerin içine "(int, int)" yazdık. Bu metot farklı sayı ve tipte bilgiyi geriye gönderiyor olsaydı parantezin içini ona göre düzenlerdik. Metodun imzasını bu şekilde düzenledikten sonra metot dahilinde bu 2 bilgiyi geriye göndermek üzere **return** anahtar kelimesini aşağıdaki gibi düzenledik.

```
static (int, int) Karekup(int X)
{
 int kare = X * X;
 int kup = X * X * X;
 return (kare, kup);
}
```

Burada return anahtar kelimesi ile geriye gönderilen nesne yukarıda hakkında bilgi verilen örneklerden farklı olarak **ValueTuple** tipindedir. Başka bir deyişle programcı return deyiminin yanına parantezlerin içine yazdığı bilgi veya değişkenler, derleyici tarafından geri planda ValueTuble nesnesi olarak paketlenip öyle geriye gönderilmektedir.

Geriye 2 değer gönderecek metodumuzu bu şekilde düzenledikten sonra üzerinde çalıştığımız uygulamanın Main() metodunu aşağıdaki düzenleyip bu metodu işlettik. Metodun geriye gönderdiği nesnenin tipini seçme işini derleyiciye havale etmek üzere **var** anahtar kelimesinden yaralandık.

```
public static void Main()
{
 var sonuc = Karekup(4);
 Console.WriteLine(sonuc.Item1);
 Console.WriteLine(sonuc.Item2);
}
```

Burada Karekup() adını verdiğimiz metot geriye 2 bilgi gönderdiği için, başka bir deyişle metot imzasında metot adından önce 2 veri tipi (int) adı yazdığımız için return anahtar kelimesi ile geriye gönderilen nesne Item1 ve Item2 adında 2 alana sahip olur. Bu alanlara kendiniz ad verebilirsiniz. Return deyimi ile geriye gönderilen ValueTuple nesnesinde derleyici tarafından geri planda hazırlanan alanlara nasıl ad verildiğini aşağıda görebilirsiniz.

```
static (int alan1, int alan2) Karekup(int X)
{
  int kare = X * X;
  int kup = X * X * X;
  return (kare, kup);
 }

static void Main(string[] args)
{
 var sonuc = Karekup(4);
 Console.WriteLine("Sayının Karesi : " + sonuc.alan1);
 Console.WriteLine("Sayının Küpü   : " + sonuc.alan2);
}
```

Yukarıda Karekup() adını verdiğimiz metodun geriye gönderdiği ValueTuple nesnesini yakalarken "var" anahtar kelimesinden yararlanmıştık. Konu üzerinde düşünmenizi sağlamak için var anahtar kelimesinden yararlanmak yerine aşağıdaki gibi kendimiz bir ValueTuple nesnesi hazırladık.

```
ValueTuple<int, int> deste = new ValueTuple<int, int>();
```

ValueTuple yapısının yapıcı metodunu kullanırken bu nesnenin hizmet edeceği metodun yapısını ve geriye göndereceği bilgilerin tip ve sayısını gözettik. Bu nedenle burada Karekup() adını verdiğimiz metodu işletip geriye gönderdiği nesnenin referansını "deste" adını verdiğimiz değişkene aktarıp kullanabiliriz.

```
static void Main(string[] args)
{
 ValueTuple<int, int> deste = new ValueTuple<int, int>();
 deste = Karekup(4);
 Console.WriteLine("Sayının Karesi : " + deste.Item1);
 Console.WriteLine("Sayının Küpü   : " + deste.Item2);
}
```

Karekup() metodunu hazırlarken kendimiz alan adı seçmemize rağmen burada alan adlarını derleyici seçti. Çünkü ValueTuple nesnesini yapıcı metodu kullanarak hazırlarken alan adları konusunda herhangi bir seçimimiz olmadı.

Metotları Aşırı Yüklemek

Şimdiye kadar metotlar hakkında anlatılanlara göre metotlar eksik veya fazla parametre ile işletilmek istendiğinde hata meydana gelmektedir. Bu konuda bilgi vermek için aşağıda verilen konsol uygulamasını hazırladık. Bu örnekte "Procedure1" adını verdiğimiz metot **static** olduğu için sınıfın örneğini almadan kullanabildik.

```
class Program
{
 public static void Main()
 {
  int Sayi1 = 100;
  int Sayi2 = 200;
  int Toplam = Procedure1(Sayi1, Sayi2);
  Console.WriteLine(Toplam);
 }
```

```
 static int Procedure1(int X, int Y)
  {
    return (X+Y);
  }
}
```

Bu uygulamadaki "Procedure1" metodu çağırılırken 2 yerine 1 veya 3 parametre kullanılırsa hata meydana gelir. Ancak bazen hazırlanan metodun kaç parametreye ile çağrılacağı önceden belli değildir. Örnek olsun diye hazırladığımız bu metot 2 sayıyı toplamaktadır. Ya toplanmak üzere 3 sayı bu metoda gönderilmek istenirse? Bu gibi durumlarda aynı metodun 2. veya 3. kopyası hazırlanabilir. Her kopyanın parametreleri farklı olur. Aynı ada sahip birden fazla metot hazırlamaya metodun aşırı yüklenmesi (overload) denilmektedir.

```
static int Procedure1(int X, int Y)
{
 return (X + Y);
}
static int Procedure1(int X, int Y, int Z)
{
 return (X + Y + Z);
}
```

Programcı int tipinde 2 sayıyı toplamak istiyorsa metodu 2 parametre ile çağırır. Bu durumda bu metodun 2 parametreli kopyası devreye girer. Metot 3 parametre ile çağrıldığında ise 3 parametreye sahip 2. kopya işlev görür. Derleyici, metodun çağrıldığı satırda kullanılan parametre yapısına bakarak metodun hangi kopyasının kullanılacağına karar verir.

Derleyici işleteceği metodu seçerken metodun geri dönüş tipini bir kriter olarak almıyor. Başka bir deyişle, metodun imzasına metodun geri dönüş tipi dahil değildir. Aşağıda verilen uygulama derlenip işletildiğinde metodun 3. parametreye sahip versiyonu işletilir.

```
class Program
{
 public static void Main()
 {
  int sayi1 = 100;
  int sayi2 = 200;
  int sayi3 = 300;
  int toplam = Procedure1(sayi1, sayi2, sayi3);
  Console.WriteLine(toplam);
 }
 static int Procedure1(int X, int Y)
 {
   return (X + Y);
 }
 static int Procedure1(int X, int Y, int Z)
  {
    return (X + Y + Z);
  }
}
```

Yukarıda hazırladığımız metodun her 2 kopyası da **int** tipinde bilgiler alıyordu. Ancak bazen hazırladığınız metot ile değişik tipteki bilgiler üzerinde işlem yapmak isteyebilirsiniz. Nasıl ki

aynı metodun değişik sayıda parametreye sahip versiyonlarını hazırlamak mümkün oluyorsa, değişik tipte bilgiler üzerinde işlem yapan kopyalarını hazırlayabilirsiniz. Aşağıda verilen metot hem float hem de int bilgileri toplayabilir.

```
static int Procedure1(int X, int Y)
{
 return (X + Y);
}
static float Procedure1(float X, float Y)
{
 return (X + Y);
}
```

Bu metoda parametre olarak 2 float sayıyı gönderirseniz metodun 2. kopyası işletilir. Yukarıda verilen uygulamanın Main() metodu aşağıdaki gibi düzenlediğinde bu metodun float bilgiler üzerinde işlem yapan kopyası işletilir. Procedure1() adlı metodun işletildiği satırda 2 parametreden birisi float, diğeri int tipinden seçilirse, metodun float parametrelere sahip kopyasını işletmektedir.

```
public static void Main()
{
 float Toplam = Procedure1(100.25f, 200.75f);
 Console.WriteLine(Toplam);
}
```

Aynı örnekten yola devam edecek olursak: Bazen toplanacak sayıların kaç adet olduğu programın yazımı sırasında belli değildir. Bu durumda her ihtimal için metodun ayrı bir kopyasını hazırlamak gerekir ki bu pratik değildir. Bu gibi durumlarda parametreler dinamik bir diziye yerleştirilir. Bu işlemin nasıl yapıldığını anlatmak için yukarıda verdiğimiz konsol uygulamasını aşağıdaki gibi düzenledik. Aşağıda verilen kodu dikkatlice incelerseniz **params** anahtar kelimesinden yararlandığımızı fark edersiniz.

```
public static void Main()
{
 int Toplam = Procedure1(100, 200);
 Console.WriteLine(Toplam);
}
static int Procedure1(params int[] X)
{
 int T = 0;
 foreach(int i in X)
   T = T + i;
 return T;
}
```

Bu şekilde düzenlenen metot istenilen sayıda parametre ile çağrılıp işletilebilir. Bu metot ile float tipindeki sayısal bilgileri de toplayabilmek istiyorsanız bu metodun aşağıdaki gibi 2. bir kopyasını hazırlamanız gerekir.

```
static int Procedure1(params int[] X)
{
 int T = 0;
 foreach(int i in X)
```

```
    T = T + i;
 return T;
}
static float Procedure1(params float[] X)
{
 float T = 0;
 foreach(float i in X)
   T = T + i;
 return T;
}
```

Konu metotların aşırı yüklenmesi olduğunda **dynamic** ve **object** tipinde parametreler kullanıldığında ne gibi sonuçların alındığı konusunda ayrı bir başlık açmak gerekiyor. Konu üzerinde düşünmenizi sağlamak için yukarıda hazırlayıp "Procedure1" adını verdiğimiz metodun 2 kopyasını aşağıdaki gibi düzenledik.

```
static int Procedure1(dynamic X, int Y)
{
  return (X + Y);
}
static float Procedure1(float X, dynamic Y)
{
  return (X * Y);
}
```

Daha önceki konulardan bildiğiniz gibi dynamic değişkenlerin tipleri çalışma anında aktarılan bilgiye göre belirlenmektedir. Buna göre bu metodun geriye int bilgi gönderen kopyasının işletilmesi için ikinci parametrenin int tipinde olması gerekir. Bu tespitin ışığında bu metot aşağıdaki gibi işletilmek istenirse hangi kopya kullanılır?

```
float toplam = Procedure1(100.25f, 200);
```

Bir soru daha: Birinci parametre olarak verilen float sayı metodun ilk kopyasındaki dynamic tipteki parametre olarak mı yoksa ikinci kopyadaki float tipteki parametre olarak mı kullanılır? Bu sorunun cevabı olmadığı için derleyici bu satırda hata rapor eder. Bu nedenle konu metotların aşırı yüklenmesi olduğunda dynamic tipte parametreleri kullanırken dikkatli olmak gerekir. Aynı dikkati object tipli parametreler için de göstermek gerekir.

Derleyicinin İşleteceği Metodu Seçme Mekanizması

Yukarıdaki sayfalarda işaret edildiği üzere üzerinde çalışılan sınıfta aynı ada sahip birden fazla metot olduğunda, programcının işletilmesini istediği metodun parametre yapısıyla (metodun imzası) derleyiciye yol göstermeye çalışılır. Yine işaret edildiği gibi metodun imzasına geri dönüş tipi dahil değildir. Bu konudaki bazı ayrıntıları ortaya çıkarmak için aşağıdaki gibi 2 metot hazırladık.

```
static void Procedure1(long x, long y)
{
 Console.WriteLine("long parametreli metot işletildi");
}
static void Procedure1(int x, int y)
{
 Console.WriteLine("int parametreli metot işletildi");
}
```

Adı ve parametre sayıları aynı olan bu 2 metodun parametrelerinin tipleri farklıdır. Dolayısıyla programcı long tipinde parametreler ile metodu çağırdığında derleyici long parametreli kopyayı işletir. Ancak programcı aşağıda yapıldığı gibi bu metodu byte tipinde parametreler ile işletmek istediğinde derleyici önceliği hangisine verir?

```
public static void Main()
{
 Procedure1((byte)10, (byte)20);
}
```

İşletmek istediğimiz metodun byte tipinde parametreye sahip kopyası olmadığı için kapalı dönüştürme yapılır ve byte daha yakın olan tipe sahip parametreli olanı, yani int'li metodu işletir. Aynı ada sahip bu 2 metottan int parametreli olanı aşağıdaki gibi değiştirilirse bu kez derleyici long parametreli olanı seçer. Çünkü long, decimal'e göre byte'ta daha yakındır.

```
static void Procedure1(decimal x, decimal y)
{
 Console.WriteLine("decimal parametreli metot işletildi");
}
```

Benzer şekilde aynı ada sahip bu metotlarımızın birisi float, diğeri double tipli parametreye sahip olsaydı? Programcı aşağıda yapıldığı gibi int veya long tipli parametre ile işletmek isteseydi derleyicinin önceliği float tipli kopya olurdu.

```
public static void Main()
{
  Procedure1(10, 20);
 }
static void Procedure1(Single x, Single y)
 {
  Console.WriteLine("Float parametreli metot işletildi");
 }
static void Procedure1(double x, double y)
{
 Console.WriteLine("Double parametreli metot işletildi");
}
```

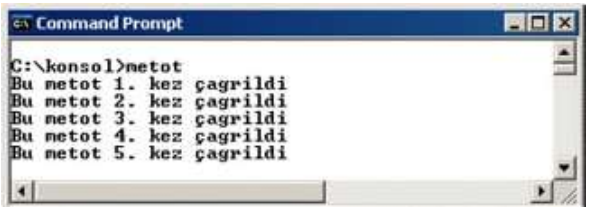
Metotların Kendi Kendilerini Çağırması

C# derleyicisi metotların kendi kendilerini çağırmalarına izin vermektedir. Bu işlemin nasıl yapıldığını anlatmak için aşağıda verilen metodu hazırladık. Bu metot kendi kendisini 5 kez çağırarak belirtilen mesajı 5 kez konsola yazmaktadır.

```
class Program
{
 public static void Main()
 {
  int sayi =5;
  procedure1(sayi);
 }
static void procedure1(int X)
{
 if (X>0)
  {
```

```
    Console.WriteLine("Bu metot {0:X}. kez çağrıldı", 6-X);
    X = X - 1;
    procedure1(X);
   }
  return;
  }
}
```

Belli bir şarta bağlı olarak metottan çıkılmadığı zaman metot kendisini sürekli olarak çağıracağı için sorun çıkar. Pratik değeri olmayan bu konsol uygulaması çalıştırıldığı zaman aşağıdaki gibi bir sonuç alınır.



Metotların kendi kendilerini çağırmaya örnek olması için faktöriyel hesabı yapan bir örnek hazırlayacağız. Bu amaçla forma 2 TextBox yerleştirip aşağıda verilen kodu hazırladık. Kullanıcı ilk TextBox'a faktöriyelini bulmak istediği sayıyı yazıp bu kodu işletecek.

```
private void Faktoryel_bul_Click(object sender, EventArgs e)
{
 int sayi = Convert.ToInt32(textBox1.Text);
 int sonuc = Faktoryel(sayi);
 textBox2.Text = sonuc.ToString();
}
int Faktoryel(int i)
{
 if (i == 0)
  return 1;
 else
  return (i * Faktoryel(i - 1));
}
```

Metotlara Nesneleri Parametre Olarak Vermek

Yukarıdaki sayfalarda metotlara int ve string tipindeki değişkenleri parametre olarak verdik. Ancak metotlar nesneleri parametre olarak alabilmektedir. Bu konuda bilgi vermek için forma 2 ListBox yerleştirip ilk ListBox'ın içeriğini diğerine aktaracak bir metot hazırladık. Bu metot ListBox sınıfının **Items** koleksiyonunu parametre olarak almaktadır.

```
private void aktar(ListBox Liste)
{
 foreach (string i in Liste.Items)
   listBox2.Items.Add(i);
}
```

Benzer şekilde parametre olarak istediğiniz nesneyi metotlara gönderebilirsiniz. Bu metodu kullanmak için formun Click olayını temsil eden metodu aşağıdaki gibi düzenledik.

```
private void Form1_Click(object sender, EventArgs e)
{
 aktar(listBox1);
}
```

Şimdi yukarıda yaptığımız işlemin tersini yapıp metodun geriye nesne göndermesini sağlayacağız. Bu amaçla ilk olarak bir dizi değişken tanımlayıp bilgi aktardık. Devamında bu dizi değişkeni **Gonder**() adı verilen metoda parametre olarak verdik.

```
private void Form1_Click(object sender, EventArgs e)
{
 ListBox List = new ListBox();
 string[] Dizi = new string[3];
 Dizi[0] = "Fenerbahçe";
 Dizi[1] = "Galatasaray";
 Dizi[2] = "Beşiktaş";
 List = Gonder(Dizi);
 foreach (string i in List.Items)
   listBox1.Items.Add(i);
}
```

"Gonder()" metodu geriye ListBox nesnesi gönderdiği için önceden "List" adında bir ListBox nesnesi hazırladık. Gonder() metodunun geriye gönderdiği ListBox nesnesinin elemanlarını formdaki ListBox'a aktardık. String tipinde dizi değişkeni parametre olarak alıp geriye ListBox nesnesi gönderen metodu aşağıda verdik.

```
private ListBox Gonder(string[] Diz)
{
 ListBox Liste = new ListBox();
 int Sayi = Diz.Length;
 for (int i = 0; i < Sayi; i++)
  {
   Liste.Items.Add(Diz[i]);
  }
 return Liste;
}
```

Partial Metotlar

Class'lar hakkında bilgi verilen bölümde işaret edildiği gibi C# 2.0'dan itibaren class'ları oluşturan kodları birden fazla parça olarak düzenlemek, ayrı kod dosyaları arasında paylaştırtmak mümkündür. Class'ların birden fazla parça halinde düzenlenmesine **Partial Class** denilmektedir. Bu başlık altında metotların birden fazla class arasında nasıl paylaşıldığından söz edilecektir.

Konu, metodu oluşturan kod satırlarını birden fazla class arasında paylaştırma olunca (partial method) class'taki gibi metodun istenen kısmını bir class'a diğer kısmını ise başka class'a almak şeklinde olamıyor. C# 3.0'dan itibaren desteklenen Partial Method konusu adım adım anlatılması gereken bir konu olduğu için ayrıntıları birazdan vereceğiz.

Yine de partial metotları tek cümle ifade etmek gerekirse şu söylenebilir: Partial Method'lar sayesinde metotların deklarasyon ve imlementation kısımlarını ayrı dosyalardaki partial

sınıflarda tutmak mümkün hale geldi. Bu konuda bilgi vermek için üzerinde çalıştığımız uygulamanın "Program.cs" adlı kod dosyasında aşağıdaki gibi 2 partial sınıf hazırladık.

```
public partial class Parcali
{
}
public partial class Parcali
{
}
```

Çünkü partial metotlar ancak partial class'larda yer alabiliyorlar. Yani metodunuzu birden fazla parçaya bölmek istiyorsanız önce elinizde 2 parçaya ayrılmış bir class olmalıdır. Çünkü metotlar, ister tümü ister bir kısmı her türlü class'ların içinde olmalıdır. Bu şekilde partial metotlarımız için 2 partial sınıf hazırladıktan sonra sınıflardan birisinde hazırlamak istediğimiz partial metodun imzasına yer verdik.

```
public partial class Parcali
{
 partial void Karesi(int X);
}
public partial class Parcali
{
}
```

Bu şartlarda 2 parça halinde düzenlenmiş ve bir parçasında partial bir metodun imzasını içeren bu class'ın yer aldığı uygulama hatasız derlenir. Hatta bu partial class'tan yola çıkılıp nesne hazırlanabilir. Tabi burada "Karesi()" adını verdiğimiz metot henüz işletilebilinecek durumda değildir. Daha doğrusu, sadece deklerasyon kısmı yazılan partial metotlar derleme sırasında göz ardı ediliyor. Şimdi partial class'ın ikinci parçasında partial metodumuzu kodlayacağız.

```
public partial class Parcali
{
  partial void Karesi(int X);
}

public partial class Parcali
{
 partial void Karesi(int X)
 {
  int sayi = X * X;
  Console.WriteLine(sayi);
 }
}
```

Bu şartlarda ilk partial metodumuzu hazırlamış durumdayız. Bu partial sınıfı dikkatlice incelerseniz birisinde sadece metodun imzası olurken diğerinde metodun imzası dahil tümü var. Bu bize ne mi kazandırır? Ekip halinde kapsamlı bir proje üzerinde çalışıyorsak elimizde sadece metotların imzalarının olduğu partial bir class'ın olması, projemizdeki bir sınıfı kuşbakışı değerlendirme imkanını bize kazandırabilir. Yani bir bakıma elimizde sınıftaki metotların listesi olacaktır.

Verilen kodu dikkatlice incelediyseniz partial metodumuz için erişim belirleyici anahtar kelimelerden herhangi birisini kullanmadığımızı ve dolayısıyla bu metot private olduğu için başka sınıfta kullanılmayacağını fark etmişsinizdir. C# derleyici partial metotların public yapılmasına izin vermediği için ister istemez bu partial class'lardan birisinde private olan partial metodumuzu işletecek public bir metot hazırlamak zorundayız.

Kodu okunur yapmak için partial metotları işletmek amacıyla hazırladığımız public metotlar için partial class'ın 3. bir parçası hazırlanabilir. Bu işlemi nasıl yaptığımızı net olarak görebilmeniz için şimdiye kadar partial metotlarla ilgili olarak yazdığımız kodun tamamını aşağıda bir arada verdik.

```
public partial class Parcali
{
  partial void Karesi(int X);
}
public partial class Parcali
{
 partial void Karesi(int X)
 {
  int sayi = X * X;
  Console.WriteLine(sayi);
 }
}
public partial class Parcali
{
 public void islet(int i)
  {
    Karesi(i);
  }
}
class Program
{
 public static void Main()
 {
   Parcali nesne = new Parcali();
   nesne.islet(10);
  }
}
```

Yukarıda ifade edildiği gibi, partial metotlar public olamıyorlar. Bu engel partial metotların kullanımını engellemiyor ama programcıyı tek satırlık ve public olan ikinci bir metot hazırlamak zorunda bırakıyor. Partial metotların kullanım alanını daraltan ikinci bir engel daha var: Partial Metotlar void olmak zorundalar.

Yerel Metotlar – Local Functions

Yerel metotlardan söz etmeden isimsiz tipleri hatırlamak gerek. İsimsiz tiplerin bir metot üzerinde çalışırken yeni bir tipe ihtiyaç duyulduğunda ve uygulamanın başka bir yerinde bu tipin kullanılma ihtimali yoksa ayrı bir yerde class hazırlamak kod kalabalığından başka bir şey değildir. İsimsiz tip desteği C# diline LINQ ile birlikte geldiği için isimsiz tiplerden LINQ sorgularında sıklıkla yararlanılmaktadır.

Bu gibi durumlarda programcılar class anahtar kelimesini kullanmadan, ayrı kod dosyası hazırlamadan hemen bulundukları yerde isimsiz tip tanımlayıp kullanabiliyorlar. C# 7.0 ile birlikte programcıların daha temiz ve okunur kod yazmalarına hizmet edecek bir yenilik kullanıma sunuldu: Yerel Metotlar.

Nasıl ki yerel değişkenler sadece tanımlandığı metot içinde yaşayıp metot içinde kullanılabiliniyorsa yerel metotlar sadece içinde kodlandıkları metotta kullanılabiliniyorlar. Metot içinde metotların nasıl hazırlandığı üzerinde konuşmaya devam etmek üzere üzerinde çalıştığımız uygulamanın Main() metodu içinde aşağıdaki gibi bir metot hazırladık.

```
public class Program
{
 static void Main(string[] args)
  {
   int islem (int par1, int par2)
    {
     return par1 + par2;
    }
  }
}
```

C# konulu dokümanlarda Local Functions olarak geçen yerel metotlar sadece kodlandıkları metot içinde kullanılabildikleri için geçerlilik alanları önceden bellidir. Bu nedenle metot içinde hazırlanan metotlar için erişim belirleyici anahtar kelimelerden herhangi birisi kullanılamaz. Ayrıca bu metotlar statik olmazlar. Main() metodu içinde hazırladığımız bu metodu nasıl kullandığımızı aşağıda görebilirsiniz.

```
public class Program
{
 static void Main(string[] args)
  {
    int Islem (int par1, int par2)
     {
       return par1 + par2;
     }
    int sonuc = Islem(100, 200);
    Console.WriteLine(sonuc);
  }
}
```

Daha önceki konulardan bildiğiniz gibi Main() gibi statik metotların içinde statik olmayan metotları aynı sınıfın içinde hazırlanmış olsalar bile nesne üzerinden işletebiliyorduk. Verilen koddan fark etmiş olabileceğiniz yerel metotları statik metotların içinde bile adıyla işletebiliyoruz.

Yerel metotlarla ilgili birkaç önerimiz olacaktır:

- Kodunuzu hazırlayıp ilk testlerden geçirdikten sonra mevcut bir metodu sadece bir yerde çağırıp işlettiyseniz o metodu işletildiği metodun içine yerel metot olarak taşımak kodunuzun daha okunur ve düzenli olmasına bir katkıdır.

- Bildiğiniz gibi çok uzun metotların hazırlanması önerilmiyor. Çünkü uzun metotlar hem programcının veriminin düşmesine hem de kodun okunurluğunun azalmasına neden olmaktadır. Hazırladığınız metodun çok uzadığını fark ettiğinizde uygun kısımlarını ayırıp aynı metot içinde yerel metot olarak düzenlemek iyi bir tercihtir.
- Yerel metotlarda kapsayıcı metodun içinde tanımlanan değişkenler kullanılabilmektedir. Buda yerel metodun birden fazla değeri geri göndermesi gerektiğinde programcıya kolay kullanılan bir alternatif olmaktadır.

12. BÖLÜM

INTERFACE'ler - ARAYÜZLER

Bir görüşe göre interface'ler birkaç sayfada anlatılacak konudur. Birkaç metot ve özelliğe sahip bir interface hazırlanır. Sonra bir sınıfın bu interface'i uygulaması sağlanır. Bize sorarsanız interface'ler iyice kavranmadan nesneye yönelik programcılık ve C# gibi programlama dillerini öğrenme yarım kalacaktır. Bu nedenle bu bölümde yazılanları iyice kavramadan bir sonraki bölüme geçmenizi önermeyiz. Bizce interface konusu önemli olduğu için, bu bölüm tek satır kod yazılmadan sayfalarca yazı içeren bir bölüm oldu. Bu kadar açıklamayı şimdi kim okuyacak diyorsanız buraya sonra dönmek üzere ilk interface'in hazırlandığı sayfaya geçebilirsiniz.

Bu kitapta daha sonra verilecek koleksiyon sınıfları hakkında bilgi içeren bölüme baktığınızda neredeyse sınıflar kadar interface'lerden söz edildiğini göreceksiniz. Bu nedenledir ki işlevleri birbirinden oldukça farklı olan koleksiyon sınıflarında birçok işlem aynı ada sahip metotlarla yapılmaktadır. Bunun nedeni interface'lere verilen önemden kaynaklanmaktadır. İleri bir tarihte C# kodu yazmak üzere kurumsal bir firmada çalışmaya başladığınızda abstract class veya interface'te adı geçmeyen metotların çok az kodlandığını göreceksiniz.

.NET uygulaması geliştiren firmaların eleman seçerken veya işe almak istedikleri programcının bilgisini ölçerken sıklıkla başvurdukları sorulardan birisinin abstract class-interface farklılıkları-benzerlikleri üzerine olduğu biliniyor. Bu bölüm üzerinde çalışmaya başlamadan önce abstract class'lar hakkında bilgi içeren sayfaları okuduysanız ve bu bölümde interface'ler hakkında yazılanları baştan sona okur ve verilen basit kod örneklerini denerseniz interface'ler konusunda size gelmesi muhtemel soruları rahatlıkla cevaplarsınız.

Bu bölümde anlatılan interface kavramının Türkçe karşılığı olan arayüz kavramını, programların kullanıcı ile yüz yüze gelen görsel haliyle karıştırmamak gerek. Çünkü burada sözü edilen arayüzler, programcıya rehberlik etmektedir, programcının yazması gereken metotları kendisine söylemektir.

Değişik programcılar tarafından geliştirilen sınıflar farklı işlemleri yapıyor olsalar bile bazı ortak noktalara sahip olmaları istenebilir. Örneğin geliştirilen uygulamadaki her sınıfın **Sirala**() adında bir metoda veya "firma" adında bir özelliğe sahip olması istenebilir. Bu gibi sorunların çözümü **interface** veya **abstract class** hazırlamaktan geçiyor.

Programcılık üzerine konuşup yazanlar interface ve abstract class kavramlarını aynı cümle içinde sıklıkla kullanırlar. Çünkü bir birine yakın işleve sahiptirler. Bu nedenle bu bölüm üzerinde çalışmaya başlamadan önce bu kitapta abstract class'larla ilgili olarak yazılanları okuyup anlamış olmanızda fayda vardır.

Bu kitabın şimdiye kadar verilen sayfalarından fark etmiş olacağınız gibi .NET Framework ile birlikte çok sayıda interface hazır olarak gelmektedir. Standart işlemler söz konusu olduğunda kendiniz interface hazırlamak yerine .NET Framework ile gelen interface'leri kullanmanızı öneririz. Bu sayede kodunuz diğer C# programcıları için okunur olur. Örneğin

.NET Framework ile gelen **IDisposeable** yerine 3-5 satır kod yazarak "IYokEdilebilir" adında bir interface hazırlayıp kullanabilirsiniz. Ancak bu durumda kodunuzu başka bir programcı incelediğinde "bu da nedir" sorusu kaçınılmaz olacaktır.

Kısaca anlatmak gerekirse, C# programlama dilinde interface'ler bir sınıfta olması gereken metotları, olayları, özellikleri ve indeksleyicileri işaret eden yapılardır. Yani interface'ler o sınıfta nelerin olacağını, nelerin yapılacağını işaret ederken nasıl yapılacağıyla ilgilenmezler. Interface'lerde metotların imzaları, delegate'ler, indexer'lar ve özelliklerin tanımlama satırları olmaktadır. Bu kitabın bundan önceki sayfalarında olayları ve indeksleyicileri henüz anlatmadığımız için bu bölümde interface'lerde sadece metot ve özelliklere yer vereceğiz. İlgili bölümlerde konunun eksik bırakılan yanları tamamlanacaktır.

Bize sorulursa, C# uygulaması geliştirmek isteyen programcılar işe uygulamalarındaki interface'leri hazırlamakla başlamalıdır. Söz konusu uygulama bir ekip çalışması ile hayata geçirilecekse çalışmaya interface'lerden başlamak daha bir önem kazanır. Uygulamada gerek duyulacak interface'ler hazırlandıktan sonra sırayı abstract class'lar almalıdır diyoruz. Abstract class'lar hazırlandıktan sonra class'lar üzerinde çalışmaya başlanabilir. Uygulamada gerek duyulan enum ve struct'lar class'larla paralel olarak hazırlanabilir. Sealed class'lar ise bizce en son hazırlanması gereken uygulama bileşenleridir.

Interface'lerin sayesinde .NET Framework ile gelen çok sayıda kontrolde benzer işlev ve aynı ada sahip çok sayıda metot bulunmaktadır. Örneğin .NET Framework ile gelen ArrayList sınıfı hakkında bilgi edinen bir programcı günün birisinde List sınıfını kullanmak istediğinde 2 sınıf arasında çok sayıda ortak metot ve özellik olduğundan kısa sürede adapte olacaktır. Çünkü List ve ArrayList sınıfının uyguladıkları interface'lerin bazıları aynıdır.

Yukarıda interface'lerde nelere yer verilebildiğini işaret ettik. Bunları bir de maddeler halinde vermek akılda kalmalarına yardımcı olacaktır:

- Metotlar (Method)
- Olaylar (Event)
- İndeksleyiciler (Indexer)
- Özellikler (Properties)

Class'larda kullanıp interface'lerde yer veremediğimiz dil öğeler ise şunlardır:

- Alanlar (Fields)
- Yapıcı Metotlar (Constructor)
- Yok Edici Metotlar (Destructor)

Listeden gördüğünüz gibi class'ların önemli öğelerinin hepsi kendisine interface'te yer bulabiliyor. Alanların olmaması normal. Çünkü C# kodunda özellikler zaten alanlara erişimde kullanılan özel metotlardır. Özellikler olabildiğine göre alanların interface'lerde bulunmasına gerek yoktur. Diğer yandan yapıcı ve yok edici metotların olmaması son derece normaldir. C# dilini geliştiren ekip, sınıfların yapıcı ve yok edici metotlarının interface'ler aracılığıyla işaret edilmesini tercih etmemiştir.

C#'ın çoklu kalıtımı desteklemiyor olmasından kaynaklanan eksiklikler interface'ler sayesinde biraz giderilmektedir. Aslında kalıtım denildiğinde base class'ın mevcut halinden yararlanmak anlaşılıyor. Interface'lerde ise yukarıda sayılan class öğelerinin tanımlama satırlarından başka bir şey yoktur. Yani bu açıdan bakıldığında interface'lerin çoklu kalıtım adına uygulandıkları sınıfa bir katkıları yoktur. Çünkü interface'ler programcıyı kendisine sunulan şartların hepsini yerine getirmek, interface'ten işaret edilen bütün metotları kodlamak zorunda bırakmaktan öte önemli işlev sunmuyor. Bu zorunluluk sayesinde sınıfta nelerin hangi ad ile olacağı önceden belli oluyor ki interface'lerin en önemli işlevi budur. Bu söylenenin de tekrar kullanımla (reusability) ile bir ilgisi yoktur. Yani konu, kalıtım ve interface olduğunda konu üzerinde düşünmek gerekiyor.

Son paragraftan interface'lerin NET programcılığındaki önemiyle ilgili olarak şüpheye düşmemelisiniz. Çünkü C# uygulaması geliştirirken karşınıza sürekli olarak interface'ler çıkacaktır. Ayrıca .NET Framework ile birlikte çok sayıda interface'in hazır olarak geldiğini de unutmamak gerek. Interface'ler hakkında yeteri kadar bilgi edindikten sonra .NET Framework ile gelen class ve interface'lerin işlevlerini anlamak kolaylaşacaktır. Bütün bu nedenlerden dolayı bu bölümde anlatılanların dikkatli okunup küçük örneklerin kodlanıp denenmesini öneriyoruz.

En başta belirtelim; bu kitabın daha önce verilen abstract sınıflarla ilgili sayfalarını okuyan birisi bu bölümde interface'ler hakkında şimdiye kadar yazılanları gördüğünde işlev bakımından bazı benzerliklerin olduğunu fark etmiş olabilir. Bu fark etme soru işaretlerini birlikte getirecektir. Bu soru işaretlerini ilerleyen sayfalarda gidermeye çalışacağız.

Hem abstract class'lar hem de interface'ler C#'ın ilk sürümünden beri desteklendiği için sonradan ihtiyaçtan veya başka bir nedenden dolayı dile eklenmiş değillerdir. Ötesi, bu 2 dil öğesi muhtemelen uzun değerlendirme ve araştırmalardan sonra dile eklenmiştir. Bu yorumdan şu sonucu çıkarabiliriz: C# uygulamalarında hem abstract class'lar hem de interface'ler sıklıkla gerek duyulan dil öğeleridir. Madem, programcılıkta yolun başında olanlar ne zaman interface, ne zaman abstract class hazırlamaları gerektiğine karar vermekte zorlanıyorlar konuyu uzatıp bazı soruları tekrar sorup kendimizce cevap vereceğiz.

- Hazırlamak istediğiniz abstract class veya interface'de gövdesi olacak metotlar var mı? Sorunun cevabı evet ise abstract class hazırlamalısınız.
- Hazırlayacağınız class'a rehber olacak yapıda çok sayıda metot olacak mı? Metotların sayısı fazla ise bunları gruplamak anlamı mıdır? Class'a rehber olarak hazırlamak istediğiniz kodda çok sayıda metot varsa ve bunları gruplamak iyi bir fikirse interface hazırlamalısınız.

Interface'ler üzerinde bu kadar konuştuktan sonra şimdi anlatmaya kod üzerinde devam edeceğiz. Bu amaçla üzerinde çalıştığımız konsol uygulamasındaki kod dosyasında aşağıdaki gibi "Matematik" adında bir class ve "IArayuz" adında bir interface hazırladık.

```
namespace ConsoleApp1
{
 public interface IArayuz
  {
  }
```

```
class Matematik
 {
 }
class Program
 {
  public static void Main()
   {
   }
 }
}
```

Interface'ler **interface** anahtar kelimesi ile başlamaktadır. Genel alışkanlıkla Interface'lerin adlarının başına "I" harfi konulmaktadır. Burada hazırladığımız interface ile class aynı kod dosyasında bulunmaktadır. İsteseydik bu Interface'i ayrı bir kod dosyasında hazırlayabilirdik.

Hazırlayıp "IArayuz" adını verdiğimiz interface kısıtlı özelliklere sahip olduğundan ayrı bir kod dosyası olarak değerlendirmeyi tercih etmedik. Bu şartlarda "Matematik" adını verdiğimiz sınıf, bu arayüzü uygulasa bile değişen bir şey olmaz. Çünkü "IArayuz" adını verdiğimiz bu interface henüz herhangi bir olay, metot veya özelliğe sahip değildir. Interface'lerin class'lara nasıl uygulandığını aşağıda görebilirsiniz.

```
public interface IArayuz
{
}
class Matematik : IArayuz
{
}
```

Bu interface için erişim belirleyici olarak **public**'i seçmiş olmakla birlikte erişim belirleyici kullanmayabilirsiniz. Interface'ler için erişim belirleyici olarak **internal** seçilebilir. Bu durumda söz konusu interface sadece geçerli assembly'e dahil olan class veya interface'lere uygulanabilir.

Şimdi bu interface'in bir metoda sahip olmasını sağlayacağız. "Topla()" adını verdiğimiz bu metot **int** tipinde 2 parametreye sahip olacak ve geriye int bilgi gönderecektir. Interface'lerde metotların yalnızca imzaları bulunur.

```
public interface IArayuz
{
 int Topla(int x, int y);
}
```

Interface dahilinde deklare edilen özellik ve metotlar için erişim belirleyicilerden birisini kullanırsanız hata meydana gelir. Madem interface'ler imzalarını içerdikleri metotlar, özellikler, olaylar ve indeksleyiciler bağlamında kendisini uygulayacağı sınıflara rehberlik ediyorlar, söz konusu üyenin geçerlilik alanı sınıfta belirlenmelidir.

Interface'ler işlev bakımından abstract sınıflara çok benzemektedir. Abstract (özet) sınıflarda hem abstract hem de normal metotlara yer verme imkanı olmasına rağmen interface'lerde

gövdesi olan metotlara yer verilemez. Interface'lerdeki bütün metotlar doğaları gereği abstract oldukları için interface'lerin abstract olduğunu ayrıca belirtmeye gerek yoktur.



Yukarıda hazırladığımız interface deneysel olduğu için sadece bir üyeye yani bir metoda sahiptir. Her ne kadar bir interface'de istenilen sayıda üyeye yer verilmesi teknik olarak mümkün olsa bile sayının fazla olması interface'i karmaşık hale getirebilir. Bu nedenle programcıların hazırlayacakları interface'lerdeki üye sayıları konusunda önceden aralarında tartışmaları yerinde olacaktır. Bize sorarsanız bir interface'deki üye sayısı ortalama 10-15 olmalıdır. Hazırlanmak istenen interface'teki her metodun 3-5 overlaod sürümü varsa bu sayının kolayca yükselebileceğini unutmamak gerekir.

Bu şartlarda, yani uygulamadaki sınıf, hazırladığımız interface'i uygulamış halde iken proje çalıştırılırsa hata meydana gelir. Çünkü bu arayüzü uygulayan "Matematik" sınıfında **Topla()** adında ve int tipinde 2 parametreye sahip bir metodunun olması gerekir. Bu nedenle "Matematik" adını verdiğimiz sınıfı aşağıdaki gibi düzenledik. Burada "Matematik" adını verdiğimiz sınıf aynı zamanda bir sınıfın mirasçısı olabilir. Bu durumda interface'ten önce base class'ın adını yazmak gerekir.

```
public interface IArayuz
{
 int Topla(int x, int y);
}
class Matematik : IArayuz
{
 public int Topla(int sayi1, int sayi2)
  {
   return sayi1 + sayi2;
  }
}
```

Bu şartlarda uygulama çalıştırılırsa hata meydana gelmez. Çünkü "Matematik" sınıfı uyguladığı interface'in gereklerini yerine getirmiş oldu. Bu sınıfta Topla() metodunun parametre olarak aldığı bilgileri toplayıp geriye göndermesini sağladık. Bu arayüzü uygulayan başka bir sınıfta bu metotta yapılan işlem farklı olabilir.

Tekrar etmek gerekirse, interface'ler yalnızca metotların adları, geriye gönderdikleri bilginin türü ve parametrelerinin sayı ve tipiyle ilgilenirler. Bunu göstermek için bu arayüzü uygulayan 2. bir sınıf hazırladık. Bu sınıfta Topla() metoduna farklı bir işlem yaptırdık. Bu metot farklı bir işlem yapsa bile interface'in gereklerini yerine getirmektedir.

```
class Deneme : IArayuz
{
 public int Topla(int sayi1, int sayi2)
  {
   int X = sayi1 * sayi1;
   int Y = sayi2 * sayi2;
   return X + Y;
  }
}
```

Interface'lerde Özelliklere Yer Vermek

Bu bölümün ilk sayfalarında ifade edildiği üzere interface'lerde metot imzalarından başka özelliklere yer verilebilmektedir. Bu nedenle sınıflarınızda mutlaka bulunmasını istediğiniz özellikleri interface'de tanımlayarak ilgili sınıfta bulunmasını garanti altına alabilirsiniz. Interface'lerde alanlara yer verilmesine derleyicinin izin vermediğini biliyorsunuz.

Şimdi yukarıda "IArayuz" adını verdiğimiz interface'in ayrıca bir özelliğe sahip olmasını sağlayacağız. Interface'lerde özelliklerin **get** ve **set** bloklarına bir şey yazılmaz. Özellik **ReadOnly** olacaksa yalnızca **get** bloğuna yer verilmelidir.

```
public interface IArayuz
{
 int Topla(int x, int y);
 string islem_adi
 {
  get; set;
 }
}
```

Bu şekilde düzenlenen interface'i uygulayan sınıfın burada "islem_adi" adını verdiğimiz bir özelliğe sahip olması gerekir. Sınıfta aynı ad ve tipe sahip bir özellik hazırlanmazsa interface'in gerekleri yerine getirilmediği için hata meydana gelir. Bu interface'i uygulayan "Matematik" sınıfının interface'in gereklerini yerine getirmesi için aşağıdaki gibi düzenledik.



C# 3.0'den itibaren özellikler için alan tanımlama bir zorunluluk olmaktan çıkarıldı. Çünkü derleyici, kendisi için alan tanımlanmamış özellikler için geri planda otomatik olarak alan tanımlıyor. Bu bölümde interface'lere odaklandığımız için C# derleyicisinin sağladığı bu kolaylığa burada kayıtsız kaldık.

```
class Matematik : IArayuz
{
 private string islem_a;
 public string islem_adi
 {
  get
   {
    return islem_a;
   }
  set
   {
    islem_a = value;
   }
 }
 public int Topla(int sayi1, int sayi2)
 {
  return sayi1 + sayi2;
 }
}
```

Interface'ler indeksleyici içerebilir. Interface'lerde indeksleyicilere nasıl yer verildiğini anlatmak için yukarıda verilen interface'i aşağıdaki gibi düzenledik. İndeksleyici içeren interface'i uygulayan sınıfta nasıl bir hazırlık yapılması gerektiği konusu ilgili bölümde işlenecektir.

```
interface IArayuz
{
 int Topla(int x, int y);
 string islem_adi
 {
  get;
  set;
 }
 string this[int i]
 {
  get;
  set;
 }
}
```

Bir Sınıfa Birden Fazla Interface Uygulamak

Bir sınıfa birden fazla interface uygulanabilir. Bu işlemin nasıl yapıldığını anlatmak için 2. bir interface hazırladık. Bu interface'i yukarıda hazırladığımız sınıfa 2. interface olarak uygulayacağız.

```
interface IAlanlar
{
 double Kare(double X, double Y);
}
```

Yalnızca bir metodun deklare edildiği bu interface'i de sınıfa uygulamak istiyorsanız sınıfın deklare edildiği satıra bu interface'in de adını yazmalısınız. Aşağıda 2. interface'i sınıfa nasıl uyguladığımızı ve bu interface'in gereklerini nasıl yerine getirdiğimizi görebilirsiniz.

```
class Matematik : IArayuz, IAlanlar
{
 private string islem_a;
 public string islem_adi
 {
  get
   {
    return islem_a;
   }
  set
  {
   islem_a = value;
  }
 }
 public int Topla(int sayi1, int sayi2)
 {
   return sayi1 + sayi2;
 }
```

```
 public double Kare(double kenar1, double kenar2)
  {
   return kenar1 * kenar2;
  }
}
```

Bu kodda "Matematik" adını verdiğimiz sınıftaki Topla() metodu "IArayuz" adlı interface'in gereklerini yerine getirirken Kare() metodu IAlanlar'ın gereklerini yerine getirmektedir. Bu sınıfa uygulanan 2 arayüz de "Kare" adında bir metot olsaydı ne olurdu? Bu soruya cevap vermek amacıyla aşağıda verilen 2 interface ve sınıfı hazırladık.

```
interface Iislemler1
{
 int Kare(int X);
}
interface Iislemler2
{
 int Kare(int X);
}
class Matematik : Iislemler1, Iislemler2
{
 public int Kare(int kenar)
  {
   return kenar * kenar;
  }
}
```

"Matematik" sınıfının uyguladığı bu 2 interface aynı ada sahip birer metoda sahiptir. Bu durumda bu interface'leri uygulayan sınıfta yalnızca bir metot hazırlanırsa her iki interface'in de gerekleri yerine getirilmiş olunur.

"Matematik" sınıfının uyguladığı interface'lerdeki "Kare" metotları farklı imzaya sahip olsaydı ne olurdu? Bu durumda "Matematik" sınıfında "Kare" adında 2 metot hazırlamak gerekirdi. Aşağıda verilen hali ile "Matematik" sınıfı hataya neden olmaz ve işletilmek istenen metodun hangisi olduğuna parametre yapısına bakılarak karar verilir.

```
interface Iislemler1
{
 int Kare(int X);
}
interface Iislemler2
{
 double Kare(double X);
}
class Matematik : Iislemler1, Iislemler2
{
 public int Kare(int kenar)
 {
  return kenar * kenar;
 }
 public double Kare(double kenar)
 {
  return kenar * kenar;
 }
}
```

Bu sınıfın örneği aşağıdaki gibi alınıp her iki metot sorunsuzca işletilebilir. Burada yapılan, metotların aşırı yüklenmesi gibidir. Metoda int tipinde bilgi parametre verilirse sınıfa ilk sırada uygulanan interface'in gereklerini yerine getirmek için hazırlanan metodun işletilmek istendiğine karar verilir.

```
private void islet_Click(object sender, EventArgs e)
{
 Matematik mat = new Matematik();
 int sonuc_int = mat.Kare(2);
 textBox1.Text = sonuc_int.ToString();
 double sonuc_double = mat.Kare(2.5);
 textBox2.Text = sonuc_double.ToString();
}
```

Şimdi bu 2 metodun geri dönüş tipine sahip olmasını sağlayıp farklı işlemler yaptırtacağız. Daha doğrusu programcının hangi metodu kullanmak istediği konusunda derleyiciyi kararsızlığa sevk etmek istiyoruz. Bu düşünce bu 2 interface'i ve bu interface'leri uygulayan sınıfı aşağıdaki gibi düzenledik.

```
interface Iislemler1
 {
  int Kare(int X);
 }
interface Iislemler2
 {
  int Kare(int X);
 }
class Matematik : Iislemler1, Iislemler2
{
 public int Kare(int sayi)
  {
   return sayi * sayi;
  }
 public int Kare(int sayi)
  {
   return sayi * sayi * sayi;
  }
}
```

Tahmin edeceğiniz gibi bu sınıf derleme sırasında hataya neden olur. Çünkü C# derleyicisi aynı sınıf içinde aynı parametre yapısına sahip birden fazla metodun olmasına izin vermez. Bu sorunun çözümü, sınıfta metodu kodlarken o metodun hangi interface'in gereklerini yerine getirmek için hazırlandığını işaret etmektir. Bu sınıf aşağıdaki gibi düzenlenirse hata meydana gelmez.

```
interface Iislemler1
 {
  int Kare(int X);
 }
interface Iislemler2
 {
  int Kare(int X);
 }
```

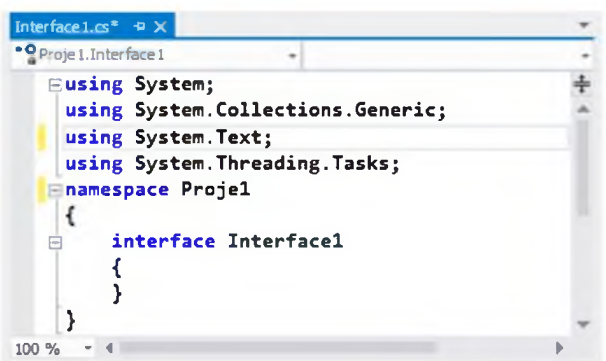
```
class Matematik : Iislemler1, Iislemler2
{
 int Iislemler1.Kare(int sayi)
  {
   return sayi * sayi;
  }
 int Iislemler2.Kare(int aayi)
  {
   return sayi * sayi * sayi;
  }
}
```

Metotları kodlarken erişim belirleyici olarak public'i sildiğimizi fark etmiş olmalısınız. Çünkü buna gerek yoktur. Bu sınıfın örneğini alıp metotları kullanırken hangini kullanmak istediğinizi belirtmeniz gerekir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
private void islet_Click(object sender, EventArgs e)
{
 Matematik mat = new Matematik();
 Iislemler1 islem1 = (Iislemler1)mat;
 int sonuc1 = islem1.Kare(2);
 textBox1.Text = sonuc1.ToString();
 Iislemler2 islem2 = (Iislemler2)mat;
 int sonuc2 = islem2.Kare(2);
 textBox2.Text = sonuc2.ToString();
}
```

Burada örneğini aldığımız sınıfa uyguladığımız her iki interface'te aynı ad ve parametre yapısına sahip birer metot olduğu için çalıştırmak istediğimiz metodun hangi interface'in gereklerini yerine getirmek için hazırlandığını işaret ettik. Böyle bir mecburiyet yokken bile metodun hangi interface kaynaklı olduğunu belirtip kodu okunur yapabilirsiniz.

Üzerinde çalıştığınız uygulamada çok sayıda interface hazırlayacaksanız bu interface'leri ayrı bir CS dosyasına almanız önerilir. Interface'leri içerecek CS dosyasını **Add New Item** diyalog kutusunda **Interface** adlı şablondan yararlanarak hazırlayıp zaman kazanabilirsiniz. **Add New Item** diyalog kutusunda **Interface** adlı şablonu seçerseniz Visual Studio sizin için aşağıdaki gibi bir CS dosyası hazırlar.



GetInterface() Metodu

Bir sınıfın ilgilendiğiniz bir interface'i uygulayıp uygulamadığını öğrenmek istiyorsanız **GetInterface()** metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Bu kodda önce bir ArrayList nesnesi hazırlayıp bu nesneye kaynaklık eden sınıfa **IEnumerable** arayüzünün uygulanıp uygulanmadığını araştırdık.

```
ArrayList Liste = new ArrayList(10);
if (Liste.GetType().GetInterface("IEnumerable") == null)
  MessageBox.Show("Bu sınıf bu interface'i uygulamamış");
else
  MessageBox.Show("Bu sınıf IEnumerable'i uygulamış");
```

İlgilendiğiniz sınıfın bir interface'i uygulayıp uygulamadığını öğrenirken "is" operatöründen yararlanabilirsiniz. Söz konusu sınıfa uygulanıp uygulanmadığı öğrenilmek istenen arayüz "is" operatörüne parametre olarak verilmektedir.

```
ArrayList Liste = new ArrayList(10);
if (Liste is IEnumerable)
  MessageBox.Show("Bu sınıf bu interface'i uygulamış");
```

Örneğini aldığınız sınıfın ilgilendiğiniz interface'i uygulayıp uygulamadığını öğrenirken "is" yerine "as" operatöründen yararlanabilirsiniz. Aşağıda verilen **as** operatörünü kullanma örneğinde önce "Matematik" adını verdiğimiz sınıfın örneğini aldık. Ardından bu nesneden yola çıkıp ilgilendiğimiz interface'in örneğini as operatöründen yararlanarak aldık. Eğer as operatörüne 1. parametre olarak verilen nesne örneği operatörün soluna yazılan Interface'i uygulamamışsa geriye **null** gönderir.

```
matematik mat = new matematik();
Iislemler islem = mat as Iislemler;
if (islem != null)
  MessageBox.Show("Bu sınıf bu interface'i uygulamış");
```

Interface'lerde Kalıtım

Hazırladığınız interface başka bir interface'in mirasçısı yapmak istiyorsanız yapmanız gereken tek şey interface'in deklare edildiği satırda base durumundaki interface'lerin adlarını yazmaktır. Aşağıda verilen örnekte "IArayuz" adlı interface, "IAlanlar" adını verdiğimiz diğer interface'in her şeyini miras almaktadır. Dolayısıyla IArayuz'ü herhangi bir sınıfa uyguladığınızda gerçekte 2 interface'i uygulamış olursunuz.

```
interface IArayuz : IAlanlar
{
 int Topla(int x, int y);
 string islem_adi
  {
   get;
   set;
  }
}
```

Interface'ler kalıtımı desteklediği için mevcut bir interface'i genişletip yeni bir interface hazırlamak mümkündür. Interface'lerin kalıtım özelliklerinden söz etmek amacıyla yukarıda hazırladığımız "Islemler1" ve "Islamler2" adlı interface'leri aşağıdaki gibi düzenledik.

```
public interface Islemler1
{
 int Kare(int X);
}
public interface Islemler2 : Islemler1
{
 int Kup(int X);
}
```

Bu hazırlıktan sonra Islemler2 adını verdiğimiz interface herhangi bir sınıfa uygulanırsa o sınıfta hem "Kare()" hem de "Kup()" metodu kodlanmalıdır. Aşağıda verilen kodda "Program" adını verdiğimiz sınıfa Islemler2'yi uyguladık.

```
class Program : Islemler2
{
 public static void Main()
 {
  Program nesne = new Program();
  int sonuc = nesne.Kup(10);
  System.Console.WriteLine(sonuc);
 }
 public int Kare(int sayi)
 {
  return sayi * sayi;
 }
 public int Kup(int sayi)
 {
  return sayi * sayi * sayi;
 }
}
```

Bu kitapta daha önce ifade edildiği üzere C#'ta çoklu kalıtım direk desteklenmiyor. C#'ın bu eksikliği interface'lerle, abstract sınıflarla ve nested class'larla giderilmeye çalışılmaktadır. Konu interface'leri genişletmek veya kalıtımdan yararlanmak olduğunda ":" operatöründen sonra birden fazla interface'in adını yazmak mümkündür. Bu nedenle aşağıda verilen interface'ler sorun çıkarmazlar.

```
public interface Islemler1
{
 int Kare(int X);
}
public interface Islemler2
{
 int Kup(int X);
}
public interface Islemler3 : Islemler1, Islemler2
{
}
```

Interface'ler ve Abstract Sınıflar

Bu kitapta daha önce abstract sınıflar hakkında bilgi verilen bölümden hatırlayacağınız gibi abstract sınıfların asıl kullanım amacı, hazırlanacak sınıflara kalıtım desteği vermektir. Dolayısıyla abstract class'lar hakkında bilgi sahibi olmak, interface'leri anlayıp kavramaya yardımcı olacaktır. Bu düşünce ile burada abstract class'lar hakkında bilmenizi istediğimiz özellikleri maddeler halinde sıralayacağız:

- Abstract class'ta hem abstract hem de normal metotlar olabilmektedir.
- Abstract class'taki abstract üyeler konusunda derived class'lara seçim şansı verilemez ve abstract class'ın mirasçısı sınıfın bütün abstract üyelerinin hazırlanması şarttır.

Interface'ler ile abstract sınıflar benzer işleve sahiptir. Daha önce sözü edildiği üzere abstract sınıflar, abstract olmayan yani normal metotlara sahip olabilirken interface'ler sahip olamıyorlar. Bazen içinde bulunulan şartlara göre mevcut interface'ler, hazırlanacak abstract sınıfa uygulanmak istenebilir. Bu konuda bilgi vermek üzere aşağıdaki gibi herhangi bir metoda sahip olmayan bir abstract sınıf hazırladık ve bu abstract sınıfa elimizdeki mevcut interface'lerden birisini uyguladık.

```
public interface Islemler1
{
 int Yil
  {
   get; set;
  }
 int Kare(int X);
}
public abstract class Ozet : Islemler1
{
}
```

C# derleyicisi burada yaptığımız işleme itiraz eder. Çünkü "Ozet" sınıfı "Islemler1" adını verdiğimiz interface'in gereklerini yerini getirmiş değildir. Başka bir deyişle bu andan itibaren "Ozet" adını verdiğimiz sınıf Islemler1'den gelen metot ve özelliklere sahip olmalıdır. Bu amaçla burada "Ozet" adını verdiğimiz sınıfı aşağıdaki gibi düzenledik.

```
public abstract class Ozet : Islemler1
{
 private int yil;
 public int Yil
 {
  get
  {
   return Yil;
  }
  set
  {
    Yil = value;
  }
 }
```

```
 public int Kare(int X)
 {
   return X * X;
 }
}
```

Bu abstract sınıf hangi sınıfa uygulanırsa o sınıfta Interface1'den gelen metotların override edilmesi şart değildir. Çünkü yukarıda örnek olması için hazırladığımız Ozet adlı sınıftaki Kare() metodu interface kaynaklıdır. Bu metot abstract class kaynaklı olsaydı durum değişebilirdi. Konumuz interface'ler olduğu için interface'i uyguladığımız sınıfta abstract metotlara yer vermedik. Aşağıda verilen kodda bu abstract sınıfı normal bir sınıfa nasıl uygulayıp kullandığımızı görebilirsiniz.

```
class Program : Ozet
{
 public static void Main()
 {
  Program nesne = new Program();
  int sonuc = nesne.Kare(10);
  Console.WriteLine(sonuc);
 }
}
```

Interface Referansları

Interface'lerle ilgili olarak şimdiye kadar anlatılardan fark etmiş olacağınız gibi interface'ler metotların, özelliklerin ve indeksleyicilerin imzalarını içerebilmektedir. Dolayısıyla, içinde işletilecek herhangi bir satır olmadığı için **new** anahtar kelimesi kullanarak söz konusu interface tipinde nesneler hazırlanamaz.

Bu tespiti yaptıktan sonra nesneye yönelik programlama dillerinde interface'lerin neden önemli olduğuna cevap niteliğinde interface referanslarından söz edeceğiz. Konuyu adım adım anlatabilmek için önce aşağıdaki gibi kod yazarak bir ArrayList nesnesi hazırlayıp birkaç eleman ekledik.

```
private void button1_Click(object sender, EventArgs e)
{
 ArrayList liste;
 liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
}
```

Sizin için bilinmedik herhangi bir şey içermeyen bu kodda önce ArrayList tipinde yerel bir değişken tanımladık. Sonrasın new anahtar kelimesinden yararlanıp ArrayList nesnesi hazırlayıp bu nesnenin referansını "liste" adını verdiğimiz yerel değişkene aktardık.

Bu şekilde ArrayList tipinde nesne hazırlandıktan sonra ArrayList sınıfının statik olmayan metotlarını bu nesne üzerinden işletmenin mümkün olduğunu biliyorsunuz. Yukarıda bu metotlardan Add'i kullandık. Şimdi ise bu ArrayList nesnesi üzerinden ArrayList sınıfının **GetEnumerator**() metodunu işleteceğiz. Bu metot geriye IEnumerator nesnesi gönderdiği için aşağıdaki gibi 2 satır yazdık.

```
private void button1_Click(object sender, EventArgs e)
{
 ArrayList liste;
 liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 IEnumerator numaraliListe;
 numaraliListe = liste.GetEnumerator();
}
```

Koda sonradan eklediğimiz ilk satırda IEnumerator tipinde bir yerel değişken tanımladık. Devamında ArrayList sınıfının GetEnumerator() metodunu yukarıda hazırlayıp liste adını verdiğimiz ArrayList nesnesi üzerinden işlettik. Bu metodun geriye gönderdiği IEnumerator nesnesinin referansını ise IEnumerator tipindeki yerel değişkene aktardık. Bu andan itibaren ArrayList sınıfının IEnumerator arayüzü kaynaklı metotlarını bu IEnumerator nesnesi üzerinden işletebileceğinizi biliyorsunuz. Şimdi bu koda sonradan ekleyip IEnumerator nesnesi hazırladığımız 2 satır üzerinde biraz konuşacağız.

- Burada button1_Click() adlı metodun içinde bulunduğunuz sırada bellekte 4 elemanlı bir ArrayList nesnesinin bulunduğunu ve bu nesnenin referansının burada "liste" adını verdiğimiz yerel değişkende tutulduğunu biliyoruz.
- Bu yerel değişken aracılığıyla bellekte tutulan ArrayList nesnesi üzerinde ArrayList sınıfının statik olmayan metotları ile istenen işlemin yapılabildiğini biliyoruz.
- Kodun devamında bellekteki bu ArrayList nesnesinden yola çıkıp bir IEnumerator nesnesi hazırladık. Bu nesne sayesinde elimizdeki ArrayList nesnesinin IEnumerator arayüzü kaynaklı metotlarını işletebiliriz.
- Kodun işletimi metodun sonuna geldiğinde ArrayList nesnesinin referansını tutan yerel değişkenler bellekten silineceği için muhtemelen bellekte bir süre daha yaşayacak ArrayList nesnesine erişim sağlama imkanımız kalmaz.

Konunun iyice anlaşılması için bu kodda değişikliğe gidip new anahtar kelimesi ile hazırladığımız ArrayList nesnesinin referansını IList tipindeki yerel değişkene aktardık. En başta belirtelim ki bu 2 satırlık kodun hataya neden olmama nedeni ArrayList sınıfının IList arayüzünü uygulamış olmasından kaynaklanıyor.

```
private void button1_Click(object sender, EventArgs e)
{
 IList liste;
 liste = new ArrayList();
}
```

Tam bu noktada bu kitabın Koleksiyon Sınıfları adlı bölümüne bakıp IList arayüzünün yapısına bakmak faydalı olabilir. 2 satırlık bu kodun üzerinde biraz konuşup konuyu iyice netleştirmeyi hedeflemekteyiz:

- İlk satırda IList tipinde yerel bir değişken tanımladık.
- İkinci satırda ise new anahtar kelimesinden yararlanıp bir ArrayList nesnesi hazırladık ve bu nesnenin referansını IList tipindeki yerel değişkene aktardık.
- IList tipindeki bu yerel değişken aracılığıyla, ArrayList sınıfının IList arayüzü kaynaklı metotlarını işletip bellekteki ArrayList nesnesi üzerinde işlem yapmak mümkündür. (Bu cümle hem uzun hem de çok şeye gönderme yaptığı için size karışık gelmiş olabilir. Dönüp tekrar okumakta fayda olabilir.)

Şimdi yukarıda verdiğimiz 2 satırlık koda ekleme yapıp ArrayList sınıfının IList kaynaklı metotlarını işletmeyi deneyeceğiz. Aşağıda verilen kodda kullandığımız Add() metodu, IList kaynaklı bir metot olmasaydı kullanamazdık.

```
private void button1_Click(object sender, EventArgs e)
{
 IList liste;
 liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
}
```

Bu kodda özellikle vurgulamak istediğimiz önemli ayrıntılar var:

- Burada Add() metodunu IList nesnesi üzerinden işlettik. Başka bir anlatımla burada IList tipindeki "liste" adındaki yerel değişken, Add() metodunu işletmemize aracılık etti. Çünkü IList nesnesi, ArrayList nesnesinin referansını bilmektedir.
- Add() metodu bu şekilde IList nesnesi üzerinde işletildiğinde gerçekte bellekteki ArrayList nesnesine eleman ekler. Yoksa, IList nesnesine eklenen bir şey yoktur. IList nesnesi sadece ArrayList sınıfının IList arayüzü kaynaklı metotlarının işletilmesine, IList kaynaklı özelliklerinin içeriklerinin okunmasına aracılık eder.
- Kodun işletimi bu metodun sonuna geldiğinde IList tipindeki yerel değişken silineceği için bellekte bir süre daha yaşayacak ArrayList nesnesine erişmek mümkün olmamaya başlar.

Şimdi yukarıda interface referansları hakkında söylediklerimiz konusunda sizleri ikna etmek amacıyla yukarıda verilen koda ekleme yapacağız. Yoksa burada yapacağımız işlemin pratik bir değeri yoktur. Tabi, burada interface'lerle ilgili olarak söylenenlerin yerli yabancı başka kaynaklarda bulunma ihtimali çok düşüktür.

```
private void button1_Click(object sender, EventArgs e)
{
 IList liste;
 liste = new ArrayList();
 liste.Add("Fenerbahçe");
```

```
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 ArrayList takimlar = (ArrayList)liste;
 foreach (var takim in takimlar)
  {
    listBox1.Items.Add(takim);
  }
}
```

Burada ne mi yaptık? Önce durum tespiti yapmak gerekir:

- Bellekte bir ArrayList nesnemiz vardı.
- Bu nesnenin referansını IList tipindeki yerel değişkende tutuyorduk.
- Bu yerel değişken aracılığıyla belleğin heap kısmındaki ArrayList nesnemize birkaç eleman ekledik.
- En son olarak bu yerel IList tipindeki değişkenden yararlanıp belleğin heap kısmında tutulan ArrayList nesnemizin referansını bu kez ArrayList tipindeki yerel değişkene aktardık. Bu hazırlıktan sonra belleğin heap kısmında tutulan ArrayList nesnesi üzerinde işlem yapmak için kullanabileceğimiz 2 yerel değişkenimiz vardır. Birisi IList, diğeri ArrayList tipindedir. Bu nedenle aşağıda verilen kod işletildiğinde ListBox'a 6 takımın adı aktarılır.

```
private void button1_Click(object sender, EventArgs e)
{
 IList liste;
 liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 ArrayList takimlar = (ArrayList)liste;
 liste.Add("Göztepe");
 takimlar.Add("Bursaspor");
 foreach (var takim in takimlar)
  {
   listBox1.Items.Add(takim);
  }
}
```

Şimdiye kadar interface referansları üzerine konuşurken .NET Framework ile gelen class ve interface'leri kullandık. Şimdi benzer işlemi kendimiz yapacağız. Bu amaçla yukarıda hazırladığımız kısıtlı özelliklere sahip 2 interface'i aşağıda tekrar verdik.

```
public interface IIslemler1
{
 int Kare(int X);
}
public interface IIslemler2
{
 int Kup(int X);
}
```

Yararlanacağımız interface'leri bu şekilde hazırladıktan sonra üzerinde çalıştığımız uygulamada bu interface'leri uygulayan bir class hazırladık. Bu class'a bu interface'leri uyguladığımız için sahip oldukları metotları class'ta kodladık.

```
public class Islem : IIslemler1, IIslemler2
{
 public int Kare(int Sayi)
 {
  return Sayi * Sayi;
 }
 public int Kup(int Sayi)
 {
  return Sayi * Sayi * Sayi;
 }
}
```

Buraya kadar sizin için yeni bir şey yoktur. Burada "Islem" adını verdiğimiz sınıf çok kapsamlı bir sınıf olabilirdi ve bize sadece "Islemler1" arayüzü kaynaklı metotlar gerekebilirdi. Bu durumda yapmamız gereken "Islem" sınıfının yapıcı metodunun geriye gönderdiği referansı "IIslemler1" tipindeki bir yerel değişkene aktarmaktır. Bu amaçla uygulamamızda Main() metoduna sahip class'ı aşağıdaki gibi düzenledik.

```
class Program
{
 public static void Main()
 {
  IIslemler1 nesne1 = new Islem();
  int sonuc = nesne1.Kare(10);
  Console.WriteLine(sonuc);
 }
}
```

Main() metodunda "Islem" sınıfı tipindeki nesnenin referansını "IIslemler1" tipindeki yerel değişkene aktardığımız için sadece "IIslemler1" arayüzü kaynaklı metotları kullanabiliriz. Bu arayüzde "Kare()" adında bir tek metot vardı. Eğer sadece "IIslemler2" arayüzü kaynaklı metot ve sabitlere gerek duymuş olsaydık aşağıdaki gibi düzenleme yapabilirdik.

```
class Program
{
 public static void Main()
 {
  IIslemler1 nesne1 = new Islem();
  int sonuc = nesne1.Kare(10);
  Console.WriteLine(sonuc);
  IIslemler2 nesne2 = new Islem();
  sonuc = nesne2.Kup(10);
  Console.WriteLine(sonuc);
 }
}
```

13. BÖLÜM

DELEGATE HAZIRLAMAK ve KULLANMAK

C# programcıları için Visual Studio'dan yararlanıp .NET Framework ile gelen kontrollerin olaylarını temsil eden metotları hazırlamak son derece kolay olsa bile bu işlemin geri planında bulunan delegate'leri kavramanın zor olduğu görüşü hakimdir. Buna rağmen bu bölümde yazılanları sonuna kadar okuyup verilen kısa örnekleri denerseniz delegate kavramı sizin için anlaşılmaz olmaktan çıkacaktır.

Bu bölümü hazırlarken şunları hedefledik:

- Ortalama bir C# programcısının delegate'lerle ilgili olarak öğrenmek isteyeceğini düşündüğümüz bütün ayrıntılardan söz etmek.
- Her yazılan cümlenin ilk okumada anlaşılmasını sağlamak. Bunu yaparken tekrarların olacağını bilmek.
- Delegate'leri olaylarla (events) birlikte anlatmak daha anlamlı olsa bile başlangıçta delegate'leri olaylardan bağımsız ele alacağız. Bu tercihimizin nedeni, zor anlaşılan 2 konuyu birlikte ele alıp anlama zorluğunu arttırmamaktır.

Delegate'lerle ilgilenmeden C# ile standart uygulamalar geliştirmek mümkündür. Delegate'lerle ilgilenmeyen C# programcıları muhtemelen aynı işlem için daha fazla kod yazarlar. Bize sorarsanız delegate'lere kayıtsız kalmayın. Çünkü birçok yerde karşınıza çıkacaktır. Örneğin delegate'lere kayıtız kalırsanız .NET Framework ile gelen kontrollere ait olayları, konunun geri planını bilmeden kullanırsınız; isimsiz metotları anlamakta zorlanırsınız; lambda ifadelerine uzak durmak zorunda kalırsınız.

Konu .NET Framework ile gelen görsel kontrolleri kullanmak olduğunda mevcut olayları için metot yazarken Visual Studio yardımıyla hazır delegate'leri kullanmak isterseniz. Bu tercih programcının işini kolaylaştırır. Delegate ve event hazırlamaya daha çok kendi sınıflarını hazırlayacak olanlar gerek duyar. C# uygulaması geliştiren herkes ister istemez kendi sınıflarını hazırlayacaktır. Bizim burada sözünü ettiğimiz sınıflar, library haline getirilip diğer uygulamalarda kullanılan sınıflardır.



Delegate'ler .NET dillerine özel (C# ve VB) bir yapıdır. Başka bir deyişle C++ ve Java gibi nesneye yönelik programlama dillerinde delegate'ler yoktur. Bu cümleye düz mantıkla yaklaşılırsa şunlar söylenebilir:

* Madem, C# programlama dili, yapı ve yaklaşım olarak Java'ya çok yakındır; Java'da delegate'ler olmadığına göre, C# programcıları delegate'lerle fazla yüz göz olmadan iyi programlar yazabilmelidir.

* Bu düz mantığı destekleyen bir diğer konu ise birçok kaynakta delegate'lerin yeteri ayrıntıda işlenmediğini görüp, delegate'lerin C# programcısı için çok önemli olmadığını düşünmüş olabilirsiniz.

* Bu kitapta, delegate'lerle ilgili olarak yukarıda sözü edilen düz mantığı destekleyen bir deneme yaptık: Bu kitabın Generic koleksiyon sınıfları hakkında bilgi içeren bölümünde

List sınıfının **ForEach()** metodunu bir sayfa boyunca anlatırken delegate kelimesini hiç kullanmadık. Daha sonra başka bir bölümde aynı ForEach() metodunu .NET Framework ile gelen bir delegate'i anlatmak üzere kullandık.

* Ezbere anlatım tekniğini isteseydik bu kitapta .NET Framework ile gelen Button, ListBox vs gibi kontrollerin olaylarını anlatırken delegate kelimesini hiç kullanmayabilirdik. Öyle bir anlatımda bile okurlar o kontrol ile yapılan işlemleri anlardı. Ancak bu anlama ezbere bir anlama olurdu.

* Düz mantık böyle dese bile C#'ta, dolayısıyla Visual Basic'te her şey dönüp dolaşıp delegate'lere uğruyor. Yani delegate'leri iyice kavramazsanız ya bazı şeyleri zorlanma pahasına ezbere yaparsanız ya da hiç yapamazsınız. Bu nedenle bu bölümü dikkatlice okumanızı öneriyoruz.

* Java'da delegate'ler yoksa Java programcılarının elleri bağlı mıdır sorusu akıllarına gelenlere cevabımız şudur: Java'da metotları işletmek, olayları yakalamak için başka mekanizmalar kullanılıyor. Dolayısıyla C#'ta her yerde karşınıza çıkan delegate'lerin Java'da olmaması soruna neden olmaz.

Tek cümle ile ifade etmek gerekirse, programcılar metotların adreslerini tutma gereğini duyduklarında **delegate** hazırlarlar. Bu sayede işletilecek metotları çalışma anında belirleme imkanını elde ederler. Yoksa, mevcut metotların ne zaman ve hangi şartlarda işletileceği kod yazımı sırasında belirtilir ki bu da programcının elini ayağını bağlamakla eş anlamlıdır.

Delegateleri kullandığımız ikinci alan olaylardır. Bir olay gerçekleştiği zaman bir veya daha fazla metot, bundan haberdar edilirken delegate'lerden yararlanılır. Olay gerçekleştiği zaman hangi metotların işletileceği delegeler yardımıyla işaret edilmektedir. Ayrıca delegeler çok kanallı programlamada ve asenkron programcılık tekniklerinde kullanılmaktadır.

Bu bölümde bütün yazdıklarımızın ilk okumada anlaşılmasını hedeflediğimiz için konuyu uzatmayı göze alıp sıfır noktasından yola çıkıp anlatmaya başlayacağız. Bu amaçla aşağıda verilen uygulamada parametre olarak aldığı 2 int sayıyı toplayan **Topla()** adında bir metot hazırladık. Yani delegate'leri anlatmaya basit bir metot hazırlayarak başladık.

```
using System;
class Program
{
 public static int Topla(int x, int y)
 {
  return (x + y);
 }
 public static void Main()
  {
   int sayi1 = 100;
   int sayi2 = 150;
   int sonuc = Topla(sayi1, sayi2);
   Console.WriteLine(sonuc);
  }
}
```

Sizin için yeni bir şey içermeyen bu uygulama derlenip çalıştırıldığında kod Main() metodundan itibaren işletilmeye başlanır.

Kodun işletimi Main() metodu içinde Topla() metodunun çağrıldığı satıra geldiğinde, adıyla çağrılan bu metot "bulunup" işletilir.

Bu uygulamadaki Topla() ile Main() metotlarının yerleri değiştirilse değişen bir şey olmaz.

Programcılar bazen ellerindeki mevcut metotları bu şekilde adlarıyla işletmek istemezler. Çünkü metotları adları yerine referanslarını kullanarak işletmek programcıların elini güçlendirir. Metotların referanslarına ihtiyaç duyulduğunda **delegate**'ler hazırlanır. Hemen hatırlamak gerekirse, olaylara dayalı bir programcılık modeli olan NET programcılığında olaylar önemlidir ve delegate'siz olay olmaz.



Yukarıda verdiğimiz uygulamanın derlenip çalıştırılması demek, uygulamanın makine diline çevrilmiş halinin belleğe yüklenmesi demektir. Derleyici Main() metodunu uygulamanın başlangıç noktası olarak ayarlayacağı için hemen uygulanın bellekteki kopyasında Main() metodunun olduğu adrese geçilir. Ardından programın işletimi Topla() metodunun çağrıldığı satıra gelince oradan Topla() metodunun bellekte bulunduğu adrese geçilir. Topla() metodunun işletimi tamamlandığında ise Main() metodunda kalınan satıra geçilir ve kodun işletimine oradan devam edilir.

Kısa bir cümle ile özetlemek gerekirse; delegate'ler metotların adreslerini tutabilmekte ve işletilmelerine aracılık etmektedirler. Delegate'lerin nasıl hazırlanıp kullanıldığını anlatmaya devam etmek için yukarıda verilen konsol uygulamasına aşağıdaki gibi bir satır ekledik. Bu satır sayesinde "Temsilci" adında bir delegate "tipi" hazırlamış olduk.

```
class Program
{
 delegate int Temsilci (int X, int Y);
 public static int Topla(int x, int y)
  {
   return (x+y);
  }
 public static void Main()
 {
  int sonuc = 0;
  int sayi1 = 100;
  int sayi2 = 150;
  sonuc = Topla(sayi1, sayi2);
  Console.WriteLine(sonuc);
 }
}
```

Delegate'lerin asıl kullanım amacı; çağrılıp işletilecek metotları program yazılırken değil çalışma anında belirlemektir. Örneğin yukarıda hazırlayıp uygulamanın Main() metodu içinde adıyla çağırıp işlettiğimiz Topla() metodunun ne zaman ve hangi şartlarda işletileceği programın yazımı sırasında belirtiliyor. Delegate'lerle ilgilenme nedenlerimizden birisi işletilecek metodu çalışma anında belirleyebilmektir.

Verilen koddan fark etmiş olacağınız gibi delegate tipi hazırlanırken **delegate** anahtar kelimesinden yararlanılmaktadır. Gerçekte, bu anahtar kelime ile delegate tipi hazırlamak demek, geri planda System'de bulunan **Delegate** sınıfının örneğini almaktır. Delegate sınıfı **abstract** bir sınıf olduğu için bilinen şekilde **new** anahtar kelimesiyle örneği alınamaz.



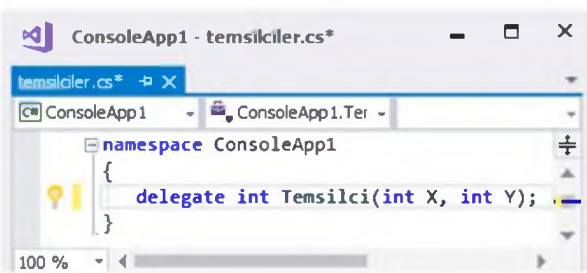
Oluşturduğumuz delegate tipi ile aynı ada sahip ve System.MulticastDelegate sınıfının mirasçısı bir sınıf hazırlanır. Yani delegate tipleri dolaylı olarak System.MulticastDelegate sınıfından türetilir. Çünkü o da System.Delegate sınıfından türetilmektedir. Bundan şunu anlamamız gerekebilir: Delegate hazırlar ve kullanırken Delegate ve MulticastDelegate sınıfının metotlarını kullanacağız demektir.

Yukarıda verdiğimiz kodda işaret ettiğimiz satır ile "Temsilci" adında bir **Delegate** tipi hazırlamış olduk. Bu delegate tipinden yola çıkılarak hazırlanacak delegate nesneleri geriye int tipinde değer gönderen ve int tipinde 2 parametreye sahip metotları temsil edebilir. Başka bir deyişle, burada "Temsilci" adını verdiğimiz delegate tipindeki nesneler anılan özelliklere sahip metotların adreslerini tutabilir.

Yukarıda verilen delegate hazırlama satırından "delegate" anahtar kelimesini çıkardığımızda geride kalanın bir metodun tanımlama satırından bir farkı olmadığını görebilirsiniz. Bu nedenledir ki, delegate'leri bir veya birden fazla metodun tetiklenmesine yardım eden metottur diye tanımlayanlar vardır. Yani delegate'ler sayesinde metotların tetiklenmesi dinamik olarak gerçekleşebiliyor.

Delegate konusunun kolay anlaşılması hedefimizi riske etmemek için fazla ilerlemeden şu noktalara vurgu yapmamız şarttır:

- Yukarıda delegate anahtar kelimesi ile delegate tipi tanımlamayı class bloğu içinde yaptık. İsteseydik bu tanımlamayı class bloğu dışında yani namespace bloğu içinde yapabilirdik. Delegate tipleri metotların içinde tanımlanamaz.
- Nasıl ki **class** anahtar kelimesi ile sınıf tanımı yapıldıktan sonra, o class'tan yararlanmak için **new** anahtar kelimesi ile nesne hazırlamak gerekiyorsa; **delegate** anahtar kelimesi ile tanımlanan tipten yola çıkıp **new** anahtar kelimesi ile delegate nesnesi hazırlamak gerekecektir.
- Uygulamanızda çok sayıda delegate tipi hazırlamanız gerekiyorsa delegate tipi tanımlamayı ayrı bir kod dosyasında yapabilirsiniz. Bu ayrıntı açığa çıksın diye üzerinde çalıştığımız uygulamada ayrı bir kod dosyası hazırladık ve yukarıda class bloğu içinde delegate tipi tanımlamak amacıyla yazdığımız satırı bu kod dosyasına taşıdık.



- Bir delegate nesnesi hazırladığınızda elinize, uygun bir metodun adresini tutabilecek bir araç geçmiştir denilebilir. Her metodun bellekte fiziksel bir konumu vardır ve metodun başlangıç adresi, metot işletilmek istendiğinde atlanılan adrestir. Bu adres, bir delegate nesnesine atanabilir. Bu atama işleminden sonra ilgili metot delegate nesnesi aracılığıyla işletilebilir. Çalışma anında, delegate nesnesinin tuttuğu adresi, başka bir deyişle referans ettiği metodu değiştirmek demek bu delegate nesnesi ile gelişen şartlara bağlı olarak işletilecek metodu seçebilmek demektir.

Yukarıda "Temsilci" adını verdiğimiz delegate tipini hazırlarken üzerinde çalıştığımız konsol uygulamasındaki **Topla()** metodunun parametre yapısını ve geri dönüş tipini gözettik. Daha

doğrusu bu delegate tipindeki nesnelerin Topla() metodunu temsil edebilmesini istedik. Dolayısıyla, C# programcısı ya önce metotlarını hazırlar sonra o metotlara uygun delegate tipleri tanımlar, ya da ihtiyaç duyacağını düşündüğü delegate'leri önceden hazırlar.

Ayrı kod dosyasına taşıdığımız satır ile delegate tipi hazırladıktan sonra **Main**() metoduna yazılacak bir satır ile delegate nesnesi hazırlayıp bu delegate nesnesinin hangi metodu temsil edeceğini belirteceğiz. Bunu günlük hayatta bir avukata vekâlet vermeye benzetmek mümkündür. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz. Uygulamanın bu son halinde new anahtar kelimesi ile örneği alınan delegate tipinin yukarıda hazırladığımız kod dosyasında tanımlı olduğunu unutmamak gerek.

```
class Program
{
 public static void Main()
 {
  Temsilci temsilciNesne = new Temsilci(Topla);
 }
 public static int Topla(int x, int y)
 {
  return (x + y);
 }
}
```

Bu satır ile "Temsilci" adını verdiğimiz delegate tipinden yola çıkılarak Topla() adlı metodun "temsilciNesne" ile temsil edilmesi sağlanmaktadır. Bu satırı dikkatlice incelerseniz "Temsilci" adı verilen delegate tipinden yola çıkılarak **new** anahtar kelimesi ile nesne hazırlanmaktadır. Delegate tiplerinin örneği alınırken yani bir metodu temsil etmesini sağlama işlemi aşağıda yapıldığı gibi 2 satırda gerçekleştirilebilir.

```
Temsilci temsilciNesne;
temsilciNesne = new Temsilci(Topla);
```

Burada new anahtar kelimesiyle delegate nesnesi hazırlanırken delegate tipinin yapıcı metoduna, imzası uygun bir metodun parametre olarak verilmesi zorunludur. Yani önce delegate nesnesi hazırlayıp, temsil edeceği metodu daha sonra belirtmeye C# derleyicisi izin vermemektedir.

C# 2.0 ile birlikte gelen bir özellik olarak delegate tipinden yola çıkılarak bir metodu temsil edecek nesnenin hazırlanıp bir metotla ilişkilendirilmesi işleminde sadeleştirme yapıldı. Bu sayede aşağıda verilen 2 satır aynı işleve sahip olmaya başladı.

```
public static void Main()
{
 Temsilci temsilciNesne1 = new Temsilci(Topla);
 Temsilci temsilciNesne2 = Topla;
}
```

C# 2.0 ile gelen bu sadeleştirme bize göre henüz yolun başında olan birisi için kodu daha anlaşılmaz yaptığı için eski ve **new** anahtar kelimesinin olduğu stili tercih edeceğiz.



Bu bölümde anlatılan delegate tipi ile, delegate nesnesini karıştırmamak gerekir. Delegate tipini class'a, mevcut bir metodu temsil etmesi amacıyla delegate tipinden yola çıkılarak **new** anahtar kelimesi ile hazırlanan delegate nesnesini ise nesneye benzetmek mümkündür.

Yukarıda verilen koddan fark etmiş olacağınız gibi, delegate tipinden yararlanıp Topla() metodumuzu temsil etmesi amacıyla hazırladığımız delegate nesnemiz ile, Topla() metodumuz aynı sınıfın içinde bulunuyorlar. Dolayısıyla, burada "Temsilci" adını verdiğimiz delegate tipinin yapıcı metoduna Topla() metodunu parametre olarak verirken metot adından önce sınıf adını yazmamız gerekmedi.

Kendisi için delegate nesnesi hazırladığımız metot başka bir sınıfın içinde olsaydı bunu işaret etmemiz gerekirdi. Bu konu açığa çıksın diye üzerinde çalıştığımız uygulamada 2. bir class hazırladık ve Topla() metodunu bu class'a taşıdık. Bu nedenle delegate nesnesi hazırladığımız satırda değişikliğe gitmemiz gerekti.

```
namespace ConsoleApp1
{
 class Program
 {
   public static void Main()
   {
    Temsilci temsilciNesne = new Temsilci(Program1.Topla);
   }
  }
 public class Program1
 {
  public static int Topla(int x, int y)
   {
    return (x + y);
   }
 }
}
```

Yukarıda hazırlayıp kendisi için delegate nesnesi hazırladığımız Topla() metoduna dikkatlice baktığınızda statik bir metot olduğunu fark ederseniz. Dolayısıyla bu metoda erişimi sınıf üzerinden yaptık. İlk başta delegate nesnesini Topla() metodunun bulunduğu sınıfın sınırları içinde hazırladığımız için metot adından önce sınıf adını yazmak gerekmemişti. Bu metot static olmasaydı metoda erişim sınıf üzerinden değil nesne üzerinden sağlanırdı. Bu ayrıntıyı netleştirmek için Topla() metodumuzun tanımlama satırındaki "static" anahtar kelimesini kaldırdık ve delegate nesnesi hazırlama satırında değişikliğe gittik.

```
class Program
{
 public static void Main()
 {
  Program1 p1 = new Program1();
  Temsilci temsilciNesne = new Temsilci(p1.Topla);
 }
}
```

```
public class Program1
{
 public int Topla(int x, int y)
  {
    return (x + y);
  }
}
```

Buraya kadar anlatılanlara göre; nasıl ki **class** anahtar kelimesi ile sınıf hazırlıyorsak **delegate** anahtar kelimesi ile tek satırda delegate tipi hazırlıyoruz. Ardından tıpkı class'lardan yola çıkıp **new** anahtar kelimesi ile nasıl nesne hazırlıyorsak, aynı şekilde delegate'in örneğini alıyoruz. new anahtar kelimesi ile delegate'in örneğini alırken hangi metodu temsil edeceğini işaret ediyoruz. Bu satırlardan sonra Topla() metodunun yerine onun temsilcisi olan delegate nesnesini kullanabiliriz. Bu delegate nesnesini nasıl kullandığımızı aşağıda görebilirsiniz.

```
class Program
{
 public static void Main()
 {
  Program1 p1 = new Program1();
  Temsilci temsilciNesne = new Temsilci(p1.Topla);
  int sayi1 = 100;
  int sayi2 = 150;
  int sonuc = temsilciNesne(sayi1, sayi2); // Metot işletildi
  Console.WriteLine(sonuc);
 }
}
public class Program1
{
 public int Topla(int x, int y)
 {
   return (x + y);
 }
}
```

Burada ilk dikkat etmeniz gereken nokta, delegate nesnesi aracılığıyla ilgili metot işletilirken sanki delegate'in kendisi metotmuş gibi işletilmesidir. Delegate nesnesi aracılığıyla metotların işletilmesine delegate'lerin **invoke** edilmesi denilmektedir. Delegate nesneleri aracılığı ile metotları işletirken **Delegate** sınıfının **Invoke**() metodu kullanılabilir. Yani yukarıda verilen koddaki Main() metodu aşağıdaki gibi düzenlenirse değişen bir şey olmaz.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = new Temsilci(p1.Topla);
 int sayi1 = 100;
 int sayi2 = 150;
 int sonuc = temsilciNesne.Invoke(sayi1, sayi2);
 Console.WriteLine(sonuc);
}
```

Sanırız konu biraz anlaşıldı. Ancak sormakta haklısınız, böyle bir işlem için delegate nesnesi hazırlayıp kullanmaya gerek var mı? Elbette bu kadar basit bir işlem için delegate tipi hazırlamaya gerek yoktur. Yukarıdaki sayfalarda işaret edildiği gibi delegate'ler daha çok işletilecek metot çalışma anında belirlenmek ve metotlar olaylarla ilişkilendirilmek istendiği zaman işlevsel olmaktadır. Bildiğiniz gibi Windows Forms Applicaiton'larda formlar ve formlara yerleştirilen nesnelerle ilgili olaylar genellikle kullanıcının katkısıyla çalışma anında meydana gelir ve bu olaylara karşılık işletilecek metotlar delegate'lerin yardımıyla çalışma anında seçilip işletilir.

Delegate ve olayların nasıl birlikte işlev gördükleriyle ilgili ayrıntılar bir sonraki bölümde verilecektir. Bu nedenle bir sonraki bölümde **olaylar** hakkında bilgi sahibi olmadan ve delegate'lerle ilgili olarak yazılanların hepsini okumadan delegate'ler hakkında yorum yapmanızı önermeyiz. Yine hatırlatmak gerek; programcılara kolaylık sağlayan isimsiz metotları kullanırken delegate'lerden yararlanmak gerekmektedir.

Şimdi yukarıda delegate'lerle ilgili olarak yapmış olduğumuz işlemleri maddeler halinde özetleyeceğiz:

- Önce üzerinde çalıştığımız uygulamada Main() metodunun olduğu sınıfın içinde Topla() adında ve int tipinde 2 parametreye sahip bir metot hazırladık. Daha sonra bu metodu başka bir sınıfa taşıdık.
- Uygulamamızdaki tek sınıfın sınırları içinde yani class bloğunda **delegate** anahtar kelimesini kullanarak, geriye int tipinde bilgi gönderen ve int tipinde 2 parametreye sahip metotları temsil edebilecek bir delegate tipi hazırladık. Sonra delegate tipi hazırlama işlemini yaptığımız satırı ayrı bir kod dosyasına taşıdık.
- Bu delegate tipinden yola çıkıp new anahtar kelimesi ile delegate nesnesi hazırladık. New anahtar kelimesiyle delegate nesnesini hazırlarken yapıcı metoda elimizdeki metodu parametre olarak verdik. Dolayısıyla delegate nesnemiz Topla() adını verdiğimiz metodumuzu temsil eder hale geldi.
- Metodumuza temsilci yaptığımız delegate nesnesinden yararlanıp metodumuzu işlettik.

Delegate'lerle ilgili olarak şimdiye kadar yaptıklarımızı özetledikten sonra şimdi sırada henüz değinilmeyen bazı noktaları ifade etmek için bir şeyler yazacağız. Çünkü yukarıdaki sayfalarda yazdıklarımızın hepsi delegate hazırlayıp kullanmakla ilgili değildi.

Delegate tiplerinin varsayılan erişim belirleyicisi private'dir. O yüzden tanımlama satırının başına private veya protected yazmaya kalkarsak hata alırız. Yani üzerinde çalıştığımız uygulamaya sonradan eklediğimiz kod dosyasında delegate tipini aşağıdaki gibi tanımlamış olsaydık hata mesajı alırdık.

```
namespace ConsoleApp1
{
 private delegate int Temsilci(int X, int Y); //hata meydana gelir
}
```

Delegate tipleri hazırlanırken erişim belirleyicisi olarak **public**'i kullanabiliriz. Hatırlarsanız yukarıda aynı namespace içinde yer alan CS uzantılı bir kod dosyası hazırlamış ve delegate tipi tanımladığımız satırı Main() metoduna sahip sınıftan bu kod dosyasına taşımışık. Yani tanımladığımız delegate tipi için public erişim belirleyicisi kullanmamış olmamıza rağmen Main() metodu içinde delegate nesnesi hazırlarken sorun yaşamamıştık. Bir ayrıntıyı ortaya çıkarmak için delegate tipi tanımladığımız kod dosyasının farklı bir namespace'te olmasını sağlayacağız. Bu amaçla sözünü ettiğimiz kod dosyasını aşağıdaki gibi düzenledik.

```
namespace ConsoleApp2
{
 delegate int Temsilci(int X, int Y);
}
```

Bu değişiklikten sonra artık delegate tipimiz farklı bir namespace'tedir. Yani uygulamamızda 2 namespace bulunmaktadır. Dolayısıyla aşağıdaki gibi delegate nesnesi hazırlamaya çalışırsak hata meydana gelir.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = new Temsilci(p1.Topla); // hata
}
```

Çünkü bu şekilde yapılan kodlama, delegate tipinin aynı namespace'te olduğu varsayımına dayanır. Delegate tipimiz şimdi farklı bir namespace'te bulunduğu için delegate tipinden önce namespace adını yazmalıyız.

```
Program1 p1 = new Program1();
ConsoleApp2.Temsilci temsilciNesne = new
                ConsoleApp2.Temsilci(p1.Topla);
```

Bu delegate tipini uygulamanızda sıklıkla kullanıyorsanız her seferinde delegate tipi adından önce namespace adını yazmak istemiyorsanız **using** bloğunda söz konusu namespace'teki sınıf veya delegate'leri kullanmak istediğinizi aşağıda yapıldığı gibi belirtebilirsiniz.

```
using System;
using ConsoleApp2;
namespace ConsoleApp1
{
 class Program
 {
  public static void Main()
  {
   Program1 p1 = new Program1();
   Temsilci temsilciNesne = new Temsilci(p1.Topla);
   int sayi1 = 100;
   int sayi2 = 150;
   int sonuc = temsilciNesne(sayi1, sayi2); // Metot işletildi
   Console.WriteLine(sonuc);
   Console.ReadLine();
  }
 }
```

```
public class Program1
{
 public int Topla(int x, int y)
 {
  return (x + y);
 }
}
}
```

Üzerinde çalıştığımız uygulamada sadece bir delegate olması nedeniyle ayrı bir namespace'te olmasına gerek olmadığı için delegate tipimizi tanımladığımız satırı ayrı kod dosyamızdan yine eskisi gibi Main() metoduna sahip sınıfla aynı namespace'te olmasını sağladık.

Ayrıca yukarıdaki sayfalarda delegate tipi tanımlama satırını ayrı bir kod dosyasına almıştık. Uygulamamızda fazla delegate tipi tanılamayacağımız için bu satırı Main() metodunun olduğu kod dosyasında namespace bloğuna taşıdık. Bu sayede delegate'lerle ilgili olarak yazdığımız kodların hepsi bir arada olacaktır. Yani organizasyon aşağıdaki gibi olacaktır.

```
namespace ConsoleApp1
{
 delegate int Temsilci(int X, int Y);
 class Program
 {
  public static void Main()
   {
    Program1 p1 = new Program1();
    Temsilci temsilciNesne = new Temsilci(p1.Topla);
    int sayi1 = 100;
    int sayi2 = 150;
    int sonuc = temsilciNesne(sayi1, sayi2);
    Console.WriteLine(sonuc);
    Console.ReadLine();
   }
 }
 public class Program1
 {
  public int Topla(int x, int y)
  {
    return (x + y);
  }
 }
}
```

Bu koda yakından baktığınızda;

- Temsilci" adını verdiğimiz delegate tipinin erişim belirleyici anahtar kelime kullanılmadan namespace bloğunda hazırlandığı,
- "Temsilci" tipindeki delegate nesnesinin Main() metodu içinde hazırlandığı,
- Kendisi için delegate nesnesi hazırlanan Topla() adlı metodun statik olmadığı ve burada "Program1" adı verilen sınıfta kodlandığı fark edilir.

Namespace bloğunda delegate tipini tanımlarken public anahtar kelimesine yer verebilirdik. İleriki sayfalarda yapacağımız işlemlerden dolayı bu delegate tipini public yapmak durumunda kalacağız



Daha önce class'lar hakkında yazılanlardan bildiğiniz gibi C# derleyicisi class'ların statik olmasına izin vermektedir. Bu sayede class'ın bütün üyelerinin statik olması garanti altına alınmış oluyordu. Akıllara delegate'ler static olur mu sorusunun gelme ihtimali düşük olsa bile cevaplamakta fayda olabilir: Delegate'ler static olarak tanımlanamazlar.

.NET Framework ile birlikte sıklıkla kullanılan çok sayıda delegate hazır olarak gelmektedir. Bunların en ünlüsü ve çok kullanılanı **EventHandler** adlı delegate'tir. Bu delegate tipinin ayrıca generic versiyonu bulunmaktadır. Sıklıkla karşınıza çıkacak bu 2 delegate'in orijinal halleri aşağıda verildi. Bu bölüm dahilinde ileriki sayfalarda bu delegate'leri inceleyeceğiz.

```
public delegate void EventHandler(object sender, EventArgs e);
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Bir Delegate ve Birden Fazla Metot

Yukarıda delegate'lerle ilgili olarak işlem yaparken elimizde sadece bir metot vardı. Gerçekte delegate'ler, işletilebilinecek birden fazla metot söz konusu iken işlevsel olmaktadırlar. Bu nedenle delegate'lerle ilgili bazı noktaları açığa çıkarmak için elimizde birden fazla metodun olması gerekir. Bu amaçla uygulamamızdaki 2. sınıfta parametre olarak aldığı **int** tipindeki 2 sayıyı çarpıp sonucu geriye gönderen "Carp()" adında 2. bir metot hazırladık.

```
public class Program1
{
 public int Topla(int x, int y)
 {
  return (x + y);
 }
 public int Carp(int X, int Y)
 {
  return (X * Y);
 }
}
```

Bu metodu dikkatlice incelerseniz parametre yapısı ve geriye gönderdiği değerin tipi bakımından yukarıda tanımlayıp "Temsilci" adını verdiğimiz delegate tipi ile uyumludur. Başka bir deyişle yukarıda tanımlayıp "Temsilci" adını verdiğimiz delegate tipinden yola çıkılarak **new** anahtar kelimesi ile hazırlanacak nesne, Carp() adını verdiğimiz metodu da temsil edebilir. Çünkü **delegate** anahtar kelimesi ile "Temsilci" adını verdiğimiz delegate' tipini tanımlarken int tipinde 2 parametreye sahip metotları temsil etmesini istemiştik.

Şimdi uygulamamızdaki Main() metoduna ekleme yaparak mevcut delegate nesnemizle ayrıca Carp() adını verdiğimiz metodu da işleteceğiz. Yani "temsilciNesne" adını verdiğimiz delegate nesnesinin Topla() metodu temsil etmesini sağlayıp işlettikten sonra aynı işlemleri

bu kez Carp() metodu için yapacağız. Şu an itibarıyla delegate'lerle ilgili yazmış olduğumuz kodun tamamını aşağıda bir arada verdik.

```
namespace ConsoleApp1
{
 delegate int Temsilci(int X, int Y);
 class Program
  {
   public static void Main()
   {
    Program1 p1 = new Program1();
    Temsilci temsilciNesne = new Temsilci(p1.Topla);
    int sayi1 = 100;
    int sayi2 = 150;
    int sonuc = temsilciNesne(sayi1, sayi2);
    Console.WriteLine(sonuc);
    temsilciNesne = new Temsilci(p1.Carp);
    sonuc = temsilciNesne(sayi1, sayi2);
    Console.WriteLine(sonuc);
   }
  }
 public class Program1
 {
  public int Topla(int x, int y)
   {
     return (x + y);
   }
  public int Carp(int X, int Y)
   {
    return (X * Y);
   }
 }
}
```

Aynı delegate nesnesi 2. ve 3. metoda temsilci yapılırken delegate tipinin yapıcı metodunu kullanmak yeterli olmaktadır. Aşağıda verilen satır ile daha önce hazırladığımız delegate'in Carp() metodunu temsil etmesini sağladığımız için delegate nesnesinin Topla() metodu ile ilişkisi kalmamış oluyor.

```
temsilciNesne = new Temsilci(p1.Carp);
```

Burada kullanıldığı şekliyle "=" operatörü "temsilciNesne" adını verdiğimiz delegate nesnesinin mevcut içeriğini silmektedir. Nasıl ki normal bir değişkenin mevcut içeriğine başka bir bilgiyi eklerken "+=" operatöründen yararlanıyorsak delegate nesnelerine atama yaparken de "+=" operatöründen yararlanabiliyoruz. İleriki sayfalarda bu operatörden yararlanacağız.

Uygulamamızın en son halinde Main() metodunda önce Topla() metodu için bir delegate nesnesi hazırlanmaktadır. Başka bir deyişle Topla() metodunun "Temsilci" tipinde bir nesne tarafından temsil edilmesi sağlanmaktadır. Bu delegate nesnesi için Topla() metodunu işlettikten sonra "temsilciNesne" adını verdiğimiz nesnenin bu kez Carp() metodunu temsil etmesi sağlanmaktadır.

Multicast Delegeler

Yukarıdaki sayfalarda hazırladığımız delegate nesneleri bir metodu temsil ediyorlardı. Aslında bir delegate nesnesi aynı anda birden fazla metodu temsil edebilir. Bunun için, yani bir delegate nesnesinin 2. veya 3. bir metodu temsil etmesi için ek bir hazırlık veya ayarlama yapmak gerekmiyor. Farklı olarak yapılması gereken, delegate nesnesi 2. veya 3. metodun temsilcisi yapılırken "=" yerine "+=" operatörünü kullanmaktır. Bu operatör, Windows Forms uygulamalarında, **EventHandler** adlı delegenin metotları temsil etmesi sağlanırken sıklıkla kullanılmaktadır.



Konunun başlığını Multicast Delegeler olarak seçmiş olmamıza rağmen bu başlık bizce uygun değildir. Bu başlık hakkında biraz konuşmak gerekecek:

* Programcı önce bir delegate tipi hazırlar. Sonra bu delegate tipinden yola çıkıp bir delegate nesnesi hazırlar. Programcı delegate nesnesini hazırlarken bu nesneye mevcut bir metodu atayabilir. Bu durumda delegate nesnesi bir metodu temsil ediyor olur.

* Gelişen şartlara göre bu delegate nesnesinin 2. veya 3. bir metodu da temsil etmesi ihtiyacı olursa atama işlemi "+=" operatörü ile yapılırsa delegate nesnesinin mevcut temsilciliğine dokunulmadan 2. metodun da temsilcisi olması sağlanır.

* Gelişen şartlara göre programcı elindeki delegate nesnesinden bazı temsilciliklerini geri alabilir. Bütün bunlar olurken kullanılan delegate nesnesi aynı nesnedir.

* Bundan şu sonucu çıkarmak mümkündür, bütün delegate'ler yapı olarak zaten birden fazla metodu temsil edebilirler. Bu nedenle yerli-yabancı kaynaklardaki singel cast-multi cast delegate başlıklarıyla ilgilenmenize gerek yoktur.

Delegate'lerin birden fazla metodu temsil edebilme özelliği hakkında konuşmak için üzerinde çalıştığımız konsol uygulamasında "Bol()" adında 3. bir metot hazırladık. Bu kez yukarıda yaptığımızdan farklı olarak 3 metot için 3 ayrı delegate nesnesi hazırlayıp her metodu ayrı satırlarda çağırmayacağız.

Verilen örneğin daha anlamlı olması için Topla(), Carp() ve Bol() metotlarının geriye değer göndermesini engelleyip toplama, çarpma ve bölme işlemleri ile bulunan sonuçları metot dahilinde konsola yazdırdık. Çünkü yapacağımız ayarlama sonucu bir delegate nesnesi birden fazla metodu aynı anda temsil edebilecektir.

Mevcut bilgilerimiz ışığında bir değerlendirme yapmak gerekirse şunları söyleyebiliriz. Delegate nesnemiz birden fazla metodu temsil edip hepsi bir seferde işletilirse geriye gönderilen bilgilerden sadece birisini yakalayabiliriz. Bu nedenle yukarıda hazırlayıp kullandığımız Topla(), Carp() ve Bol() metotlarını **void** yaptık.

Bu metotları void yapınca, bu metotları temsil edecek ve "Temsilci" adını verdiğimiz delegate tipinin de void olmasını sağladık. Bu işlemin yani bir delegate'in aynı anda birden fazla metodu temsil etmesinin nasıl sağlandığını aşağıda görebilirsiniz.

```
delegate void Temsilci(int X, int Y);
class Program
{
 public static void Main()
  {
   Program1 p1 = new Program1();
   Temsilci temsilciNesne = null;
   temsilciNesne += new Temsilci(p1.Topla);
   temsilciNesne += new Temsilci(p1.Carp);
   temsilciNesne += new Temsilci(p1.Bol);
   temsilciNesne(30, 10);
  }
}
public class Program1
{
 public void Topla(int x, int y)
  {
   Console.WriteLine(x + y);
  }
 public void Carp(int x, int y)
  {
   Console.WriteLine(x * y);
  }
 public void Bol(int x, int y)
  {
   Console.WriteLine(x / y);
  }
}
```

Burada ne mi yaptık? "temsilciNesne" adını verdiğimiz delegate nesnesinin elimizdeki her 3 metodu da temsil etmesini sağladık. Bu işlemi yaparken önce aşağıda verilen satırı yazdık.

```
Temsilci temsilciNesne = null;
```

Bu sırada "temsilciNesne" henüz herhangi bir metodu temsil etmediği için **null** bilgisini aktardık. Ardından artı(+=) operatörü ile metotların referanslarını bu delegate nesnesine aktardık. Aşağıda verilen satırlar sayesinde aynı delegate nesnesi aynı parametre yapısına sahip 3 metodu da aynı anda temsil etmeye başlar.

```
temsilciNesne += new Temsilci(p1.Topla);
temsilciNesne += new Temsilci(p1.Carp);
temsilciNesne += new Temsilci(p1.Bol);
```

Burada kullandığımız "+=" operatörü, delegate nesnesinin daha önceden temsilciliğini yapması sağlanan metotları listeden çıkarmadan başka bir metodu da temsil etmesini sağladık. "+=" haricinde "-=", "=" operatörleri de kullanılabilir.

Mevcut bir metot eldeki delegate nesnesine "=" operatörü ile bağlandığında, delegate nesnesi sadece o metodu temsil edebilecektir. Yani delegate'lere yapılan ilk atamada sadece "=" operatörü kullanılabilir. Bu nedenle kafalar karışmasının diye yukarıda "+=" operatörünü kullanmadan önce null bilgisini delegate nesnesine aktardık. Konu üzerinde düşünmenizi sağlamak için Main() metodunun yukarıda verilen içeriğini aşağıdaki gibi düzenledik. Bu düzenlemenin kodun işlevi üzerinde bir etkisi yoktur.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = new Temsilci(p1.Topla);
 temsilciNesne += new Temsilci(p1.Carp);
 temsilciNesne += new Temsilci(p1.Bol);
 temsilciNesne(30, 10);
}
```

Bu şekilde düzenlenen konsol uygulaması çalıştırıldığında **temsilciNesne(30, 10);** satırı sayesinde Topla(), Carp() ve Bol() metotları sırayla işletilip konsola 40, 300 ve 3 yazılır. Gelişen şartlara göre bu delegate nesnesinin Carp() metodunu artık temsil etmesini istemiyorsanız aşağıdaki gibi bir satır yazmanız yeterlidir.

```
temsilciNesne -= new Temsilci(Carp);
```

Şimdi yukarıda verdiğimiz kodda biraz değişiklik yapıp bir delegate nesnesinin birden fazla metodu temsil etme ayarlamasını "Delegate" sınıfının **Combine**() metodu ile yapacağız. Bu amaçla yukarıda verilen koddaki Main() metodunu aşağıdaki gibi düzenledik.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci temsilci1 = new Temsilci(p1.Topla);
 Temsilci temsilci2 = new Temsilci(p1.Carp);
 Temsilci temsilci3 = new Temsilci(p1.Bol);
 Temsilci Temsilci_hepsi = null;
 Temsilci_hepsi = (Temsilci)Temsilci.Combine(temsilci1,
                                temsilci2, temsilci3);
 Temsilci_hepsi(30, 10);
}
```

Main() metodumuzun yeni halinde, önce elimizdeki her metot için ayrı bir delegate nesnesi hazırladık. Ardından başlangıçta herhangi bir metodu temsil etmeyen, yani null aktarılan "Temsilci_hepsi" adını verdiğimiz 4. bir delegate nesnesi hazırladık. En son olarak bu 3 delegate nesnesini Delegate sınıfının **Combine**() metoduna parametre olarak verdik. Bu sayede "Temsilci_hepsi" nesnesini 3 metodu birden temsil eder hale getirmiş olduk.

Yukarıda Combine() metodunu kullanırken kodun ilk bakışta anlaşılmasını istedik. Yoksa Combine() metodunu aşağıdaki gibi kullanıp kodu kısaltmak mümkündür.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci Temsilci_hepsi = null;
 Temsilci_hepsi = (Temsilci)Temsilci.Combine(
       new Temsilci(p1.Topla),
            new Temsilci(p1.Carp),
                 new Temsilci(p1.Bol));
 Temsilci_hepsi(30, 10);
}
```

Konunun ayrıntılarını ortaya çıkarmak için **Combine**() metodunu şimdi farklı bir şekilde kullanmayı deneyeceğiz. Bu amaçla önce "Temsilci" adını verdiğimiz delegate tipinden yola çıkarak **Topla**(), **Carp**() ve **Bol**() metotlarını temsil etmesi için 3 adet delegate nesnesi hazırladık.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci tNesne1 = new Temsilci(p1.Topla);
 Temsilci tNesne2 = new Temsilci(p1.Carp);
 Temsilci tNesne3 = new Temsilci(p1.Bol);
}
```

Ardından "Temsilci" yani kendi hazırladığımız delegate tipinde bir dizi değişken tanımladık. Bu dizi değişkeni tanımlama satırında bu 3 delegate nesnesini ilk değerler olarak verdik.

```
delegate void Temsilci(int x, int y);
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci tNesne1 = new Temsilci(p1.Topla);
 Temsilci tNesne2 = new Temsilci(p1.Carp);
 Temsilci tNesne3 = new Temsilci(p1.Bol);
 Temsilci[] Temsilci_dizi = {tNesne1, tNesne2, tNesne3};
}
```

Bu noktada hazırlayıp tNesne1, tNesne2 ve tNesne3 adını verdiğimiz delegate nesneleri 3 elemanlı dizi değişkende tutulmaktadır. Şimdi 3 elemanlı dizi değişkende tutulan delegate nesnelerini ayrı bir delegate nesnesinde birleştireceğiz. Birleştirme işlemini **Delegate** sınıfının **Combine**() metodu ile yapacağız.

```
delegate void Temsilci(int x, int y);
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci Temsilci1 = new Temsilci(p1.Topla);
 Temsilci Temsilci2 = new Temsilci(p1.Carp);
 Temsilci Temsilci3 = new Temsilci(p1.Bol);
 Temsilci[] Temsilci_dizi = {Temsilci1, Temsilci2, Temsilci3};
 Temsilci Temsilci_hepsi =
              (Temsilci)System.Delegate.Combine(Temsilci_dizi);
 Temsilci_hepsi(10, 30);
}
```

Bu şekilde düzenlenen uygulama çalıştırıldığında **Temsilci_hepsi(30, 10)** satırı sayesinde Topla(), Carp() ve Bol() metotları sırayla işletilip konsola 40, 300 ve 3 yazılır. Gelişen şartlara göre delegate nesnesinin Carp() metodunu artık temsil etmesini istemiyorsanız aşağıdaki gibi bir satır yazmanız yeterlidir. Remove() metodunun kullanıldığı bu satır yukarıda verdiğimiz kodun sonuna eklenip "Temsilci_hepsi(30, 10)" gibi bir satırla delegate işletilirse bu kez 2 metot işletilir.

```
Temsilci_hepsi =
       (Temsilci)System.Delegate.Remove(Temsilci_hepsi, Temsilci2);
```

Delegate nesnesinin temsil ettiği bütün metotlarla olan bağlantısını veya temsilciliklerini bir seferde iptal etmek istiyorsanız Delegate sınıfının **RemoveAll**() metodunu kullanabilirsiniz.

Şimdi bu "Temsilci_hepsi" adını verdiğimiz delegate nesnesinin temsil ettiği metotları listeleyeceğiz. Bunu yaparken Delegate sınıfının **GetInvocationList**() metodundan yararlanacağız. Bu amaçla yukarıda verilen Main() metodunu aşağıdaki gibi düzenledik.

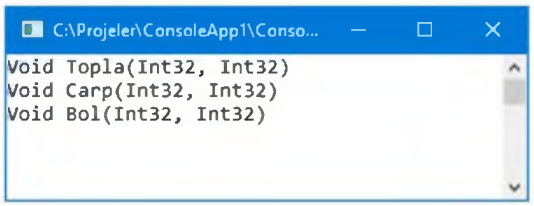
```
static void Main(string[] args)
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = null;
 temsilciNesne += new Temsilci(p1.Topla);
 temsilciNesne += new Temsilci(p1.Carp);
 temsilciNesne += new Temsilci(p1.Bol);
 Delegate[] Temsilci_dizi;
 Temsilci_dizi = temsilciNesne.GetInvocationList();
}
```

Burada sizin için yeni olan 2 satır var. İlkinde Delegate tipinde bir dizi değişken tanımladık. Çünkü bir delegate nesnesinin temsilcisi olduğu metotlar öğrenilmek istenildiğinde başvurulan **GetInvocationList**() metodu geriye bir Delegate dizisi göndermektedir. Sizin için yeni dediğimiz 2. satırda ise bu metodu işletip elimizdeki delegate nesnesinin temsil ettiği metotların birer Delegate nesnesi olarak dizi değişkene aktarılmasını sağladık.

Birden fazla metodu temsil eden delegate nesnemizin temsil ettiği metotları **GetInvocationList**() metodu sayesinde birer Delegate nesnesi olarak dizi değişkende topladıktan sonra bu Delegate nesneleri üzerinde istenilen işlemler yapılabilir. Şimdi bu Delegate nesnelerinin adlarını ve temsil ettikleri metotları konsola listeleyeceğiz. Bu amaçla aşağıdaki gibi bir foreach döngüsü hazırladık.

```
static void Main(string[] args)
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = null;
 temsilciNesne += new Temsilci(p1.Topla);
 temsilciNesne += new Temsilci(p1.Carp);
 temsilciNesne += new Temsilci(p1.Bol);
 Delegate[] Temsilci_dizi;
 Temsilci_dizi = temsilciNesne.GetInvocationList();
 foreach (Delegate item in Temsilci_dizi)
 {
  Console.WriteLine(item.Method.ToString());
 }
}
```

Foreach döngüsü içinde Delegate nesnelerinin **Method** özelliğinin içeriğini konsola yazdırdık. Bu şekilde düzenlenen kod işletilirse aşağıdaki gibi bir sonuç alınır. Verilen ekran görüntüsünden fark edeceğiniz gibi ToString() metodu sayesinde ilgili metotların imzaları hakkında bilgi edinilmektedir.



GetInvocationList() metodu sayesinde dizi değişkende toplanan Delegate nesneleriyle ilgili metotları işletmek istiyorsanız Delegate sınıfının **DynamicInvoke**() metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 Program1 p1 = new Program1();
 Temsilci temsilciNesne = null;
 temsilciNesne += new Temsilci(p1.Topla);
 temsilciNesne += new Temsilci(p1.Carp);
 temsilciNesne += new Temsilci(p1.Bol);
 Delegate[] Temsilci_dizi;
 Temsilci_dizi = temsilciNesne.GetInvocationList();
 Temsilci_dizi[0].DynamicInvoke(10, 20);
}
```

Burada yapılan işlemi başka bir şekilde ifade etmek gerekirse, birden fazla metodu temsil ettiği için multicast delegate adı verilen delegate nesnesindeki metotları GetInvocationList() ve DynamicInvoke() metotları yardımıyla tek tek işletmek mümkündür. Bu sayede kullanıcıya işleteceği metodu seçme imkanı verilebilir. Bir sonraki başlıkta, işletilecek metotların çalışma anında seçilmesi üzerinde durulacaktır.

İşletilecek Metodu Çalışma Anıdan Seçmek

Yukarıdaki sayfalarda delegate'lerin en önemli işlevinin, işletilecek metodun çalışma anında seçilmesine ön ayak olmalarıdır demiştik. Şimdi uygulamamızda yapacağımız değişiklik sonucu işletilecek metodun çalışma anında seçilmesini sağlayacağız. Bunu yapmamızdaki amaç, delegate'ler hakkında şimdiye kadar edindiğiniz bilgileri netleştirmektir. Uygulamamızın son hali aşağıdaki gibidir.

```
delegate void Temsilci(int X, int Y);
public class Program
{
 public static void Main()
 {
  Program1 p1 = new Program1();
  Temsilci delege = null;
  Console.Write("İşletmek istediğiniz metot:");
  string str = Console.ReadLine();
  if (str == "Topla")
  {
   delege = new Temsilci(p1.Topla);
   delege(30, 10);
  }
```

```
    if (str == "Carp")
    {
     delege = new Temsilci(p1.Carp);
     delege(30, 10);
    }
   }
  }
```

Bu şekilde düzenlenen uygulamada kullanıcıya hangi metodu işletmek istiyorsunuz sorusu yönetiliyor. Kullanıcının cevabına göre delegate nesnesi hazırlanıp istenilen metot işletiliyor. Burada işletmeyi denediğimiz "Carp()" ve "Topla()" metotlarının "Program1" adını verdiğimiz sınıfta kodlamış olduğumuzu biliyorsunuz.

İşletilecek metodu çalışma anında bu şekilde seçmek pratik değildir. Buradaki amacımız konu üzerinde düşünmenizi sağlamaktır. Diğer yandan metotlar delegate nesnelerini parametre olarak alabilmektedir. Delegate nesnelerinin metotlara nasıl parametre olarak verildiğini aşağıda görebilirsiniz. Uygulamamanız yeni halinde **islet()** adını verdiğimiz metot, kendi hazırladığımız delegate tipinde bir parametreye sahiptir.

```
namespace ConsoleApp1
{
 public delegate void Temsilci(int X, int Y);
 public class Program
 {
  public static void Main()
  {
   Program1 p1 = new Program1();
   Temsilci delege = new Temsilci(p1.Topla);
   Islet(delege);
  }
  public static void Islet(Temsilci dlg)
  {
   dlg(30, 10);
  }
 }
}
```

Bu kodun işletilebilmesi için "Temsilci" adını verdiğimiz delegate tipinin public olması gerekiyor. Burada Islet() adını verdiğimiz metoda hangi delegate nesnesi gönderilirse o nesnenin temsil ettiği metot işletilir. Örneğin Main() metoduna aşağıdaki gibi 2 yeni satır eklenirse elimizdeki diğer metot işletilir.

```
public static void Main()
{
 Program1 p1 = new Program1();
 Temsilci delege = new Temsilci(p1.Topla);
 Islet(delege);
 delege = new Temsilci(p1.Carp);
 Islet(delege);
}
```

EventHandler Delegate'ten Yararlanmak

Şimdiye kadar 20 sayfa boyunca delegate'ler üzerine konuşmamıza rağmen bizce konu zihinlerde henüz netleşmiş değildir. Bu düşünce ile Windows Forms uygulamalarında yararlanılan ve .NET Framework ile gelen **EventHandler** adlı delegate tipinin üzerinde biraz duracağız. Bu amaçla ilk olarak bu delegate tipinin orijinal halini vereceğiz.

```
public delegate void EventHandler(object sender, EventArgs e);
```

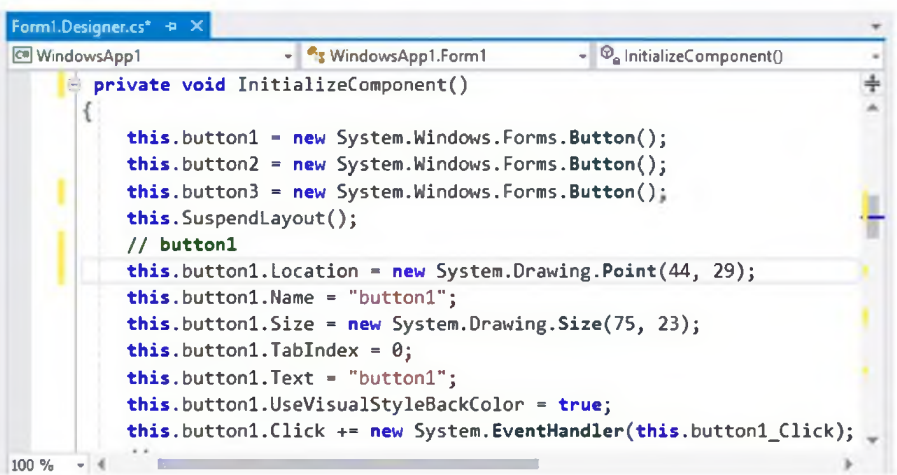
Bu satırı dikkatlice incelerseniz bu delegate'in geriye değer göndermeyen ve 2 parametreye sahip olan metotları temsil edebileceğini görürsünüz. Bu parametrelerden ilkinin **object**, ikincisinin **EventArgs** tipinde olması gerekiyor. Bu kısa açıklamadan sonra bir Windows Forms uygulaması hazırlayıp "Form1" adlı forma 2 button nesnesi yerleştirdik.



Daha önceki konulardan bildiğiniz gibi tasarım anında bu button nesneleri çift tıklandığında, Visual Studio, programcının söz konusu button nesnesi çalışma anında tıklandığında işletilecek bir metodu hazırlamak istediğini varsaymaktadır. Başka bir anlatımla tasarım anında formdaki button nesneleri çift tıklandığında, Visual Studio tarafından, adında nesnenin adıyla birlikte olay adının olduğu bir metot hazırlamaktadır.

```
private void button1_Click(object sender, EventArgs e)
{
}
```

Tasarım anında programcının button nesnesini çift tıklaması sonucu Visual Studio'nun tek yaptığı işlem bu şekilde bir metot hazırlamak değildir. Visual Studio, ayrıca bu metodun **EventHandler** adlı delegate tarafından temsil edilmesini sağlamaktadır. Bu ayarlamayı "Form1.Designer.cs" adlı kod dosyasında yapmaktadır.



Form1.Designer.cs dosyasında forma yerleştirilen button1 adlı nesnesinin özellikleri ayarlandıktan sonra buttun1_Click() metodunun EventHandler tarafından temsil edilmesi sağlanmaktadır. Burada, eşitliğin sol tarafı "button1" nesnesinin **Click** olayıyla ilgilidir. Henüz olaylarla ilgilenmediğimiz için Visual Studio tarafından hazırlanıp Form1.Designer.cs dosyasına eklenen bu satırı sileceğiz. Çünkü buradaki amacımız, Visual Studio tarafından geri planda yapılan işlemi kendimiz yapacağız.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Benzer şekilde tasarım anında button2 nesnesini de çift tıklayıp Visual Studio'nun bu nesne için button2_click() adında bir metot hazırlamasını sağladık. Ardından yine Form1.Designer.cs dosyasını açıp EventHandler'la ilgili satırı sildik. Çünkü buradaki amacımız delegate'leri anlamaktır ve bu nedenledir ki Visual Studio'nun sağladığı kolaylıklardan yararlanmayacağız.

"Form1.Designer.cs" dosyasının "button1" ve "button2" nesneleriyle ilgili kısımlarını aşağıda verdik. Burada dikkatinize sunmak istediğimiz ayrıntı şudur: formdaki button1 ve button2 nesnelerinin Click olayları artık herhangi bir metotla ilişkili değildir.

```
private void InitializeComponent()
{
 this.button1 = new System.Windows.Forms.Button();
 this.button2 = new System.Windows.Forms.Button();
 this.SuspendLayout();
 this.button1.Location = new System.Drawing.Point(44, 29);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(75, 23);
 this.button1.TabIndex = 0;
 this.button1.Text = "button1";
 this.button1.UseVisualStyleBackColor = true;
 this.button2.Location = new System.Drawing.Point(140, 29);
 this.button2.Name = "button2";
 this.button2.Size = new System.Drawing.Size(75, 23);
 this.button2.TabIndex = 1;
 this.button2.Text = "button2";
 this.button2.UseVisualStyleBackColor = true;
}
```

Şimdi içinde bulunduğumuz şartlarda üzerinde çalıştığımız Windows Forms uygulamasının formunda button1 ve button2 adında 2 button nesnesi bulunmaktadır. Ayrıca Form1 sınıfımızın kodlandığı Form1.cs dosyamızda button_Click() ve button2_Click() adında 2 metodumuz vardır. Yukarıda bu metotları hazırlarken Visual Studio'dan yararlanmıştık. Sizler bu metotları kendiniz kodlayabilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 MessageBox.Show("Button 1 tıklandı");
}
private void button2_Click(object sender, EventArgs e)
{
 MessageBox.Show("Button 2 tıklandı");
}
```

Bu metotlarla ilgili button nesnelerinin Click olayı arasındaki bağlantıyı Form1.Designer.cs dosyasına müdahale ederek kopardığımız için çalışma anında bu button nesneleri tıklansa bile bu metotlar işletilmez. Buradaki amacımız button nesnelerinin tanımlı olaylarını kullanmadan .NET Framework ile gelen **EventHandler** adlı delegate'ten yararlanıp bu metotları işletmektir. Yapmak istediğimiz bir bakıma kulağımızı tersten göstermek gibidir.

Bu hazırlıklardan sonra, yani;

- Forma 2 button nesnesi yerleştirip tasarım anında bu button nesnelerin çift tıklayıp button nesneleri için button1_Click() ve button2_Click() adında 2 metot hazırladık.
- Ardından Form1.Designer.cs dosyasına müdahale edip EventHandler aracılığıyla bu metotların button nesnelerinin Click olayları arasında ilişki kuran satırları sildikten sonra;

"Form1.cs" dosyamızda class bloğunda bir satır ekleyerek Delegate tipinde bir dizi değişken tanımladık. Bu dizi değişkeni daha sonra kullanacağız. Konu ezbere olmasın diye Form1.cs dosyamızın bütün içeriğini aşağıda verdik.

```
namespace WindowsApp1
{
 public partial class Form1 : Form
 {
  public Form1()
   {
    InitializeComponent();
   }
   Delegate[] Temsilci_dizi;
   private void Form1_Load(object sender, EventArgs e)
   {
   }
   private void button1_Click(object sender, EventArgs e)
   {
     MessageBox.Show("Button 1 tıklandı");
   }
   private void button2_Click(object sender, EventArgs e)
   {
     MessageBox.Show("Button 2 tıklandı");
   }
  }
}
```

Şimdi sırada Form1 sınıfımızın Form_Load() metoduna ekleme yapıp **EventHandler** tipinde bir delegate nesnesi hazırlamak ve bu nesnenin 2 metodumuzu temsil etmesini sağlamak var.

```
Delegate[] Temsilci_dizi;
private void Form1_Load(object sender, EventArgs e)
{
 EventHandler temsilciNesne = null;
 temsilciNesne += new EventHandler(this.button1_Click);
 temsilciNesne += new EventHandler(this.button2_Click);
}
```

Bu satırların yukarıdaki sayfalarda verdiklerimizden tek farkı şudur: Yukarıdaki sayfalarda Topla() ve Carp() adını verdiğimiz metotları temsil etmesi için hazırladığımız delegate nesneleri kendi hazırlayıp "Temsilci" adını verdiğimiz delegate tipinde iken bu kez "EventHandler" tipindedir. Elbette burada yapmak istediğimiz işlem için .NET Framework ile gelen EventHandler adlı delegate tipini kullanmak zorunda değiliz. Ancak genel anlayış şudur: .NET Framework ile gelen sınıf, interface veya delegate işinizi görüyorsa onu kullanmakta fayda vardır.

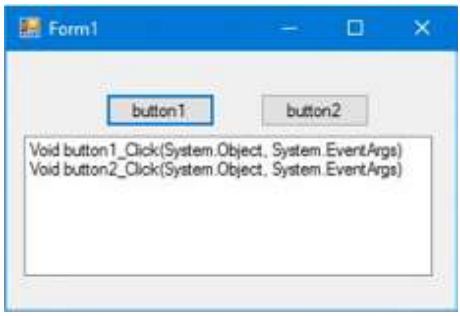
Yukarıda verilen kod için konuşmak gerekirse, kodun işletimi Form1_Load() metodunun sonuna ulaştığında elimizdeki "temsilciNesne" adlı delegate nesneniz hem button1_Click() hem de button2_Click() metotlarını temsil etmektir. Bu metotlardan istediğimizi çalışma anında işletebilmek amacıyla forma bir ListBox nesnesi yerleştirdik ve Form_Load() metoduna eklemeler yaptık.

```
Delegate[] Temsilci_dizi;
private void Form1_Load(object sender, EventArgs e)
{
 EventHandler temsilciNesne = null;
 temsilciNesne += new EventHandler(this.button1_Click);
 temsilciNesne += new EventHandler(this.button2_Click);
 Temsilci_dizi = temsilciNesne.GetInvocationList();
}
```

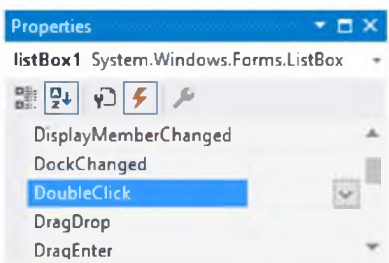
Bu noktada burada "temsilciNesne" adını verdiğimiz ve EventHandler tipindeki delegate nesnemizin içerdiği delegate nesneleri "Temsilci_dizi" adını verdiğimiz dizi değişkenin birer üyesi durumundadır. Bu değişkenin elemanlarını ListBox'ta listeleyebilmek için Form1_Load() metoduna yine ekleme yaptık.

```
Delegate[] Temsilci_dizi;
private void Form1_Load(object sender, EventArgs e)
{
 EventHandler temsilciNesne = null;
 temsilciNesne += new EventHandler(this.button1_Click);
 temsilciNesne += new EventHandler(this.button2_Click);
 Temsilci_dizi = temsilciNesne.GetInvocationList();
 foreach (Delegate item in Temsilci_dizi)
 {
  listBox1.Items.Add(item.Method.ToString());
 }
}
```

Form1_Load() metoduna sonradan eklediğimiz foreach döngüsüne yabancı değilsiniz. Dizi değişkende tutulan Delegate nesnelerinin "Method" özelliklerinin ListBox'ta listelenmesi sağlanmaktadır. Form1_Load() metodu bu şekilde düzenlediğimiz uygulamamızı çalıştırınca aşağıdaki gibi bir sonuç aldık.



Şimdi öyle bir ayarlama yapacağız ki kullanıcı ListBox'ta listelenen hangi metodu çift tıklarsa onun işletilmesini sağlayacağız. Tabi bunu yaparken, başka bir deyişle ListBox nesnesiyle ilgili olarak double click olayı meydana geldiğinde işletilecek metodu hazırlama ve bu metodu olayla ilişkilendirmede Visual Studio'dan yararlanacağız. Bu amaçla formdaki ListBox nesnesi seçili durumda iken Visual Studio'nun Properties penceresinde ListBox'a ait olayların listelenmesini sağladık.



Properties penceresinde ListBox1 nesnesinin olayları listelenirken ilgilendiğimiz DoubleClick olayını çift tıkladığımızda, Visual Studio'nun bu olayı temsil edecek metodu hazırlayıp olayla ilişkilendirdiğini biliyorsunuz. Çalışma anında ListBox nesnesi çift tıklandığında işletilecek metodu aşağıdaki gibi düzenledik. Bu metoda yazdığımız satırlara yabancı değilsiniz.

```
private void listBox1_DoubleClick(object sender, EventArgs e)
{
 int secilen = listBox1.SelectedIndex;
 Temsilci_dizi[secilen].DynamicInvoke(button1, e);
}
```

Şimdiye kadar delegate'ler hakkında azımsanmayacak kadar bilgi verildi. Delegate'ler hakkında bu bölümde okuduklarınız, bundan sonra anlatılacak konu veya yapılacak işlemin ucu delegate'lere dokunduğunda anlamakta zorluk çekmeyeceksiniz. Örneğin bir sonraki bölümde olaylar anlatılacaktır. Olaylar delegate'lerin üzerine temellendiği için delegate'ler hakkında bilgi edinmeden olayları kavramak zordur. Yani artık olaylar hakkında bilgi edinmeye hazırsınız, .NET Framework ile gelip programcıların sıklıkla kullandıkları delegate'leri zorlanmadan kullanabilirsiniz.

Delegate'lerle ilgili olarak henüz anlatılmayan önemli bir konu var: Generic Delegate'ler. Bu kitapta henüz generic delegare'lerden söz etmediğimiz için generic delegate'lerin üzerinde durmayı erteledik.

14. BÖLÜM
OLAY HAZIRLAMAK ve KULLANMAK

Bu kitapta şimdiye kadar .NET Framework ile gelen çok sayıda sınıfın çok sayıda olay ve metodundan söz edildi. Kendi hazırladığınız sınıflardaki metotların belli olaylar meydana geldiğinde işletilmesini istiyorsanız **delegate** ve **event** hazırlamanız gerekecektir.



Olaylar, delegate'lerin üzerine inşa edilen yapılardır. Bu nedenle delegate'lerin üzerinde durulan bölümde anlatılanları henüz kavramadıysanız Event'leri anlayıp benimsemekte zorlanabilirsiniz. Buna rağmen bu bölümü baştan sona okumanızı öneriyoruz. Bu sayede delegate'ler üzerinde tekrar çalışmayı denediğinizde konular sizin için daha anlaşılır olacaktır. Ayrıca bu bölümün ilk birkaç sayfası Windows Forms uygulamalarıyla ilgili olduğu için ilk okumada anlaşılmaya uygundur. Bu bölümde event'lerle ilgili olarak bazı ayrıntılar bilinçli olarak tekrar anlatıldı. Bu tekrarların nedeni bölümün sonuna gelindiğinde event kavramını kafanızda iyice netleşmesini garanti etmektir.

Bu arada Windows'un "mesajlar" ile işlem yapan bir işletim sistemi olduğunu ve kullanıcının yaptığı işlemlere göre ilgili uygulamaya mesajlar gönderdiğini hatırlamak gerekir. C# uygulaması dahilinde gerçekleşen olayları, Windows'la ilgili olaylarla karıştırmamak gerekir. Kullanıcının klavye ve fare ile yaptığı işlemleri temsil eden mesajları Windows uygulamaya gönderir. C# uygulaması da bu mesajları alıp delegate'ler aracılığı ile ilgili metotları işletir. Olayların zor anlaşılmasının en önemli nedeni bizce şudur:

- C# konulu kaynaklarda C# ve .NET programcılığı konusunda yolun başında olan birisine forma yerleştirilen "button1" nesnesinin Click olayının nasıl kullanıldığı, Click olayını temsil eden metodun nasıl hazırlandığı anlatılır.
- Programcı adayımız bunları okuduktan sonra olay kavramı kafasında ilk gördüğü şekliyle netleşir. Programcı adayı için artık olay demek Properties penceresinde Button veya başka bir sınıfın olaylarını listeleyip olayın adı üzerinde çift tıklayıp Visual Studio'nun metot hazırlaması ve bu metodun içine bir şeyler yazmasıdır.
- Programcı adayımız Delegate kavramı hakkında yeteri bilgiye sahip olmadan .NET Framework ile hazır olarak gelen bir delegate olan **EventHandler**'ı kullanmayı öğrenir.
- Başka bir deyişle, C# ve .NET programcılığını öğrenmek üzere kursa gidenler veya C# kitabı okuyanlar, **event** anahtar kelimesini hiç kullanmadan deneysel de olsa programlar yazarlar. Tekrar etmek gerekirse, bu şartlarda programcı adayının olay kavramı diye bir sorunu henüz yoktur.
- Derken programcının önüne **event** anahtar kelimesini çıkarırız. Tabi event anahtar kelimesiyle ilgili olarak söylenenler eski öğrendikleriyle hiç alakalı değildir.
- Bu şartlarda event kavramı C# dilinin zor anlaşılan konularından birisi olur çıkar.

.NET Framework İle Gelen Kontrollerin Olayları

.NET Framework ile gelen ve programcıların sıklıkla kullandıkları kontrollerin olaylarını kullanmak ile programcının kendi hazırladığı sınıflarda olay hazırlamak ve kullanmak

birbirinden oldukça farklıdır. Bu farklılığı ortaya çıkarmak için önceliği Windows Forms uygulamaları dahilinde kullanılan olaylara verdik.



Burada olaylar, metotlar ve delegate'lerle ilgili olarak anlatılanların yararlı olmasını garanti etmek için yapılan basit işlemleri yapmanız önerilir.

Bu amaçla Visual Studio ile yeni bir C# projesi hazırlayıp forma bir düğme yerleştirdik. Tasarım anında düğmeyi çift tıklarsanız Visual Studio tarafından düğmenin Click olayı için aşağıdaki gibi bir metodu hazırlandığını biliyorsunuz. Size düşen, bu metodun içine yapılmasını istediğiniz işlemlere uygun kod satırlarını yazmaktır.

```
private void button1_Click(object sender, System.EventArgs e)
 {
 }
```

Çalışma anında düğme tıklandığı zaman bu metot işletilir. Ancak tasarım anında düğmenin çift tıklanıp Visual Studio tarafından hazırlaması yerine kendiniz metot hazırlarsanız çalışma anında düğme tıklandığı zaman hazırladığınız metot işletilmez. Çünkü tasarım anında düğme çift tıklanıp metot hazırlandığında Visual Studio ayrıca **Designer.cs** dosyasında "Windows Form Designer generated code" bölgesine aşağıda verilen satırı yazmaktadır.

```
this.button1.Click +=new System.EventHandler(this.button1_Click);
```

Bu satır sayesinde "button1" nesnesinin **Click** olayı ile, yukarıda verilen metot arasında .NET Framework ile gelen **EventHandler** aracılığıyla ilişki kurulmuş olmaktadır. Bu satır silinirse yukarıda verilen metot ile "button1" nesnesinin **Click** olayı arasındaki ilişki kopmuş olacağı için çalışma anında düğme tıklansa bile **button1_Click()** metodu işletilmez.

Button nesnesi ile Click olayı arasında ilişki kurmak üzere Designer.cs dosyasında **System**'de yer alan ve gerçekte bir delegate olan **EventHandler**'a ekleme yapılmakta ve EventHandler'ın bu metodu temsil etmesi sağlanmaktadır. Forma yerleştirdiğimiz düğme için Visual Studio sayesinde metot hazırlayıp bu metot ile Click olayının ilişkilendirilmesini sağladıktan sonra formun Load olayı ile ilişkilendirilmiş metodu aşağıdaki gibi düzenledik.

```
private void Form1_Load(object sender, EventArgs e)
{
 this.button1.Click -= new System.EventHandler(this.button1_Click);
}
```

Tahmin edeceğiniz gibi bu satır sayesinde "button1" nesnesinin Click olayı ile button1_Click() metodu arasındaki bağlantı (temsil) kesilmekte, başka bir deyişle EventHandler'ın elinden bu metodu temsil etme hakkı alınmaktadır. Bu şartlarda çalışma anında formdaki "button1" nesnesi tıklansa bile button1_Click() metodu işletilmez.

Visual Studio'nun button nesneleri için **EventHandler**'da yaptığı ayarlamalar hakkında biraz konuştuktan sonra "Designer.cs" dosyasında aşağıda verilen satırı sildik. Bu sayede formdaki button1 nesnesi ile Visual Studio tarafından hazırlanan button1.Click() metodu arasındaki ilişkiyi koparmış olduk.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Konunun ayrıntıları ortaya çıksın diye forma "button2" adında 2. bir button nesnesi yerleştirip formun Load olayını temsil eden metodu aşağıdaki gibi düzenledik. Bu şartlarda çalışma anında ister "button1" ister "button2" tıklansın "button1_Click()" metodu işletilir.

```
private void Form1_Load(object sender, EventArgs e)
{
 this.button1.Click += new System.EventHandler(this.button1_Click);
 this.button2.Click += new System.EventHandler(this.button1_Click);
}
```



Bu sırada forma "button2" adında Button nesnesi yerleştirdik ama tasarım anında bu button nesnesinin üzerinde çift tıklama yapmadık. Yoksa, Visual Studio devreye girip button2 nesnesi için bir metot hazırlayıp bu metodu Click olayıyla ilişkilendirirdi.

.NET Framework ile gelen sınıfların olayları için metot hazırlamayı ve bu metotları olaylarla ilişkilendirmeyi bildiğimize göre şimdi Form sınıfının Click olayını temsil etmek üzere hazırlayıp aşağıda verilen metodu mercek altına alacağız.

```
private void Form1_Click(object sender, EventArgs e)
 {
 }
```

Bu metottaki **object** tipindeki **sender** adlı parametreyi biliyoruz. Her ne kadar bu metodun adının ilk yarısı "Form1" olsa bile programcı metoda istediği adı verebilir. Metodun hangi nesnenin Click olayı aracılığı işletildiğini öğrenmek istediğinizde **sender** adlı parametreye bakabilirsiniz. Şimdi sender parametresinden yararlanıp bu metodu işleten formun **Text** özelliğinin içeriğine bakacağız.

```
private void Form1_Click(object sender, EventArgs e)
{
 Form1 F1 = (Form1)sender;
 MessageBox.Show(F1.Text);
}
```

Bu kodda önce **Form** tipinde bir değişken tanımladık. Ardından object tipindeki **sender** parametresini dönüştürüp bu değişkene aktardık. Tabi bu metodun işletilmesine neden olan nesne "Form1" tipinde değilse dönüştürme sırasında hata meydana gelir. Örneğin forma bir button nesnesi yerleştirilip düğmenin Click olayı ile ilişkilendirilmiş metot aşağıdaki gibi düzenlenirse yukarıda verilen kod hataya neden olur.

```
private void button1_Click(object sender, EventArgs e)
{
 Form1_Click(sender, e);  // hata meydana gelir.
}
```

Bu hatanın önüne geçmek için dönüştürme yapılmadan önce **sender** parametresinin tipinin **Form1** olup olmadığı kontrol edilebilir. Bu kontrol 'is' operatörü ile aşağıdaki gibi yapılabilir.

```
private void Form1_Click(object sender, EventArgs e)
{
 if (sender is Form1)
 {
  Form1 F1 = (Form1)sender;
  MessageBox.Show(F1.Text);
 }
}
```

Formların Click olaylarını temsil etmesi için hazırlanan metotların **sender** parametresi hakkında biraz konuştuktan sonra sırada **EventArgs** tipindeki "e" parametresi var. EventArgs parametresi ile, Click olayını temsil eden metoda bilgi gönderiliyor. EventArgs sınıfının yapısı incelendiğinde **Object** sınıfı kaynaklı olup işimize pek yaramayacak birkaç metoda ve **Empty** adında bir özelliğe sahip olduğu görülür. Aslında **EventArgs** tipindeki parametre ile metoda önemli sayılabilinecek bilgi gönderilmemesi normaldir. Çünkü konu Click olayını temsil eden metodu işletmek olduğuna göre kullanıcının formun neresini tıkladığının önemi yoktur.

Konu Form sınıfının **MouseDown** olayı olduğunda ise "e" adındaki ve **EventArgs** tipindeki parametre yeterli olmaz. Çünkü konu MouseDown olayı olduğunda formun tıklanan yerinin koordinatlarına gerek duyulabilir. Bu konunun ayrıntılarını ortaya çıkarmak için Visual Studio'dan yararlanıp hem Form1 sınıfının hem Click hem de MouseDown olaylarını temsil etmek üzere aşağıda verilen metotları hazırladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 MessageBox.Show("Click olayı meydana geldi");
}
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
 MessageBox.Show("MouseDown olayı meydana geldi");
}
```

Bu iki metodu incelediğinizde MouseDown olayını temsil etmek üzere hazırladığımız metodunun "e" adındaki parametresinin Click olayıyla ilgili metottan farklı olarak **MouseEventArgs** tipinde olduğunu görürsünüz. MouseEventArgs sınıfı EventArgs sınıfından türetilip bazı ek özellikler kazandırılmış bir sınıftır. EventArgs sınıfı System'de yer alırken MouseEventArgs sınıfı **System.Windows.Forms**'de bulunmaktadır.

Bu şartlarda proje çalıştırılıp Form tıklanırsa MouseDown olayı ile ilişkilendirilmiş metot işletilir. Başka bir deyişle her ikisi de form tıklandığında meydana gelen bu olaylardan öncelik **MouseDown** olayınındır. Formun Click olayı için metot hazırlayıp işletmek istiyorsanız MouseDown olayı için metot hazırlamamanız veya mevcut metodun **EventHandler** tarafından temsil edilmesinin önüne geçmeniz gerekir. MouseDown olayı için aşağıdaki gibi bir metot hazırlayarak formun tıklanın konumu hakkında bilgi alabilirsiniz.

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
 string str;
 str = e.Location.X.ToString() +"," + e.Location.Y.ToString();
 MessageBox.Show("Formun tıklamam yerin koordinatı :" + str);
}
```

Tahmin edeceğiniz gibi Form ve TextBox sınıfının KeyPress olayı için hazırlanacak metodun 2. parametresi farklı olacaktır. **KeyPressEventsArgs** tipindeki 2. parametre ile metoda, basılan tuşla ilgili bilgi gönderilmektedir.

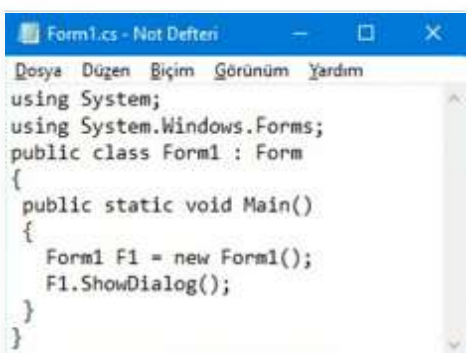
```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
}
```

Gelenek olduğu üzere olaylarla ilgili bilgi vermeye Visual Studio ile hazırlanan projedeki formun üzerine yerleştirilen button nesnesi tıklandığı zaman meydana gelen Click olayından söz edildi. Ne var ki bu şekilde forma yerleştirilmiş olan Button nesnesi üzerinde olayları anlatmak **Event** kavramının bazen yanlış bazen de zor anlaşılmasına neden olmaktadır. Çünkü çalışma anında formdaki düğme tıklandığı zaman focus olabilen ilgili pencereye (burada Button nesnesi) Windows bir mesaj göndermektedir.

Başka bir deyişle olayın tetiklenmesi geri planda işletim sisteminin yardımıyla gerçekleştirilmektedir. Ancak kendi hazırladığımız sınıflardaki olayların tetiklenmesi konusunda kendimiz ayarlama yapmak zorundayız. Tekrar etmek gerekirse, formdaki TextBox'a bilgi girildiğinde veya button nesnesi tıklandığında, .NET Framework ile gelen Button ve TextBox sınıflarına yapıları gereği işletim sistemi bize yardım ederken kendi sınıfımız için olay hazırladığımızda bu olayın nasıl tetikleneceğini düşünmek zorundayız.

Windows Forms Uygulamasında Event Hazırlamak

.NET Framework ile gelen Form, Button, TextBox gibi sınıflarının olaylarından nasıl yararlanıldığı konusuna başka bir pencereden bakmak amacıyla bir forma sahip bir konsol uygulaması hazırladık. Not Defteri'ni kullanarak hazırladığımız bu uygulamada **System.Windows.Forms**'deki sınıfları kullanmak üzere **using** bloğunda ayarlama yaptık.



Bu C# programı derlenip çalıştırılırsa bir formun pencere olarak ekrana geleceğini tahmin edebilirsiniz. Şimdi bu uygulamada değişiklik yapıp forma bir düğme yerleştireceğiz. Forma düğmeyi yerleştirecek satırları "Form1" sınıfının yapıcı metoduna yazdık.

```
using System.Windows.Forms;
public class Form1 : Form
{
```

```
    private Button dugme1;
    public Form1()
     {
      dugme1 = new Button();
      dugme1.Text = "Kaydet";
      dugme1.Location = new System.Drawing.Point(75, 75);
      dugme1.Size = new System.Drawing.Size(100, 40);
      this.Text = "Form1";
      this.Controls.Add(dugme1);
     }
     public static void Main()
     {
      Form1 F = new Form1();
      F.ShowDialog();
     }
}
```

Bu kodda yabancısı olduğunuz önemli bir şey yoktur. Asıl anlatmak istediğimiz, Visual Studio'dan yararlanmadan hazırlanan Button nesnelerinin tanımlı olayları için metot hazırlamaktır. Bu amaçla "Form1" sınıfının yapıcı metoduna aşağıda verilen satırı ekledik.

```
dugme1.Click += new EventHandler(dugme1_Click);
```

Önce bu satırda eşitliğin sağ tarafına odaklanacağız. EventHandler'ın .NET Framework ile gelen bir delegate tipi olduğunu ve 2 parametreye sahip metotları temsil edebildiğini biliyorsunuz. Burada **EventHandler** tipinden yola çıkılarak new anahtar kelimesiyle delegate nesnesi hazırlanırken yapıcı metoda dugme1_Click() adını verdiğimiz metodu parametre olarak verdiğimiz için bu delegate nesnesi bu metodu temsil etmeye başlar. Bu metodu henüz hazırlamadığımız için bu şartlarda uygulama derlenmek istenirse derleyici hata rapor eder.

Şimdi sırada bu satırın sol tarafına odaklanmak var: Daha önceki konulardan hatırlayacağınız gibi mevcut bir delegate tipinden yola çıkarak delegate nesnesi hazırlandığında eşitliğin sol tarafında delagete nesnesine verilen ad olurdu. Örneğin bu kitabın delegatelerle ilgili bölümünde aşağıdaki gibi 2 satıra yer vermiştik.

```
delegate int Temsilci (int X, int Y);
Temsilci temsilciNesne = new Temsilci(Topla);
```

Kendi yazdığımız konsol uygulamasında ise eşitliğin sol tarafında "dugme1" adını verdiğimiz Button tipindeki nesnenin **Click** olayının adı yazılmış durumdadır. Dolayısıyla dugme1_Click() metodunu temsil etmek üzere hazırladığımız delegate nesnesinin yerini artık dugme1'in Click olayı almaya başlamıştır. Bunun anlamı şudur: "dugme1" nesnesinin Click olayı tetiklediğinde burada "dugme1_Click" adını verdiğimiz metot işletilecektir.

Geriye bu metodu yazmak kaldı. Bu şekilde düzenlenen konsol uygulaması derlenip çalıştırılırsa üzerinde bir düğme olan form ekrana gelir. Çalışma anında formdaki düğme tıklandığında dugme1_Click() metodu işletilir.

```
using System.Windows.Forms;
public class Form1 : Form
{
 private Button dugme1;
 public Form1()
 {
  dugme1 = new Button();
  dugme1.Text = "Kaydet";
  dugme1.Location = new System.Drawing.Point(75, 75);
  dugme1.Size = new System.Drawing.Size(100, 40);
  this.Text = "Form1";
  dugme1.Click += new EventHandler(dugme1_Click);
  this.Controls.Add(dugme1);
 }
 public static void Main()
 {
  Form1 F = new Form1();
  F.ShowDialog();
 }
 void dugme1_Click(object nesne, EventArgs e)
 {
  MessageBox.Show("Düğme tıklandı");
 }
}
```

Kendi Sınıflarımız İçin Event Hazırlamak

Bu bölümün ilk sayfasında ifade edildiği üzere gelenekselleştiği üzere C# uygulamalarında .NET Framework ile gelen sınıflara ait Event'lerin nasıl kullanılacağı anlatılırken button sınıfının Click olayından yararlandık. Gerçekte yukarıdaki sayfalarda anlatılanlar olay yani event hazırlamak değildir, ilgili sınıfın tanımlı olaylarını kullanmaktır. Ötesi, bu olayları kullanırken işletim sisteminden yararlanıldığı için konu oldukça farklılaşmaktadır.

Şimdi adım adım kendi hazırlamış olduğunuz sınıflarda nasıl olay tanımlayacağınızı göreceğiz. Bu amaçla aşağıda verilen konsol uygulamasını hazırladık. Olaylar birden fazla sınıf söz konusu olduğu zaman işlevsel olduğu için uygulamada birden fazla sınıfa yer verdik.

```
namespace ConsoleApp1
{
 class Program
 {
  public static void Main()
  {
  }
 }
 public class Dugme
  {
  }
}
```

Bu konsol uygulamasındaki "Dugme" adını verdiğimiz sınıfta olaylarla ilgili satırlar olacak. Main() metodunun olduğu diğer sınıfta ise tanımlayacağımız olay meydana geldiği zaman işletilecek metotlara yer verilecek. Maddeler halinde yazmak gerekirse;

- Burada "Dugme" adını verdiğimiz sınıf için olay tanımlayacağız. (Tıpkı .NET Framework ile gelen Button sınıfında Click adında bir olayı olduğu gibi.)
- Dugme adlı sınıfta tanımlayacağımız olay bir şekilde gerçekleştiğinde işletilecek veya olayı temsil edecek metot, Main() metodunun bulunduğu sınıfta olacaktır. (Windows Forms uygulamasında Toolbox'tan yararlanılıp Forma yerleştirilen düğme tasarım anında çift tıklandığında "Form1" sınıfında olay meydana geldiğinde işletilecek metot Visual Studio tarafından hazırlanmaktadır.)
- Sonra Main() metodunun olduğu sınıfta Dugme sınıfının örneğini alıp tanımladığımız olay ile delegate aracılığıyla bir metotla ilişkilendireceğiz. (Windows forms uygulamalarında olay ile bir metodun ilişkilendirilmesi işlemi Designer.cs dosyasında yapılmaktadır).

Yukarıda ifade edildiği gibi delegate olmadan olay olmaz. Çünkü event'ler zaten delegate'lerin bir nevi özel halidir ve delegate'lerin üzerine inşa edilirler. Başka bir deyişle olayları tetiklemek için delegate'ler gereklidir. Bu nedenle burada Dugme adını verdiğimiz sınıfımız için olay hazırlamaya bir delegate tipi tanımlamakla başlayacağız.

"Temsilci" adını verdiğimiz bu delegate tipini uygulamadaki diğer sınıflarda kullanabilmek için delegate tipi ile ilgili satırı namespace bloğuna yazdık. Burada "Temsilci" adını verdiğimiz delegate tipinden yola çıkılarak hazırlanacak delegate nesneleri, geriye değer göndermeyen ve parametreleri olmayan metotları temsil edebilecektir.

```
namespace ConsoleApp1
{
 public delegate void Temsilci();
 class Program
  {
   public static void Main()
   {
    Console.ReadLine();
   }
 }
 public class Dugme
  {
  }
}
```

Bu delegate tipinden yola çıkılıp **new** anahtar kelimesiyle hazırlanacak nesnenin temsil etmesini istediğimiz metot geriye değer göndermeyeceği için **void** anahtar kelimesine yer verdik. Bu şekilde **delegate** tipi tanımladıktan sonra "Dugme" adlı sınıfın içinde **event** anahtar kelimesi ile "Click" adında bir olay tanımladık. Bu satırda "event" anahtar kelimesi ile birlikte daha önce tanımlanan delegate tipine yer verildiği için olay meydana geldiği zaman işletilecek metodu bu delegate temsil eder.

```
namespace ConsoleApp1
{
 public delegate void Temsilci();
 class Program
 {
```

```
    public static void Main()
    {
      Console.ReadLine();
    }
  }
  public class Dugme
   {
     public event Temsilci Click;
   }
}
```



Bu olaya istediğimiz adı verebilirdik. Burada **Click** adını seçmemizin nedeni konuya dikkatinizi çekmektir. Yoksa, "Dugme" sınıfı için hazırlayacağımız olayın, tıklama yapılarak tetiklenmesi burada söz konusu değildir.

Bu şartlarda derleme yapılırsa hata meydana gelmez, derleyici uyarıda bulunmakla yetinir. Çünkü henüz "Dugme" sınıfının "Click" olayından yararlanmadık. Toplam 4 kelimeden meydana gelen bu satıra yakından bakmakta fayda var. Çünkü olay hazırlayıp kullanma işlemini adım adım anlatırken arada yapılan herhangi bir işlem anlaşılmadığında daha sonra anlatılacakları anlamak zorlaşıyor.

- Burada amacımız Dugme sınıfımız için olay tanımlamak olduğundan **event** anahtar kelimesine yer verdik. Olaylar doğaları gereği başka sınıflardan kullanılacakları için public veya internal olmalılar. Yoksa buradaki adıyla "Dugme" sınıfının örnekleri üzerinden Click adını verdiğimiz olayı erişip kullanmak mümkün olmaz.
- Event'ler delegate'lerin üzerine temellenen yapılardır. Dolayısıyla sınıfımız için event hazırlamak istiyorsanız daha önceden söz konusu event'in yararlanacağı bir delagete hazırlamanız gerekir. Burada "Dugme" sınıfımı için hazırlamak istediğimiz event'ın "Temsilci" adını verdiğimiz delegate'ten yararlanmasını istediğimiz için **event** anahtar kelimesinden hemen sonra bu delegate'in adını yazdık.
- Event tanımladığımız satırda en son olarak event'a vermek istediğimiz adı yazdık. Event'a verilen adların olayın tetiklenme şeklini hatırlatması tercih edilmektedir. Burada hazırlamak istediğimiz olay geri planda gelişen işlemler sonucu tetiklenmeyeceği için kendimiz tetikleyeceğiz. Buna rağmen Windows Forms uygulamalarında kullanılan sınıfların çoğunun Click adında bir olayı olduğu için konu üzerinde düşünmenize vesile olması için "Click" adını seçtik.

Bu şekilde olay tanımladıktan sonra bu olay meydana geldiği zaman bu olayı karşılayacak veya olayı tespit edecek bir metodun olması gerekir. Bu amaçla kendisi için olay tanımladığımız "Dugme" sınıfında aşağıdaki gibi **OnClick()** adında bir metot hazırladık. .NET programcılarının hazırladıkları olayı karşılayacak, yani olayı tetikleyecek metoda olay adının önüne "On" kelimesini getirmek gibi bir alışkanlıkları vardır.

```
namespace ConsoleApp1
{
 public delegate void Temsilci();
 class Program
 {
  public static void Main()
```

```
    {
     Console.ReadLine();
    }
   }
   public class Dugme
   {
    public event Temsilci Click;
    public void OnClick()
    {
     Click();
    }
   }
}
```

Olaylarla ilgili olarak kafaların en fazla karıştığı noktadayız. Bu nedenle burada OnClick() adını verdiğimiz metodun üzerinde durup biraz konuşmamız gerekir.

- Windows Forms uygulamasında ListBox'tan bir eleman seçtiğinizde veya formdaki bir düğmeyi tıkladığınızda, Windows, kullanıcının başrolde olduğu işlem hakkında ilgili uygulamaya mesaj gönderir.
- Uygulama ise gönderilen mesaja yani olaya karşılık vermek amacıyla programcı tarafından hazırlanan metodu işletir. Programcı söz konusu olay meydana geldiğinde işletilecek metodu hazırlamadıysa hata meydana gelmez. Yani uygulama kullanıcıya herhangi bir tepkide bulunmaz.
- Burada ise "Dugme" adını verdiğimiz sınıfımız için **event** anahtar kelimesinden yararlanarak "Click" adında bir olay tanımladık. Olayı tanımladık tanımlamasına olay nasıl meydana gelecek, meydana geldiğinde neler yapılacak hazırlık yapmalıyız.
- Burada hazırladığımız olay, ilgili düğme tıkladığında veya veritabanındaki stok belli bir düzeyin altına indiğinde tetikleyecek durumda olmadığımıza göre diyoruz ki "Dugme" sınıfının OnClick() metodu işletildiğinde bu sınıfla ilgili Click olayı meydana gelmiştir. Başka bir anlatımla, Dugme sınıfımızdaki OnClick metodunun işletilmesi demek bir bakıma Click olayının tetiklenmesi anlamına gelecektir. Çünkü burada tetikleme işlemini kendimiz yapacağız.
- Burada OnClick() adını verdiğimiz metoda dikkatlice bakarsanız sanki Click() adında bir metodumuz varmış gibi işletiyoruz. Gerçekte Click bir metot olmayıp bir olaydır. Ancak daha önceden Click olayını "dugme1_Click()" adını vermeyi düşündüğümüz metotla ilişkilendirdiğimiz için OnClick() metodunda Click() yazmakla gerçekte 'dugme1_Click()' metodunu işletiyoruz. Baştan dugme1_Click() adını vermeye karar verdiğimiz metodumuzu henüz yazmadığımızı biliyorsunuz.

OnClick() adını verdiğimiz bu metotta yalnızca söz konusu olay meydana geldiği zaman işletilmesini istediğimiz metodun adı olacaktır. Bu satırlar sayesinde herhangi bir şekilde "Dugme" adlı sınıfın **Click**() metodu işletildiği zaman henüz hazırlamadığımız bir metodun işletilmesini sağlayacağız.

Başka bir deyişle bu metot sayesinde bu sınıfla ilgili olarak tanımlanan olay meydana geldiği zaman belirtilen metot (henüz hazırlamadığımız metot) işletilir. Ancak daha sonra sınıftan

yola çıkılarak nesne hazırlandığı zaman bu olay meydana geldiğinde işletilecek metot yoksa hatalı bir durumla karşılaşılmaması için **OnClick**() adını verdiğimiz bu metodu aşağıdaki gibi düzenledik. Bu düzenlemeden sonra uygulamamız hatasızca derlenebilir duruma geldi.

```
public void OnClick()
{
 if (Click != null)
   Click();
}
```

Bu noktada artık "Dugme" sınıfı için olay tanımlama ve olay meydana geldiği zaman işletilecek metodu işaret etmenin ön hazırlıkları tamamlandı. Geriye bu sınıftan yola çıkarak nesne hazırlanıp olay meydana geldiği zaman işletilecek metodu yazmak kaldı. Bu amaçla Main() metodunda "Dugme" sınıfından yararlanarak "dugme1" adında bir nesne hazırladık.

```
Dugme dugme1 = new Dugme();
```

Bu satır sayesinde elimizde "Dugme" sınıfından yola çıkılarak hazırlanmış "dugme1" adında bir nesne vardır. Şimdi sırada forma düğme yerleştirilip düğmenin üzeri çift tıklandığı zaman forma ait kod penceresinde "Windows Form Designer generated code" adlı blokta Visual Studio tarafından otomatik olarak hazırlanan satırın benzerini yazmak var. Bu amaçla uygulamamızda Main() metoduna yazdığımız 2 satırı aşağıda görebilirsiniz.

```
public static void Main()
{
 Dugme dugme1 = new Dugme();
 dugme1.Click += new Temsilci(dugme1_Click);
 dugme1.OnClick();
}
```

Main() metoduna yazdığımız ilk 2 satıra yabancı değilsiniz. Aslında 3. satırda sizin için yeni bir şey yoktur: Dugme sınıfının OnClick() metodu "dugme1" adını verdiğimiz nesne üzerinden işletilmektedir. Bu işletmenin amacı Click adını verdiğimiz olayı tetiklemektir. Çünkü bu uygulamamızda kısıtlı özelliklere sahiptir ve Click olayı kendimiz tetikliyoruz.

Bu noktada olay tanımlamakla bütün işlemler tanımlanmış oldu. Geriye "Dugme" sınıfından yol çıkılarak hazırlanan nesnenin kullanıldığı sınıfta dugme1_Click() metodunu yazmak kaldı. Çünkü "Temsilci" adını vermiş olduğumuz delegate tipinin yapıcı metoduna böyle bir metot adını parametre olarak verdik. Yoksa metoda başka bir ad verebilirdik. Bu amaçla uygulamamızdaki Main() metodunu içeren 2. sınıfta aşağıdaki gibi basit bir metot hazırladık.

```
public static void dugme1_Click()
{
 Console.WriteLine("Düğme tıklandı ve Click olayı meydana geldi");
}
```

Bu şartlarda konsol uygulaması işletildiğinde ekrana bu mesaj yazılır. Hazırladığımız konsol uygulamasında Main() metodunda sanki "Dugme" sınıfından türetilen "dugme1" nesnesi ile ilgili **Click** olayı meydana gelmiş gibi ilgili metot işletilmektedir. Yazılanların kolay anlaşılması için yukarıda parça parça verilen kodu aşağıda toplu halde verdik.

```
namespace ConsoleApp1
{
  public delegate void Temsilci();
  public class Program
  {
    public static void Main()
    {
     Dugme dugme1 = new Dugme();
     dugme1.Click += new Temsilci(dugme1_Click);
     dugme1.OnClick();
     Console.ReadLine();
    }
    public static void dugme1_Click()
    {
     Console.WriteLine("Düğme tıklandı ve Click olayı meydana geldi");
    }
  }
  public class Dugme
  {
   public event Temsilci Click;
   public void OnClick()
   {
    if (Click != null)
       Click();
   }
  }
}
```

Bu kodu tekrar anlatmak gerekirse şunlar söylenebilir; öncelikle bir **delegate** tanımladık. Devamında "Temsilci" adını verdiğimiz bu delegate'ten yararlanıp bir event ve event meydana geldiğinde işletilecek metodu hazırladık. Event ile event meydana geldiğinde işletilecek metodu ayrı sınıflarda hazırladık. Bu hali ile bu konsol uygulaması derlenip çalıştırıldığında Main() metodu içinde tanımlanan olay tetiklendiği için ilgili metot işletilir.

Yukarıda belirtildiği gibi olaylar, delegate nesneleri ile birlikte kullanılmaktadır. Bu nedenle yukarıda olay hazırlamadan önce delegate tipi hazırlamıştık. İsterseniz **System**'de bulunan **EventHandler** adlı mevcut delegate tipinden yararlanabilirsiniz. Bu delegate tipinden nasıl yararlanıldığını göstermek için yukarıda hazırladığımız uygulamayı aşağıdaki gibi değiştirdik.

```
public class Dugme
{
 public event EventHandler Click;
 public void OnClick()
 {
  if (Click != null)
   Click(this, new EventArgs());
 }
}
public class Program
 {
  public static void Main()
  {
   Dugme dugme1 = new Dugme();
   dugme1.Click += new EventHandler(dugme1_Click);
   dugme1.OnClick();
  }
```

```
    public static void dugme1_Click(object sender, EventArgs e)
    {
      Console.WriteLine("Click olayı meydana geldi");
    }
}
```

Konsol uygulamasının bu son haline dikkat ederseniz delegate hazırlamadığımızı fark edersiniz. Delegate olarak **System**'deki **EventHandler** kullanıldığında söz konusu olay meydana geldiğinde işletilecek metodun 2 parametreye sahip olması gerekmektedir. Bu nedenle delegate'i kendiniz hazırlasanız bile ilgili metodun **Object** ve **EventArgs** tipinde 2 parametreye sahip olmasını gözetmeniz önerilir.

Bu örnekte olay meydana geldiği zaman işletilecek metoda (Click) **this** ve **new EventArgs** parametre olarak verilmesine rağmen EventArgs'den yararlanmadık. Bu gibi durumlarda "new EventArgs" yerine **EventArgs.Empty** kullanılabilir.

```
class Dugme
{
 public event EventHandler Click;
 public void OnClick()
 {
  if (Click != null)
    Click(this, EventArgs.Empty);
 }
}
```

Yukarıda birkaç kez söylendiği gibi Delegate ve Event'lar C# programcıları için zor anlaşılan konuların başında gelmektedir. Bu nedenle aynı konuyu tekrar tekrar anlatma gereğini duyup pratik değeri olmasa bile Delegate ve olay kavramı hakkında yukarıda söylenenleri gözden geçirmenize neden olacak bir örnek daha hazırladık. Bu örnekte 2 Delegate hazırlayacağız. Çünkü konsol uygulamasında 2 olaya gerek duymaktayız.

Yine tahmin edeceğiniz gibi bu konsol uygulamamızda 2 sınıf olacak. İlk sınıfta olaylar ve söz konusu olaylar meydana geldiği zaman olayları tetikleyen metotlar. 2. sınıfta ise asıl işlem yapan yani olaylar meydana geldiği zaman işletilecek metotlar olacak. Öncelikle hazırladığımız örnekteki ilk sınıfı vereceğiz.

```
delegate void Temsilci1(int x);
delegate void Temsilci2();
class Karekok
{
 public event Temsilci1 Kokbul;
 public event Temsilci2 Mesajver;
 public void Tetikle1(int sayi)
 {
  if (Kokbul != null)
    Kokbul(sayi);
 }
 public void Tetikle2()
 {
  if (Mesajver != null)
```

```
        Mesajver();
    }
}
```

Şimdi bu kod üzerinde biraz konuşalım. İlk olarak 2 delegate tanımladık. Bu delegate'lerden birisi **int** tipinde bir parametreye sahip metotları temsil edebilecekken ikincisi parametresiz ve geriye bilgi göndermeyen metotları temsil edebilir. Devamında bu delegate'lerden yararlanıp "Kokbul" ve "Mesajver" adlarını verdiğimiz 2 olay tanımladık. Buna göre olaylara neden olacak sınıf, "Kokbul" ve "Mesajver" adında 2 metoda sahip olmalıdır.

Delegate ve olayları tanımladığımız bu sınıfta söz konusu olayları temsil edecek metotları tetikleyecek 2 ayrı metodun olması gerekiyordu. Buraya kadar zaten sizin için yeni bir şey yoktur. "Tetikle1" adını verdiğimiz metot ilk olayı tetiklerken "Tetikle2" metodu 2. olayın gereklerini yerine getirip "Mesajver" adını verdiğimiz metodun işletilmesini sağlayacak. Tabi bu metot Main() metodunu da içeren 2. sınıfta olacak. Bu örneğin 2. sınıfını aşağıda verdik.

```
class Program
{
 public static void Main()
  {
   Karekok Kok1 = new Karekok();
   System.Console.Write("Sayı girin   : ");
   string  str = System.Console.ReadLine();
   int sayi = System.Int16.Parse(str);
   if (sayi > 0)
    {
     Kok1.Kokbul += new Temsilci1(Kokbul);
     Kok1.Tetikle1(sayi);
    }
   else
    {
     Kok1.Mesajver += new Temsilci2(Mesajver);
     Kok1.Tetikle2();
    }
  }
 public static void Mesajver()
 {
  Console.WriteLine("Girdiğiniz sayı negatif");
 }
 public static void Kokbul(int sayi)
 {
  double sonuc = Math.Sqrt(sayi);
  Console.WriteLine("Sayının karekökü:"+ Convert.ToString(sonuc));
 }
}
```

Bu sınıfta ilk olarak yukarıda verilen ve delegate nesnelerini hazırladığımız "Karekok" adlı sınıfın örneğini aldık. Devamında kullanıcıdan konsoldan bir sayı girmesini istedik. Kullanıcının girdiği sayı 0'dan büyükse bu sınıfın "Kokbul" olayının 2. sınıftaki Kokbul metodu tarafından temsil edilmesini sağlayıp ilk sınıftaki Tetikle1() metodunu işlettik. Kullanıcının konsoldan girdiği sayı sıfırdan büyük değilse ikinci olay tetiklenecek. Dolayısıyla burada Mesajver() adını verdiğimiz metot işletilecek.

15. BÖLÜM

OPERATÖRLERİ AŞIRI YÜKLEMEK

Değişik tipler ve bu tiplerin içerikleri üzerinde bilinen işlemler yapılırken operatörlerin kullanıldığını biliyorsunuz. .NET dillerinin önemli artılarından birisi, operatörlere standart işlevlerinin yanında ek görevlerin verilebilmesidir. Operatörlere ek işlevler kazandırmaya, operatörlerin aşırı yüklenmesi denilmektedir. Bu bölümde operatörleri aşırı yüklemeye örnek olması için birkaç basit örnek yapmakla yetineceğiz. Çünkü operatörleri aşırı yüklemeyle yapılabilinecek işlemlere yüzlerce örnek verilebilir.

Operatörleri aşırı yükleme işlemi, ilgili sınıfta özel bir **public static** metot hazırlamakla mümkün olmaktadır. Hangi sınıf üzerinde işlem yaparken mevcut operatörlerden birisine ek görev vermek istiyorsanız o operatör için static bir metot hazırlamanız gerekiyor. Önemli bir ayrıntı şudur: Operatörleri aşırı yüklemenin geçerlilik alanı ilgili sınıfla sınırlıdır. Bu anlatılanlara göre hangi sınıfınızda operatörlerden bir veya birden fazlasını aşırı yüklemek istiyorsanız o sınıfta statik metot hazırlamanız gerekiyor. İlgili sınıfta 2 operatörü aşırı yükleme ihtiyacı duyuyorsanız 2 statik metot hazırlamalısınız.

Operatörleri aşırı yükleme, nesneye yönelik programcılıkla direk ilgili bir konudur. Çünkü nesneye yönelik programcılıkta mevcut operatörler birer metot olarak ele alınıp bu metotlara istenirse ek beceriler kazandırılır. C# uygulamaları dahilinde operatörlerin nasıl aşırı yüklenip ek işlemler yapabilme becerisine nasıl kavuşturuldukları konusunda bilgi vermeden önce bu konuya ısınmanızı sağlamaya çalışacağız. Bu amaçla ilk olarak aşağıda verilen 3 satırı incelemenizi istiyoruz.

```
int X = 10;
int Y = 20;
int Z = X + Y; // Z 30 olur
```

Bu işlemde '+' operatörü, sağına ve soluna yazılan integer bilgileri toplayıp geriye göndermektedir. Burada "X" ve "Y" değişkenleri + (artı) operatörü için birer operand, yani parametredir. Bu işlemi bit düzeyinde incelemek gerekirse aşağıdaki gibi resmedilebilir.

```
  0 0 0 0 1 0 1 0
+ 0 0 0 1 0 1 0 0
-----------------
  0 0 0 1 1 1 1 0
```

Aslında burada yapılan işlem son derece anlaşılırdır. Bütün programlama dillerinde "+" operatörünün soluna ve sağına yazılan sayısal bilgiler toplanır ve toplama işlemi sonucu bulunan değer geriye gönderilir. Burada "+" işaretini 2 parametreye sahip bir metot gibi düşünmek gerekir.

Şimdi "+" operatörü ile string bilgileri birleştirmeyi deneyeceğiz. C# ve birçok programlama dilinde "+" operatörünün soluna yazılan string bilgiye sağına yazılan bilgi eklenir. Bu 2 string bilginin birleştirilmesi işlemi için C# derleyicisinin geri planda birçok işlem yaptığını eski konulardan biliyorsunuz.

```
private void islem_Click(object sender, EventArgs e)
{
 string takim = "Fener" + "bahçe";
 MessageBox.Show(takim);  // takim 'Fenerbahçe' olur.
}
```

İlk bakışta bu kısacık kodda size yabancı gelen bir şey olmasa bile gerçekte burada yapılanlar oldukça farklıdır. Şimdi tıpkı C'de olduğu gibi C#'ın string'leri "+" operatörü ile birleştiremediğini varsayıp kendimiz birleştirmek için kod yazacağız. Bunu yapmamızdaki amaç, konu üzerinde düşünmenizi sağlamaktır. Yoksa bu kodun pratik bir değeri yoktur.

```
private void islem_Click(object sender, EventArgs e)
{
 string str1 = "Fener";
 string str2 = "bahçe";
 string Takim = "";
 int uz1 = str1.Length;
 int uz2 = str2.Length;
 char[] dizi = new char[uz1 + uz2];
 for (int i = 0; i<uz1; i++)
   dizi[i] = (char)str1[i];
 for (int i = uz1; i<uz1+uz2; i++)
   dizi[i] = (char)str2[i-uz1];
 for (int i = 0; i<uz1+uz2; i++)
   Takim = Takim + dizi[i];
 textBox1.Text = Takim;
}
```

Gerçekte ne zaman C# uygulamasında 2 string bilgi birleştirilmek istenirse derleyici geri planda buna benzer işlemler yapar. Bu işlemi buradaki gibi yapmak yerine String sınıfının **Concat**() metodu kullanılabilinir. Burada asıl anlatmak istediğimiz şudur: Sayısal bilgileri toplayan operatörle, string bilgileri birleştiren operatör aynı operatör olmasına rağmen birbirinden farklı işlevler görüyorlar. Başka bir deyişle C# derleyicisi "+" operatörünü aşırı yükleyip stringleri birleştirebilmektedir. Operatörlere ek görevler verip başka işlemler yaptırtmayı kendi hazırladığınız sınıflarda gerçekleştirebilirsiniz.

Şimdi C# derleyicisinin aşırı yüklediği başka bir operatörden söz edeceğiz. Bildiğiniz gibi .NET Framework ile gelen **DateTime** ve **TimeSpan** adında 2 yapı (struct) vardır. TimeSpan yapısının aşırı yüklenmiş 4 yapıcı metodu bulunmaktadır. İlk olarak **Tick** bilgisini parametre olarak alan yapıcı metodu kullanacağız. Bu amaçla DateTime yapısının **Ticks** özelliğinden yararlanarak 2 tarih arasındaki gün farkını bulduk.

```
private void islem_Click(object sender, EventArgs e)
{
 DateTime Tarih1 = DateTime.Parse("15.08.2017");
 DateTime Tarih2 = DateTime.Parse("15.08.2016");
 long ilk_tick = Tarih1.Ticks;
 long ikinci_tick = Tarih2.Ticks;
 System.TimeSpan Fark = new TimeSpan(ilk_tick - ikinci_tick);
 textBox1.Text = Fark.Days.ToString(); // 365 yazılır.
}
```

Gördüğünüz gibi burada TimeSpan yapısının yapıcı metoduna **Tick** birimi ile zaman bilgisini parametre olarak verdik. Aşağıda verilen kodda ise 2 tarih bilgisini birbirinden çıkarıp TimeSpan nesnesine aktardık. Yapılan işlem aynı gibi görünse bile burada TimeSpan yapısı tarafından operatörlere ek işlemler yaptırılmaktadır.

```
private void islem_Click(object sender, EventArgs e)
{
 DateTime Tarih1 = DateTime.Parse("15.08.2017");
 DateTime Tarih2 = DateTime.Parse("15.08.2016");
 System.TimeSpan Fark = Tarih1 - Tarih2;
 textBox1.Text = Fark.Days.ToString(); // 365 yazılır.
}
```

Operatörlerin aşırı yüklenmesiyle ilgili olarak "+" operatörü için şöyle bir açıklama getirmek mümkündür: "her ne kadar artı(+) operatörü sayısal bilgileri topluyor ve string bilgileri birleştiriyorsa da öyle bir ayarlama yapmalıyız ki '+' operatörü bilinen standart işlemlere ek olarak başka işlemler yapabilsin." Operatöre yüklenen ek görevler o sınıfla sınırlıdır. Ayrıca aşırı yüklenen operatör orijinal işlevini yitirmez.



Şimdiye kadar operatörlerin aşırı yüklenmesi ile ilgili olarak yazılanlara bakıp "operatörlerin aşırı yüklenmesi önemli midir?" sorusunu soruyor olabilirsiniz. Buna cevabımız net: Konu nesneye yönelik programcılık olduğunda önemlidir. Bu soruya cevabımız net olsa bile nesneye yönelik programlama dilleri sayılmak istendiğinde akla mutlaka gelen Java'da operatörleri aşırı yükleme imkanı yoktur. Bundan şu sonucu çıkaranlarınız olabilir: Madem Java'da operatörler aşırı yüklenemiyor, o halde olmazsa olmaz konulardan birisi değildir. Bize sorarsanız, bu bölümü sonuna kadar okuyup ondan sonra değerlendirme yapmanız daha sağlıklı olacaktır.

Mevcut operatörlerden birisini aşırı yüklemek için söz konusu sınıf dahilinde yani operatörün aşırı yüklenmiş haline gerek duyulan sınıfta **Operator** tipinde statik bir metot hazırlanması gerekiyor. Başka bir deyişle operatörleri aşırı yüklerken "Operator" anahtar kelimesinden yararlanıyoruz.

C# kodu yazarken yararlanabildiğiniz operatörlerin çoğunu aşırı yükleyebilme imkanı vardır. Derleyicinin overload etmesine izin vermediği operatörlere new, as, is, sizeof, typedef, =, ?:, ->, (), [], || ve && örnek olarak gösterilebilir. Yani, C# derleyicisi "()" ve "[]" gibi önemli operatörlere ek görevler vermemize izin vermemektedir.

Şimdi kendi hazırlayacağımız bir sınıf dahilinde "+" operatörü kullanıldığında sağına ve soluna yazılacak sınıfın (parametre) **int** tipindeki alanın içeriklerinin karelerinin alınıp toplanmasını sağlayacağız. Böyle bir işlemin pratik değerinin olmayacağı açıktır. Buradaki amacımız operator metotlarının nasıl hazırlandığını en basit bir şekilde anlatmak olduğu için basit bir işlemi seçtik.

Bu amaçla ilk olarak int tipinde bir alana sahip basit bir sınıf hazırladık. Kod uzamasın diye sınıfımızdaki alan için özellik hazırlamadık.

```
public class Islecler
{
 public int Sayi;
 public Islecler(int x)
  {
   Sayi = x;
  }
}
```

Gördüğünüz gibi bu sınıfın örneği alınırken yapıcı metoda parametre olarak verilen integer bilgiyi saklayabilecek bir alan bulunmaktadır. Bir bakıma bu sınıftan yola çıkılarak hazırlanan nesne bu hali ile işlev bakımından int tipindeki değişkenden farklı değildir.

Şimdi yapacağımız hazırlık sonucu bu sınıfın 2 örneğini toplayabileceğiz. Dolayısıyla hazırlayacağımız operatör metodu "Islecler" tipinde 2 parametreye sahip olacaktır. Hazırlayıp "Islecler" adını verdiğimiz sınıfta kodladığımız **operator** metodunun imzasını aşağıda görebilirsiniz.

```
public class Islecler
{
 public int Sayi;
 public Islecler(int x)
 {
  Sayi = x;
 }
 public static Islecler operator +(Islecler Par1, Islecler Par2)
  {
  }
}
```

Henüz bu metodun işlem yapacak satırlarını yazmamış olsak bile dikkatinizi bu metodun imzasına çekmek için maddeler halinde açıklama yapmayı deneyeceğiz:

- Bu operator metodunu "Islecler" adını verdiğimiz sınıfın içinde hazırladığımız için geçerlilik alanı bu sınıfla sınırlıdır. Başka bir deyişle diğer sınıflarda burada aşırı yüklemek istediğimiz "+" operatörü orijinal işlevini korumaya devam eder.
- Operatörleri aşırı yüklerken kullanılan metotların hem public hem de static olmaları gerektiği için metodun imzasında public ve static anahtar kelimelerine yer verdik. Metodun static yapılma nedeni sınıfının örneğini almaya gerek olmadan kullanmak gerekeceği içindir.
- Hazırlamak istediğimiz "operator" tipindeki metodu public ve static olarak işaret ettikten sonra sıra geriye göndereceği bilginin tipini yazmaya gelmektedir. Bu metot "Islecler" sınıfı tipinde bilgiyi, geriye göndereceği için "Islecler" sınıfının adını yazdık. Bu operator metodu geriye farklı bilgi gönderecek olsaydı onu yazardık.
- Operator metodumuz tarafından geriye gönderilecek bilginin tipini yazdıktan sonra sıra "operator" anahtar kelimesine gelmektedir. Bu anahtar kelime sayesinde derleyici hazırlanmak istenen metodun normal metot olmayıp mevcut bir operatö-

rün görevini yeniden tanımlanmak üzere hazırlanan bir metot olduğunu anlamaktadır. Bundan şu sonucu çıkarmak mümkündür: C# programlama dilinde **Operatör Metotları** adında bir metot türü bulunmaktadır.

- Operator anahtar kelimesinden sonra sıra aşırı yüklenmek istenen operatörü belirtmeye gelmektedir. "Islecler" adını verdiğimiz sınıf dahilinde artı (+) operatörünü aşırı yükleyip ek görev vermek istediğimiz için "+" yazdık. Artı yerine mevcut operatörlerden başka birisini yazabilirdik.
- En son olarak operatör metodu ile üzerinde işlem yapılacak bilgi veya bilgilerin parametre olarak parantezlerin içine yazılması gerekmektedir. Bu operatör metodu ile "Islecler" sınıfının 2 örneğini toplamak istediğimiz için parametrelerin "Islecler" tipinde olmasını sağladık.

Operator tipindeki metodun imzasını bu şekilde hazırladıktan sonra şimdi sırada bu metodun gövdesini yazmak var. Bu metotta yapılmasını istediğimiz şudur: "Islecler" tipindeki parametrelerin "Sayi" adını verdiğimiz alanların içeriklerinin karelerini toplayıp bulunan değeri kullanıp "Islecler" sınıfının 3. bir örneğini hazırlayıp geriye göndermektir. Bu işlemleri nasıl yaptığımızı aşağıda görebilirsiniz.

```
public static Islecler operator +(Islecler Par1, Islecler Par2)
{
 int sayi1 = Par1.Sayi;
 int sayi2 = Par2.Sayi;
 int sonuc = (sayi1 * sayi1) + (sayi2 * sayi2);
 Islecler nesne = new Islecler(sonuc);
 return nesne;
}
```

Operator metodunun gövdesinde yaptığımız işlemler son derece basittir. Yine de birkaç cümle yazmakta fayda var:

- Hazırladığımız operatör metodunun parametreleri "Islecler" tipinde olup "Islecler" sınıfının "Sayi" adlı int tipindeki alanın içeriğiyle ilgilendiğimiz için bu alanın içeriğini okuyup int tipindeki değişkenlere aktardık. Yani "+" operatörünün soluna yazılacak "Islecler" tipindeki nesnenin "Sayi" alanın içeriğini "sayi1" adını verdiğimiz değişkene, sağına yazılacak nesnenin "Sayi" alanını ise "sayi2" adlı değişkene aktardık.
- "Islecler" tipindeki nesnelerin "Sayi" alanların içeriklerini elde ettikten sonra kareleri toplamını bulup burada "sonuc" adını verdiğimiz değişkene aktardık.
- Anlatılan şekilde elde ettiğimiz kareler toplamından yola çıkıp "Islecler" tipinde yeni bir nesne hazırlayıp geriye gönderdik. Tahmin edeceğiniz gibi operatör metotların geri dönüş tipi void olamaz.

Şimdi sırada bu operatör metodunu kullanmak var. Bu nedenle hazırladığımız konsol uygulamasının Main() metodunu aşağıdaki gibi düzenledik. Main() metodu içinde önce "Islecler" tipinde 2 nesne hazırladık. Çünkü yukarıda hazırladığımız operatör metodu Islecler tipinde 2 parametreye sahiptir. Devamında bu 2 nesneyi toplayıp 3. nesneye aktardık.

Yukarıda yaptığımız ayarlamaya göre "Islecler" sınıfının örnekleri "+" operatörü ile toplandığında geriye yine bu sınıf tipinde nesne gönderildiğini biliyorsunuz. Yukarıda parça parça verdiğimiz kodun tamamı aşağıda bir arada verildi. Böylece operatörlerin nasıl aşırı yüklendiğini en basit hali ile görmüş olduk.

```
class Program
{
 public static void Main()
  {
   int s1 = 100;
   int s2 = 100;
   Islecler islec1 = new Islecler(s1);
   Islecler islec2 = new Islecler(s2);
   Islecler islec3 = islec1 + islec2;
   Console.WriteLine(islec3.Sayi);
  }
}
public class Islecler
{
 public int Sayi;
 public Islecler(int x)
  {
    Sayi = x;
  }
  public static Islecler operator +(Islecler Par1, Islecler Par2)
  {
   int sayi1, sayi2;
   sayi1 = Par1.Sayi;
   sayi2 = Par2.Sayi;
   int sonuc = (sayi1 * sayi1) + (sayi2 * sayi2);
   Islecler nesne = new Islecler(sonuc);
   return nesne;
  }
}
```

Yapılan işlemleri özetlemek gerekirse;

- Yukarıda overload ettiğimiz "+" operatörü 2 parametreye sahiptir. Yani ikili (binary) bir operatördür. Ayrıca bu opetator metodumuzun her iki parametresi de, bu metodu içinde yazdığımız sınıf (Islecler) tipindeydi.
- C# derleyicisi binary operatörler (yani 2 parametreye sahip) overload edilirken hazırlanacak metodun iki parametresinden birisinin başka tipte olmasına izin vermektedir. Bu şu demektir: Konu binary operatörleri overload etmek olduğunda parametrelerden birisi int veya double gibi built-in tiplerden olabilir.
- Artı(+) operatörünü "Islecler" sınıfımız için overload edip sonra Main() metodunda şu satırı yazıp işletmek demek "Islecler islec3 = islec1 + islec2;" gerçekte "Islecler" sınıfı içinde hazırladığımız operatör metodunu çağırıp işletmek demektir.

Operatörleri Aşırı Yüklemeye 2. Örnek

Yukarıda operatörlerin nasıl aşırı yüklendiğini anlatmak için verdiğimiz örnek son derece basitti ve anlaşılması kolaydı. Ayrıca örneğin pratik bir değeri yoktu. Şimdi vereceğimiz örnek yine basit olacak ve 2 karenin alanlarını toplanması sonucu elde edilecek alan değerini verecek 3. bir kareyi hazırlayacağız. Bu amaçla ilk olarak aşağıdaki gibi "Kare" adında bir sınıf hazırladık.



Overlaod etmek istediğiniz operator tek parametreye gerek duyuyorsa yani unary bir operatörse hazırlanacak operator metodundaki parametrenin tipi mutlaka ilgili sınıf tipinde olmalıdır. Operator metotlarında "ref" ve "out" anahtar kelimelerinin kullanılmasına izin verilmediğini bilmek gerekir.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
  Kenar = x;
  Alan = x * x;
 }
}
```

"Kare" adını verdiğimiz bu sınıf **double** tipinde 2 public alana sahiptir. 1 parametreye sahip yapıcı metotta karenin yüzölçümü bulunup "Alan" adını verdiğimiz public field'e aktarılmaktadır. Elbette alan bilgisini "Kare" nesnesi içinde tutmak pek anlamlı olmasa bile buradaki amacımız deneysel.

Kendisi için operatör metodu hazırlayacağımız sınıfı bu şekilde hazırladıktan sonra sıra "+" operatörü için metot hazırlamaya gelir. Amacımız 2 "Kare" nesnesini toplamak olduğu için operatör metodumuz "Kare" tipinde 2 parametreye sahip olacaktır. "Kare" adını verdiğimiz sınıfımızın operatör metodu eklenmiş hali aşağıdadır.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
  Kenar = x;
  Alan = x * x;
 }
 public static Kare operator +(Kare par1, Kare par2)
 {
  double alan1 = par1.Alan;
  double alan2 = par2.Alan;
  double toplam = alan1 + alan2;
  double uzunluk = Math.Sqrt(toplam);
```

```
  Kare nesne = new Kare(uzunluk);
  return nesne;
 }
}
```

Burada hazırladığımız operatör metodunda "+" operatörüne parametre olarak verilen 2 karenin alanları toplanıp bulunan alanı verecek kenar uzunluğu **Math** sınıfının **Sqrt**() metodu ile bulunmaktadır. Devamında bu kenar uzunluğundan yararlanılıp 3. bir "Kare" nesnesi hazırlanıp geriye gönderilmektedir.

Bu şekilde aşırı yüklenmiş artı(+) operatöründen yararlanmak için üzerinde çalıştığımız konsol uygulamasının Main() metodunu aşağıdaki gibi düzenledik. Bu örneği Windows Form uygulaması üzerinde yapıp bu işlemi görsel olarak göstermek mümkündür.

```
class Program
{
 public static void Main()
 {
  double kenar1 = 10;
  double kenar2 = 10;
  Kare kare1 = new Kare(kenar1);
  Kare kare2 = new Kare(kenar2);
  Kare kare3 = kare1 + kare2;
  Console.Write("Yeni karenin kenar uzunluğu : ");
  Console.WriteLine(kare3.Kenar);
  Console.Write("Yeni karenin Alanı : ");
  Console.Write(kare3.Alan);
 }
}
```

Yukarıdaki sayfalarda ifade edildiği gibi operatör metotları hangi sınıfın içinde kodlanıyorsa o sınıf tipinde bir parametreye sahip olmak zorundadır. Kendisi için metot hazırlanan operator binary ise parametrelerin birisi istenen tipte olabilmektedir. Bu zorunluluk geriye gönderilen bilgi için geçerli değildir. Şimdi konunun bu yanını ortaya çıkarmak için "+" operatörü için hazırladığımız metodun geriye "Kare" nesnesi yerine "Alan" bilgisini göndermesini sağlayacağız. Bu amaçla operatör metodunda yaptığımız değişikliği aşağıda görebilirsiniz. Tabi "+" operatörü için hazırlan metot bu şekilde değiştirildiğinde bu metodu yani "+" operatörünü kullanma şekli değişecektir.

```
public static double operator +(Kare par1, Kare par2)
{
 double alan1 = par1.Alan;
 double alan2 = par2.Alan;
 double toplam = alan1 + alan2;
 double uzunluk = Math.Sqrt(toplam);
 Kare nesne = new Kare(uzunluk);
 return nesne.Alan;
}
```

Şimdi farklılık olsun diye "++" operatörünü "Kare" sınıfımız için aşırı yükleyeceğiz. Yani bu operatör sayesinde "Kare" sınıfımızdan yola çıkılarak hazırlanacak yeni "Kare" nesnesinin kenar uzunluğu bu operatör sayesinde bir birim arttırılacaktır. Bu amaçla "Kare" sınıfımızda hazırladığımız 2. operatör metodu aşağıda verildi.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
   Kenar = x;
   Alan = x * x;
 }
 public static Kare operator +(Kare par1, Kare par2)
 {
  double alan1 = par1.Alan;
  double alan2 = par2.Alan;
  double toplam = alan1 + alan2;
  double uzunluk = Math.Sqrt(toplam);
  Kare nesne = new Kare(uzunluk);
  return nesne;
 }
 public static Kare operator ++(Kare par1)
 {
  double uzunluk = par1.Kenar + 1;
  Kare nesne = new Kare(uzunluk);
  return nesne;
 }
}
```

Bu operatör metodunun yukarıda hazırladığımızdan en büyük farkı bir parametreye sahip olmasıdır. Dolayısıyla bu parametrenin, ilgili sınıf tipinde (burada Kare) olması şarttır. Bu metot dahilinde yaptığımız, parametre olarak gelen nesnenin kenar uzunluğuna 1 ekleyip ve yeni kenar uzunluğundan yararlanıp yeni bir "Kare" nesnesi hazırlayıp geriye göndermektir. Bu operatör metodunu nasıl kullandığımızı aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 double kenar = 10;
 Kare kare1 = new Kare(kenar);
 Console.Write("Karenin kenar uzunluğu : ");
 Console.WriteLine(kare1.Kenar);
 Console.Write("Karenin Alanı : ");
 Console.WriteLine(kare1.Alan);
 Kare kare2 = ++kare1;
 Console.Write("Karenin yeni, kenar uzunluğu : ");
 Console.WriteLine(kare2.Kenar);
 Console.Write("Karenin Yeni Alanı : ");
 Console.Write(kare2.Alan);
}
```

Operatörlerin overload edilmesiyle ilgili olarak şimdiye kadar yazılanları okuyup verilen kısa ve basit örnekleri kodladıysanız konunun zor anlaşılan bir tarafının olmadığını fark etmişsinizdir. Tabi, kendisini overload etmek istediğiniz operatörlerle ilgili olarak "Operatörler" adlı bölümde yazılanları okumuş veya ilgilendiğiniz operatörün .NET Framework ile gelen built-in tiplerle nasıl kullanıldığını bilmeniz gerekiyor.

ArrayList'leri Birbirine Eklemek

Şimdi "+" operatörüne kazandıracağımız bir ek işlev sayesinde sağına ve soluna yazılacak ArrayList nesnelerini parametre olarak kabul edebilecektir. Yapacağımız ayarlama sayesinde artı (+) operatörü, sağ tarafına yazılan ArrayList'i soluna yazılana ekleyip geriye gönderecektir. Başka bir deyişle "+" operatörü sayesinde 2 ArrayList nesnesinin elemanlarını başka bir ArrayList nesnesinde birleştirebileceğiz. Bu amaçla hazırladığımız ve henüz bir işlem yapabilecek beceriye sahip olmayan sınıfı aşağıda görebilirsiniz.

```
class Program
{
public static void Main()
 {
  Console.ReadLine();
 }
}
public class Listeler
{
 public static Listeler operator +(Listeler par1, Listeler par2)
 {
   return par1;
 }
}
```

ArrayList nesnelerini birleştirmek üzere kullanacağımız bu sınıf bu hali ile "operator +" adında ve public static bir metoda sahiptir. Burada dikkat edilmesi gereken önemli nokta, metodun geri dönüş değerinin bu sınıfın tipinde olması gerektiğidir.

Bu sınıfta başka bir operatörü de aşırı yüklemek istemiş olsaydık o operatör için de bir metot hazırlardık. Örneğin çıkarma(-) operatörünü aşırı yüklemek istemiş isteseydik aşağıdaki gibi "operator" anahtar kelimesiyle başka bir metot hazırlardık.

```
class Listeler
{
 public static Listeler operator +(Listeler par1, Listeler par2)
 {
   return par1;
 }
 public static Listeler operator -(Listeler par1, Listeler par2)
 {
   return par1;
 }
}
```

Konuyu dağıtmamak için operatörlerin nasıl aşırı yüklendiğini anlatmak amacıyla hazırladığımız bu sınıftan çıkarma(-) operatörü ile ilgili metodu sildik.

"Listeler" adını verdiğimiz bu sınıfın ArrayList tipinde public bir alanı ve yapıcı metodunun ArrayList tipinde bir parametresi olacaktır. Bu sınıfın operator tipindeki metodun gövdesi hariç son hali aşağıdaki gibidir. Yani bu sınıfın yapıcı metodu ArrayList nesnesi talep edecek ve bu nesne "Dizi" adını verdiğimiz public alana aktarılacaktır.

```
public class Listeler
{
 public ArrayList Dizi;
 public Listeler(ArrayList liste)
  {
   Dizi = liste;
  }
 public static Listeler operator +(Listeler par1, Listeler par2)
  {
  }
}
```

Şimdi sırada bu operator metodunu kodlama var. Yukarıda söylendiği gibi bu metot 2 parametreye sahip olacak ve sınıf içinde barındırılan ArrayList nesnelerini birleştirip geriye gönderecektir. Tabi, ArrayList birleştirme denilince bunu her iki ArrayList nesnesinin elemanlarını başka bir ArrayList'te bir araya getirmeyi anlamak gerekir.



Yukarıdaki sayfalarda artı (+) operatörünü overload ederken operatöre verdiğimiz 2 parametrenin sıraları önemli değildi. Bu sınıf dahilinde yapacağımız overload işleminde parametrelerin sırası önemlidir. Çünkü 2. parametre olarak verilen nesnede bulunan ArrayList'in elemanları ilk parametredeki ArrayList'in elemanlarının arkasında yer alacaklardır. ArrayList sınıfı elemanları otomatik olarak sıralıyor olsaydı parametrelerin yerleri önemli olmazdı.

Hazırlamak istediğimiz operatör metodunda, parametre olarak kullanılacak "Listeler" sınıfımızın "Dizi" alanındaki ArrayList nesnelerini başka bir ArrayList nesnesinde birleştirmek istediğimiz için önce yeni bir ArrayList nesnesi hazırladık. Devamında "+" operatörüne parametre olarak verilecek "Listeler" tipindeki nesnelerin ArrayList tipindeki "Dizi" alanın elemanlarını **AddRange()** metodu ile yeni ArrayList nesnesine ekledik. En son olarak bu yeni ArrayList nesnesinden yola çıkarak yeni bir "Listeler" tipinde nesne hazırlayıp geriye gönderdik.

```
public class Listeler
{
 public ArrayList Dizi;
 public Listeler(ArrayList liste)
  {
   Dizi = liste;
  }
  public static Listeler operator +(Listeler par1, Listeler par2)
  {
```

```
        ArrayList takimlar = new ArrayList();
        takimlar.AddRange(par1.Dizi);
        takimlar.AddRange(par2.Dizi);
        Listeler nesne = new Listeler(takimlar);
        return nesne;
      }
    }
```

Bu metot yukarıdaki sayfalarda hazırlanan operator tipindeki metotlardan yaklaşım tarzı bakımından farklı değildir. Artı(+) operatörünün aşırı yüklendiği bu sınıfı hazırladıktan sonra üzerinde çalıştığımız uygulamadaki diğer sınıftaki Main() metodunu aşağıdaki gibi düzenledik. Bu uygulama derlenip çalıştırılırsa 2 ArrayList nesnesine aktarılan toplam 6 takımın adı ekrana yazılır.

```
class Sinif
{
 public static void Main()
 {
  ArrayList basket = new ArrayList(3);
  basket.Add("Anadolu Efes");
  basket.Add("Fenerbahçe Doğuş");
  basket.Add("Karşıyaka");
  ArrayList futbol = new ArrayList(3);
  futbol.Add("Beşiktaş");
  futbol.Add("Galatasaray");
  futbol.Add("Fenerbahçe");
  Listeler islec1 = new Listeler(basket);
  Listeler islec2 = new Listeler(futbol);
  Listeler islec3 = islec1 + islec2;
  foreach (var item in islec3.Dizi)
  {
   Console.WriteLine(item);
  }
 }
}
```

Implicit ve Explicit Operatörlerini Aşırı Yüklemek

Yukarıdaki sayfalarda amacımız sıklıkla kullanılan operatörlere ek işlevlerin nasıl kazandırıldığını basit şekilde anlatmaktı. Şimdi ise C# programcılarının sıklıkla kullandıkları Implicit ve Explicit operatörlerinden söz edeceğiz. Dolayısıyla bu kitabın "Değişken Tanımlamak ve Kullanmak" adlı bölümünde sözünü ettiğimiz bazı dönüştürmeleri hatırlayacağız.

Değişkenler ve operatörler hakkında bilgi verilen bölümlerde .NET Framework :le gelen built-in tiplerin birbirine nasıl dönüştürüldüğünden söz edilmişti. Örneğin aşağıda verilen 2 satırda önce byte tipinde bir değişken tanımlayıp kapasitesini aşmayacak büyüklükte bir sayıyı aktarmıştık. Devamında int tipinde bir değişken tanımlayıp byte değişkenin içeriğini int değişkene aktarmıştık.

```
byte b = 225; //
int i = b; // kapalı (implicit) dönüştürme
```

Başka bir deyişle byte bilgiyi dönüştürüp int tipindeki değişkene aktarmıştık. Burada sözünü ettiğimiz dönüştürme geri planda derleyici tarafından yapılmaktadır. "=" operatörüyle yapılan aktarma sırasında otomatik olarak yapılan bu dönüştürmeye kapalı (implicit) dönüştürme adı verilmektedir.

C# dilinde bu şekilde yapılan kapalı dönüştürme işlemi için kullanılan özel bir operatör yoktur. Derleyici bu kapalı dönüştürmeyi yaparken herhangi bir bilgi kaybının olmayacağını biliyor. Çünkü byte ile int birlikte değerlendirildiğinde byte bellekte daha az yer kaplamaktadır.

Yine kitabın değişkenlerin anlatıldığı bölümünde int tipinde bir değişken tanımlayıp 255 sayısını aktarmıştık. int değişkene aktardığımız sayıyı 255 seçmenizin nedeni byte değişkenin kapasitesini gözetmiş olmamızdı. Devamında int değişkenin içeriğini byte değişkene aktarmak istediğimizde derleyici buna itiraz etmişti.

```
int i = 255;
byte b = i; // hata meydana gelir.
```

Bu yaptığımız işlemde 255 sayısı byte değişkenin kapasitesini aşmıyor olmasına rağmen derleyici bu satıra itiraz eder. Çünkü burada yapılan işlem bilgi kayıplarına açık kapı bırakıyor. Buna rağmen programcı "ben ne yaptığımın farkındayım" deyip int tipindeki değişkenin içeriğini cast operatörünü "()" kullanarak byte değişkene aktarabilir. Casting operatörü kullanılarak yapılan dönüştürmeye ise açık (explicit) dönüştürme denilmektedir.

```
int i = 255; //
byte b = (byte)i; // Açık((explicit) dönüştürme.
```

Kendi yazdığınız sınıflarda açık ve kapalı dönüştürmeler yapılabilinsin istiyorsanız implicit ve explicit operatörlerini overload etmeniz gerekir. Bu işlemleri en basit şekilde anlatabilmek için yukarıda "+" operatörünü overload ederken hazırladığımız "Kare" sınıfından yararlanacağız. "Kare" sınıfımızın son hali aşağıdaki gibiydi.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
  Kenar = x;
  Alan = x * x;
 }
 public static Kare operator +(Kare par1, Kare par2)
 {
  double alan1 = par1.Alan;
  double alan2 = par2.Alan;
  double toplam = alan1 + alan2;
  double uzunluk = Math.Sqrt(toplam);
  Kare nesne = new Kare(uzunluk);
  return nesne;
 }
}
```

Şimdi öyle bir ayarlama yapacağız ki "Kare" sınıfı tipindeki nesne long tipindeki değişkene aktarılmak istendiğinde "Kare" tipindeki nesnenin "Kenar" adlı field'ın içeriğinin aktarılmasını sağlayacağız. Yani normalde hata verecek aşağıdaki satırın hata vermemesini sağlayacağız.

```
class Program
{
 public static void Main()
  {
   Kare kare1 = new Kare(10);
   long area = kare1; // hata meydana gelir.
  }
}
```

Bu tür bir aktarmanın mümkün olmasını sağlamak üzere "Kare" sınıfımızda aşağıdaki gibi yeni bir statik metot hazırlayarak **implicit** operatörünü overload ettik. Kalabalık etmesin diye Kare sınıfımızın "+" operatörüyle ilgili metodunu buraya almadık.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
  Kenar = x;
  Alan = x * x;
 }
 public static implicit operator long (Kare par)
 {
  return (long)par.Kenar;
 }
}
```

Kapalı dönüştürme yaparken yararlanacağımız bu metodun imzasını dikkatlice incelerseniz şunları görebilirsiniz:

- **implicit** anahtar kelimesi, bu operatör metodunun kapalı dönüştürme yapılırken kullanılacağını işaret ediyor. Bu metodu açık dönüştürme yapmak üzere hazırlamış olsaydık implicit yerine **explicit** anahtar kelimesini kullanırdık.
- Metodun imzasında kullanılan long, geri dönüş tipini yani bu metot geri planda kullanıldığında eşitliğin sol tarafında yer alacak hedef veri tipini işaret etmektedir.
- "Kare" ise kaynak veri tipini yani kullanım sırasında eşitliğin sağ tarafında yer alacak veri tipini işaret ediyor.
- Kapalı dönüştürme yaparken geri planda kullanılacak metodumuzun imzasını bu şekilde ele aldıktan sonra sıra metodun tek satırdan meydana gelen gövdesi hakkında konuşmaya geldi:
- Operatör metotları doğaları gereği void olamadıkları için **return** anahtar kelimesine yer verilmesi bir zorunluluktur.

- Örnek olsun diye hazırlayıp ve imzasında **implicit** anahtar kelimesine yer verdiğimiz metodumuzda geri dönüş tipini long olarak seçmiştik. Bu sınıfı kullanan herhangi bir programcı "Kare" nesnesini long tipinde bir değişkene aktarmak istediğinde "Kare" nesnesinin double tipindeki "Kenar" alanı long'a dönüştürülüp geriye gönderilecektir.

Yukarıda dönüştürme yaparken explicit operatörden yararlandık. İsteseydik buradaki dönüştürmeyi Convert sınıfının **ToInt64**() metodundan yararlanarak yapabilirdik. Konular karışmasın diye yukarıda hazırladığımız implicit metodunu aşağıdaki gibi değiştirdik.

```
public class Kare
{
  public double Alan;
  public double Kenar;
  public Kare(double x)
  {
   Kenar = x;
   Alan = x * x;
  }
 public static implicit operator long(Kare par1)
  {
   return Convert.ToInt64(par1.Kenar);
  }
}
```

Bu hazırlıktan sonra bu operatör metodunu aşağıdaki gibi kullandık. Tahmin edeceğiniz gibi implicit metodunu bu şekilde kullanmak veri kaybına neden olur. Çünkü "Kare" sınıfımızın "Kenar" alanı double iken long'a dönüştürdük. Dolayısıyla "Kenar" alanın bu kullanımda virgülden sonrası kaybolur.

```
static void Main(string[] args)
{
 Kare kare1 = new Kare(10.40);
 long kenar = kare1; // implicit metot çağrılır.
 Console.WriteLine(kenar);
}
```

Buradaki bilgi kaybı sizin için önemli ise dönüştürmeyi yukarıda yaptığımız gibi yapmamalısınız. Daha önceki konulardan bildiğiniz gibi kapalı dönüşümlere, bilgi kayıplarının ve taşma olasılığının olmadığı durumlarda başvurulması önerilmektedir. Buradaki amacımız deneysel olduğu için double'den long'a dönüşümde kaybolan bilgiyi göz ardı ettik.

Bu şartlarda kapalı dönüştürme yapmak üzere hazırladığımız implicit metodunu aşağıda yapıldığı gibi açık dönüştürme yaparken kullanırsak derleyici hiçbir itirazda bulunmaz. Sanki explicit operatör kullanılmamış gibi yine implicit metodunu işletir.

```
static void Main(string[] args)
{
 Kare kare1 = new Kare(10.40);
 long kenar = (long)kare1; // implicit metot çağrılır.
 Console.WriteLine(kenar);
}
```

Çünkü derleyici aynı dönüştürmede kullanılmak üzere implicit operator metodu hazırlandıysa bunu explicit dönüştürmelerde de kullanmaktadır. Örneğin aşağıda verilen 4 satırda derleyicinin aynı davranışını görebilirsiniz. Yani bir değişkenin içeriği kapalı dönüştürme yapılarak bir başka tipteki değişkene aktarılıyorsa açık dönüştürme de yapılabilmelidir.

```
byte sayi1 = 100;
byte sayi2 = 200;
int i1 = sayi1;
int i2 = (byte)sayi2;
```

Yani bir sınıf için sadece implicit operatörünün aşırı yüklemesi halinde, çalışma zamanında hem explicit hem de implicit dönüşümlere izin vermiş oluruz. Yukarıda "Kare" sınıfı için yaptığımız kapalı dönüştürmede bilgi kaybının önüne geçmek için implicit metotta long yerine double tipini kullandık.

```
public class Kare
{
  public double Alan;
  public double Kenar;
  public Kare(double x)
  {
   Kenar = x;
   Alan = x * x;
  }
  public static implicit operator double (Kare par1)
  {
   return par1.Kenar;
  }
}
```

Bu değişiklikten sonra bu implicit metot aşağıdaki gibi kullanılırsa istenen sonuç alınır ve bilgi kaybı olmaz. Bu değişiklikten sonra açık dönüştürme yapılarak implicit operatörünü kullanabileceğinizi biliyorsunuz.

```
public class Program
{
 static void Main(string[] args)
 {
  Kare kare1 = new Kare(10.40);
  double kenar = kare1; // implicit metot çağrılır.
  Console.WriteLine(kenar);
 }
}
```

Şimdi "Kare" sınıfımızda bir explicit operatöre yer vereceğiz. Bu amaçla "Kare" sınıfımızda yazdığımız yeni metodu görebilirsiniz. Explicit metotta, Kare nesnesinin "Alan" adlı double tipteki field'in içeriğini geriye göndermekle yetindik.

```
public class Kare
{
 public double Alan;
 public double Kenar;
 public Kare(double x)
 {
  Kenar = x;
  Alan = x * x;
 }
 public static implicit operator double (Kare par1)
 {
   return par1.Kenar;
 }
 public static explicit operator double (Kare par)
 {
   return par.Alan; // hata meydana gelir.
 }
}
```

Derleyici yazdığımız bu explicit metoda itiraz eder. Çünkü aynı kaynak ve geri dönüş tipi için (burada double) hem implicit hem de explicit metot yazılmasına izin vermiyor. Bu hatanın önüne geçmek için implicit metodun geri dönüş tipini float yaptık.

```
public class Kare
{
  public double Alan;
  public double Kenar;
  public Kare(double x)
  {
   Kenar = x;
   Alan = x * x;
  }
  public static implicit operator float (Kare par)
  {
   return (float) par.Kenar;
  }
  public static explicit operator double (Kare par)
  {
   return (double) par.Alan;
  }
}
```

Bu şartlarda "Kare" nesnesi kapalı dönüştürme yapılarak float değişkene aktarılmak istenirse "Kenar" alanın içeriği geriye gönderilir. Açık dönüştürmeye ile Kare nesnesi double değişkene aktarılmak istenirse "Alan" adlı field'in içeriği geriye gönderilir. Kare sınıfımız için hazırladığımız implicit ve explicit metotlarını nasıl kullandığımızı aşağıda görebilirsiniz.

```
public class Program
{
 static void Main(string[] args)
 {
  Kare kare1 = new Kare(10.40);
  float kenar = kare1; // implicit metot çağrılır.
```

```
        double alan = (double) kare1; // explicit metot çağrılır.
        Console.WriteLine(kenar);
        Console.WriteLine(alan);
      }
    }
```

Değişkenlerin anlatıldığı bölümden hatırlayacağınız gibi .NET Framework ile gelen standart veri tipleri bağlamında kapalı dönüştürmede sorumluluk derleyicinin iken açık dönüştürmede sorumluluk programcınındır. Benzer durum kendi hazırladığınız sınıflarda implicit metotları kullanırken de karşınıza çıkar. Örneğin Kare sınıfı için implicit metodu aşağıdaki gibi kullanırsak ne olur?

```
static void Main(string[] args)
{
 Kare kare1 = new Kare(10.40);
 double kenar = kare1; // implicit metot çağrılır.
 Console.WriteLine(kenar);
}
```

"Kare" sınıfıyla ilgili implicit metodunun bu şekilde kullanılmasına derleyicinin bir itirazı olmaz. Çünkü implicit metodun geriye gönderdiği float bilgiyi bellekte daha fazla yer kaplayan double değişkene aktarılması soruna neden olmaz. Implicit metodun geriye gönderdiği float bilgiyi long bir değişkene aktarılmasına derleyici izin vermez. Dolayısıyla implicit metodun içinde bilgi kaybına ve taşmaya neden olacak bir işlem yapmadıysak kullanma sırasında derleyici genel kurallara uyacaktır.

Konu explicit metot olduğunda ise derleyici devreden çıkıp sorumluluğu programcıya verdiği için dikkatli olmak gerekir. Örneğin explicit metodu aşağıdaki gibi kullanmak isterseniz yanlış sonuç alırsınız. Çünkü derleyici gider implicit metodu işletir.

```
static void Main(string[] args)
{
 Kare kare1 = new Kare(10.40);
 long alan = (long) kare1;
 Console.WriteLine(alan);
}
```

Bundan şu sonucu çıkarmak mümkündür: Kendi sınıfınız için Explicit metodu hazırladığınızda bütün olasılıklar için sonuçları test edip gerekli tedbirleri almalısınız. Yoksa beklenmeyen sonuçlar alınabilir.

Şimdiye kadar implicit ve explicit metotlar için olası bütün durumlar için konuştuk. Ancak hazırladığımız örneğin pratik değeri yoktu. Çünkü amacımız bu metotları overload ederken yaşanması muhtemel sorunlardan söz etmekti. Şimdi inch-cm çevrimi yaparken kullanabileceğimiz başka bir örnek hazırlayacağız.

Günlük hayatta karşımıza inch birimi ile ifade edilen bir boyut veya uzunluk geldiğinde bunu 2.54 ile çarpıp cm'ye dönüştürürüz. Benzer şekilde elimizdeki uzunluk cm iken bizden inch istendiğinde ise cm'yi 2.54'e bölüp inch değerini elde ederiz. Şimdi yazacağımız 2 sınıf

sayesinde bu işlemleri kolaylaştıracağız. Bu amaçla ayrı bir kod dosyasında aşağıdaki gibi 2 sınıf hazırladık.

```
public class Santim
{
 public double uzunluk;
 public static explicit operator Santim(Inch par1)
 {
  return new Santim
  {
   uzunluk = par1.uzunluk * 2.54
  };
  }
 }
 public class Inch
 {
  public double uzunluk;
  public static explicit operator Inch(Santim par1)
  {
   return new Inch
   {
    uzunluk = par1.uzunluk / 2.54
   };
  }
}
```

Bu sınıfları daha işlevsel yapmaya çalışmadık. Çünkü amacımız explicit ve implicit operatör metotları hakkında bilgi vermektir. Örneğin bu 2 sınıf için yapıcı metot hazırlayabilirdik.

Bu 2 sınıfı dikkatlice incelediğinizde her ikisinde de double tipinde ve "uzunluk" adında public bir alan bulunmaktadır. Bu sınıfların explicit metotlarında "uzunluk" alanında tutulan bilgiyi 2.54 ile çarpıp veya bölüp dönüştürme yaptık. Tabi burada bizi asıl ilgilendiren explicit metotların imzalarıdır. Bu nedenle "Santim" adını verdiğimiz sınıftaki implicit metodu mercek altına aldık.

- Santim sınıfımızda hedef tip Santim olduğu için "operator" anahtar kelimesinden sonra "Santim" tipine yer verdik. Hedef tip Santim olduğuna göre metodun parametresi "Inch" tipinde olacaktır. Çünkü dönüştürmeler santimden inch'e veya inch'ten santime olacaktır.
- Santim sınıfındaki explicit metot geriye Santim tipinde bir nesne göndereceği için (çünkü operator anahtar kelimesinden sonra Santim yazdık) önce Santim tipinde bir nesne hazırladık.
- Devamında parametre olarak gelen Inch tipindeki nesnenin "Uzunluk" alanını 2.54 ile çarpıp Santim tipindeki nesnenin aynı adı verdiğimiz alanına aktarıp Santim nesnesini geriye gönderdik.

Bu 2 sınıf sayesinde ne zaman elimizde Inch birimiyle ifade edilmiş bir uzunluk olduğunda bunu kolayca Santim olarak ifade edebiliriz. Elimizdeki inch birimi ile ifade edilmiş değeri cm'ye nasıl dönüştürdüğümüzü aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 Inch par = new Inch();
 par.uzunluk = 3;
 Santim cm = (Santim) par;
 Console.Write("inç ile uzunluk :" );
 Console.WriteLine(par.uzunluk);
 Console.Write("cm ile uzunluk  :");
 Console.WriteLine(cm.uzunluk);
 Console.ReadLine();
}
```

16. BÖLÜM

İNDEKSLEYİCİ HAZIRLAMAK ve KULLANMAK

C# konulu kaynaklarda indexer'lar hakkında bilgi verilirken dizi değişkenlerden mutlaka söz edilir. Çünkü indeksleyiciler hakkında bilgi içeren makale veya kitabı okuyan birisi muhtemelen daha önce dizi değişkenler hakkında bilgi edinmiştir, dizi değişken tanımlayıp kullanmıştır. Son derece teknik bir konu olan indeksleyiciler anlatırken Array sınıfı hakkında edinilmiş bilgilerden yararlanma isteği yerinde bir seçimdir. Kitabımızın bu bölümünde bu geleneğe uyduk. Bunu yaparken konunun arka planını biraz deştik.



Bu kitabın bu bölümü de dahil olmak üzere bir kavram veya sınıf hakkında bilgi verirken ister istemez örnek kodlar hazırlıyoruz. Genellikle pratik değeri var mı yok mu sorusunu sormadan hazırladığımız bu kodlarda sınıflara, metotlara, alanlara vs yer veriyoruz. Sonra bu kodlar aracılığıyla ilgilenilen kavram veya .NET Framework ile gelen sınıf hakkında bilgi verirken kendi hazırladığınız sınıfın ve metodun adını zikretmek durumunda kalıyoruz. İşte asıl sorun burada ortaya çıkıyor: Kitap aracılığıyla ilgili bölümde işlenen konu üzerinde çalışan programcı kitapta verilen örnek koddan biraz uzaklaşır ve farklı sınıf metot adlarını tercih ederse metnin okunması, okunanı anlamak zorlaşıyor. Bu nedenle okurların başlangıçta bu kitapta verilen kodlara bağlı kalmalarını, ilgili bölümde yapılan işlem anlaşıldıktan sonra kitapta verilen örnekleri değiştirmelerini veya başka örnekler hazırlamalarını öneriyoruz.

Dizi değişkenlerde elemanların içeriği elde edilirken veya elemanlara bilgi aktarılırken operatör görevi yapan köşeli parantezlerden "[]" yararlanılıyor. Her elemanın dizi içindeki yerini işaret eden sıra numarası köşeli parantezlerin içine yazılıyor. Çünkü dizi değişkenlerin elemanları geri planda indekslenmektedir. Başka bir deyişle dizi değişkendeki her elemana int tipinde birer sıra numarası veriliyor. Bu sıra numarasından yararlanarak dizi değişkenin istediğimiz elemanına erişim sağlayabiliyoruz.

Benzer şekilde kendi hazırladığınız sınıflarda dahili bir koleksiyona yer verip bu koleksiyonun üyelerini indeksleyebilirsiniz. Özelliklerde olduğu gibi **Indexer**'lerde de **set** ve **get** blokları bulunmaktadır. Bu nedenle indexer'leri **özellik dizisi** olarak görmek yanlış olmaz. Kendi hazırladığınız sınıftaki elemanların bir veya birden fazlası içsel koleksiyon gibi işlev görecekse indeksleyici hazırlamanız önerilir. Bu sayede bu özelliklere "[]" operatörü ile erişim imkanı elde edilir.

Burada sözünü ettiğimiz "[]" köşeli parantezlerin bir operatör olduğunu unutmamak gerekir. Bu kitabın" Operatörleri Aşırı Yüklemek" adlı bölümünü teknik bir konu olarak görüp atlamadıysanız indeksleyicilerle ilgili olarak anlatılan her şeyin daha anlaşılır olduğu göreceksinizdir.

Dizi değişkenlerle ilgili olarak yukarıda yaptığımız bu açıklamaya ek olması için dizi değişkenlere başka bir pencereden bakmayı deneyeceğiz. En başta şunu söyleyelim, dizi değişkenler yani **Array** tipindeki nesneler, istenilen sayıda nesne veya tipteki bilgileri bellekte tutabilmektedir. Array nesnesinde kaç tane bilgi veya nesne saklanmak isteniyorsa

bu en başta belirtiliyor. Bu nesnelere bilgi aktarma veya okuma indeksleyici ile yapılmaktadır. Örneğin aşağıdaki gibi bir dizi değişken tanımlandığında bu nesnede string tipte 10 adet bilgi saklanabilmektedir.

```
string[] strDizi = new string[10];
```

Başka bir anlatımla, böyle bir satırın koda yazılıp işletilmesiyle Array tipinde bir nesne hazırlanır ve bu nesnenin dahili koleksiyonunda string tipte 10 bilginin saklanabilmesi için ayarlama yapılır. Array sınıfının metotlarının çoğu statik olsa bile bu nesne üzerinden işletebileceğimiz bir kaç metot ve özellik bulunmaktadır. Çünkü Array sınıfı **IList** ve **ICollection** interface'lerini uygulamaktadır.

Burada Array sınıfından asıl söz etme nedenimiz, bu sınıfın dahili bir koleksiyona sahip olması ve bu koleksiyon indekslenmiş olmasıdır. Bu dahili koleksiyonda tutulacak bilgilerin tipi Array sınıfının örneği hazırlanırken yani dizi değişken tanımlanırken belirleniyor. Bu yönüyle dizi değişkenler generic tiplere benzemektedir.

Yukarıda dizi değişken tanımlarken, başka bir ifade ile Array tipinde nesne hazırlarken nesnenin dahili koleksiyonunda string tipte 10 bilginin saklanacağını işaret etmiş olduk. Dizi değişkeni aşağıdaki gibi tanımlamış olsaydık 'yillar' adını verdiğimiz Array tipindeki nesnenin dahili koleksiyonunda int tipinde 5 bilginin saklanabilmesi için ayarlama yapmış olurduk.

```
int[] intDizi = new int[5];
```

Burada "intDizi" adını verdiğimiz nesnenin dahili koleksiyonuna erişimde indeksleyici kullanıldığı için içeride neler olduğuyla ilgilenmeyiz. Ötesi, dahili koleksiyonun adını bilmeyiz. Bilmemize gerek yoktur. Bu dahili koleksiyonun adını bilsek bile private olduğu için erişemeyiz.

Benzer şekilde, hazırlamak istediğimiz sınıfta dahili olarak koleksiyon gibi işlev görecek bir alana yer verip bu koleksiyonun elemanlarına sıra numaralarıyla erişim sağlamak istiyorsanız sınıfınızda indeksleyici hazırlamanız gerekir.

Burada indeksleyicilerin nasıl hazırlanıp kullanıldığını adım adım anlatacağız. Hatta ilk hazırlayacağımız indeksleyicinin pratik değeri olmayacak. Bu amaçla aşağıda verilen uygulamayı hazırladık. Bu konsol uygulamasında "Yayincilar" adını verdiğimiz 2. sınıfta "dizi" adında bir alan olacak. Tahmin edeceğiniz gibi burada "dizi" adını verdiğimiz alanı "Yayincilar" sınıfımızda dahili koleksiyon olarak kullanacağız. Bu hali ile bu uygulama derlenirse hata meydana gelmez.

```
namespace ConsoleApp1
{
 public class Program
  {
   public static void Main()
    {
    }
  }
```

```
public class Yayincilar
 {
  private string[] dizi = new string[3];
 }
}
```

Bu sınıf dahilinde dizi değişken gibi işlev görecek yani içinde birden fazla bilgi barındırabilecek bir alana yer vermiş olduk. Bu alana erişimde kullanmak üzere sınıf için indeksleyici tanımlamamız gerekecektir. Yoksa sınıf dahilinde tutulacak bilgilere erişim sorunlu olacaktır. Elbette bu sınıfta bir özellik hazırlayıp veya bu alanı public yaptıktan sonra bu sınıftan yola çıkıp hazırlayacağımız nesne üzerinden bilinen şekilde bu alanda tutulan bilgilere erişim sağlamak mümkündür. Buradaki amacımız bu alana indeksleyici kullanarak erişim sağlamaktır. Bu nedenle bu alanın **private** olarak kalmasını istedik.

Şimdi "Yayincilar" sınıfındaki string tipinde ve "dizi" adındaki değişkenin veya alanın elemanlarına erişebilmek için indeksleyici tanımlayacağız. Bu alan **private** olduğu için başka sınıflardan zaten erişilemez. "dizi" adını verdiğimiz ve string tipindeki bu alan dizi değişken olmasaydı indeksleyici yerine özellik tanımlamaktan söz etmek daha anlamlı olurdu. Bu arada, indeksleyicinin özel işlevi olan bir özellikten (property) başka bir şey olmadığını bilmek gerekir.

Dizi değişkenin veya alanın elemanları için indeksleyici hazırlarken tip belirtecinden sonra **this** anahtar kelimesi kullanılmakta, köşeli parantezin arasında ise elemanın dizi içindeki sırasını gösterecek bilginin tipi (genellikle sayısal) yazılmaktadır. Yani sınıfta hazırlamak istediğiniz özelliği indexer olarak kullanmak istiyorsanız herhangi bir isim vermeden this'i kullanmak gerekiyor. İndeksleyici olarak kullanmak istediğimiz özelliği nasıl hazırladığımızı aşağıda görebilirsiniz.

```
public class Program
{
 public static void Main()
  {
  }
}
public class Yayincilar
{
  private string[] dizi = new string[3];
  public string this[int indis]
  {
   get
    {
    }
   set
    {
    }
  }
}
```

Daha önceki konulardan bildiğiniz gibi sınıfların private alanlarına erişimde kullanılan property'ler için erişim belirleyici olarak genellikle public seçilmektedir. Aynı seçimi burada

yaptık. Erişim belirleyici anahtar kelimeden sonra burada kullanılan **string**, geri dönüş tipidir. Yani indexer aracılığıyla içeriği okunacak bilginin tipini string olarak seçtik. Çünkü bu indexer aracılığıyla sınıfın erişeceğimiz alanı ('dizi' adını verdiğimiz alan) **string** tiptedir. Bir indeksleyicinin geri dönüş tipine mutlaka sahip olması gerektiğini tahmin etmiş olmalısınız.

Bu property'yi "Yayincilar" adını verdiğimiz sınıfta indexer olarak kullanmak istediğimiz için geri dönüş tipinden sonra yani özelliğin adının yazıldığı yere **this** anahtar kelimesini yazdık. Burada kullanılan this, yazdığımız property'nin indexer olarak görülmesine neden olmaktadır. This'ten sonra ise "[]" operatörüne yer verdik. Bu hali ile uygulama derlenmek istenirse hata meydana gelir. Çünkü indeksleyicinin en azından get bloğunun yazılmış olması gerekirdi.

Bu örnekte indeksleyicinin kullanacağı "indis" bilgisinin tipini **int** olarak seçtik. Kendi hazırladığınız sınıflarda indeksleyici için int dışında başka bir tipi seçebilirsiniz. C# programcıları indis olarak kullanılacak bilgiyi genellikle int olarak seçiyorlar.

Şimdi **get** ve **set** bloklarına birer satır yazıp dahili koleksiyon olarak kullanacağımız alana başka sınıflardan erişilebilmeye imkan sağlayacağız. Dolayısıyla indeksleyicinin **public** olması gerektiğini tahmin ediyorsunuzdur. İndeksleyici olarak kullanacağımız özelliğin set ve get bloklarına yazdıklarımız, özellikler için hazırlanan bloklardan farklı değildir.

```
public class Yayincilar
{
 private string[] dizi = new string[3];
 public string this[int indis]
 {
  get
   {
    return dizi[indis];
   }
  set
   {
    dizi[indis] = value;
   }
  }
}
```

Bu şekilde düzenlenen set ve get blokları sayesinde "Yayincilar" sınıfının "dizi" adlı alanın elemanlarına "[]" operatörü ile erişilebilinir. Yani henüz işlevsel olmasa bile teorik olarak indeksleyicimiz hazır sayılır. Bu nedenle indeksleyicimizin mevcut hali üzerinde konunun anlaşılmasına katkı niyetine buraya birkaç paragraf daha yazacağız:

- "Yayıncılar" adını verdiğimiz sınıfımızda 3 elemanlı ve string tipte bir dizi değişken tanımladık. Yani bir alan hazırladık. Bu dizi değişken private olduğu için başka sınıflardan, bu sınıf tipinde nesne hazırlandığında erişilemez.
- Bu alan private olduğundan (ayrıca sınıfımız yapıcı metoda henüz sahip değildir) bu alana bilgi aktarırken ve mevcut içeriği okurken kullanmak üzere özellik hazırla-

mak gerekecektir. Hazırlamak istediğimiz özellik aracılığıyla sınıftaki ilgili alana erişim sağlanırken "[]" operatöründen yararlanmayı düşündüğümüz için tanımlayacağımız özellik standart dışı bir özellik olacaktır.

- Nesneye yönelik programcılıkta "[]" aracılığıyla kullanılan özelliklere indexer adı verilmektedir. Başka bir anlatımla, programcı ne zaman (buradaki sınıf adlarıyla) "Yayincilar" tipindeki nesnenin adının yanına "[]" yazarsa, derleyici programcının kendisi için özellik hazırlanan alana erişmek istediğine karar verecektir.
- Sınıf için hazırlanacak indexer, "[]" operatörü aracılığıyla kullanılacağı için bir ada sahip olması gerekmiyor. Tabi derleyici normalde isimsiz özelliklere izin vermez. Bu nedenle "this" anahtar kelimesinden yararlanılır.
- Özelliğin imzasında "bu sınıf" anlamında "this" anahtar kelimesine yer yerdik. Bunu, buradaki sınıf adıyla anlatmak gerekirse, programcı bu özellikten yararlanmak istediğinde önce "Yayincilar" (tabi direk Yayincilar yazmayacak, "Yayincilar" tipindeki nesnenin adını) yazacak. This anahtar kelimesinden sonra [] operatörüne yer verdiğimiz için "Yayincilar" tipindeki nesnenin yanına "[]" yazılır. Hazırlanan özelliğin "[]" operatörü aracılığıyla kullanılmak istendiği ifade edildikten sonra sıra "[]" arasına yazılacak ve indis görevi yapacak bilginin tipini işaret etmeye gelir. Burada hazırladığımız indexer'da bu tipin int olmasına karar verdik.

Konunun iyice netleşmesi için daha önce özellikler hakkında bilgi verirken hazırladığımız bir özelliğe ait kodları buraya aldık. Bu normal property ile, indexer işlevi görecek property'e ait kodları birlikte incelerseniz konu netleşir.

```
public string Numara_kontrol
{
 get
  {
  }
 set
  {
  }
}

public string this[int indis]
{
 get
  {
  }
 set
  {
  }
}
```

Şimdi sırada kendisi için indexer hazırladığımız sınıfın örneğini alıp dahili koleksiyona bilgi aktarıp sonra bu bilgileri okumak var. Bu amaçla uygulamamızın Main() metoduna sahip diğer sınıfında "Yayincilar" sınıfının örneğini hazırlayıp nesnenin dahili koleksiyona bilgi aktardık.

```
namespace ConsoleApp1
{
 public class Program
 {
  public static void Main()
  {
    Yayincilar yay = new Yayincilar();
    yay[0] = "Seçkin Yayıncılık";
    yay[1] = "Alfa Yayınları";
    yay[2] = "Dikeyeksen Yayın Dağıtım";
    Console.WriteLine(yay[0]);
    Console.WriteLine(yay[1]);
    Console.WriteLine(yay[2]);
  }
 }
 public class Yayincilar
 {
  private string[] dizi = new string[3];
  public string this[int indis]
  {
   get
    {
     return dizi[indis];
    }
   set
    {
     dizi[indis] = value;
    }
   }
  }
}
```

Verilen kodda "Yayincilar" adını verdiğimiz sınıfta alanın adı "dizi" iken Main() metodunda bu sınıf tipindeki nesneyi kullanırken yani "dizi" alanına erişim sağlarken "[]" operatöründen yararlandık. Bu alana erişim için indexer hazırlamamış olsaydık yukarıda Main() metodu içinde kullanılan kodda bu property'in adı mutlaka olurdu.

Bu örnekte dahili koleksiyon işlevi gören alan **string** tipindeydi. Bu alanın, dolayısıyla indexer'ın geriye gönderdiği bilginin tipini **object** olarak seçip istediğiniz tipteki bilgileri sınıfın dahili koleksiyonunda aktarabilirsiniz. Her şeyin generic olduğu bu zaman diliminde bu öneri kabul edilebilir değildir. Konunun anlaşılması için bu sınıfa birisi string diğeri int tipinde 2 alan ekledik.

```
public class Yayincilar
{
 public string kategori;
 public int yil;
 private string[] dizi = new string[3];
 public string this[int indis]
 {
  get
   {
    return dizi[indis];
   }
```

```
    set
     {
      dizi[indis] = value;
     }
    }
}
```

Konu uzamasın diye bu 2 alan için özellik hazırlamak zorunda kalmamak için public yaptık. Bu elemanlara Main() metodu içinde bilgi aktarıp içeriğini konsola yazacağız. Bu amaçla Main() metodunu aşağıdaki gibi düzenledik. Bu kod incelendiğinde indeksleyici içeren sınıfın örneği alınıp bu örneğe (yani nesneye) köşeli parantezler eklendiğinde birden fazla elemana sahip alan işaret edilmiş olunuyor.

```
public static void Main()
{
 Yayincilar yay = new Yayincilar();
 yay[0] = "Seçkin Yayıncılık";
 yay[1] = "Alfa Yayınları";
 yay[2] = "Dikeyeksen Yayın Dağıtım";
 Console.WriteLine(yay[0]);
 Console.WriteLine(yay[1]);
 Console.WriteLine(yay[2]);
 yay.kategori = "Bilişim";
 yay.yil = 2017;
 Console.WriteLine(yay.kategori);
 Console.WriteLine(yay.yil);
}
```

"Yayincilar" adını verdiğimiz sınıfta kendisi için indexer hazırladığımız alanın baştan 3 elemana sahip olmasını istediğimiz için bu sınıfın örneği alınıp köşeli parantezin içine 3 veya daha büyük bir değer yazılırsa hata meydana gelir. Çünkü "Yayincilar" adını verdiğimiz sınıfta kendisi için indexer hazırladığımız string tipteki alanın eleman sayısını baştan 3 olarak sabitlemiştik. Bu sınıf için yapıcı metot hazırlayıp dahili koleksiyon olarak kullandığımız elemanın sayısını çalışma anında tespit edebilirsiniz. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
public class Yayincilar
{
  public string kategori;
  public int yil;
  private string[] dizi;
  public Yayincilar(int sayi)
  {
   dizi = new string[sayi];
  }
  public string this[int indis]
  {
   get
   {
     return dizi[indis];
   }
```

```
        set
        {
         dizi[indis] = value;
        }
      }
    }
```

Bu değişiklikten sonra bu sınıfın örneği alınırken yapıcı metoda hangi sayı parametre olarak verilirse kendisi için indexer hazırlanan alanın eleman sayısı o kadar olur. Buna rağmen kullanıcı veya programcı indexer aracılığı ile dahili koleksiyon olarak kullandığımız alana bilgi aktarırken sınırları aşabilir. Başka bir deyişle sınıfın dahili koleksiyonunda olmayan bir eleman kullanılmak istenirse **IndexOutRangeExection** hatası meydana gelir. Olası hataların önüne geçip programın kırılmasını önlemek için set veya get blokları içinde kontrol işlemi yapılabilir. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
namespace ConsoleApp1
{
 public class Program
 {
  public static void Main()
  {
   Yayincilar yay = new Yayincilar(3);
   yay[0] = "Seçkin Yayıncılık";
   yay[1] = "Alfa Yayınları";
   yay[2] = "Dikeyeksen Yayın Dağıtım";
   Console.WriteLine(yay[0]);
   Console.WriteLine(yay[1]);
   Console.WriteLine(yay[2]);
   yay.kategori = "Bilişim";
   yay.yil = 2017;
   Console.WriteLine(yay.kategori);
   Console.WriteLine(yay.yil);
   Console.ReadLine();
  }
 }

 public class Yayincilar
 {
  public string kategori;
  public int yil;
  private string[] dizi;
  public Yayincilar(int sayi)
  {
    dizi = new string[sayi];
  }
  public string this[int indis]
  {
   get
    {
     if (indis < dizi.Length)
      {
       return dizi[indis];
      }
```

```
            else
             {
              throw new IndexOutOfRangeException("Index Hatası");
             }
            }
            set
             {
              if (indis < dizi.Length)
               {
                dizi[indis] = value;
               }
              else
              {
               throw new IndexOutOfRangeException("Index Hatası");
              }
             }
           }
        }
    }
```

Kendimize Array sınıfını örnek aldığımız için olmayan eleman okunmak istendiğinde **throw** deyimi ile ortama **IndexOutRangeExection** tipinde nesne fırlattık. Tabi sizler bu hatayı yakalayıp uygulamanın çalışmasının kırılmasının önüne geçebilirsiniz. Ayıca programcı indis olarak eksi bir sayıyı kullanabilir. Bunu da düşünüp kontrolü ona göre yapmak gerekir.

Indexer'ları Aşırı Yüklemek

Buraya kadar anlatılanlara göre sınıf içinde indekslenecek herhangi bir alan varsa **this** anahtar kelimesini kullanıp set ve get bloklarını hazırlamak gerekmektedir. Bu tespitten dolayı sınıflarda en fazla bir alanın indekslenebileceği sonucu çıkarılabilir. Indexer'lar aşırı yüklenebildiği için sınıftaki birden fazla alanı indekslemek mümkündür. Indexler'ların aşırı yüklenmesine veya bir sınıfta birden fazla indexer'a ender durumlarda gerek duyulmaktadır.

Bu konuda bilgi vermek için yukarıda hazırlayıp "Yayincilar" adını verdiğimiz sınıfı biraz değiştireceğiz. Burada amacımız indexer'ların nasıl hazırlandığını anlatmak olduğu için örneğin pratik değerinin olmasını gözetmedik. Kodu kısa tutmak için hata kontrolü yapılan satırları kaldırdık. Ayraca sınıfımıza sonradan eklediğimiz 2 alanı da kaldırdık. Önce sınıfımızın son halini verip ondan sonra üzerinde konuşacağız.

```
public class Yayincilar
{
 private string[] dizi1;
 private string[] dizi2;
 public Yayincilar(int sayi)
 {
  dizi1 = new string[sayi];
  dizi2 = new string[sayi];
 }
 public string this[int indis]
 {
  get
   {
```

```
        return dizi1[indis];
       }
       set
       {
        dizi1[indis] = value;
       }
      }
     }
```

"Yayincilar" adını verdiğimiz sınıfımızın son halinde ilk fark etmenizi istediğimiz değişiklik dahili koleksiyon olarak kullanabileceğimiz 2. bir alanın olmasıdır. Burada ikinci fark edip üzerinde düşünmenizi istediğimiz ayrıntı şudur: Dahili koleksiyon olarak kullanacağımız her iki alan da aynı tipe sahiptir. Dolayısıyla aşırı yüklemede farklılık veri tipi üzerinden sağlanamaz. Sınıfta dahili koleksiyon olarak kullanabileceğimiz 2 alan olunca indexer işlevi görecek 2 property hazırlayabiliriz. Sınıfımızın 2. indexer hazırlanmış hali aşağıdadır.

```
public class Yayincilar
{
 private string[] dizi1;
 private string[] dizi2;
 public Yayincilar(int sayi)
 {
  dizi1 = new string[sayi];
  dizi2 = new string[sayi];
 }
 public string this[int indis]
 {
  get
   {
    return dizi1[indis];
   }
  set
   {
    dizi1[indis] = value;
   }
  }
  public string this[byte indis]
  {
   get
   {
    return dizi2[indis];
   }
   set
   {
    dizi2[indis] = value;
   }
  }
}
```

"Yayincilar" adlı class'ın son halinde birisinin indisi int, diğerin ise byte tipinde olan 2 indexer bulunmaktadır. Tahmin edeceğiniz gibi int tipinde indis kullanıldığında ilk indexer, byte tipinde indis kullanıldığında ise ikincisi işlevsel olacaktır. Çünkü sınıfta tanımladığımız ve dahili koleksiyon olarak kullanabileceğimiz her 2 alan string tipindedir. Dolayısıyla derle-

yici hangi indexer'ın kullanılmak istendiğini indisin tipine bakarak anlayabilecektir. Sınıfımızda dahili koleksiyon olarak kullanabileceğimiz 2 alan olduğu için yapıcı metotta 2 dizi değişken hazırladık.

Burada dikkat edilmesi gereken nokta, burada yapılanın indexer'ın aşırı yüklenmesi olduğu ve 2. indexer'ın tanımlama satırındaki köşeli parantezlere farklı bir tipte indis kullanılması gerektiğidir. İlk indexer için **int** tipinde indis kullandığımız için ikincisinde farklılık olması için byte tipini kullandık.

Şimdi sırada hazırladığımız bu indexer'ları kullanmak var. Konu üzerinde düşünmenizi sağlamak için öncelikle üzerinde çalıştığımız uygulamadaki diğer sınıfta bulunan Main() metodunun eksik halini verdik. Main() metodunun bu hali yukarıda verdiğimizden pek farklı değildir. Önce "Yayincilar" sınıfının örneği alınıp sonra kendisi için indexer hazırlanan alanın elemanlarına bilgi aktarılıp bu elemanların içerikleri konsola yazılmaktadır.

```
public static void Main()
{
 Yayincilar yay = new Yayincilar(3);
 yay[0] = "Seçkin Yayıncılık";
 yay[1] = "Alfa Yayınları";
 yay[2] = "Dikeyeksen Yayın Dağıtım";
 Console.WriteLine(yay[0]);
 Console.WriteLine(yay[1]);
 Console.WriteLine(yay[2]);
}
```

Burada örneğini aldığımız sınıfta (Yayincilar) yalnızca bir indexer olsaydı üzerinde tekrar konuşmak gerekmezdi. Dikkat ederseniz bu kodda aktarılan bilgilerinin sınıfın indekslenen hangi alanın elemanlarına aktarılacağı konusunda bir işaret yoktur. Tam bu noktada bu kitabın değişkenlerin anlatıldığı bölümde üzerinde konuşulan bir ayrıntıdan söz edeceğiz. Bu ayrıntıyı şu satır üzerinde konuşarak hatırlatacağız.

```
yay[0] = "Seçkin Yayıncılık";
```

Sizce bu satırda köşeli parantezin içine yazılan sıfırın (0) tipi nedir? Değişkenler adlı bölümden hatırlayacağınız gibi bu değerin tipi int olarak kabul edilir. Çünkü C#'ın böyle bir varsayımı vardır. Konudaki diğer ayrıntıları ortaya çıkarmak için "Yayincilar" sınıfında kendisi için indexer hazırladığımız 2. alanın elemanlarına bilgi aktarmak için Main() metoduna ekleme yaptık.

```
public static void Main()
{
 Yayincilar yay = new Yayincilar(3);
 yay[0] = "Seçkin Yayıncılık";
 yay[1] = "Alfa Yayınları";
 yay[2] = "Dikeyeksen Yayın Dağıtım";
 yay[0] = "Hukuk";
 yay[1] = "Teknik";
 yay[2] = "Bilişim";
```

```
 Console.WriteLine(yay[0]);
 Console.WriteLine(yay[1]);
 Console.WriteLine(yay[2]);
}
```

Tahmin edeceğiniz gibi bu şekilde düzenlenen Main() metodu hataya neden olmasa bile beklenen sonucu vermez. Çünkü burada örneğini aldığımız nesneye köşeli parantez içinde verdiğimiz değerin int tipinde olduğu varsayılır ve ikinci indexer aracılığıyla sınıftaki dahili koleksiyon olarak kullandığımız 2. alana bilgi aktarılmaz. Bu sorunu gidermek için köşeli parantezlerin içindeki sayısal değerin tipi işaret edilmelidir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
public static void Main()
{
 Yayincilar yay = new Yayincilar(3);
 yay[0] = "Seçkin Yayıncılık";
 yay[1] = "Alfa Yayınları";
 yay[2] = "Dikeyeksen Yayın Dağıtım";
 yay[(byte)0] = "Hukuk";
 yay[(byte)1] = "Teknik";
 yay[(byte)2] = "Bilişim";
 Console.WriteLine(yay[0]);
 Console.WriteLine(yay[1]);
 Console.WriteLine(yay[2]);
 Console.WriteLine(yay[(byte)0]);
 Console.WriteLine(yay[(byte)1]);
 Console.WriteLine(yay[(byte)2]);
}
```

Şimdi benzer bir örnek daha yapacağız. Bu amaçla forma 5 TextBox yerleştirdik. Kullanıcı çalışma anında TextBox'lara ilgili basketbol takımının ilk beşinde yer alan oyuncuların adlarını yazıp "Ekle" düğmesini tıklayacak. Visual Studio ile hazırladığımız projedeki 2. class'ı aşağıda verdik.

```
public partial class Takim
{
 ArrayList ilk_5 = new ArrayList();
 public object this[int indeks]
 {
  get
   {
    return ilk_5[indeks];
   }
  set
  {
   ilk_5.Add(value);
  }
  }
}
```

Çalışma anında formdaki TextBox'lara bilgi girilip "ekle" düğmesi tıklandığında önce "Takim" adını verdiğimiz class'ın örneğini hazırlayıp ardından girilen bilgilerin ArrayList nesnesine aktarılmasını sağlayacağız.

```
public partial class Form1 : Form
{
 public Takim Takim_nesne;
 private void ekle_Click(object sender, EventArgs e)
 {
  Takim_nesne = new Takim();
  Takim_nesne[0] = textBox1.Text;
  Takim_nesne[1] = textBox2.Text;
  Takim_nesne[2] = textBox3.Text;
  Takim_nesne[3] = textBox4.Text;
  Takim_nesne[4] = textBox5.Text;
  for(int i=0; i<5; i++)
   listBox1.Items.Add(Takim_nesne[i]);
 }
}
```

Bu sınıfın örneği alındığında bir **ArrayList** nesnesi hazırlanacaktır. "Takim" sınıfından yola çıkılarak hazırlanan nesnenin elemanlarına bilgi aktarılan koda şöyle bir satır eklenirse ne olur?

```
Takim_nesne[2] = textBox4.Text;
```

Bu satır sayesinde TextBox'ın içeriği ArrayList nesnesine 6. eleman olarak eklenir. Bunun önüne geçmek için kontrol yapmak gerekir. Yani yukarıda verilen koda göre ArrayList nesnesinin 3. elemanı mevcut olduğu için söz konusu TextBox'ın içeriğini ArrayList nesnesinin sonuna yeni bir eleman olarak eklemek yerine mevut elemanın içeriği değiştirilmelidir. Bu amaçla indeksleyicinin **set** kısmını aşağıdaki gibi değiştirdik.

```
public object this[int indeks]
{
 get
  {
   return ilk_5[indeks];
  }
 set
 {
  if (indeks >0 && ilk_5.Count > indeks)
    ilk_5[indeks] = value;
  else
   ilk_5.Add(value);
  }
}
```

Benzer kontrol indeksleyicinin **get** metodunda yapılabilir. Verilen örnekte ArrayList nesnesinin elemanlarını forma yerleştirdiğimiz ListBox'a aktardık. İlk sınıftaki for döngüsünü aşağıdaki gibi düzenleseydik hata meydana gelirdi.

```
for(int i=0; i<6; i++)
  listBox1.Items.Add(Takim_nesne[i]);
```

ArrayList nesnesi 5 elemana sahip olmasına rağmen 6. elemanın içeriği okunup ListBox'a aktarılmak istenmektedir. ArrayList'te 6. eleman olmadığı için hata meydana gelir. Bunun önüne geçmek için indexleyicinin **get** bloğunu aşağıdaki gibi düzenledik.

```
public partial class Takim
{
 ArrayList ilk_5 = new ArrayList();
 public object this[int indeks]
  {
   get
    {
     if ( indeks >=0 && ilk_5.Count > indeks)
       return ilk_5[indeks];
     else
     {
      MessageBox.Show((1+indeks).ToString() +". eleman yok");
      return "Boş";
     }
    }
   set
    {
     if (indeks >0 && ilk_5.Count > indeks)
       ilk_5[indeks]= value;
     else
      ilk_5.Add(value);
    }
  }
}
```

17. BÖLÜM

OBJECT SINIFI

C# programlama dili anlatılırken **Object** sınıfına sık sık gönderme yapılmak zorunda kalındığı için bu sınıf hakkında bilgi sahibi olmak gerekmektedir. Bu bölüm bir bakıma nesneye yönelik programcılık kavramları hakkında bildiklerinizi test etmeye yaramaktadır.

Object sınıfı C# kodu içinde sıklıkla kullanıldığı için **object** anahtar sözcüğü ile temsil edilmektedir. C#'ta hazırlanan her sınıf doğrudan veya dolaylı olarak Object sınıfından türetilmiştir. Herhangi bir sınıfı kodlarken kalıtım imkanından yararlanmamış olsak bile derleyici yine de o sınıfın Object'ten türetildiğini varsayar. Object tipinde nesne hazırlamak için new anahtar kelimesini kullanmak zorunda değilsiniz. Çünkü "object" ile değişken tanımlanırken gerçekte Object tipinde nesne hazırlanıyor. Bu sınıfın orijinal yapısının bir kısmı aşağıdadır.

```
namespace System
{
 public class Object
 {
  protected virtual void Finalize()
     {
     }
  public virtual int GetHashCode()
     {
     }
  public virtual bool Equals(System.Object obj)
     {
     }
  public virtual string ToString()
     {
     }
  public static bool Equals(System.Object objA, System.Object objB)
     {
     }
  public static bool ReferenceEquals(System.Object objA,System.Object objB)
     {
     }
  public Type GetType()
     {
     }
}
```

Bu sınıfın private oldukları için kullanamadığımız **FieldSetter**(), **FeildGetter**(), **InternalGetType**(), **GetFieldInfo**() ve **FastGetExisting**() metotlarını burada göstermedik. Bildiğiniz gibi **static** metotları hazırlık yapmadan istediğiniz yerde kullanabilirsiniz. Bu nedenle bu listede gösterilen ReferenceEquals() ve Equals() metodu nesne üzerinden değil direk Object sınıfı üzerinden kullanılabilmektedir.

En başta tekrar belirtelim, ister sadece .NET Framework ile gelen hazır sınıfları kullanın ister kendi özel sınıflarınız oluşturup DLL haline getirip diğer uygulamalarınızda kullanın: Kaçınılmaz olarak Object sınıfı kaynaklı metotlar karşınıza çıkacaktır. Bu nedenle bu bölümde yazılanları önemsemenizi öneriyoruz.

Bu yapıdaki bazı metotların virtual olduğunu fark etmişsiniz. Bu demektir ki kendi sınıflarınızda virtual olarak işaret edilen bu metotları yeniden yazmanıza bir engel yoktur. Konu Object sınıfı ve bu sınıftan miras yoluyla gelen metotları yeniden yazmak olduğunda ilk akla gelen **ToString**() metodudur.

İlk olarak **Equals**() metodunu kullanıp anlatmayı deneyeceğiz. Equals() metodu object veya başka bir tipteki 2 nesne veya değişkeni parametre olarak alıp bunların birbirine eşit olup olmadıklarını araştırmaktadır.

Equals() metodundan söz etmeden önce Object sınıfının metotları kullanılarak mevcut nesnelerin birbirine eşitliği araştırılırken yararlanılan 2 farklı modelin olduğunu belirtmek istiyoruz. Bunlar;

- Referansa göre yani nesnenin kimliğine göre yapılan karşılaştırma,
- Nesnenin hem kimliğine hem de içeriğine göre yapılan karşılaştırma.

İleriki sayfalarda hangi metodun hangi modele göre işlev gördüğünü belirteceğiz. Equals() metodundan söz etmek amacıyla **new** anahtar kelimesinden yararlanıp Object sınıfının 2 örneğini hazırladık. Object sınıfının yapıcı metodu standart bir nesne hazırlamaktadır. Hemen hatırlatalım: Object sınıfına ait **Equals** metodunun 2 farklı sürümü bulunmaktadır.

```
private void Form1_Click(object sender, EventArgs e)
{
 Object nesne1 = new Object();
 Object nesne2 = new Object();
}
```

Bu şekilde **Object** sınıfının 2 örneği hazırlanıp bu nesneler statik **Equals**() metodu ile karşılaştırılırsa nesnelerin birbirine eşit olmadıkları rapor edilir. Yani aşağıda verilen kodda Equals() metodu "nesne1" ile nesne2'in birbirine eşit olmadıklarına hükmedir.

```
private void Form1_Click(object sender, EventArgs e)
{
 Object nesne1 = new Object();
 Object nesne2 = new Object();
 bool sonuc = Object.Equals(nesne1, nesne2);
 if (sonuc == true)
   MessageBox.Show("Bu nesneler birbirine eşit");
}
```

Object tipindeki bu 2 nesneyi **Equals**() metodu ile birbiri ile karşılaştırmadan önce aşağıda yapıldığı gibi aynı değeri aktarmış olsaydık Equals() metodu bu kez geriye true gönderirdi. Bu kodda kullanılan Object tipindeki bu 2 değişkene sayısal bilgi yerine **null** aktarmış olsaydık bile Equals() metodu geriye true gönderirdi.

```
private void Form1_Click(object sender, EventArgs e)
{
 Object nesne1 = new Object();
 Object nesne2 = new Object();
```

```
    nesne1 = 1234;
    nesne2 = 1234;
    bool sonuc = Object.Equals(nesne1, nesne2);
    if (sonuc == true)
      MessageBox.Show("Bu nesneler birbirine eşit");
    else
      MessageBox.Show("Bu nesneler birbirine eşit değil");
  }
```

Bu sonuca bakıp Equals() metodunun karşılaştırmayı yalnızca referans bazında yapmadığına karar verebilirsiniz. Şimdi Object sınıfının **Equals**() adındaki statik metoduna biraz yakından bakacağız. Bu metot kendisine parametre olarak verilen 2 nesne veya değişkenin (bu değişkenlerin object tipinde olması şart değildir) referanslarının aynı olup olmadığına bakıyor. Her 2 parametre veya nesne de aynı referansı işaret ediyorsa geriye true göndermektedir.

Equals() metoduna parametre olarak verilen nesneler veya değişkenler aynı adreste saklanmıyorsa yani referansları farklı ise bu kez içeriklerinin aynı olup olmadığına bakmaktadır. Yukarıda ilk yaptığımız karşılaştırma "nesne1" ve "nesne2" adını verdiğimiz nesnelerin referansları zaten farklı olur. Bu nesnelerin içerikleri boş iken Equals() metodu geriye false göndermişti. İkinci karşılaştırmada bu kez içerikleri aynı olduğu için geriye true gönderildi.

Yukarıda Equals() metoduna object tipindeki bilgileri direk parametre olarak verdik. Bu metot istenen tipteki bilgi veya nesneleri karşılaştırabilir. Aşağıda verilen örmekte ise Int32 tipindeki 2 değişkeni yani **System.Int32** yapısından yola çıkılarak hazırlanmış 2 nesneyi Equals() metoduna parametre olarak verip birbiri ile karşılaştırdık.

```
private void Form1_Click(object sender, EventArgs e)
{
 System.Int32 sayi1 = new System.Int32();
 System.Int32 sayi2 = new System.Int32();
 sayi1 = 100;
 sayi2 = 100;
 bool sonuc = Object.Equals(sayi1, sayi2);
 if (sonuc == true)
   MessageBox.Show("Bu nesneler birbirine eşit");
}
```

Equals() metodu System.Int32 yapısından yola çıkılarak oluşturulmuş nesneleri karşılaştırırken onların referansları yerine içeriklerine baktığı için burada yapılan karşılaştırma doğru değerini verir. Burada Int32 yapısının örneklerini **new** anahtar kelimesi ile aldıktan sonra bilgi aktardık. Bu kodun değişkenlere bilgi aktarılan satırlarını kaldırırsanız değişen bir şey olmaz ve yine değişkenlerin birbirine eşit olduğu rapor edilir. Çünkü değer tipindeki değişkenler **new** anahtar kelimesi ile hazırlandığında varsayılan içerik otomatik olarak aktarılmaktadır.

"Değişkenler" adlı bölümden hatırlayabileceğiniz gibi double ve int tipindeki değişkenler new anahtar kelimesi ile tanımlanıp değer aktarılmadan kullanılmak istenirse hata meydana gelmemektedir. Konunun iyice anlaşılmasını sağlamak için aşağıda verilen örnekte TextBox sınıfından yola çıkılarak hazırlanmış 2 nesneyi karşılaştırdık.

```
private void Form1_Click(object sender, EventArgs e)
{
  System.Windows.Forms.TextBox text1;
  System.Windows.Forms.TextBox text2;
  text1 = new TextBox();
  text2 = new TextBox();
  bool sonuc = Object.Equals(text1, text2);
  if (sonuc == true)
    MessageBox.Show("Bu nesneler birbirine eşit");
  else
    MessageBox.Show("Bu nesneler birbirine eşit değil");
}
```

Bu örnekte TextBox'lara bilgi aktarmadan karşılaştırdık. Bu sırada her 2 TextBox'ın özellikleri aynı olsa veya ikisinin de içeriği boş olsa bile bu karşılaştırmada nesnelerin birbirine eşit olmadığına hükmedilir. Çünkü Equals() metodu referans tipleri karşılaştırırken nesnelerin referanslarına yani bellekteki adreslerine bakmaktadır.

Bu şartlarda TextBox sınıfının "text1" ve "text2" adını verdiğimiz her iki nesne her şeyi ile aynı özelliklere sahip olsalar bile referansları farklı olduğu için Equals() metodu bu 2 nesnenin eşit olmadığını rapor eder. Ancak aşağıda yapıldığı gibi bu TextBox nesneleri birbirine eşitlenip bellekte aynı adresi kullanmaları sağlandığında Equals() metodu geriye **true** gönderir ve parametre olarak aldığı nesnelerin birbirine eşit olduğuna hükmeder.

```
TextBox text1 = new TextBox();
TextBox text2 = new TextBox();
text2 = text1;
bool sonuc = Object.Equals(text1, text2);
if (sonuc == true)
  MessageBox.Show("Bu nesneler birbirine eşit");
```

Tekrar etmek gerekirse; Equals() metodunu çağıran ve parametre olarak verilen nesne **Referans Type** olmasaydı içeriklerinin aynı olup olmadığına bakılırdı. Bu nedenle aşağıda verilen kodda Equals() metodu geriye true gönderir.

```
private void Form1_Click(object sender, EventArgs e)
{
 int sayi1 = 100;
 int sayi2 = 100;
 bool sonuc = sayi1.Equals(sayi2);
 MessageBox.Show(sonuc.ToString());
}
```

Bu sayfaya kadar Object sınıfının Equals() metodunu anlatırken sürekli .NET Framework ile gelen hazır tip ve sınıfları kullandık. Equals() metodunun kendi hazırladığınız tiplere davranışını göstermek için aşağıdaki gibi bir class hazırladık.

```
public class Takim
{
 public string takim_adi;
 public string baskan;
```

```
 public Takim(string par1, string par2)
 {
  takim_adi = par1;
  baskan = par2;
 }
}
```

Bu şekilde 2 alana ve bir yapıcı metoda sahip sınıf hazırladıktan sonra aşağıdaki gibi kod yazıp bu sınıfın 2 örneğini alıp Equals() metodu ile karşılaştırdık. Beklenilenin aksine Object sınıfının Equals() metodu bu 2 nesnenin birbirinden farklı olduğunu rapor eder.

```
private void Form1_Click(object sender, EventArgs e)
{
 Takim t1 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 Takim t2 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 if (Object.Equals(t1, t2))
   MessageBox.Show("Bu nesneler birbiri ile aynı");
 else
   MessageBox.Show("Bu nesneler birbirinden farklı");
}
```

Yukarıdaki sayfalarda .NET Framework ile değişik tipteki nesneleri karşılaştırırken, nesneler aynı içeriğe sahip olduklarında Equals() metodu nesnelerin birbirine eşit olduğunu söylüyordu. "t1" ve "t2" adı verilen bu 2 nesnenin birbirinden farklı olarak görülmesinin nedeni Equals() metodunu "Takim" adını verdiğimiz sınıf için yeniden yazmamış olmamızdır. "Takim" adlı sınıfın Object sınıfından gelen Equals() metodunu override edip nasıl yeniden yazdığımızı aşağıda görebilirsiniz. Bu hazırlıktan sonra Equals() metodu Compare işlemi yapıp kendisine parametre olarak verilen nesnedeki alanların içeriklerine göre karar verir.

```
public class Takim
{
 public string takim_adi;
 public string baskan;
 public Takim(string par1, string par2)
 {
  takim_adi = par1;
  baskan = par2;
 }
 public override bool Equals(object obj)
 {
  Takim T = (Takim)obj;
  if ((T.takim_adi ==this.takim_adi) && (T.baskan == this.baskan))
    return true;
  else
    return false;
 }
}
```

Bu hazırlıktan sonra yukarıda verilen "Object.Equals(T1, T2)" satırında Equals() metodu geriye true gönderir. Şimdi benzer ayarlamayı TextBox sınıfının mirasçısı bir sınıfta yapacağız. Yapacağımız ayarlama sonucu birbirleri ile karşılaştırılan TextBox nesnelerinin Text özellikleri aynı ise bir birine eşit oldukları kabul edilecektir.

```
public class Edit : TextBox
{
 public override bool Equals(object obj)
 {
  Edit Edit1 = (Edit)obj;
  if (Edit1.Text == this.Text)
    return true;
  else
    return false;
 }
}
```

Bu kodu yazmamızdaki amaç TextBox ve başka sınıfların Object sınıfından miras aldıkları metotların nasıl override edildiklerine dikkatinizi çekmektedir. Bu metodun nasıl kullanıldığını göstermek için aşağıda verilen kodu yazdık.

```
private void button1_Click(object sender, EventArgs e)
{
 Edit Text1 = new Edit();
 Edit Text2 = new Edit();
 bool sonuc = Object.Equals(Text1, Text2);
 if (sonuc == true)
   MessageBox.Show("Bu nesneler birbirine eşit");
 else
   MessageBox.Show("Bu nesneler birbirine eşit değil");
}
```

Şimdiye kadar Object sınıfının **Equals**() metodunu kullanırken karşılaştırmak istediğimiz 2 nesneyi bu metoda parametre olarak verdik. Şimdiye kadar kullandığımız Equals() metodu Object sınıfının static bir metodu olduğu için direk kullanıyorduk.

Ancak isterseniz Object sınıfının static olmayan ve nesne üzerinden çağrılan Equals() metodunu aşağıdaki gibi kullanabilirsiniz. Burada Equals() metodunu çağıran nesne ile metoda parametre olarak verilen nesne ile aynı referansa veya içeriğe sahipse geriye true gönderilir. Bu örnekte bu nesnelerin referansları farklı olduğu için geriye false gönderilir.

```
TextBox Text1 = new TextBox();
TextBox Text2 = new TextBox();
bool sonuc = Text1.Equals(Text2);
MessageBox.Show(sonuc.ToString());
```

ReferanceEquals Metodu

Bu bölümün ilk sayfasında verilen Object sınıfının yapısını incelediğinizde Equals() metodundan başka **ReferanceEquals**() adında statik bir metodun olduğunu görürsünüz. Bu metot kendisine parametre olarak verilen 2 nesne veya değişkenin referanslarının yani bellekteki adreslerinin aynı olup olmadığını araştırmaktadır.

Yukarıda hakkında bilgi verilen Equals() metodu konu **Değer Tipleri** olduğunda değişken veya nesnenin içeriğiyle ilgileniyordu. ReferenceEquals() metodu ise kendisine parametre olarak verilen tiplerin referanslarının aynı olup olmadığını araştırmaktadır. Bu nedenle **ReferenceEquals**() metodu aşağıdaki gibi kullanıldığında geriye false gönderir.

```
private void Form1_Click(object sender, EventArgs e)
{
 int sayi1 = 100;
 int sayi2 = 100;
 if (Object.ReferenceEquals(sayi1, sayi2))
   MessageBox.Show("Bu değişkenlerin referansları aynı");
 else
   MessageBox.Show("Bu değişkenlerin referansları farklıdır");
}
```

Yukarıda yazılanlardan bildiğiniz gibi Equals() metoduna aynı içeriğe sahip 2 string değişken parametre olarak verildiğinde geriye true göndermektedir. Ne var ki konu **ReferenceEquals**() metodu olduğunda beklenilenden farklı bir sonuç alınmaktadır. Yani aynı içeriğe sahip 2 string değişkeni aşağıda yapıldığı gibi **ReferenceEquals**() metoduna parametre olarak verirseniz bu 2 değişkenin referanslarının birbirine eşit olduğu rapor edilir.

```
string Takim1 ="Fener";
string Takim2 = "Fener";
if (String.ReferenceEquals(Takim1, Takim2))
  MessageBox.Show("Bu değişkenler birbirine eşit");
else
  MessageBox.Show("Bu değişkenler birbirine eşit değil");
```

Bu sonucun açıklaması şudur:.NET Framework ile gelen String sınıfında **ReferenceEquals** metodu override edilip aynı içeriğe sahip string değişkenlerin birbirine eşit olması sağlanmaktadır. Benzer durum object tipindeki değişkenler için de geçerlidir. Bu örnekte değişkenlerin tipi object yapılırsa **ReferenceEquals**() metodu geriye yine true gönderir.

```
object Takim1 = "Fener";
object Takim2 = "Fener";
if (Object.ReferenceEquals(Takim1, Takim2))
  MessageBox.Show("Bu değişkenler birbirine eşit");
else
  MessageBox.Show("Bu değişkenler birbirine eşit değil");
```

Tekrar etmek gerekirse; string ve object değişkenler hariç **ReferenceEquals**() metodu kendisine parametre olarak verilen değişken veya nesneleri karşılaştırırken referanslarının aynı olup olmadığına bakıyor. Bu nedenle aşağıdaki gibi 2 TextBox nesnesi hazırlayıp bunları **ReferenceEquals**() ile karşılaştırırsanız nesnelerin birbirine eşit olmadıkları rapor edilir.

```
private void Form1_Click(object sender, EventArgs e)
{
 TextBox Nesne1 = new TextBox();
 TextBox Nesne2 = new TextBox();
 if (Object.Equals(Nesne1, Nesne2))
   MessageBox.Show("Bu nesneler birbirine eşit");
 else
   MessageBox.Show("Bu nesneler birbirine eşit değil");
}
```

ReferenceEquals() metodu benzer davranışı kendi hazırladığınız sınıfların örneklerini karşılaştırırken de gösterir. Örneğin aşağıdaki gibi bir sınıf hazırlayıp bu sınıfın aynı içeriği sahip 2 kopyası hazırlanıp birbiri ile karşılaştırıldığında birbirine eşit olmadıkları rapor edilir.

```
private void Form1_Click(object sender, EventArgs e)
{
 Takim Takim1 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 Takim Takim2 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 if (Object.ReferenceEquals(Takim1, Takim2))
   MessageBox.Show("Bu nesneler eşit");
}
public class Takim
{
  public string takim_adi;
  public string baskan;
  public Takim(string par1, string par2)
  {
   takim_adi = par1;
   baskan = par2;
  }
}
```

ToString() ve GetType() Metotları

Bütün sınıflar Object sınıfından türetildiği için ToString() metodu bütün yapılar ve sınıflar tarafından desteklenmektedir. Örneğin aşağıda verilen satırda kullanılan ToString() metodu, Int32 yapısının (struct) **Object** sınıfından miras aldığı metottur.

```
Int32 Sayi = new Int32();  // veya int Sayi;
Sayi = 1234;
textBox1.Text = Sayi.ToString();
```

Her ne kadar Int32 yapısı ToString() metodunu Object sınıfından miras almış olsa bile bu metot ezilip (override) yeniden yazılmıştır. Herhangi bir müdahalede bulunmadığınız sürece **ToString**() metodu kendi sınıfınızın tam adını verir. Bu konuda bilgi vermek için yukarıda verilen "Takim" sınıfını aşağıdaki gibi düzenledik.

```
public class Takim
{
 public string takim_adi;
 public int kurulus;
 public Takim(string par1, int par2)
  {
   takim_adi = par1;
   kurulus = par2;
  }
}
```

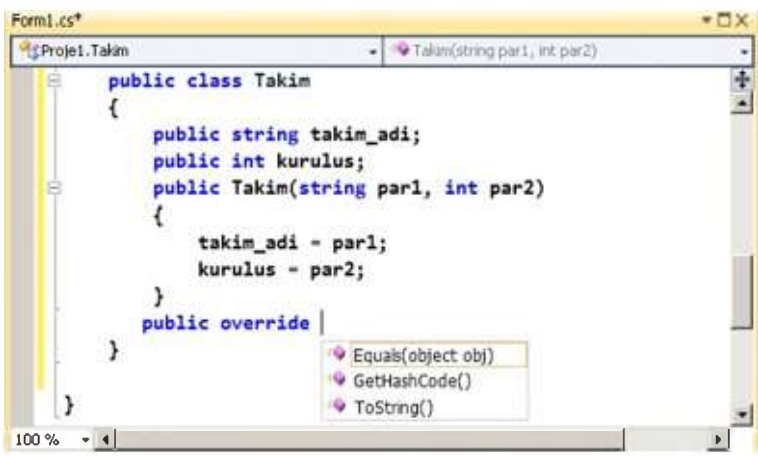
Bu sınıfın örneğini alıp nesneye ToString() metodunu aşağıdaki gibi uygularsanız formun başlığına önce class'ın dahil olduğu namespace'in, ardından class'ın adı yazılır. Başka bir deyişle "Takim" tipindeki "takim1" nesnesine **ToString**() metodu uygulandığında sınıfın adı elde edilmektedir.

```
Takim takim1 = new Takim("Fenerbahçe", 1907);
this.Text = takim1.ToString();
```

Hal bu ki Int32 yapısının örneğine (değişken) ToString() metodu uygulandığı zaman değişkenin içerinin string halini elde ediyorduk. Kendi hazırladığınız sınıfta ToString() metodundan farklı işlevler bekliyorsanız Object sınıfından miras alınan bu metodu yeniden yazmalısınız. ToString() metodunu aşağıdaki gibi kullansaydık formun başlığına 1907 yazılırdı. Çünkü "Takim" sınıfının "kurulus" alanı Int32 tipinde olduğu için burada kullanılan ToString() metodu Int32 yapısına ait metottur.

```
private void Form1_Click(object sender, EventArgs e)
{
 Takim takim1 = new Takim("Fenerbahçe", 1907);
 this.Text = takim1.kurulus.ToString();
}
```

Şimdi Object sınıfından gelen ToString() metodunu "Takim" adlı sınıfta override edip yani yeniden yazıp farklı bir işleve sahip olmasını sağlayacağız. Ancak ToString() metodunu ezip yeniden yazmadan önce yukarıda "takim1" nesnesine uyguladığımız ToString() metodunun neden namespace adına kendi hazırladığımız sınıfın adını ekleyip geriye gönderdiğini açıklayacağız. "Takim" sınıfının sınırları içinde **public override** yazıp boşluk tuşuna basınca Visual Studio aşağıdaki gibi 3 metot listeledi.



Çünkü bu sırada "Takim" adını verdiğimiz sınıf dahilinde override edilebilinecek 3 metot vardı. Bu 3 metodun da Object sınıfından miras yoluyla alınan metotlar olduğunu biliyorsunuz. Listelenen bu 3 metottan **ToString**'i seçmeniz halinde Visual Studio otomatik olarak ToString() metodunun Object sınıfından alınan orijinal halini hazırlar.

```
public class Takim
{
 public string takim_adi;
 public int kurulus;
 public Takim(string par1, int par2)
 {
  takim_adi = par1;
  kurulus = par2;
 }
```

```
    public override string ToString()
     {
      return base.ToString();
     }
}
```

Gördüğünüz gibi ToString() metodunun orijinal halinde **base** (taban) sınıfın adı geriye gönderilmektedir. Şimdi Visual Studio tarafından hazırlanan bu ToString() metodunda öyle bir değişiklik yapacağız ki geriye "Takim" sınıfından yola çıkılarak hazırlanan nesnenin "takim_adi" ve "kurulus" alanların birleştirilmiş halini geriye gönderecektir.

```
public class Takim
{
 public string takim_adi;
 public int kurulus;
 public Takim(string par1, int par2)
  {
   takim_adi = par1;
   kurulus = par2;
  }
 public override string ToString()
  {
   return "Takim_adi : " + this.takim_adi +
     " " + "\n" + "Kuruluş Yılı : " + this.kurulus.ToString();
  }
}
```

Bu şekilde düzenlenen ToString() metodunun pratik değeri yoktur. Çünkü geriye gönderilecek bilginin formatını seçerken MessageBox'ın **Show**() metodunu gözettik. Yukarıda Form1_Click() metoduna yazdığımız kod işletilirse aşağıdaki gibi bir sonuç alınır.



Şimdi yukarıda **Object** sınıfından yola çıkarak hazırladığımız nesnenin türünü Object sınıfının **GetType**() metodu ile öğreneceğiz. Bütün sınıflar Object sınıfının mirasçısı olduğu için bütün sınıflar **GetType**() metoduna sahiptir.

```
private void Form1_Click(object sender, EventArgs e)
{
 Object nesne = new Object();
 object Tip = nesne.GetType();
 textBox1.Text = Tip.ToString();
}
```

Bu kodda object tipinde bir değişken tanımlayıp GetType() metodunun geriye gönderdiği bilgiyi bu değişkene aktardık. Bu kod işletildiğinde TextBox'a **System.Object** yazılır. **GetType**() metodunu aşağıdaki gibi kullanmış olsaydık TextBox'a **System.Byte** yazılırdı.

```
private void Form1_Click(object sender, EventArgs e)
{
 byte Sayi = 128;
 object Tip = Sayi.GetType();
 textBox1.Text = Tip.ToString();
}
```

Nesnenin tipini veya oluşturulduğu sınıfı öğrenmek istiyorsanız **typeof** operatöründen yararlanabilirsiniz. Bu operatör parametre olarak hakkında bilgi alınmak istenen tipi alıp geriye **System.Type** tipinde bilgi göndermektedir. Bu sınıfın ve typeof() operatörünün nasıl kullanıldığını anlatmak için aşağıda verilen örneği hazırladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 matematik nesne = new matematik();
 if (nesne.GetType() == typeof(matematik))
   MessageBox.Show("Bu nesne matematik sınıfından türetilmiş");
}
```

Object sınıfının yukarıdaki sayfalarda verilen yapısı incelenirse geriye **int** tipinde bilgi gönderen **GetHashCode()** adında bir metoda sahip olduğu görülür. Bu metot, nesnelerin **Hash** kodu öğrenilmek istendiği zaman kullanılmaktadır.

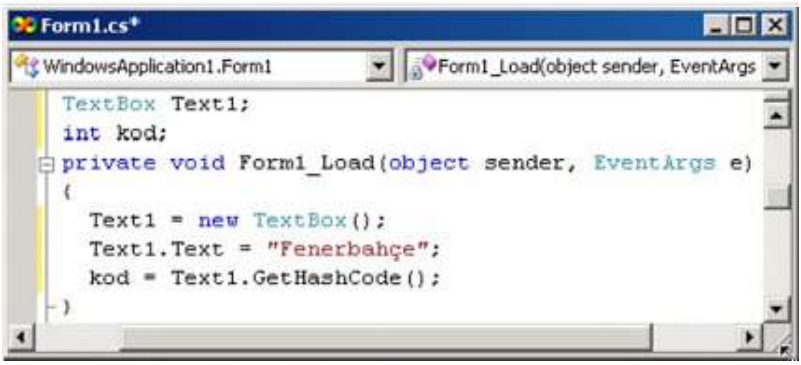
.NET uyumlu uygulamalarda hazırlanan her nesne veya tanımlanan her değişken **HashCode** adı verilen sayısal bir koda sahiptir. Bu kod öğrenilerek söz konusu nesneye kolayca ulaşmak mümkün olmaktadır. Bu metodun nasıl kullanıldığını anlatmak için forma 2 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 Button Dugme1 = new Button();
 Button Dugme2 = new Button();
 int kod1 = Dugme1.GetHashCode();
 int kod2 = Dugme2.GetHashCode();
 textBox1.Text = kod1.ToString();
 textBox2.Text = kod2.ToString();
}
```

Bu kod işletildiğinde aşağıdaki gibi bir sonuç alınır. Aynı oturumda bile aynı kodu tekrar işlettiğinizde farklı hash kodları elde edilir. Çünkü kod tekrar işletildiğinde nesne yeniden hazırlanır ve belleğin farklı bir yerinde kendisine yer bulup farklı bir hash koduna sahip olur.



Konuya başka bir açıdan bakabilmek için **Windows Forms** uygulamasında "Form1" sınıfının her yerinde yaşayacak TextBox tipinde değişken tanımladık. Ardından bu sınıfın Form1_Load() metodunda **TextBox** nesnesi hazırladık.



Proje çalıştırıldığında bu satırlar sayesinde TextBox nesnesi hazırlanıp HashCode'u elde edilir. Programın ileri aşamalarında bu nesneye gerek duyduğunuzda HashCode'dan yararlanabilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 if (kod == this.Text1.GetHashCode())
   MessageBox.Show("ilgilendiğim nesne bu");
}
```

GetHashCode() metodunun diğer işlevlerinden söz etmek için aşağıdaki gibi aynı içeriğe sahip 2 string değişken tanımlayıp hash kodlarını öğrendik.

```
private void Form1_Click(object sender, EventArgs e)
{
 string str1 = "Fenerbahçe";
 string str2 = "Fenerbahçe";
 int kod1 = str1.GetHashCode();
 int kod2 = str2.GetHashCode();
 textBox1.Text = kod1.ToString();
 textBox2.Text = kod2.ToString();
}
```

Burada tanımladığımız string değişkenler aynı içeriğe sahip oldukları için aynı hash koduna sahip olurlar. Şimdi ise yukarıdaki sayfalarda hazırlayıp "Takim" adını verdiğimiz sınıfın aynı içeriğe sahip 2 örneğini alıp hash kodlarını öğreneceğiz.

```
private void Form1_Click(object sender, EventArgs e)
{
 Takim Takim1 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 Takim Takim2 = new Takim("Fenerbahçe", "Aziz Yıldırım");
 int kod1 = Takim1.GetHashCode();
 int kod2 = Takim2.GetHashCode();
 textBox1.Text = kod1.ToString();
 textBox2.Text = kod2.ToString();
}
```

Bu kod işletildiğinde bu iki nesnenin farklı hash kodlarına sahip oldukları görülür. "Takim" sınıfını struct olarak düzenlemiş olsaydık bu kez farklı bir sonuç alınırdı ve her iki nesne de aynı hash koduna sahip olurdu.

18. BÖLÜM

STRUCT HAZIRLAMAK ve KULLANMAK

C# konulu hangi kaynağa baksanız, konu struct'lar olduğunda anlatılanların çoğunluğunu class'lar ile struct'lar arasındaki farklar oluşturuyor. Bizce böyle bir anlatım struct'lar hakkında öğrenilenlerin kolay unutulmasına neden oluyor. Bu kitapta class'ları zaten şimdiye kadar anlattık. Şimdi sırada struct'lar var. Elbette class'lar hakkında ayrıntılı bilgi edinmiş birisine struct'lar anlatılırken özet yapmakla yetinmek gerekir. Örneğin struct'lar için yapıcı metotları anlatırken daha önce class'lar bağlamında yapıcı metotlar hakkında bilgi edindiğinizi varsayıp yapıcı metotları uzun uzun anlatmayacağız.

C# uygulamaları dahilinde Struct'ların nasıl hazırlanıp kullanıldığından söz etmeden önce önemli gördüğümüz bazı noktaları buraya yazacağız. Bu önbilgilerin ışığında struct'lara ne kadar ilgi duyacağınıza daha sağlıklı karar verebileceksiniz.

Struct'lar C ve eski Visual Basic'te bile vardı. Konuya C penceresinden bakılırsa şunlar söylenebilir: C ile uygulama geliştirmek isteyen programcı yapmak istediği işlemlere uygun fonksiyonları hazırlayıp kod dosyalarında topluyordu. Daha sonra gerek duyduğu anlarda bu fonksiyonları çağırıp işletiyordu. C programcısı ne zaman birden fazla bilgiyi bir araya getirip gruplamak ve dolayısıyla bu grup bilgi üzerinde topluca işlem yapmak istediğinde struct hazırlıyordu. Aşağıda C diliyle hazırlanmış 2 struct bulunmaktadır.

```
struct tarih {
  int gun;
  int ay;
  int yil; };
struct kisi {
  char ad[20];
  char soyad[20];
  struct tarih dogum; };
```

Burada "tarih" adı verilen struct'ta int tipinde 3 bilgi bulunmaktadır. Bundan şu sonucu çıkarabiliriz, C programcısı ne zaman int tipinde 3 bilgiyi bir araya getirip kullanmak istediğinde böyle bir struct hazırlamak durumunda kalıyordu. Burada "kisi" adını verdiğimiz struct'ta yine 3 veri vardır. Ancak bu verilerin birisi "tarih" tipindedir. C'de struct'lar bu kadardı, sadece birden fazla bilgiyi paketlemeye yarıyorlardı.

Bazı arkadaşlar kaşlarını çatıp şimdi C mi öğreneceğiz diye soru sorabilirler. Hayır, size C'yi anlatmaya çalışmayacağız. O zaman neden C'den söz ediyoruz? Çünkü bu kitapta şimdiye kadar class'larla ilgilendik. Şimdi ise diyoruz ki bir de struct'lar var. Mevcut kaynakların neredeyse tamamında struct'lar class'ların üzerinden anlatılmaktadır. Bu bölümde struct'ları class'larla karşılaştırmadan bağımsız olarak ele almaya çalıştık.

Bu kitabın bundan önce verilen bölümlerini okuyan birisi C#'ta her şeyin class'ların etrafında döndüğünü fark etmiş olmalıdır. Buna rağmen .NET Framework ile birlikte çok sayıda struct'ın (örneğin Int32, Int64) geldiğini biliyoruz. Bundan şu sonucu çıkarabiliriz: Programcıların struct'lara ihtiyaçları er veya geç olacaktır.

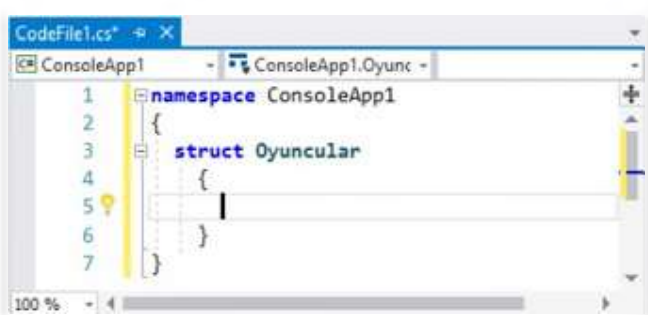
C# kodu içinde kullanılan tipler gruplara ayrılırken en önemli gruplama kriteri bellekte saklanılan yerdir. Şimdiye kadar değişik konular anlatılırken ifade edildiği üzere .NET programcılığı jargonunda .NET Framework tarafından kontrol edilen bellek 2 kısım halinde organize ediliyor: **Stack** ve **Heap**. Stack'te, bellekte kaplayacakları yer miktarı önceden belli olan tipler olurken, Heap'te ise ne kadar yer kaplayacakları önceden bilinmeyen tipler olmaktadır. Örneğin int, long, double tipler stack'te yer alması sağlanırken dizi değişkenler, stringler ile .NET Framework ile hazırlana gelen class'ların örnekleri heap'te tutulmaktadır. Programcılar, bir değişkenin veya nesnenin stack'te olmasıyla heap'te olmasını çok önemserler. Çünkü bellekteki yeri yazacakları kodu temelden etkilemektedir.

Struct'larla ilgili olarak bilmeniz gereken en önemli nokta, struct tipindeki nesnelerin belleğin stack adı verilen kısmında tutuluyor olmalarıdır. Bu tespiti yaptıktan sonra struct'ların hangi şartlarda hangi amaçlarda kullanıldığını maddeler halinde sıralayacağız.

- Yukarıda C için hazırlanan struct'ta olduğu gibi birden fazla veriyi struct adı altında paketlemek için,
- Bazı işlemleri yapmak için hazırlayacağınız bir veya birden fazla metodu struct olarak bir araya getirmek ve gerektiğinden bu metotları işletmek için,
- Hem veri hem de metotları içeren bir yapıya (burada geçen yapı, struct'ın Türkçe karşılığı değildir) ihtiyacınız olduğunda,
- Ne zaman class, ne zaman struct hazırlanması gerektiği tartışmaya açık bir konu olmakla birlikte struct'ın örneği bellekte fazla yer kaplamayacaksa tercih edilebilir.
- C# programlama dilinin birçok yönden benzerlik gösterdiği Java'da struct'ların olmadığını, Java'da her şeyin sınıflarla yapıldığını bilmekte fayda vardır.

Kurumsal bir firmada C# kodu yazmak durumunda kaldığınızda muhtemelen hangi durumlarda struct hazırlanması gerektiği önceden kurala bağlanmıştır. Benzer durum, bir ekip olarak C# kodu yazmaya başladığınızda hangi işlemlerin struct yazılarak yapılacağı konusunun konuşulup karar verilmesi yerinde olacaktır.

İçinde struct kodlamak istediğiniz CS uzantılı dosyayı hazırlarken Class ve Interface şablonlarından da yararlanabilirsiniz. Visual Studio'nun **Add New Item** diyalog kutusunda Code File şablonunu seçip varsayılan dosya adına bağlı kalınıp **Add** düğmesi tıklanırsa içeriği boş bir dosya hazırlanır. Hazırlamak istediğimiz struct'ın üzerinde çalıştığımız uygulamadaki tek namespace'e dahil etmek istediğimiz önce namespace bloğu hazırladık. Ardından namespace bloğu içinde **struct** anahtar kelimesini kullanarak struct hazırladık.



Bu şekilde hazırlanan struct'ta metotlara, yapıcı metotlara, sabitlere, özelliklere ve alanlara yer verebiliriz. Struct'lar başka struct'ın mirasçısı olamaz. Çünkü struct'lar kalıtımı desteklemiyor. Bundan şu sonucu çıkarabiliriz: Struct'lar çok kapsamlı ve çok amaçlı olmamalıdır. Çünkü sıfırdan hazırlanmaya başlanıp tamamlanmaları gerekiyor. Kalıtımı desteklemedikleri için çok biçimlilik de söz konusu olamıyor ve struct dahilinde yazılan kod başka struct'larda kullanılamıyor. C# derleyicisi struct içinde struct hazırlanmasını desteklediği için daha önce hazırlanan bir struct'ın kaynaklarından yeni hazırlanan struct'ta faydalanmak mümkündür.

Struct'ları class'ların içinde hazırlamak mümkündür. Bu imkandan şu şartlarda yararlanmanızı önerebiliriz: Üzerinde çalıştığınız sınıfta yapmak istediğiniz işlemlere bağlı olarak bir struct'a ihtiyaç duyduysanız ve başka yerde yararlanmayacaksanız o sınıfın içinde kodlayabilirsiniz. Ayrıca C# derleyicisi struct içinde class hazırlamanıza izin vermektedir.

Yukarıda "Oyuncular" adını verdiğimiz bu struct'ı hazırlarken erişim belirleyici anahtar kelimelerden herhangi birisini kullanmadık. Bu nedenle bu struct'ı üzerinde çalıştığımız uygulamadaki sınıf ve diğer structlarda kullanabiliriz.

Bu struct'ı farklı bir namespace'e üye yapsaydık bu şartlarda başka sınıf ve struct'larda kullanamazdık. Konu struct'lar olduğunda kullanabileceğimiz tek erişim belirleyici anahtar kelime **public**'tir. Hazırladığımız struct ayrı bir namespace'te ise public yapmak gerekir.

Şimdi ayrı bir kod dosyasında hazırlayıp "Oyuncular" adını verdiğimiz struct'ın 3 public alana sahip olmasını sağlayacağız. Struct'ta yer verilen alanlar istenen tipte olabilir. Doğrusu, bu alanları private yapıp her alan için bir özellik hazırlamaktır.

```
namespace ConsoleApp1
{
 struct Oyuncular
 {
   public string Ad;
   public double Boy;
   public string Takim;
 }
}
```

Struct'ta bu şekilde alan hazırlamak yerine özellik hazırlanabilir. Çünkü C# derleyicisi geri planda her özellik için bir alanı otomatik olarak hazırlamaktadır. Derleyicimin otomatik olarak alan hazırlama imkanından yararlanmak üzere struct'ımızı aşağıdaki gibi düzenleyebilirdik.

```
struct Oyuncular
{
 public string prAd { get; set; }
 public double prBoy { get; set; }
 public string prTakim { get; set; }
}
```

Bu özelliklerin get-set bloklarına bir şey yazılmadığı için struct'ın bu hali yukarıda verilenden farklı değildir. Bu şekilde alan yerine özellik hazırlayıp set-get bloklarında yer vermek, ileri bir tarihte bu set ve get bloklarının içine bir şeyler yazdığınızda, daha doğrusu hem alan hem

de özellik hazırlamaya ihtiyaç duyduğunuzda bu struct'ın kullanıldığı yerlerde revizyona gerek olmaz. Bu şekilde hazırlanan yapıyı kullanabilmek için struct'tan yola çıkıp nesne hazırlamak gerekir. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
 Oyuncular oyuncu1 = new Oyuncular();
}
```

Bu nesneyi Main() metodu içinde hazırladığımız için ancak bu metodun içinde kullanılabilir. Her ne kadar "Oyuncular" tipinde nesneyi C# uygulamalarının başlangıç noktası olan Main() metodu içinde tanımlamış olsak bile başka bir metodun içinde tanımlasaydık şimdi söyleyeceğimiz ayrıntı bağlamında değişen bir şey olmazdı: Kodun işletimi bu metodun sonuna ulaştığında burada "oyuncu1" adı verilen nesne bellekten tümden silinir.

Şimdi hazırladığımız nesnenin elemanlarına bilgi aktarmadan içeriklerine erişip konsola yazacağız. Böylece C# derleyicisinin struct'ların alanlarına ilk değeri otomatik olarak aktarıp aktarmadığını görmüş olacağız.

```
class Program
{
 static void Main(string[] args)
 {
  Oyuncular oyuncu1 = new Oyuncular();
  Console.WriteLine(oyuncu1.Ad);
  Console.WriteLine(oyuncu1.Boy);
  Console.WriteLine(oyuncu1.Takim);
 }
}
```

Bu şekilde düzenlenen C# koduna derleyici bir itirazda bulunmuyorsa, bunun anlamı şudur: **new** anahtar kelimesi yardımıyla örneği hazırlanan struct'ın alan ve özelliklerine varsayılan değerler otomatik olarak aktarılmaktadır. Structlar değer tipleri olduğu için örneklerini hazırlamak için new anahtar kelimesini kullanmak bir zorunluluk değildir. Bu nedenle yukarıda verilen kod aşağıdaki gibi düzenlenebilir.

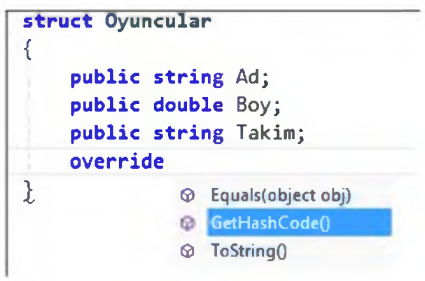
```
static void Main(string[] args)
{
 Oyuncular oyuncu1;
 Console.WriteLine(oyuncu1.Ad); // hata meydana gelir.
 Console.WriteLine(oyuncu1.Boy);
 Console.WriteLine(oyuncu1.Takim);
}
```

Ancak bu kod hataya neden olur. Çünkü Struct'ların örnekleri hazırlanırken new anahtar kelimesi kullanılmadığında alan ve özelliklere varsayılan değerler aktarılmaz. Tıpkı int ve long tipindeki değişkenlere ilk değerleri aktarmadan kullanamadığımız gibi. Burada karşılaşılan hatayı gidermek için hazırladığımız nesnenin alanlarına bilgi aktarıp içeriklerini öyle okumayı denedik.

```
namespace ConsoleApp1
{
 class Program
 {
 static void Main(string[] args)
 {
   Oyuncular oyuncu1;
   oyuncu1.Ad = "Kerem Gönlüm";
   oyuncu1.Boy = 2.10;
   oyuncu1.Takim = "Sakarya Belediye";
   Console.WriteLine(oyuncu1.Ad);
   Console.WriteLine(oyuncu1.Boy);
   Console.WriteLine(oyuncu1.Takim);
  }
 }
 struct Oyuncular
 {
  public string Ad;
  public double Boy;
  public string Takim;
 }
}
```

Yukarıda "Oyuncular" adını verdiğimiz struct'ı ayrı bir kod dosyasında hazırlamıştık. Hem kolaylık olsun hem de gerekirse struct'ın hazırlandığı ve kullanıldığı satırları birlikte gösterebilmek için üzerinde çalıştığımız "Program" adlı sınıfın olduğu kod dosyasına taşıdık.

Struct'lar base class olarak System'deki Object'i kullandıkları için Object sınıfı kaynaklı metotlar Struct dahilinde yeniden yazılabilir. Bu nedenle Struct kodlarken **override** anahtar kelimesi yazıldığında Visual Studio yeniden yazıp ezebileceğiniz metotları listeler.



Gerçekte, struct'lar ValueType'ten, ValueType sınıfı ise Object'ten türediği için Struct'lar Object'ten türemiştir demek önemsenecek bir yanlış ifade olmuyor.

Struct'lar İçin Yapıcı Metot Hazırlamak

Yukarıda belirtildiği gibi struct'lar **constructor** (yapıcı) metotlara sahip olabilirler. Daha önceki konulardan bildiğiniz gibi constructor metotlar, struct'ların örnekleri alındığı zaman otomatik olarak işletilen ve struct ile aynı ada sahip metotlardır.

Struct'ların yapıcı metotları parametreli olmak zorundadır. Başka bir deyişle, struct'lar varsayılan yapıcı metodu yeniden yazmanıza C# derleyicisi izin vermez. Struct için aşağıdaki gibi yapıcı metot hazırlanırsa derleyici hata rapor eder.

```
public struct Oyuncular
{
 public string Ad;
 public double Boy;
 public string Takim;
 public Oyuncular()
 {
 }
}
```

Çünkü bu durumda "Oyuncular" tipinde nesne hazırlanırken kullanılacak yapıcı metotta struct'ın alanlarına ilk değer aktarılmayacağı için derleme sırasında hata meydana gelir. Bu nedenle struct'lar için yapıcı metot hazırlanması halinde bütün alanlar düşünülerek parametreli bir metot hazırlanmalıdır. Başka bir deyişle bu **Struct** için parametreli yapıcı metot kullanılması halinde bütün alanlara ilk değerlerinin atanması gerekmektedir

```
public struct Oyuncular
{
  public string Ad;
  public double Boy;
  public string Takim;
  public Oyuncular(string Par1, double Par2, string Par3)
  {
   this.Ad = Par1;
   this.Boy = Par2;
   this.Takim = Par3;
  }
}
```

Struct için nesne hazırlanırken bu parametreli yapıcı metot kullanılmayabilir. Bu nedenle aşağıda verilen metot sorunsuz çalışır. Çünkü, struct için yukarıda yaptığımız gibi yapıcı metot hazırlamak, varsayılan yapıcı metodun ezilmesine neden olmaz. Yani bu şartlarda "Oyuncular" adını verdiğimiz struct 2 yapıcı metoda sahiptir.

```
static void Main(string[] args)
{
 Oyuncular nesne = new Oyuncular();
 Console.WriteLine(nesne.Ad);
}
```

Struct için yapıcı metot hazırlanmamışsa alanlara ilk değer verme zorunluluğu vardır. **new** anahtar kelimesi kullanılmadan Struct'ların örneği alınabilir. Bu durumda struct'ın alanlarına ilk değer aktarılması gerekir.

Yukarıda örnek olması için hazırladığımız struct 3 alana sahip olduğu ve dolayısıyla yapıcı metot 3 parametreye sahip olduğundan struct'ın örneği hazırlanırken parametre kullanılacaksa hepsi kullanılmalıdır.

```
static void Main(string[] args)
{
 Oyuncular nesne;
 nesne = new Oyuncular("Kerem Gönlüm", 2.10, "Sakarya");
 Console.WriteLine(nesne.Ad);
 Console.WriteLine(Convert.ToString(nesne.Boy));
 Console.WriteLine(nesne.Takim);
 Console.ReadLine();
}
```

Struct'ların yapıcı metotları aşırı yüklenebilir veya değişik sayıda parametreye sahip versiyonları olabilir. Bu imkanı göstermek için yukarıda verilen yapıcı metodun 2 parametreli versiyonunu hazırlayıp kullandık.

```
public struct Oyuncular
{
 public string Ad;
 public double Boy;
 public string Takim;
 public Oyuncular(string Par1, double Par2, string Par3)
 {
   this.Ad = Par1;
   this.Boy = Par2;
   this.Takim = Par3;
  }
  public Oyuncular(string Par1, double Par2)
  {
   this.Ad = Par1;
   this.Boy = Par2;
   this.Takim = "";
  }
}
```

2 parametreli yapıcı metotta Struct'ın "takım" adını verdiğimiz 3. elemanına varsayılan değeri aktardık. String tipindeki 3. elemana bu şekilde boşluk aktarmasaydık hata meydana gelirdi. Diğer yandan struct'lar için yok edici metot hazırlanamaz.

Şimdi örnek olması için bu struct'taki bir alan için özellik hazırlayacağız. Bu amaçla struct'taki ilk alanı private yaptık.

```
public struct Oyuncular
{
 private string Ad;
 public double Boy;
 public string Takim;
 public string prAd
 {
  get
   {
    return Ad;
   }
  set
   {
```

```
        Ad = value;
      }
    }
    public Oyuncular(string Par1, double Par2, string Par3)
    {
      this.Ad = Par1;
      this.Boy = Par2;
      this.Takim = Par3;
    }
    public Oyuncular(string Par1, double Par2)
    {
      this.Ad = Par1;
      this.Boy = Par2;
      this.Takim = "";
    }
  }
```

Yukarıdaki sayfalarda struct'larda yapıcı metotların nasıl hazırlanıp, hangi noktalara dikkat edilmesi gerektiğinden söz ettik. Şimdi sırada struct'larda normal metot hazırlamak var. Bu amaçla "Oyuncular" adlı struct'ta Yazdir() adında bir metot hazırladık.

```
public void Yazdir(Oyuncular o1)
{
 Console.WriteLine(o1.Ad);
 Console.WriteLine(o1.Boy);
 Console.WriteLine(o1.Takim);
}
```

Oyuncular adlı struct dahilinde hazırladığımız bu metot yine "Oyuncular" tipinde bir parametreye sahiptir. Metot, parametre olarak aldığı "Oyuncular" tipindeki nesnenin alanlarını konsola yazmakla yetinmektedir. Bu metodu kullanmak üzere uygulamanın Main() metodunu aşağıdaki gibi düzenledik.

```
public static void Main()
{
 Oyuncular nesne = new Oyuncular();
 nesne.Ad = "Kerem Gönlüm";
 nesne.Boy = 2.12;
 nesne.Takim = "Sakarya Belediye";
 nesne.Yazdir(nesne);
}
```

C# programcıları arasında sadece veri içeren yani önemli işlemlerin yapıldığı metotların olmadığı yapıları hazırlarken struct hazırlama tercihi yaygındır.

Struct'larda Sabit Hazırlamak

Hazırladığınız struct içinde sabit tanımlayabilir veya readonly özellikli alanlara yer verebilirsiniz. **const** anahtar kelimesi ile sabit tanımlarken ilk değerin aktarılması gerekiyor. Sabitlere ancak struct üzerinden ulaşılmaktadır. Dolayısıyla tanımlanan sabite başka sınıf veya struct'tan erişim sağlamak için erişim belirleyici olarak public'i kullanmak gerekiyor.

```
struct Oyuncular
{
 public string Ad;
 public double Boy;
 public string Takim;
 public const double PI = 3.14;
 public const double ivme = 9.81;
}
```

Tanımlamak istediğiniz sabit için seçeceğiniz tip reference type olamaz. Başka bir deyişle sabitler **value types** olmak zorundadır. Bu nedenle aşağıda verilen sabit tanımlama satırı hataya neden olur.

```
struct Oyuncular
{
 public const object Takim = "Ankaragücü"; // hata
}
```

Yukarıda anlatılanlara göre **const** olarak tanımlanan değişkenin içeriği kod yazılırken bilinmelidir. Nesne oluşturma işlemi sırasında yapıcı metotta içeriği ayarlanılabilinen değişkenlere **readonly** denilmektedir. Struct'larda readonly alanların nasıl tanımlandığını göstermek için yukarıda verdiğimiz struct'ı aşağıdaki gibi düzenledik.

```
public struct Oyuncular
{
 public string Ad;
 public double Boy;
 public string Takim;
 public readonly string lig;
 public Oyuncular(string Par1, double Par2, string Par3)
 {
  this.Ad = Par1;
  this.Boy = Par2;
  this.Takim = Par3;
  lig = "Superlig";
 }
}
```

Burada "lig" adını verdiğimiz readonly alanın içeriği yapıcı metotta ayarlandığı için bu alana nesne üzerinden erişilebilir. Bu alan readonly olduğu için nesne üzerinde içeriği okunabilir ancak mevcut içerik değiştirilemez.

Metotlara Değişken ve Nesnelerin Referanslarını Aktarmak

Metotların anlatıldığı bölümden hatırlayacağınız gibi int, long ve double gibi değer tipi değişkenler metotlara kopyalanarak gönderilmektedir. Parametre olarak metoda referans tipi bilgiler veya nesneler gönderilmek istendiğinde ise bu değişkenlerin kopyaları yerine referansları gönderiliyordu. Değer tipli değişkenlerin içerikleri yerine referansları metotlara gönderilmek istendiğinde ref anahtar kelimesinden yararlandığımızı hatırlıyor olacaksınız.

Struct'lar değer tipleri oldukları için mevcut bir struct'tan yola çıkılarak hazırlanan nesnenin kopyası metoda gönderilir. Tekrar niyetine bu konu hakkında konuşmak üzere önce aşağıdaki gibi bir struct hazırladık.

```
public struct Kare
{
 public int X { get; set; }
 public int Y { get; set; }
 public int Kenar { get; set; }
 public Kare(int konumX, int konumY, int kenar)
 {
  this.X = konumX;
  this.Y = konumY;
  this.Kenar = kenar;
 }
}
```

3 özelliğe ve 3 parametreli yapıcı metoda sahip bu struct başka metot hazırlamadık. Çünkü struct bağlamında metotlara gönderilen parametrelerden söz etmekle yetinmek istiyoruz. Bu şekilde struct hazırladıktan sonra üzerinde çalıştığımız Windows Forms uygulamasında aşağıdaki gibi "Ciz()" adında ve "Kare" tipinde bir parametreye sahip bir metot hazırladık.

```
private void Ciz(Kare k)
{
 Graphics Cizim = this.CreateGraphics();
 Pen Kalem = new System.Drawing.Pen(Color.Blue, 3);
 Cizim.DrawRectangle(Kalem, k.X, k.Y, k.Kenar, k.Kenar);
}
```

Bu metot parametre olarak aldığı Kare nesnesinin özelliklerine uygun olarak formun üzerinde bir kare şekli çizecektir. Bu metodu hazırladıktan sonra projenin formuna bir düğme yerleştirip bu düğme tıklandığında işletilecek metodu aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
  Kare k1 = new Kare(50, 50, 75);
  Ciz(k1);
}
```

Formdaki button1 nesnesi tıklandığında işletilecek bu metot sayesinde sol üst köşe koordinatı (50, 50) olan 75 piksel kenar uzunluğuna sahip bir kare şekli çizilir. Bizim buradaki amacımız referanslar hakkında bilgi vermek olduğu için Form1 sınıfı içinde Degistir() adında bir metot hazırlayarak kendisine parametre olarak verilecek "Kare" nesnenin özelliklerinde değişiklik yaptık.

```
private void Degistir(Kare k)
{
 k.X = k.X + 10;
 k.Y = k.Y + 10;
}
```

Bu hazırlıktan sonra yukarıda hazırlayıp "k1" adını verdiğimiz "Kare" nesnesini ekrana çizdikten sonra Degistir() metoduna parametre olarak verip sol üst köşesinin koordinatını değiştirip Ciz() metodu ile tekrar çizdirdik.

```
private void button1_Click(object sender, EventArgs e)
{
  Kare k1 = new Kare(50, 50, 75);
  Ciz(k1);
  Degistir(k1);
  Ciz(k1);
}
```

Burada Degistir() adını verdiğimiz metoda "k1" nesnesinin kendisi yerine kopyası gönderildiği için "k1" nesnesinin button1_Click() metodu içinde hazırlanan ilk hali korunur. Yani Degistir() metodu bu nesne üzerinde etkili olmaz. Dolayısıyla çalışma anında button1 tıklanıp bu metot işletilirse aynı yere üst üste 2 kez kare şekli çizilir. Değiştir() metoduna "Kare" nesnesinin kopyası yerine referanslarını göndermiş olsaydık sonuç farklı olurdu.

```
private void Degistir(ref Kare k)
{
  k.X = k.X + 10;
  k.Y = k.Y + 10;
}
private void button1_Click(object sender, EventArgs e)
{
 Kare k1 = new Kare(50, 50, 75);
 Ciz(k1);
 Degistir(ref k1);
 Ciz(k1);
}
```

Yukarıda ifade edildiği gibi mevcut bir struct'tan yola çıkılarak hazırlanan nesneler Value Type oldukları için metotları referansları değil yeni bir kopyası gönderilir. Metodu ilgili struct tipindeki nesnenin kopyası yerine referansını göndermek istiyorsanız ref veya out anahtar kelimesinden yararlanabilirsiniz.

Struct'lar ve Interface'ler

Interface'ler daha önce class bağlamında incelendiği için bu başlık altında, struct'ların interface desteğinin olduğunu vurgulamakla yetineceğiz. Struct'ların kalıtım desteğinin olmadığı düşünülürse, interface'lerin struct'lar için önemi artmaktadır. Diğer yandan struct'ların çok sayıda metoda sahip olmalarının önerilmediğini unutmamak gerekir.

Bu nedenle hazırlayacağımız interface'te önceliği özelliklere verdik. Interface'ler ve özellikler birlikte düşünüldüğünde, C# derleyicisinin interface'lerde alanların olmasına izin vermediğini hatırlamak gerekir. Aşağıda verdiğimiz interface bir özelliğe sahiptir.

```
public interface IArayuz
{
 string takim
  {
    get; set;
  }
}
```

Bu interface'ın ister class'a ister struct'a uygulayın "takim" adında bir özelliğe sahip olması gerekir. Bu interface'i yukarıda hazırladığımız struct'a nasıl uygulayıp, nasıl gereğini yaptığımızı aşağıda görebilirsiniz.

```
public struct Oyuncular : IArayuz
{
 private string Ad;
 public double Boy;
 public string Takim;
 public string takim
 {
  get
   {
    return Takim;
   }
  set
  {
   Takim = value;
   }
   }
 }
public interface IArayuz
{
 string takim
 {
   get; set;
 }
}
```

Ne zaman struct, ne zaman class hazırlamak gerektiği sorusuna cevap vermek kolay değildir. Buna rağmen class-struct seçiminde sizlere yardımcı olması için kendi görüşlerimizi buraya aldık.

- Hazırlamak istediğiniz tip, metot içermiyorsa veya içerdiği metotların sayısı bir ki tane ile sınırlı ise tercihinizi struct'tan yana yapabilirsiniz.
- Hazırlamak istediğiniz tipin başka sınıfların base class'ı olma ihtimali varsa tercihiniz class şeklinde olmalıdır.
- Hazırlamak istediğiniz tipi sıfırdan hazırlamayacaksanız yani mevcut tiplerden yola çıkarak hazırlayacaksanız tercihiniz class olmalı. Örneğin .NET Framework ile gelen List sınıfı benzeri bir tip hazırlamak istiyorsanız tercihinizi class'tan yana yapmalısınız. Çünkü .NET Framework ile birlikte List ve benzeri sınıfları hazırlarken base class olarak kullanmanız için hazırlanmış sınıflar bulunmaktadır.

19. BÖLÜM
ENUM TİPLERİ HAZIRLAMAK ve KULLANMAK

C# kodu yazarken sıklıkla kullandığınız sayısal sabitleriniz varsa, bunları isimlendirmek iyi bir fikirdir. Bu tip bilgilere örnek vermek gerekirse, Windows'un her mesajının 16 tabanlı sayı ile ifade edilmiş bir numarası bulunmaktadır. Hangi numaralı mesajın hangi işlemi temsil ettiğini programcı için akılda tutmak zordur. Bu nedenle mesajlara, işlevini çağrıştıran adlar verip bunları enum olarak bir CS dosyasına yazarsanız gerek duyduğunuzda mesajın işlevini kolayca hatırlayabilirsiniz. Aşağıda verilen listeyi dikkatlice incelerseniz ne demek istediğimizi anlayacaksınız.

```
WM_NULL = 0x0000
WM_CREATE = 0x0001
WM_DESTROY = 0x0002
WM_MOVE = 0x0003
WM_SIZE = 0x0005
WM_ACTIVATE = 0x0006
WM_SETFOCUS = 0x0007
WM_KILLFOCUS = 0x0008
WM_ENABLE = 0x000A
WM_SETREDRAW = 0x000B
WM_SETTEXT = 0x000C
WM_GETTEXT = 0x000D
WM_GETTEXTLENGTH = 0x000E
WM_PAINT = 0x000F
WM_CLOSE = 0x0010
```

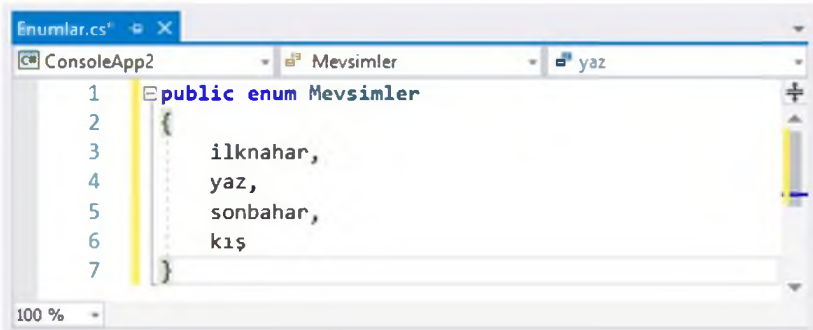
En başta belirmek gerekir; enum tipleri hazırlamanın asıl amacı programcıların işini kolaylaştırmaktır. Enum'lar, hazırlanan uygulamanın kullanıcıya yansıyan yanıyla ilgili değildir. Elbette kullanıcıların sıklıkla yararlanacağı bazı sabit bilgileri veritabanına veya source dosyalarına yazmak yerine enum olarak değerlendirip kodun içinde, dolayısıyla EXE dosyada tutmak mümkün olsa bile fazla tercih edilen bir yöntem değildir. Buna rağmen, bu bölümde ileriki sayfalarda hakkında bilgi verilecek Enum sınıfının bazı metotları sayesinde enum tiplerini kullanıcıya yönelik bazı işlemlerde kullanmak mümkün olmaktadır.

Konuyu baştan ele almak gerekirse, programcı enum'ları kendisi için hazırlar ve kod yazarken kullanır. .NET Framework ile birlikte çok sayıda **Enum** tipinin hazır olarak geldiğini düşünürsek enum'ların C# programcıları için önemini anlayabiliriz.

Enum'lar yani sayılabilir tipler, söz konusu değişkenin alabileceği değerler önceden belli ise işlevsel olmaktadır. Bu tip bilgilere haftanın günleri ve mevsimler örnek olarak verilebilir. Başka bir anlatımla, sayılabilir tipler, sayısal değerleri anlamlı adlarla ifade etmemize yardımcı olmakta ve programcının kod yazma hızını arttırmaktır.

Enum tiplerini ayrı kod dosyalarında toplayabilir veya üzerinde çalıştığınız mevcut kod dosyalarında hazırlayabilirsiniz. Bize sorarsanız hazırlamak istediğiniz enum'u uygulamadaki çok sayıda sınıfta ve birden fazla uygulamada kullanmak istiyorsanız ayrı bir kod dosyasında hazırlamanız iyi bir tercih olacaktır.

Aşağıda ekran görüntüsü olarak verilen dosyada "Mevsimler" adında bir enum tipi hazırladık. Enum tiplerini hazırlarken **public**, **private** ve **protected** erişim belirleyicilerini kullanabilirsiniz.



Bu şekilde "Mevsimler" adında bir enum tipi hazırladıktan sonra C# kodu yazarken bu tipten yararlanarak değişken tanımlayıp kullanabilirsiniz. Aşağıda "Mevsimler" tipinden yola çıkarak "bu_mevsim" adında bir değişken tanımladık. "bu_mevsim" değişkenine bilgi aktarırken Visual Studio aktarabileceğimiz seçenekleri listeledi. Visual Studio, listeleme yaparken enum tipinin elemanlarını alfabetik sıraya koyduğunu fark etmiş olmalısınız.



Bu sayede mevsimleri temsil eden sayılar yerine mevsimlere kendi vermiş olduğumuz adları kod içinde kullanma şansı elde etmiş olduk. Çünkü, programcı kod içinde kendince anlamlı isimler verdiği enum nesnesinin temsil ettiği sayısal bilgiye ihtiyaç duyacaktır. Burada hazırladığımız "Mevsimler" adlı enum için konuşmak gerekirse, programcı metoda bir mevsimi parametre olarak verirken string hali yerine int tipini tercih edecektir. Dolayısıyla kod yazarken Sonbahar'ın adı yerine temsil ettiği sayısal bilgiye daha çok gerek duyacaktır.

Şimdi "Mevsimler" tipindeki "bu_mevsim" değişkenin içeriğini **int** tipinde bir değişkene aktaracağız. Aktarma işlemi sırasında dönüştürme yapılmazsa hata meydana gelir. "bu_mevsim" değişkenine "Kış" bilgisini aktardığımız için "sira" değişkenin içeriği 3 olur.

```
private void button1_Click(object sender, EventArgs e)
{
 Mevsimler bu_mevsim = Mevsimler.kis;
 int sira = (int)bu_mevsim;
 textBox1.Text = bu_mevsim.ToString();
 textBox2.Text = sira.ToString();
}
```

Bu metot işletildiğinde ilk TextBox'a "Kış" ve 2. TextBox'a ise 3 aktarılır. Çünkü "Kış" seçeneğinin enum tipi içindeki sıra numarası 3'tür. Bu koddan ayrıca şu sonucu çıkarmak mümkündür: ToString() metodu bu şekilde kullanılarak Enum tipinde sayısal bilgiye verilen adı string olarak elde etmek mümkündür.

"Mevsimler" tipindeki değişkene "yaz" bilgisini aktarmak istiyorsanız aşağıdaki gibi bir satır yazılabilir. Tabi bu şekilde bir yazım, enum hazırlamanın arkasındaki felsefeye terstir.

```
bu_mevsim = (Mevsimler)1;
```

Yukarıda tanımlayıp "Mevsimler" adını verdiğimiz enum tipinde ilk sırada yazılan elemana 0, 2. sırada yazılan elemana ise 1 sıra numarası verilir. İsterseniz elemanlara kendiniz sıra numarası verebilirsiniz. Bu nedenle "Mevsimler" adını verdiğimiz Enum tipi aşağıdaki gibi hazırlanırsa değişen bir şey olmaz.

```
public enum Mevsimler
{
 ilkbahar = 0,
 yaz = 1,
 sonbahar = 2,
 kis = 3
}
```

Enum tipinin elemanlarına verilen sıra numaralarının veya aktarılan sayısal bilgilerin **int** tipinde olduğu varsayılmaktadır. Kendi hazırladığınız enum tipinin elemanları için int yerine **byte** veya başka bir tamsayı tipini (short, ushort, long, ulong) tercih edebilirsiniz. Hazırlamak istediğiniz enum tipinin elemanları arasında eksi değerleri tutacak olanlar varsa adında "s" harfi olan işaretli (signed) tiplerden birisini seçmeniz gerekir.

```
public enum Mevsimler : byte
{
 ilkbahar=0,
 yaz=1,
 sonbahar=2,
 kis=3
}
```

Enum tipinin elemanlarına sayısal değer aktarırken bütün elemanlara 2. elemanın içeriği ilk elemandan 1 fazla olacaksa, başka bir deyişle elemanların içeriği sıralı bir şekilde artıyorsa aşağıda yapıldığı gibi ilk eleman hariç diğerlerini belirtmek zorunda değilsiniz.

```
public enum Yillar
{
 ilkyil = 2013,
 ikinciyil,
 ucuncuyil,
 dorduncuyil
}
```

Bu şekilde tanımlanan enum tipinde "ikinciyil" elemanına 2014, diğerlerine ise 2015 ve 2016 aktarılır. Bu nedenle aşağıda verilen kod işletilirse formun başlığına 2014 yazılır.

```
private void button1_Click(object sender, EventArgs e)
{
 Yillar buyil = Yillar.ikinciyil;
 int yil = (int)buyil;
 this.Text = yil.ToString();
}
```

Burada "buyil" adını verdiğimiz Enum nesnesinin temsil ettiği sayısal değeri öğrenip formun başlığına yazdık. Zaten programcının ihtiyacı olan da budur. Yani elindeki sayısal sabitlere enum hazırlayarak kendisine göre anlamlı gelecek string karşılıklar vermiştir. Kod yazarken sayısal değeri değil, sayısal değeri temsil eden anlamlı adı kullanmayı tercih eder.

Enum nesnesinde saklanan bilginin tipini öğrenmek istiyorsanız **GetTypeCode**() metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Aşağıda verilen kod işletildiğinde TextBox'a enum tipinde tutulan bilgilerin tipi aktarılır. Enum'da tutulan sayısal bilginin tipi int ise TextBox'a "Int32", byte ise "Byte" aktarılır.

```
private void button1_Click(object sender, EventArgs e)
{
 Yillar buyil;
 buyil = Yillar.ucuncuyil;
 textBox1.Text = buyil.GetTypeCode().ToString();
}
```

Enum tiplerinin nasıl hazırlanıp kullanıldığından biraz söz ettikten sonra enum'larla ilgili birkaç teknik cümle kurabiliriz. Yukarıda anlatılan şekilde Enum anahtar kelimesini kullanarak bir enum tipi hazırladığımızda geri planda abstract bir sınıf olan **Enum** sınıfından türetme yapılır. Bu sınıfın yapısını aşağıda görebilirsiniz.

```
public abstract class Enum : ValueType, IComparable,
                IFormattable, IConvertible
{
}
```

Orijinal yapısından göreceğiniz gibi Enum sınıfı **VulueType** sınıfının mirasçısıdır. ValueType'ın ise Object sınıfının mirasçısı olduğunu biliyorsunuz. Bundan şu sonucu çıkarabiliriz: Enum'lar value type'dır. Konunun devamında bu abstract sınıfın birkaç static metodundan söz edilecektir.

Konunun iyice anlaşılmasını sağlamak için başka bir enum tipi hazırlayacağız. Bu amaçla ilk olarak "Aylar" adında bir enum tipi hazırladık.

```
public enum Aylar
{
 Ocak=1, Şubat=2, Mart=3,
 Nisan=4, Mayıs=5, Haziran=6,
 Temmuz=7, Ağustos=8, Eylül=9,
 Ekim=10, Kasım=11, Aralık=12
}
```

Bu hazırlıktan sonra üzerinde çalıştığımız Windows Forms projesi çalıştırıldığında geçerli tarihe uygun olarak ay bilgisinin forma yerleştirdiğimiz Label'a yazılmasını sağlamak üzere formun Load olayını temsil eden metodu aşağıdaki gibi düzenledik.

```
int Ay;
private void Form1_Load(object sender, EventArgs e)
{
 DateTime tarih = DateTime.Today;
 Ay = (int)tarih.Month;
 Aylar bu_ay = (Aylar)Ay;
 this.label1.Text = bu_ay.ToString();
}
```

Diğer metotlarda kullanmak üzere "Ay" adında ve int tipinde bir değişkeni class bloğunda tanımladık. Bu kodda önce geçerli tarihin ay bilgisini öğrenip "Ay" adlı değişkene aktardık. Ardından Enum tipinden ilgili ay bilgisini öğrenip Label'a yazdık. Bu sayede elimizdeki geçerli ay bilgisini temsil eden sayısal bilgiden yola çıkıp, enum tipini hazırlarken o aya verdiğimiz adı öğrendik. Aşağıda verilen ekran görüntüsünü ilk formun Load olayını temsil eden metodu bu şekilde düzenlenen projeyi çalıştırdıktan sonra aldık.



Bu sırada "Sonraki" veya "Önceki" başlıklı düğmeler tıklandığında label nesnesine yazılan ay bilgisinin değişmesini sağlamak için aşağıdaki gibi kod yazdık. Benzer kodu "Önceki" düğmesi için de yazabilirsiniz.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 Ay += 1;
 Aylar bu_ay = (Aylar)Ay;
 this.label1.Text = bu_ay.ToString();
}
```

Enum Sınıfı

Bu bölümün ilk sayfasında ifade edildiği enum tiplerini asıl kullanım amacı programcıyı sık kullanıldığı bazı sayısal değerleri hatırlamak zorunda bırakmamak ve istenen sayısal sabitleri anlamlı şekilde isimlendirip gerektiğinde kullanılabilmesini sağlamaktır. Buna rağmen Enum sınıfındaki bazı metotlar enum tiplerinde tutulan bilgilerin çalışma anında kullanıcıya sunulmasına kaynaklık edebilmektedir.

Sayılabilir tipler üzerinde işlem yaparken **System**'de bulunan **Enum** sınıfının bazı metotlarından yararlanabilirsiniz. Bu sınıftaki metotların nasıl kullanıldığını anlatmak için

"Takımlar" adında bir enum tipi hazırladık. Bu sayılabilir tipi hazırlamadan önce üzerinde çalıştığımız projeye bir kod dosyası dahil edip enum'u bu kod dosyasında hazırladık.

```
public enum Takimlar
{
 Besiktas = 1903,
 Fenerbahçe = 1907,
 Galatasaray = 1905,
 Ankaragücü = 1910,
 Trabzonspor = 1967
}
```

Hazırladığınız enum tipinin bütün elemanlarını string tipteki dizi değişkene aktarmak istiyorsanız **GetNames**() metodunu kullanabilirsiniz. GetNames() metodu, uygulandığı enum tipindeki elemanların ad kısmını string bir diziye doldurup geriye göndermektedir. Dolayısıyla bu metodu kullanmadan önce bir dizi değişken tanımlamak gerekir.

Bu metodun nasıl kullanıldığını göstermek için enum tipini hazırladıktan sonra projedeki ilk forma bir ListBox nesnesi yerleştirip aşağıda verilen kodu hazırladık.

```
private void Form1_Click(object sender, EventArgs e)
{
 string[] takimDizi = Takimlar.GetNames(typeof(Takimlar));
 for (int i = 0; i < takimDizi.Length; i++)
   listBox1.Items.Add(takimDizi[i]);
}
```

Enum tipinin bütün elemanları yerine yalnızca bir elemanla ilgileniyorsanız **GetNames**() yerine **GetName**() metodunu kullanabilirsiniz. İlgilendiğiniz elemanın sıra numarasını GetName() metoduna 2. parametre olarak vermeniz gerekir. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void button2_Click(object sender, EventArgs e)
{
 string takim = Takimlar.GetName(typeof(Takimlar), 1907);
 textBox1.Text = takim;   // Fenerbahçe
}
```

GetValues() Metodu

Yukarıdaki sayfalarda Enum sınıfının **GetNames**() metodundan yararlanıp ilgili Enum'daki bilgileri veya adları elde edip ListBox'a aktarmıştık. Daha sonra programcı seçtiği enum sabitinden yola çıkıp o sabitin temsil ettiği değeri bulup ona göre işlem yapmıştık.

Enum sınıfının **GetValues**() metodundan yararlanıp enum tipinin değerlerini elde edebilirsiniz. Bu metodun nasıl kullanıldığını anlatırken yukarıda verilen "Takimlar" adını verdiğimiz tipten yararlanacağız. Forma 2 bir ListBox yerleştirdikten sonra aşağıda verilen kodu yazdık.

```
private void Form1_Load(object sender, EventArgs e)
{
 string[] takimDizi = Takimlar.GetNames(typeof(Takimlar));
 foreach (string il in takimDizi)
   listBox1.Items.Add(il);
 int sayi = takimDizi.Length;
 Array kurulusDizi = Array.CreateInstance(typeof(byte), sayi);
 kurulusDizi = Takimlar.GetValues(typeof(Takimlar));
 foreach (int yil in kurulusDizi)
   listBox2.Items.Add(yil.ToString());
}
```

Bu kod sayesinde ilk ListBox'ta enum dahilindeki takım adları, 2. ListBox'ta ise takımların kuruluş yılları listelenir. Enum sınıfının **GetValues**() metodu geriye **Array** nesnesi gönderdiği için önceden bir dizi değişken tanımladık.

Parse() ve TryParse() Metotları İle Enum'da Arama Yapmak

Yukarıdaki sayfalarda bazı enum tiplerini hazırlarken elemanları temsil edecek sayısal bilgilerin tipini belirtmedik. Enum'daki seçenek veya elemanlara aktarılacak bilginin tipi belirtilmediğinde **int** tipinde olacağı varsayılmaktadır. Yeni hazırladığımız enum'da tutulacak sayısal bilgilerin byte tipinde olacağını belirttik.

```
enum iller : byte
{
 Adana = 1,
 Ankara = 6,
 Antalya = 7,
 Bursa = 16,
 Eskişehir = 26,
 Gaziantep = 27,
 İstanbul = 34,
 İzmir = 35
}
```

İllerin plaka numaralarını girmek yerine adlarını bir liste kutusundan seçmek için enum tipini tanımladıktan sonra projenin ilk formuna bir **ComboBox** yerleştirip bu formun Load olayını temsil eden metodu aşağıdaki gibi düzenledik.

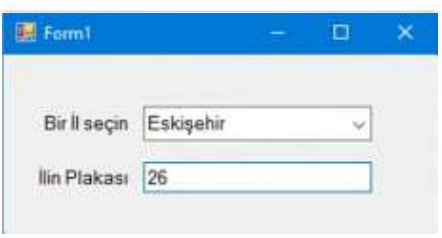
```
private void Form1_Load(object sender, EventArgs e)
{
 string[] il_dizi = iller.GetNames(typeof(iller));
 foreach (string il in il_dizi)
   comboBox1.Items.Add(il);
}
```

Bu satırlar sayesinde sayılabilir tipin elemanları ComboBox'ta listelenir. Elemanları "Bursa=16" gibi tanımlamış olmamıza rağmen ComboBox'ta yalnızca il adları listelenir. ComboBox'ta seçilen ilin plakasını göstermek için forma bir TextBox yerleştirip ComboBox'ta seçim yapıldığında işletilecek metodu aşağıdaki gibi düzenledik.

```
private void comboBox1_SelectedIndexChanged(object sender,EventArgs e)
{
  byte plaka;
  if (comboBox1.Text == "Eskişehir")
   {
    plaka = (byte)iller.Eskişehir;
    textBox1.Text = plaka.ToString();
   }
  if (comboBox1.Text == "Ankara")
   {
    plaka = (byte)iller.Ankara;
    textBox1.Text = plaka.ToString();
   }
  if (comboBox1.Text == "İstanbul")
   {
    plaka = (byte)iller.İstanbul;
    textBox1.Text = plaka.ToString();
   }
  if (comboBox1.Text == "Gaziantep")
   {
    plaka = (byte)iller.Gaziantep;
    textBox1.Text = plaka.ToString();
   }
}
```

Pratik değeri olmayan bu kodu konu anlaşılsın diye hazırladık. Çünkü bu bölümün ilk sayfasında ifade edildiği gibi enum hazırlamanın asıl amacı programcının işini kolaylaştırmaktır. Burada hazırladığımız örnek ise sanki kullanıcıya yönelikmiş gibi duruyor.

Bu şartlarda proje çalıştırılıp ComboBox'ta herhangi bir il seçilirse aşağıdaki gibi sonuç alınır. Bu kodu hazırlamamızın ikinci nedeni ise size "bunun kolay bir yolu yok mu?" sorusunu sordurmaktır. Programcı, kod dahilinde "plaka==27" yazacağına **plaka==(byte)iller.Gaziantep** yazmayı tercih edecektir. Bu sayede kod daha okunur olacaktır.



Enum sınıfının **Parse**() metodu olmasaydı yukarıda verilen koda ekleme yapmaz veya üzerinde konuşmayı burada keserdik. Enum sınıfının **Parse**() metodu, kendisine parametre olarak verilen enum tipinde arama yapılırken kullanılmaktadır. Bu metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void comboBox1_SelectedIndexChanged(object sender,EventArgs e)
{
 string secilen = comboBox1.Text;
 byte plaka = (byte)Enum.Parse(typeof(iller), secilen, true);
 textBox1.Text = plaka.ToString();
}
```

Parse() metodu kendisine 2. parametre olarak verilen bilgiyi 1. parametrede verilen Enum tipinde aramaktadır. **Parse**() metodunun bool tipte olan 3. parametresi enum tipinde arama yapılırken küçük büyük harf ayrımının yapılıp yapmayacağı ile ilgilidir. 3. parametre olarak **false** kullanılırsa aramada küçük büyük harf ayrımı yapılır.

Bu kod sayesinde ComboBox'ta seçilen ilin plakası TextBox'a aktarılır. Parse() metodu ile Enum tipte aranan bilgi bulunamazsa hata meydana gelir. Dolayısıyla olmayan bir elemanın Parse() metodu ile enum tipinde aramamak yani tedbir almak gerekir.

Şimdi konu üzerinde düşünmenizi sağlamak için aynı enum tipinde bir de konsol uygulamasında Parse() metodu ile arama yapacağız. Bu kez enum tipinde aranacak bilgiyi kullanıcının konsola yazmasını isteyeceğiz.

```
class Program
{
 static void Main(string[] args)
  {
   string il = Console.ReadLine();
   Type tip = typeof(iller);
   byte plaka = (byte)Enum.Parse(tip, il, true);
   Console.WriteLine(il + "İlinin plakası:" + plaka.ToString());
   Console.ReadLine();
  }
}
```

Gerçekte Parse() metodunun bu kullanımı yukarıda verilenden farklı değildir. Kullanıcı konsola ister il adını ister plakayı girsin, bulunursa geriye plaka bilgisi gönderilir.

C# 4.0 ile birlikte Enum sınıfına yukarıda sözü edilen metotlara ek olarak **TryParse**() metodu geldi. TryParse() metodu yine enum'da arama yapılırken kullanılmaktadır. TryParse() metodunun en önemli avantajı, yapılacak aramanın başarısız olma ihtimaline karşılık try-catch bloklarını hazırlamaya gerek olmamasıdır.

TryParse() metodunun nasıl kullanıldığını anlatırken yukarıda hazırlayıp kullandığımız "iller" adlı enum tipinden yararlanacağız. Bu amaçla ilk olarak enum tipinde aranacak il adını veya plakanın konsoldan girilmesini istedik ve aranıp bulunan bilginin aktarılacağı enum tipinde bir değişken tanımladık.

```
static void Main(string[] args)
{
 Console.Write("Aranacak il adı veya plakası :" );
 string il = Console.ReadLine();
 iller iPlaka;
 Console.ReadLine();
}
```

Burada "ilPlaka" adını verdiğimiz değişkene arama işlemi sonucu bulunacak ilin plakası aktarılacaktır. TryParse() metodu bu değişkenin önceden tanımlanmasını istemektedir. Konunun devamında kullanılacak değişkenin önceden tanımlaması konusunda C# 7.0 ile gelen değişiklikten söz edeceğiz.

```
static void Main(string[] args)
{
 Console.Write("Aranacak il adı veya plakası :" );
 string il = Console.ReadLine();
 iller ilPlaka;
 bool sonuc = Enum.TryParse<iller>(il, out ilPlaka);
}
```

Şimdi Enum sınıfının TryParse() metodunu mercek altına alacağız:

- Bu metot geriye bool bir değer göndermektedir. Yaptığı arama başarılı değilse geriye false göndermektedir.
- Enum sınıfının TryParse() metodu generic bir metottur. Bu nedenle metot adından sonra diamond "<> " operatörüne yer verilmektedir. Arama hangi enum tipinde yapılacaksa o enum tipinin adı "<>" arasında yazılmalıdır.
- Bu metodun aşırı yüklenmiş 2 sürümü olsa bile burada 2 parametreye sahip olan sürümünü kullandık.
- TryParse() metodu 1. parametre olarak enum'da aranacak bilgiyi almaktadır. Bu bilginin string tipinde olması gerekiyor.
- TryParse(), kendisine 1. parametre olarak verilen bilgiyi bulursa, bulduğunu 2. parametredeki değişkene aktarmaktadır. 2. parametre olarak kullanılan değişkenin daha önceden tanımlanması gerektiğini biliyorsunuz.
- Gelelim 2. parametrede kullanılan değişkenin önüne yazılan **out** anahtar kelimesine: Enum sınıfı için Parse() dururken bir de TryParse() metodunun hazırlanmasının tek nedeni, programcıyı try-catch blokları hazırlama zahmetinden kurtarmaktır. Bu nedenle metodun aramada başarılı olup olmadığının öğrenilmesi için geriye bool bilgi gönderilmesi gerekiyordu. Aranan bilgiyi dışarı çıkarmak için ya tuple tipi bir çözüm bulunacaktı ya da out anahtar kelimesinden yararlanılacaktı. Microsoft'un uzmanları tercihlerini out'tan yana kullandıkları için arama sonucu bulunan bilgi için bu değişkene bakmak gerekiyor.

C# 7.0 ile birlikte out anahtar kelimesi kullanılmak istendiğinden önceden değişken tanımlamak şart olmaktan çıkarıldı. Bu nedenle TryParse() aşağıdaki gibi düzenlenebilir.

```
class Program
{
 static void Main(string[] args)
 {
   Console.Write("Aranacak il adı veya plakası :" );
   string il = Console.ReadLine();
   bool sonuc = Enum.TryParse<iller>(il, out iller ilPlaka);
   if (sonuc)
   {
    Console.WriteLine("Arama başarılı");
    Console.WriteLine(ilPlaka.ToString());
   }
  Console.ReadLine();
 }
}
```

Format Metodu

Enum sınıfının yukarıda sözü edilen Parse() ve TryParse() metotlarından başka **Format**() metodu bulunmaktadır. Bu metot sayesinde enum tipinin elemanlarını biçimleyebilirsiniz. Bu metodun nasıl kullanıldığını anlatmak için elimizde "iller" adını verdiğimiz enum tipi varken aşağıda verilen kodu yazdık.

```
private void Form1_Click(object sender, EventArgs e)
{
 iller il = iller.Adana;
 textBox1.Text = il.ToString();
}
```

Tahmin edeceğiniz gibi bu kod işletildiğinde TextBox'a "Adana" yazılır. Tabii TextBox'a "Adana" yerine bu ilin Enum'daki sıra numarasının veya plakasının yazılmasını istemiş olsaydık Parse() metodunu aşağıdaki gibi kullanırdık.

```
byte plaka = (byte)Enum.Parse(typeof(iller), "Adana", true);
textBox1.Text = plaka.ToString();
```

Bu kod aşağıdaki gibi düzenlenirse aynı sonuç alınır.

```
iller il = iller.Adana;
byte plaka = (byte)il;
textBox1.Text = plaka.ToString();
```

Ne var ki bu kullanım bazen pratik olmaz. Örneğin TextBox'a "1" yerine "01" yazmak istenebilir. Bu gibi durumlarda Enum sınıfının **Format**() metodunu kullanmak gerekir. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 iller il = iller.Adana ;
 string str = Enum.Format(typeof(iller), il, "G");
 textBox1.Text =str;
}
```

Bu kodda önce kendi hazırlamış olduğumuz enum tipinde bir değişken tanımlayıp "Adana" adlı elemanı bu değişkene aktardık. Ardından hazırladığımız enum tipindeki (il) değişkeni Format() metoduna 2. parametre olarak verdik.

Format() metodunun 1. parametresi, Parse() metodunda olduğu gibi typeof() operatörü ile elde edilen tiptir. Format() metoduna 3. parametre olarak "G" karakterini verdik. Bu şekilde düzenlenen Format() metodu geriye "Adana" bilgisini gönderir. Format() metodunun geriye söz konusu elemanın enum'daki sırasını veya sayısal değerini geri göndermesini istemiş olsaydık aşağıdaki gibi kullanırdık.

```
iller il = iller.Adana ;
string str = Enum.Format(typeof(iller), il, "D");
textBox1.Text = str;
```

Bu kod işletildiğinde TextBox'a "1" yazılır. TextBox'a 1 yerine "01" yazmak isteseydik **Format**() metoduna 3. parametre olarak "X" karakterini verirdik.

```
iller il = iller.Adana ;
string str = Enum.Format(typeof(iller), il, "X");
textBox1.Text = str;
```

Enum'da Veya Operatörünü Kullanmak

.NET Framework ile gelen bir çok Enum tipinde, içinde bulunulan şartlara göre "veya" durumları söz konusu olabilmektedir. Kendi hazırladığınız Enum'da farklı seçimler yapılsa bile aynı işlemi yaptırtmak isteyebilirsiniz. Örneğin söz konusu kişinin öğrenim durumunu seçerken hazırladığınız sayılabilir tipte, hem lisans hem de yüksek lisans seçeneklerine aynı işlevi yükleyebilirsiniz. Bunu örneklemek için aşağıdaki gibi bir enum tipi hazırladık.

```
public enum OgrenimDurumu :byte
{
 ilk = 1,
 orta = 2,
 lise = 3,
 lisans = 4,
 yuksek_lisans = 5,
}
```

Bu şekilde enum tipi hazırladıktan sonra aşağıdaki gibi kod yazarak ilgili öğrenim durumu seçeneğinin sıra numarasını TextBox'a aktardık.

```
private void Form1_Click(object sender, EventArgs e)
{
 OgrenimDurumu ogrenim = OgrenimDurumu.ilk;
 byte sira = (byte)ogrenim;
 textBox1.Text = sira.ToString();
}
```

Bu kod işletildiğinde TextBox'a 1 yazılır. Öğrenim durumu ilk ve ortaokul olanları aynı grubu dahil etmek istiyorsanız bu kodu aşağıdaki gibi yazabilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 OgrenimDurumu ogrenim =
          OgrenimDurumu.ilk | OgrenimDurumu.orta;
 byte sira = (byte)ogrenim;
 textBox1.Text = sira.ToString();
}
```

Tahmin edeceğiniz gibi bu kod işletildiğinde TextBox'a 3 yazılır. Ne var ki hazırladığımız Enum'da 3 ile öğrenim durumu lise olanlar işaret edilmektedir. Bu nedenle "veya" kullanılacak enum sabitlerinin seçeneklerine sabit değerler verilirken dikkatli olmak gerekir. Bizim örnekte en basit şekli ile öğrenim durumu ilk ile orta olan toplanır ki 3 değeri elde edilir.

Buna göre 3. sıradaki "lise" öğrenim seçeneği için en az 4 değeri seçilmelidir. 4. sıradaki seçeneği temsil edecek sayısal değer ilk üçünün toplamından en azından 1 fazla olmalıdır. Enum aşağıdaki gibi düzenlenirse yukarıda sözünü ettiğimiz sorun yaşanmaz.

```
public enum OgrenimDurumu : byte
{
 ilk = 1,
 orta = 2,
 lise = 4,
 lisans = 8,
 yuksek_lisans = 16,
}
```

Bu şartlarda 2 ortayı, 3 ise "ilk veya ortayı", 5 "ilk veya liseyi", 6 "orta veya liseyi" temsil eder. Bu nedenle aşağıda verilen kod işletilirse TextBox'a 7 yazılır.

```
private void Form1_Click(object sender, EventArgs e)
{
 OgrenimDurumu ogrenim = OgrenimDurumu.ilk |
              OgrenimDurumu.orta | OgrenimDurumu.lise ;
 byte sira = (byte)ogrenim;
 textBox1.Text = sira.ToString();
}
```

Şimdi enum'ların başka bir özelliğinden söz etmek için 4 elemana sahip yeni bir tip hazırladık. Bu enum örneğinde "hepsi" seçeneğinin sıra içeriğini doğrudan yazmak yerine veya operatöründen yararlandık.

```
public enum Diller : byte
{
 ingilizce = 1,
 almanca = 2,
 fransizca = 4,
 italyanca = 8,
 hepsi = ingilizce | almanca | fransizca | italyanca
}
```

Aşağıda verilen kodu dikkatlice incelerseniz "hepsi" elemanın 15 değerini içereceğini tahmin edebilirsiniz. Bu nedenle bu kod işletilirse TextBox'a 15 yazılır.

```
private void Form1_Click(object sender, EventArgs e)
{
 Diller dil = Diller.hepsi;
 byte sira = (byte)dil;
 textBox1.Text = sira.ToString();
}
```

IsDefined() Metodu

Enum sınıfının yukarıda sözü edilen özellik ve metotlarından başka arama yapılırken kullanılan **IsDefined**() adında bir metodu vardır. Bu metot sayesinde sözü edilecek seçeneğin söz konusu Enum'da yer alıp almadığı öğrenilmektedir.

```
private void Form1_Click(object sender, EventArgs e)
{
 bool sonuc = Enum.IsDefined(typeof(OgrenimDurumu), "lise");
 textBox1.Text = sonuc.ToString();
}
```

IsDefined() metodunun ilk parametresi ile aramaya konu edilecek enum tipi işaret edilirken 2. parametrede aranmak istenen seçenek verilmektedir. Yukarıda "OgrenimDurumu" adını verdiğimiz enum'da "lise" adında bir seçenek olduğu için bu kod işletildiğinde **IsDefined**() metodu geriye **true** gönderir.

20. BÖLÜM
HATA YAKALAMAK ve EXCEPTION SINIFLARI

Programcılıkta hata denildiği zaman genellikle çalışma zamanında kullanıcının başrolde olduğu hatalar akla gelmektedir. Örneğin kullanıcıdan kaydını yapmak istediği kişinin doğum tarihinin günü istenir. Kullanıcı 31'den büyük bir değer girer ve programcı bunu öngörüp tedbir almadıysa hata meydana gelir. Bu bölümde çalışma anında karşılaşılması muhtemel hataların nasıl yakalandığından söz edilecektir.

Bu kitapta şimdiye kadar üzerinde durduğumuz konuyu anlatırken verdiğimiz kodların mümkün mertebe kısa olmasını hedefledik. Dolayısıyla çalışma anında karşılaşılması muhtemel hataları göz ardı ettik ve bu hataları yakalamak için alınması gereken tedbirleri almadık. Gerçekte, programcıları en çok yoran konuların başında muhtemel hataları yakalamak ve tedbir almak gelmektedir. Başka bir deyişle, her programcının hata yakalama ve tedbir almayı önemseme gibi bir alışkanlığının olması şarttır.

C# projelerinde hata yakalama işlemi **try-catch-finally** blokları ile yapılmaktadır. Muhtemel hataları yakalama işlemlerinin nasıl yapıldığını örnek bir olay üzerinde anlatmak istiyoruz. Kullanıcıdan bir işlem için tarih istediğinizi varsayalım. Kullanıcı TextBox'a tarih olarak değerlendirilemeyecek bilgi girerse programın çalışması kırılabilir. Böyle bir hataya sebebiyet vermek amacıyla hazırlayıp aşağıda verdiğimiz metot işletildiğinde TextBox'ın içeriğini dönüştürüp **DateTime** tipindeki değişkene aktardık.

```
private void textBox1_Leave(object sender, EventArgs e)
{
 DateTime history;
 history = Convert.ToDateTime(textBox1.Text);
}
```

Kullanıcı TextBox'a tarih veya zaman olarak değerlendirilebilinecek bilgi girdiği sürece bu metot hatasızca çalışır. Ne zaman ki kullanıcı tarih tipine dönüştürülemeyecek bilgiyi TextBox'a girip başka nesnenin üzerine giderse bu metot hata üretir ve projenin çalışması kırılır. Burada yapılması gereken şudur: Hataya neden olma ihtimali yüksek satırı **try** bloğuna almak ve meydana gelmesi muhtemel hatayı **catch** bloğunda yakalamaktır. Bu amaçla yukarıda verilen metodu aşağıdaki gibi düzenledik.

```
private void textBox1_Leave(object sender, EventArgs e)
{
 DateTime history;
 try
  {
   history = Convert.ToDateTime(textBox1.Text);
  }
 catch
 {
  MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
  textBox1.Focus();
 }
}
```

Bu kodda hataya neden olabilme ihtimali olan satırı **try** bloğuna aldık. Ardından **catch** bloğunda öngördüğümüz hata meydana geldiğinde yapılacak işlemler için hazırlık yaptık. **catch** bloğunda kullandığımız **Focus**() metodu sayesinde tekrar ilgili TextBox'a geçilir. Catch bloğuna exception handler (istisnai durum işleyicisi) denilmektedir.

Kullanıcı TextBox'a uygun bilgi girerse hata meydana gelmez ve dolayısıyla **catch** bloğuna yazılan satırlar işletilmez. catch-try bloklarıyla ilgili ayrıntıları ortaya çıkarmak için **Convert** sınıfının **ToDateTime**() metodu ile string bilginin DateTime tipine çevrildiği satırı ayrı bir metot olarak düzenledik.

```
private void textBox1_Leave(object sender, EventArgs e)
{
 DateTime history;
 try
  {
   history = Tarih_yap(textBox1.Text);
  }
 catch
  {
   MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
   textBox1.Focus();
  }
}
public DateTime Tarih_yap(string str)
{
 return Convert.ToDateTime(str);
}
```

Burada **Convert** sınıfının ToDateTime() metodu ile string bilgiyi DateTime bilgiye dönüştüren satır ayrı bir metodun içinde olsa bile Tarih_yap() adını verdiğimiz bu metot **try** bloğunun içinden çağrılıp işletildiği için bu metotta meydana gelecek hatalar yine yakalanır.

Bundan şu sonuç çıkarılabilir: C# uygulamalarının başlangıç noktası **Main**() metodu içinde try-catch bloğu hazırlanırsa, uygulamanın neresinde hata meydana gelirse gelsin yakalanır. Tabi bu önerilen bir yöntem değildir. Doğrusu, her hatayı meydana geldiği yerde yakalayıp gereğini yapmaktır.

```
static void Main()
{
 try
  {
   Application.Run(new Form1());
  }
 catch
  {
   MessageBox.Show("Herhangi bir yerde hata meydana geldi");
  }
}
```

try-catch-finally blokları ile ilgili olarak hemen söylenmesi gereken bir kural var. try bloğundan hemen sonra ya catch ya da finally bloğu gelmelidir. Bu nedenle aşağıdaki gibi düzenlenen try-catch bloğu hataya neden olur.

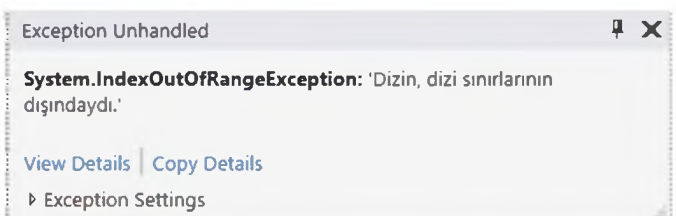
```
private void textBox1_Leave(object sender, EventArgs e)
{
 try
  {
   DateTime tarih = Convert.ToDateTime(textBox1.Text);
  }
 this.Text = "Hata yakalama blokları"; // Hata meydana gelir
 catch
  {
   MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
   textBox1.Focus();
  }
}
```

Hata Yakalama İşlemiyle İlgili 2. Örnek

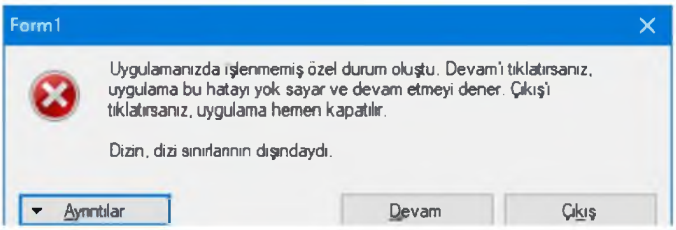
Şimdi hata yakalama konusunda 2. bir örnek vereceğiz. Hazırlayacağımız örnekte 5 elemanlı bir dizi değişken tanımlayıp **for** döngüsü ile bu dizi değişkenin elemanlarına bilgi aktaracağız. Dizi değişkene aktaracağımız bilgileri bir ListBox'tan alacağız.

```
private void Naklet_Click(object sender, EventArgs e)
{
 string[] takimlar = new string[5];
 int adet = listBox1.Items.Count;
 for (int i = 0; i < adet; i++)
   takimlar[i] = listBox1.Items[i].ToString();
}
```

ListBox, 5 veya daha az elemana sahip iken bu kod sorunsuz olarak çalışır. ListBox 5'ten fazla elemana sahip iken bu kod işletildiğinde programın çalışması kırılır ve C# derleyicisi yani debugger aşağıdaki gibi bir hata mesajı verir.



Tahmin edeceğiniz gibi uygulamanın çalışmasının kırılmasını sağlayan entegre hata ayıklayıcıdır. Bu kodu entegre hata ayıklayıcıdan bağımsız olarak çalıştırsaydık uygulamanın çalışması kırılmayıp farklı bir diyalog kutusu ekrana getirilirdi. Aşağıda verilen ekran görüntüsünü bu uygulamayı Visual Studio'nun **Debug** menüsünden **Start Without Debugging** komutunu vererek çalıştırdıktan sonra aldık.



Bu hatanın meydana gelme nedeni dizi değişkenin olmayan 6. elemanına bilgi aktarılmak istenmesidir. Yapılan tanımlamadan dolayı dizi değişken 5 elemana sahip olduğu için hata meydana gelmektedir. Bu gibi hataların önüne geçip programın çalışmasının kırılmasını önlemek için aşağıdaki gibi **try-catch** blokları hazırlanabilir.

```
private void Form1_Click(object sender, EventArgs e)
{
 string[] takimlar = new string[5];
 int adet = listBox1.Items.Count;
 try
  {
   for (int i = 0; i < adet; i++)
   takimlar[i] = listBox1.Items[i].ToString();
  }
 catch
  {
   MessageBox.Show("Dizi Değişkenin sınırı aşıldı\n"+
          "Yalnızca ilk 5 eleman diziye aktarıldı");
  }
 listBox2.Items.AddRange(takimlar);
}
```

Bu kodda hata meydana geleceğini düşündüğümüz satırları **try** bloğuna aldık. Bu satırlarda hata meydana gelmezse kod sorunsuz çalışır ve try bloğundaki satırların tümü işletildikten sonra programın işletimi **catch** bloğundan sonraki ilk satıra geçer. Hata meydana gelirse hatanın meydana geldiği noktada işleme ara verilip **catch** bloğundaki satırlara geçilir.

Yukarıda hazırladığımız örnekte **try** bloğundaki satırlar nasıl bir hataya neden olurlarsa olsunlar catch bloğu işletilir. Bu catch bloğundaki satırların yalnızca dizi değişkenlerle ilgili olarak indis sorunu yaşandığında işletilmesini isteseydik aşağıdaki gibi düzenleme yapardık.

```
private void Form1_Click(object sender, EventArgs e)
{
 string[] takimlar = new string[5];
 int adet = listBox1.Items.Count;
 try
 {
  for (int i = 0; i < adet; i++)
    takimlar[i] = listBox1.Items[i].ToString();
 }
 catch(IndexOutOfRangeException kabahat)
 {
  MessageBox.Show("Meydana gelen hatanın mesajı:\n"+kabahat.Message);
 }
 listBox2.Items.AddRange(takimlar);
}
```

try bloğundaki satırlarda çalışma anında dizi değişkenlerle ilgili bir hata meydana geldiğinde **IndexOfRangeException** sınıfının örneği hazırlanıp catch bloğuna gönderilir. Bu örnekte sistem tarafından fırlatılan istisna nesnesinin veya hazırlanıp catch bloğuna gönderilen **IndexOutOfRangeException** nesnesinin **Message** özelliğinin içeriğini ekrana getirdik.

Bu kodu tekrar yorumlamak gerekirse şunlar söylenebilir: try ve catch blokları birlikte çalışmakta ve try bloğundaki herhangi bir satırda hata meydana geldiğinde bir exception nesnesi hazırlanıp programın işletimi catch bloğuna geçmektedir. Yukarıda verilen kodda catch bloğuna gönderilen nesnenin **IndexOutOfRangeException** tipinde olup olmadığını araştırdık. Gönderilen nesne IndexOutOfRangeException tipinde ise bu bloğa yazılan satırlar işletilir. Konu üzerinde düşünmenizi sağlamak için bu bölümün ilk sayfasında verdiğimiz örneği aşağıdaki gibi düzenledik.

```
private void textBox1_Leave(object sender, EventArgs e)
{
 DateTime history;
 try
  {
   history = Convert.ToDateTime(textBox1.Text);
  }
 catch(IndexOutOfRangeException kabahat)
  {
   MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
   textBox1.Focus();
  }
}
```

Bu şartlarda TextBox'a DateTime tipine dönüştürülemeyecek bilgi girilip TextBox'tan ayrılıp **Leave** olayının meydana gelmesi sağlandığında try bloğundaki satırlar hataya neden olur. Bu hatadan dolayı programın işletimi catch bloğunda geçer. Ne var ki dönüştürme işlemi sonucu meydana gelen hata catch bloğunda yakalanamaz. Çünkü catch deyimine ait parantezin içinde **IndexOutOfRangeException** tipinde değişken tanımlandığı için catch bloğundaki satırlar işletilmeyip meydana gelen hatadan dolayı programın çalışması kırılır.

Eğer catch anahtar kelimesine ait parantezin içinde **Exception** tipinde değişken tanımlasaydık veya parantezlere yer vermeseydik dönüştürme işlemi sonucu meydana gelen hata yakalanırdı. Yani kod aşağıdaki gibi düzenlenmiş olsaydı try bloğunda karşılaşılması muhtemel dönüştürme hatası yakalanırdı.

```
private void textBox1_Leave(object sender, EventArgs e)
{
 DateTime history;
 try
  {
   history = Convert.ToDateTime(textBox1.Text);
  }
 catch(Exception kabahat)
  {
   MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
   textBox1.Focus();
  }
}
```

Sıfıra Bölme Hatasını Yakalamak

Şimdi hata yakalamanın nasıl yapıldığı anlatılırken ilk akla gelen sıfıra bölme hatasını yakalayan bir örnek vereceğiz. Bu amaçla forma 3 TextBox ve 1 düğme yerleştirdik. İlk 2

TextBox'a bilgi girilip "Hesapla" adını verdiğimiz düğme tıklandığında 1. TexttBox'a girilen sayı ikinciye bölünüp sonuç 3. TextBox'a yazılacak. Bu işlemi yapacak kod aşağıda verildi.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 byte number1 = Convert.ToByte(textBox1.Text);
 byte number2 = Convert.ToByte(textBox2.Text);
 float number3 = number1 / number2;
 textBox3.Text = number3.ToString();
}
```

Amaç sıfıra bölme hatasını yakalamayı anlatmak olduğuna göre hata yapılmadan hatanın nasıl yakalanacağından söz etme imkanı olamayacağına göre kullanıcıdan 2 sayı isteyeceğiz. Kullanıcıdan sayı isterken "bu sayıları TextBox'lara girin" dersek yerli yazarların fikirlerini çalmış olabiliriz. "TextBox'a sayı girin" şeklinde özetlenecek bu fikri çalmış olmamak için Visual Basic kaynaklı **InputBox**() metodunu aşağıdaki gibi kullandık. Tabi bu durumda "Microsoft.VisualBasic" adlı referansın Solution Explorer penceresinde yer alması gerekiyor.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 string str1 =
       Microsoft.VisualBasic.Interaction.InputBox("1. sayıyı girin");
 string str2 =
       Microsoft.VisualBasic.Interaction.InputBox("2. sayıyı girin");
 byte number1 = Convert.ToByte(str1);
 byte number2 = Convert.ToByte(str2);
 float number3 = number1 / number2;
 textBox3.Text = number3.ToString();
}
```

Formdaki ilk 2 TextBox'a veya InputBox() metoduyla ekrana getirilen diyalog kutularına 255'ten küçük bir sayı girilirse bu kod hatasızca çalışır. Ancak kullanıcı 2. TextBox'a 0 yazarsa sıfıra bölme hatası meydana gelir ve programın çalışması kırılır. Aynı şekilde TextBox'lara 255'ten büyük bir değer yazılırsa taşma hatası meydana gelir ve programın çalışması yine kırılır.

Burada TextBox'lara girilecek bilgilerin aktarılacağı değişkenlerin tipini **byte** olarak seçmemizin nedeni 255 sayısının herkes tarafından bilinmesi ve büyük bir sayı girmek zorunda kalmadan taşma hatasının meydana gelebilmesine imkan sağlamaktır. Hata meydana geldiği zaman programın çalışmasının kırılmasını engellemek için aşağıda verildiği gibi try-catch bloğu hazırladık.

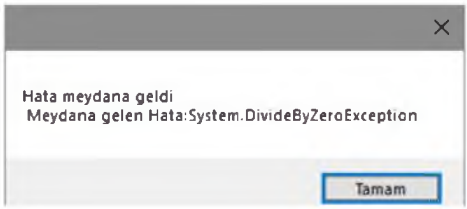
```
private void Hesapla_Click(object sender, EventArgs e)
{
 byte number1, number2;
 float number3;
 try
 {
  number1 = Convert.ToByte(textBox1.Text);
  number2 = Convert.ToByte(textBox2.Text);
```

```
        number3 = number1 / number2;
        textBox3.Text = number3.ToString();
       }
       catch
       {
        MessageBox.Show("Hata meydana geldi");
        textBox1.Focus();
       }
      }
```

Bu koddaki try bloğundaki satırlar işletilirken ister taşma ister sıfıra bölme hatası meydana gelsin her şartta kullanıcıya mesaj verilip tekrar ilk TextBox'ın üzerine gidilir. Konu üzerinde düşünmenizi sağlamak için bu kodun catch bloğunu aşağıdaki gibi düzenledik.

```
catch(Exception kabahat)
{
 Type Tip = kabahat.GetType();
 MessageBox.Show("Hata meydana geldi\n Meydana gelen Hata:"+
                 Tip.ToString());
 textBox1.Focus();
}
```

Burada yapılan şudur: catch anahtar kelimesine ait parantezlerin içinde **Exception** tipinde bir değişken tanımladık. catch deyimi, try bloğundan gönderilen nesnenin tipinin parantezlerin içinde tanımlanan değişkenle aynı olup olmadığına bakar. Gönderilen nesne **Exception** tipinde ise catch bloğundaki satırlar işletilir. Hata nedeniyle catch bloğuna gönderilen nesnenin tipi ister **IndexOutOfRangeException** olsun ister **OverflowException** her şarta catch bloğundaki satırlar işletilir. Çünkü **IndexOutOfRangeException** ve **OverflowException** sınıfları Exception'dan türetilmiştir. Dolayısıyla OverflowException ve IndexOutOfRangeException tipindeki nesneler **Exception** tipindeki değişkene aktarılabilir. Catch bloğundaki satırların işletilmesinin nedeni sıfıra bölme hatası iken aşağıdaki gibi bir mesaj alınır.

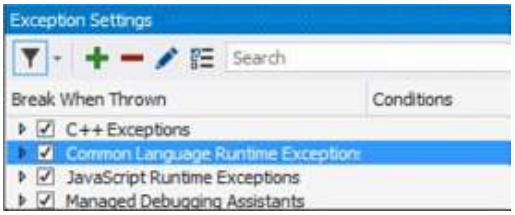


Gördüğünüz gibi bu sırada meydana gelen hata sıfıra bölme işlemi kaynaklı olduğu için catch bloğuna DivideByZeroException sınıfının örneği gönderilmiştir. TextBox'lardan birisine Convert sınıfının **ToByte**() metodu ile byte tipine dönüştürülemeyecek bilgi girip ondan sonra bu kodu işletseydik farklı bir hata mesajı alırdık.

Entegre Hata Ayıklayıcısı ve Exception Sınıfları

En çok kullanılan hata yakalama sınıflarından biraz söz ettikten sonra bu sınıflarla entegre hata ayıklayıcısının ilişkisinden söz edeceğiz. Bu amaçla Visual Stuio'nun **Debug** menüsün-

den **Windows-Exception Settings** komutunu verip ekrana aşağıda verilen diyalog kutusunu getirdik.

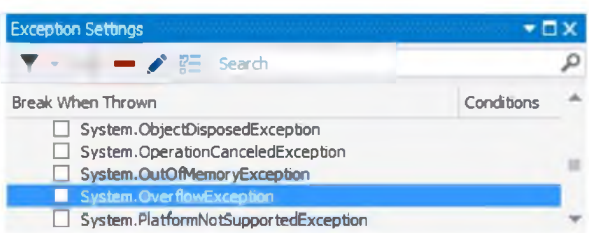


Bu diyalog kutusunda C# uygulamaları dahilinde yararlanabileceğiniz **Exception** sınıfları ile entegre hata ayıklayıcısı arasındaki ilişki ayarlanmaktadır. Bu ayarlamaya örnek olması için forma bir TextBox ve düğme yerleştirip düğme için aşağıda verilen kodu hazırladık.

```
private void Islem_Click(object sender, EventArgs e)
{
 byte number;
 try
 {
  number = Convert.ToByte(textBox1.Text);
 }
 catch
 {
  MessageBox.Show("Hata meydana geldi");
 }
}
```

Tahmin edeceğiniz gibi Debug modunda iken uygulama **Start Debugging** komutu ile çalıştırılıp TextBox'a 255'ten büyük bir değer girilmesi halinde hata meydana gelir ve **OverflowException** sınıfının örneği hazırlanıp catch bloğuna gönderilir. Burada catch anahtar kelimesinden sonra bir şey yazmadığımız için nasıl bir hata ile karşılaşılırsa karşılaşılsın catch bloğunda bu hata yakalanır.

Şimdi öyle bir ayarlama yapacağız ki uygulama Debug modunda iken Start Debbuging komutu ile çalıştırıldığında catch bloğundaki satırların işletilmesi yerine uygulamanın çalışmasının kırılmasını sağlayacağız. Bu amaçla **Exceptions** diyalog kutusunda önce **Common Language Runtime Exceptions** adlı exception sınıfı grubuna ait namespace'lerin listelenmesini sağladık. Ardından **System**'de yer alan **OverflowException** sınıfını bulduk.



Verilen ekran görüntüsünde tespit edebileceğiniz gibi bu sırada **Break When Thrown** özelliği pasif durumdadır. Başka bir deyişle kodun herhangi bir yerinde OverflowException

sınıfı tipinde bir nesnenin fırlatılmasına neden olacak hata "yakalanmazsa" programın çalışması kırılır. Yukarıda verilen kodda TextBox'a 255'ten büyük bir değer girilirse (byte değişkene 255'ten büyük bir sayı aktarılmak istenirse) meydana gelebilecek hata yakalandığı için uygulamanın çalışması kırılmaz.

Başka bir deyişle mevcut ayarlarda byte değişkene 255'ten büyük bir değer aktarılmak istenirse ve OverflowException hatasını yakalayan bir catch bloğu yoksa uygulamanın çalışması kırılır. Benzer şekilde .NET Framework ile gelen sınıfların hemen hepsi için yakalanmayan bütün hatalar Debug modunda iken çalışmanın kırılmasına neden olur.

Bu diyalog kutusunda System'de yer alan OverflowException sınıfının **Break When Thrown** onay kutusunu aktif duruma getirdikten sonra OK düğmesi ile bu diyalog kutusunu kapattık. Ardından yukarıda verilen kodu aşağıdaki gibi düzenledik.

```
private void islem_Click(object sender, EventArgs e)
{
 byte number;
 try
  {
   number = Convert.ToByte(textBox1.Text);
   if (number == 100)
      throw new OverflowException();
  }
 catch
  {
   MessageBox.Show("Hata meydana geldi veya 100 sayısını girdiniz");
  }
}
```

Bu şartlarda TextBox'a 255'ten büyük bir değer girilip bu kod işletilirse **ToByte**() metodunun kullanıldığı satır hataya neden olur ve Catch bloğu hazırlanmış olmasına rağmen uygulamanın çalışması kırılır. Çünkü Exception Settings diyalog kutusunda bu sınıf seçili iken **Break When Thrown** onay kutusunu aktif duruma getirdik

TextBox'a 100 girilip bu kod işletildiğinde ise try bloğundaki karşılaştırma doğru olacağı için OverflowException sınıfının örneği hazırlanıp catch bloğuna gönderilir ve uygulamanın çalışması yine kırılır. Başka bir anlatımla bu şartlarda yani **Exception Settings** diyalog kutusunda OverflowException sınıfının **Break When Thrown** özelliği seçili durumda iken Debug modunda uygulama **Start Debugging** komutu ile çalıştırıldığında TextBox'a ister 100 girilsin ister 255'ten büyük sayı uygulamanın çalışması kırılır. Çünkü Exception Settings diyalog kutusunda yaptığımız ayarlamanın anlamı şudur: Her ne nedenden olursa olsun ister direk **Throw** deyimi ile, ister yapılmak istenen işlem **OverflowException** hatasını üretirse uygulamanın çalışması kırılır.

Uygulamayı Build menüsündeki komutlarla derleyip sonra klasik yöntemlerle (EXE dosyanın olduğu klasöre gidip EXE dosyayı çift tıklayarak) Visual Studio'dan bağımsız olarak çalıştırırsanız TextBox'a ister 255'ten büyük sayı girilsin ister 100 girilsin catch bloğundaki satırlar işletilir ve uygulamanın çalışması kırılmaz. Visual Studio ile uygulamanızı geliştirirken bazı hataların yakalanmayıp uygulamanın çalışmasını kırılmasına neden olmasını istiyorsanız yukarıda yaptığımız gibi Exception Settings diyalog kutusunda ayarlama yapabilirsiniz.

Finally Bloğu

Bazen herhangi bir nedenden dolayı yarı yolda vazgeçilen bir işlemden dolayı geride bazı kalıntılar kalabilir. Bu konu üzerine konuşabilmek için üzerinde çalıştığımız uygulamada aşağıdaki gibi kod yazarak bir **FileStream** nesnesi hazırladık ve diskteki bir dosyanın içeriğini okuyup formdaki RichTextBox nesnesine aktarmayı denedik.

```
private void button1_Click(object sender, EventArgs e)
{
 try
  {
   string yol = "C:\\Projeler\\Nakip_ali.txt";
   FileStream akis = new FileStream(yol, FileMode.Open,
                     FileAccess.Read);
   StreamReader okuyucu = new StreamReader(akis);
   while (true)
   {
    string satir = okuyucu.ReadLine();
    if (satir == null)
      break;
    richTextBox1.Text = richTextBox1.Text + satir + '\r' + '\n';
    throw new Exception();
   }
   akis.Close();
  }
 catch (Exception)
  {
   MessageBox.Show("Hata meydana geldi");
  }
}
```

Ancak ilgili text dosyasından bir satırı okuyup RichTextBox'a aktardıktan sonra **throw** anahtar kelimesinden yararlanıp kendimiz ortama Exception nesnesi fırlattık. Dolayısıyla bu kodda FileStream nesnesinin **Close**() metoduyla kapatıldığı satır işletilmeden programın işletimi catch bloğuna ve oradan da ana forma geçer. Tam bu sırada bu text dosyasını açıp içeriğini değiştirip yeni halini kaydetmek isterseniz Windows buna izin vermez.

Çünkü bu sırada FileStream nesnesi text dosyamızı bloke etmeye devam ediyordu. Yukarda verdiğimiz kodda **finally** bloğuna yer verip FileStream nesnesini kapatma işlemini finally bloğunda yapsaydık böyle bir sorunla karşılaşmazdık. Aşağıda verilen kodda FileSteam nesnesini hem try hem de finally bloğunda kullanabilmek için değişken tanımlama satırını class bloğuna aldık.

```
FileStream akis;
private void button1_Click(object sender, EventArgs e)
{
 try
  {
   string yol = "C:\\Projeler\\Nakip_ali.txt";
   akis = new FileStream(yol, FileMode.Open, FileAccess.Read);
   StreamReader okuyucu = new StreamReader(akis);
   while (true)
   {
    string satir = okuyucu.ReadLine();
```

```
        if (satir == null)
          break;
        richTextBox1.Text = richTextBox1.Text + satir + '\r' + '\n';
        throw new Exception();
       }
     }
     catch (Exception)
     {
      MessageBox.Show("Hata meydana geldi");
     }
     finally
     {
      akis.Close();
     }
    }
```

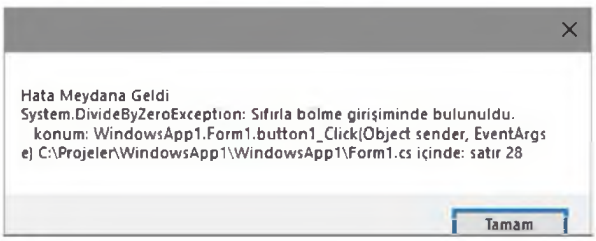
İstenirse birden fazla **catch** bloğu ve/veya iç-içe **try-catch** blokları hazırlanabilir. Söz konusu hatanın birden fazla nedenden kaynaklanması ihtimali varsa try bloğu için birden fazla catch hazırlanabilir.

Hata Yakalama Sınıfları

Yukarıdaki sayfalarda Exception, OverflowException, DivideByZeroException ve IndexOutOfRangeException sınıflarından söz edildi. Hata yakalama sınıflarının işlevleri biraz zor anlaşıldığı için yukarıda kısaca anlattığımız konulardan burada tekrar söz edeceğiz. Öncelikle aşağıda verilen kodu incelemenizi istiyoruz.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 try
  {
  byte Number1 = Convert.ToByte(textBox1.Text);
  byte Number2 = Convert.ToByte(textBox2.Text);
  float Number3 = Number1 / Number2;
  textBox3.Text = Number3.ToString();
 }
 catch(System.Exception kabahat)
 {
   MessageBox.Show("Hata Meydana Geldi" + "\n\r" + kabahat);
   textBox1.Focus();
  }
}
```

Formdaki 2. TextBox nesnesine sıfır (0) girilip bu kod işletilirse sıfıra bölme hatası meydana gelir (yani catch bloğuna **DivideByZeroException** nesnesi gönderilir) ve meydana gelen hata ile ilgili olarak ekrana aşağıdaki gibi bir hata mesajı getirilir.



Bu kodda yukarıdaki sayfalarda verilen örneklerden farklı olarak "kabahat" adını verdiğimiz bir nesne var. Bildiğiniz gibi nesneleri **new** anahtar kelimesini kullanıp ilgili sınıfın yapıcı metodundan yararlanarak hazırlanıyorduk. Ancak burada "kabahat" adını verdiğimiz **System.Exception** tipindeki nesne otomatik olarak hazırlandı. Bu kodda catch bloğuna DivideByZeroException nesnesi gönderilmesine ve catch deyimine ait parantezlerin içinde **System.Exception** tipinde değişken tanımlanmasına rağmen hata yakalandı.

Nasıl ki Object tipindeki değişkenlere istenen tipteki bilgiler aktarılabiliniyorsa bütün hata sınıflarının türetildiği **Exception** sınıfı tipindeki değişkene istenen hata sınıfının örneği aktarılabilinmektedir. try bloğundan catch bloğuna gönderilen DivideByZeroException nesnesi hatasızca dönüştürülüp Exception tipindeki değişkene aktarılabildiği için hata yakalanıp catch bloğundaki satırlar işletilebilmektedir.

Şimdi yukarıda verilen kodun **try** bloğunda değişiklik yapıp sıfıra bölme hatası meydana geldiğinde başka mesaj, TextBox'a byte tipindeki değişkene sığmayacak büyüklükte bir değer girildiğinde başka bir mesajın verilmesini sağlayacağız. Aşağıda verilen kodu incelerseniz bu 2 catch bloğunda catch deyimlerine ait parantezlerin arasına sınıfların adını yazmakla yetinip değişken tanımlamadık. try bloğundan gönderilen nesneyi catch bloğu içinde kullanmayacaksanız parantezlerin arasına ilgili sınıfın adını yazmanız yeterlidir.

```
catch(DivideByZeroException)
{
 MessageBox.Show("Sıfıra bölme hatası yaptınız");
 textBox1.Focus();
}
catch(OverflowException)
{
 MessageBox.Show("Büyük sayı girdiğiniz");
 textBox1.Focus();
}
```

Hazırladığımız kodun catch bloklarını bu şekilde değiştirip formdaki TextBox'lardan birisine karaktersel bilgi girip bu kodu işletirsek meydana gelecek hata yakalanamaz ve programın çalışması kırılır. Çünkü bu sırada **Convert** sınıfının **ToByte()** metodunun kullanıldığı satırlar hataya neden olurken ortama **FormatException** sınıfının örneği fırlatılır. Convert sınıfı ile string bilgiler byte tipine dönüştürmeye çalışılırken meydana gelebilecek hataları yakalamak istiyorsanız 3. bir catch bloğu hazırlayabilirsiniz.

```
byte Number1 = Convert.ToByte(textBox1.Text);
byte Number2 = Convert.ToByte(textBox2.Text);
```

Aşağıdaki gibi 3 catch bloğu hazırlayıp hem ilk TextBox'a byte tipindeki değişkene aktarılamayacak büyüklükte bir sayı girip hem de 2. TextyBox'a 0 yazıp sıfıra bölme hatasının meydana gelmesini sağlarsanız ilk TextBox'a yazılan bilgi byte değişkene aktarılırken hata meydana geleceği için catch bloğuna **OverflowException** nesnesi gönderilir.

```
catch(DivideByZeroException)
{
 MessageBox.Show("Sıfıra bölme hatası yaptınız");
 textBox1.Focus();
}
```

```
catch(OverflowException)
{
 MessageBox.Show("Büyük sayı girdiğiniz");
 textBox1.Focus();
}
catch (FormatException)
{
 MessageBox.Show("Yanlış türde bilgi girdiniz");
 textBox1.Focus();
}
```

Bu şekilde hazırlanan catch blokları sayesinde try bloğunda meydana gelecek 3 farklı hata yakalanır. Yukarıda verdiğimiz kodda try bloğunda yaptığımız işlem basit olduğu için bu 3 hatadan farklı bir hatanın meydana gelmesi ihtimali düşüktür. Buna rağmen programcılar meydana gelmesi muhtemel hatayı catch bloğunda yakalamayı garanti etmek için bir catch bloğu daha hazırlarlar. Bu düşünce ile yukarıda verdiğimiz kodu aşağıdaki gibi düzenledik.

```
private void Hesapla_Click(object sender, EventArgs e)
{
  byte number1, number2;
  float number3;
  try
   {
    number1 = Convert.ToByte(textBox1.Text);
    number2 = Convert.ToByte(textBox2.Text);
    number3 = number1 / number2;
    textBox3.Text = number3.ToString();
   }
   catch (DivideByZeroException)
   {
    MessageBox.Show("Sıfıra bölme hatası yaptınız");
    textBox1.Focus();
   }
   catch (OverflowException)
   {
    MessageBox.Show("Büyük sayı girdiğiniz");
    textBox1.Focus();
   }
   catch (FormatException)
   {
    MessageBox.Show("Yanlış türde bilgi girdiniz");
    textBox1.Focus();
   }
   catch // veya catch (Exception)
   {
    MessageBox.Show("Hata meydana geldi.");
    textBox1.Focus();
   }
}
```

Burada dikkat edilmesi gereken önemli nokta, her türlü hatayı yakalayacak catch bloğunun en sona yazılması gerektiğidir. Yoksa sıfıra bölme ve dönüştürme hatalarını yakalamak amacıyla hazırladığımız catch bloklarını yazmanın bir anlamı olmazdı.

FileNotFoundException Sınıfı

Hata yakalama konusuna devam etmek üzere forma bir **PictureBox** yerleştirip aşağıda verilen kodu hazırladık. Bu kodda önce "Resim" adını verdiğimiz **Image** nesnesi için değişken tanımlanmaktadır. Ardından **FromFile**() metodu ile ilgili BMP dosyası okunup **Image** nesnesi hazırlanıp bu nesnenin referansı formdaki PictureBox'a aktarılmaktadır.

```
System.Drawing.Image Resim;
Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
pictureBox1.Image = Resim;
```

Adı **FromFile**() metoduna parametre olarak verilen dosya bulunmazsa veya içeriği Image nesnesine aktarılmaya uygun olmayan bir dosyanın adı parametre olarak verilirse hata meydana gelir. Kodu aşağıdaki gibi düzenlemiş olsaydık bu hata yakalanırdı.

```
System.Drawing.Image Resim;
try
{
 Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
 pictureBox1.Image = Resim;
}
catch(System.IO.FileNotFoundException)
{
 MessageBox.Show("Dosya bulunamadı");
}
```

Bu kodda kullanılan **FileNotFoundException** sınıfı şimdiye kadar hakkında bilgi verilen sınıflardan farklı olarak **System.IO**'da tanımlıdır. Meydana gelmesi muhtemel hata ile ilgili hangi tip nesnenin oluşturulacağı tahmin edilemiyorsa **Exception** sınıfı kullanılabilir.

```
System.Drawing.Image Resim;
try
{
 Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
 pictureBox1.Image = Resim;
}
catch(System.Exception)
{
 MessageBox.Show("Hata meydana geldi");
}
```

Bu şartlarda hangi nedenden kaynaklanırsa kaynaklansın herhangi bir hata meydana geldiğinde try-catch bloğundaki **catch(System.Exception)** karşılaştırması doğru değeri vereceğinden bloklanan satırlar işletilir. Bu durumda Exception nesnesini inceleyip meydana gelen hatanın izini sürebilirsiniz. Bu konuda bilgi vermek için yukarıda verilen kodu aşağıdaki gibi düzenledik. Hata meydana geldiğinde bu catch bloğunda hazırlanan **Exception** nesnesinin hangi sınıftan türetildiği öğrenilip ona göre işlem yapılabilir.

```
catch(System.Exception kabahat)
{
 MessageBox.Show("Hata meydana geldi" + "\n\r" +
   "Orijinal Hata mesajı:"+ Convert.ToString(kabahat.GetType()));
}
```

InvalidCastException Sınıfı

Daha önceki konulardan bildiğiniz gibi değişik tipteki bilgileri başka bir tipe dönüştürmek mümkündür. Tip dönüştürme işlemleri sırasında sıklıkla **InvalidCastException** hatasıyla karşılaşılabilmektedir. Bu sınıf hakkında bilgi vermek için aşağıda verilen kodu hazırladık.

```
object nesne = 2017;
try
{
 int yil = (int)nesne;
 this.Text = yil.ToString();
}
catch (System.InvalidCastException hata)
{
  MessageBox.Show("Dönüştürme hatası meydana geldi" +
              n\r" + hata.Message);
}
```

Bu kodda "nesne" adını verdiğimiz **object** tipindeki değişkenin içeriği rakamlardan meydana geldiği için **Unboxing** işlemi yapılıp sorunsuz bir şekilde **int** tipindeki değişkene aktarılır. Object tipindeki değişkenin içeriği int tipindeki değişkene aktarılırken Unboxing işlemi başarısız olsaydı **InvalidCastException** hatası ile karşılaşılırdı.

Null yapılmış bir nesne üzerinde işlem yapılmak istenirse **NullReferenceException** hatası ile karşılaşılır. Bu hata ile nasıl karşılaşıldığını göstermek için bir Bitmap nesnesi hazırladık. Ardından bu nesneyi null yapıp **Width** ve **Height** özelliklerini kullanmayı denedik.

```
System.Drawing.Bitmap Resim;
openFileDialog1.ShowDialog();
string Dosya = openFileDialog1.FileName;
try
{
 Resim = new System.Drawing.Bitmap(Dosya);
 Resim = null;
 int i = Resim.Width;
 int j = Resim.Height;
}
catch(NullReferenceException)
{
 MessageBox.Show("Nesne null yapılmış");
}
```

Hata Yakalama Filtreleri

Bu bölümde şimdiye kadar anlatılanlara göre hata yakalama mekanizması şöyledir: try bloğunda yapılan işlemler sonucu bir hata meydana geldiğinde meydana gelen hatanın türüyle ilgili bir nesne hazırlanıp catch bloğuna gönderilir. Programcı bu hatayı yakalayıp gereğini yapmaktaydı. Konu üzerinde düşünmenizi sağlamak için yukarıdaki sayfalarda verdiğimiz ve try-catch bloklarında yapılan hata yakalama işlemlerini anlatmak üzere hazırladığımız kodu biraz değiştirip aşağıda tekrar verdik.

```
private void button3_Click(object sender, EventArgs e)
{
  byte number1, number2;
  float number3;
  try
  {
   number1 = Convert.ToByte(textBox1.Text);
   number2 = Convert.ToByte(textBox2.Text);
   number3 = number1 / number2;
   textBox3.Text = number3.ToString();
  }
  catch (Exception hata)
  {
   if (hata is DivideByZeroException)
   {
    MessageBox.Show("Sıfıra bölme hatası yaptınız");
    textBox1.Focus();
   }
   if (hata is OverflowException)
   {
     MessageBox.Show("Büyük sayı girdiniz");
     textBox1.Focus();
    }
   if (hata is FormatException)
   {
    MessageBox.Show("Yanlış türde bilgi girdiniz");
    textBox1.Focus();
   }
  }
}
```

Bu koddaki catch bloğunu dikkatlice incelediğinizde yazılan kodun işlev bakımından yukarıdaki sayfalarda verilenle aynı olduğunu, ancak yaklaşımın farklı olduğunu görürsünüz. Bu koddan şu sonucu çıkarmak mümkündür: Programcı catch bloklarında istediği kontrolleri yapıp istediği işlemleri yapabilir. C# 6.0 ile birlikte programcının hata kontrol blokları ile yapabilecekleri konusunda eli güçlendirildi. Exception filters adı verilen imkandan söz edebilmek için yukarıda verdiğimiz catch bloklarını aşağıdaki gibi düzenledik. Pratik bir değerinin olmadığı bu kodu dikkatlice incelerseniz C# 6.0 ile try-catch bloklarına getirilen yeniliği net olarak görebilirsiniz.

```
byte number1, number2;
float number3;
private void button3_Click(object sender, EventArgs e)
{
 try
  {
   number1 = Convert.ToByte(textBox1.Text);
   number2 = Convert.ToByte(textBox2.Text);
   number3 = number1 / number2;
   throw new Exception();
  }
  catch when (number1 == 100)
  {
   MessageBox.Show("ilk sayı 100");
   textBox1.Focus();
  }
```

```
    catch when (number2 == 0)
    {
     MessageBox.Show("Sıfıra bölme hatası");
     textBox1.Focus();
    }
    catch
    {
     MessageBox.Show("Herhangi bir hata");
     textBox1.Focus();
    }
}
```

Bu try bloğundaki ilk satırlardan başlayarak inceleyip üzerinde konuşacağız.

- Tahmin edeceğiniz gibi formdaki ilk 2 TextBox'a byte tipine dönüşemeyecek bilgi girilirse Convert sınıfının ToByte() metodu ile dönüştürme yapılırken FormatException veya OverFlowException hatası meydana gelir. Bu 2 hata nesnesi ortama gönderildiğinde en son catch bloğunda yakalanır.
- 2. TextBox'a sıfır girilip ondan sonra bu kod işletilirse bölme işleminin yapıldığı satırda DivideByZeroException nesnesi hazırlanır ve 2. catch bloğunda bu nesne yakalanır. Çünkü 2. catch bloğunda **when** anahtar kelimesi ile "number2" adını verdiğimiz sayının 0 olup olmadığı araştırılmaktadır.
- try bloğundaki ilk 3 satırda herhangi bir hata meydana gelmezse **throw** anahtar kelimesi bir Exception nesnesi hazırlanır. Bu sırada kullanıcı ilk TextBox'a 100 sayısını girmişse bu Exception nesnesi ilk catch bloğunda yakalanır.

Throw Deyimi İle Exception Nesnesi Hazırlamak

Şimdiye kadar anlatılanlara göre herhangi bir nedenden dolayı çalışma anında bir hata meydana geldiğinde uygulama tarafından karşılaşılan hataya uygun olarak ilgili hata sınıfının örneği hazırlanıp catch bloğuna gönderilmektedir. Bizim yaptığımız, catch bloğuna gönderilen nesnenin hangi sınıfın örneği olduğunu öğrenip ona göre kullanıcıya mesaj verip tedbir almaktır. Burada asıl vurgulamak istediğimiz konu, uygulama bir exception nesnesi fırlattığında bu exception'a kaynaklık edecek sınıfı kendisinin seçtiğidir.

Şimdi throw deyiminden yararlanıp kendimiz bir **Exception** nesnesi hazırlayacağız. Bu amaçla kullanıcıdan gün, ay ve yıl bilgilerini isteyip bunları birleştirip tarih bilgisi elde edeceğiz. Bu amaçla forma 4 TextBox yerleştirip aşağıda verilen kodu hazırladık.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 int gun = Convert.ToInt16(textBox1.Text);
 int ay = Convert.ToInt16(textBox2.Text);
 int yil = Convert.ToInt16(textBox3.Text);
 DateTime t =Microsoft.VisualBasic.DateAndTime.DateSerial(yil,ay,gun);
 textBox4.Text = history.ToShortDateString();
}
```

TextBox'lara uygun bilgi girildiği sürece bu kod sorunsuz çalışır. Gün niyetine 31'den, ay bilgisi olarak 12'den büyük değer girilip gün, ay ve yıl bilgileri birleştirilip tarih elde edilmek istendiğinde hata meydana gelir. Yanlış bilgi girişinden dolayı programın kırılmasını önlemek

için try-catch bloğu hazırlayabilir veya kullanıcının girdiği bilgileri Visual Basic kaynaklı **DateSerial**() metoduna parametre olarak vermeden önce kendiniz kontrol edebilirsiniz. Pratik değeri olmasa bile aşağıda verilen kod sorunsuz çalışır ve DateSerial() metoduna parametre olarak verilecek ay ve gün bilgilerinin yanlış olması engellenir.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 int gun = Convert.ToInt16(textBox1.Text);
 int ay = Convert.ToInt16(textBox2.Text);
 int yil = Convert.ToInt16(textBox3.Text);
 if (gun > 31 || ay > 12)
  try
  {
   throw new OverflowException("Yanlış bilgi girdiniz");
  }
  catch (Exception err)
  {
   MessageBox.Show(err.Message);
   textBox1.Focus();
  }
 DateTime t=Microsoft.VisualBasic.DateAndTime.DateSerial(yil,ay,gun);
 textBox4.Text = t.ToShortDateString();
}
```

Kullanıcı ay veya gün bilgilerinde sınırı aştığında catch bloğu işletilir. try bloğunda şimdiye kadar yapılanlardan farklı olarak **throw** anahtar kelimesi ile **OverflowException** nesnesi hazırlanmaktadır. throw ile **Exception** nesnesi hazırlandığında kodun işletimi catch bloğuna geçmektedir.

catch bloğunda ise kullanıcıya mesaj verildikten sonra programın işletimi başa alınıp kullanıcıya girdiği bilgileri düzeltme şansı veriliyor. **throw** deyimi ile **OverflowException** nesnesi yerine başka bir exception nesnesi hazırlayabilirdik. Hemen hatırlatmak gerekirse, throw deyimi ilgili exception sınıfının örneğini almaya daha çok kendi özel hata sınıflarını hazırlayan programcılar gerek duymaktadır.

Şimdi throw deyimi ile ilgili 2. bir örnek vereceğiz. Bu amaçla Bol() adında ve float tipinde 2 parametreye sahip bir metot hazırladık. Bu metotta birbirine bölünecek sayılar TextBox'lardan alınacaktır.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 float bolunen, bolen, sonuc;
 bolunen = Convert.ToSingle(textBox1.Text);
 bolen = Convert.ToSingle(textBox2.Text);
 if (bolen == 0)
  try
   {
    throw new DivideByZeroException("Sıfıra bölme hatası");
   }
  catch (DivideByZeroException Hata)
   {
    MessageBox.Show(Hata.Message);
    textBox2.Focus();
   }
```

```
 else
 {
  sonuc = Bol(bolunen, bolen);
  textBox3.Text = sonuc.ToString();
 }
}
float Bol(float number1, float number2)
{
 return number1 / number2;
}
```

İlk 2 TextBox'a yazılan bilgileri float tipindeki değişkenlere aktardıktan sonra bölme işlemini yapmadan 2. sayının yani bölenin 0 olup olmadığını araştırdık. İkinci sayı 0 ise **throw** deyimi ile **DivideByZeroException** sınıfının örneğini alıp programın işletiminin catch bloğuna geçmesini sağladık.

Tabii bu kodda TextBox'lara girilen bilgiler **Convert** sınıfının **ToSingle**() metodu ile float tipe dönüştürülmeye uygun değilse başka hatalar meydana gelir. Hazırladığımız bu kodda catch bloğunda farklı bir exception sınıfının adını yazsaydık veya DivideByZeroException dışında başka bir sınıfın örneğini hazırlasaydık sıfıra bölme hatası yakalanamazdı. Şimdi farklılık olsun diye try-catch bloğu ile **throw** deyiminin farklı yerlerde olmasını sağlayacağız.

```
private void Hesapla_Click(object sender, EventArgs e)
{
 float bolunen, bolen, sonuc;
 bolunen = Convert.ToSingle(textBox1.Text);
 bolen = Convert.ToSingle(textBox2.Text);
 try
  {
   sonuc = Bol(bolunen, bolen);
   textBox3.Text = sonuc.ToString();
  }
 catch (DivideByZeroException Hata)
  {
   MessageBox.Show(Hata.Message);
   textBox2.Focus();
  }
 }
float Bol(float number1, float number2)
 {
  if (number2 == 0)
   throw new DivideByZeroException("Sıfıra bölme hatası");
  else
   return number1 / number2;
}
```

Bu kodda throw anahtar kelimesi ile oluşturduğumuz DivideByZeroException nesnesinin yalnızca **Message** özelliğiyle ilgilendik. DivideByZeroException ve diğer Exception sınıflarının ayrıca **Source**, **StackTrace**, **TargetSite**, **InnerException** özellikleri bulunmaktadır. Örneğin Source özelliğinde hatanın meydana geldiği assembly'in adı tutulmaktadır.

Hataya neden olan metotlarla ilgileniyorsanız **StackTrace** özelliğine bakabilirsiniz. Hataya neden olan metotları araştırırken System.Diagnostics'deki **StackTrace** sınıfından yararlanabilirsiniz. Bu sınıfın nasıl kullanıldığını aşağıda görebilirsiniz.

```
StackTrace izle = new StackTrace(Hata);
foreach (StackFrame fr in izle.GetFrames())
{
 listBox1.Items.Add(fr.GetMethod());
}
```

Exception Sınıfı Hazırlamak

Hata sınıfınızı kendiniz hazırlamak istiyorsanız sınıfınız **ApplicationException** veya **Exception** sınıfının mirasçısı olmalıdır. Programcılar kendi exception sınıflarını özellikle veritabanlarından kaynaklanan hatalar söz konusu olduğunda hazırlama gereğini duyuyorlar. Bu konuda adım adım bilgi vermek aşağıdaki gibi sınırlı özelliklere sahip bir sınıf hazırladık.

```
public partial class Hata_sinifim : Exception
{
 public Hata_sinifim()
 {
 }
}
```

Bu sınıfın yapıcı metodu herhangi bir parametre ve satıra sahip değildir. Bu şekilde kendi exception sınıfınızı hazırladıktan sonra herhangi bir uygulama dahilinde kullanabilirsiniz. Bu konuda bilgi vermek için forma bir TextBox ve "Tamam" adını verdiğimiz bir düğme yerleştirip aşağıda verilen kodu hazırladık.

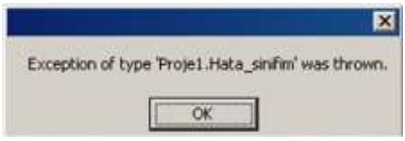
```
private void Tamam_Click(object sender, EventArgs e)
{
 if (textBox1.Text == "12345")
   MessageBox.Show("Sisteme giriş yaptınız");
 else
   throw new Hata_sinifim();
}
```

Kullanıcı çalışma anında TextBox'a şifresini girip Tamam adını verdiğimiz düğmeyi tıklayıp bu kodu işletince TextBox'a yazılan bilginin "12345" olup olmadığı araştırılıyor. Girilen bilgi 12345 değilse **throw** anahtar kelimesi ile kendi hazırladığımız exception sınıfının örneği alınıp ortama fırlatılmaktadır. Kendi hazırladığımız exception sınıfı, mirasçısı olduğu sınıftan farklı olmadığı için TextBox'a 12345'ten farklı bilgi girilirse program kırılır. "Tamam" düğmesi için yazılan kodda try-catch bloğuna yer vermiş olsaydık programın çalışması kırılmazdı. Bu amaçla yukarıda verilen kodu aşağıdaki gibi düzenledik.

```
private void Tamam_Click(object sender, EventArgs e)
{
 try
  {
   if (textBox1.Text == "12345")
     MessageBox.Show("Sisteme giriş yaptınız");
   else
     throw new Hata_sinifim();
  }
 catch (Hata_sinifim hata)
  {
   MessageBox.Show(hata.Message);
  }
}
```

TextBox'a 12345'ten farklı bir değer girilip bu kod işletilirse catch bloğundaki satır sayesinde ekrana aşağıdaki gibi hata mesajı getirilirdi. Çünkü try bloğunda kullanıcının TextBox'a şifre niyetine girmiş olduğu bilginin "12345" olup olmadığını araştırdık.

Kullanıcı TextBox'a 12345 dışında başka bilgi girdiyse throw anahtar kelimesi ile kendi hazırladığımız sınıfın örneğini aldık. Başka bir deyişle hatalı bir durumu ihbar edip programın akışının catch bloğuna geçmesini sağladık. catch bloğunda ise throw anahtar kelimesi ile hazırlanan exception nesnesinin orijinal mesajını ekrana getirdik.



Kendi hazırladığınız sınıfla ilgili olarak ekrana kendi hata mesajınızı getirmek istiyorsanız sınıfın yapıcı metodunu aşağıdaki gibi değiştirebilirsiniz. Yaptığımız değişiklik sayesinde hazırladığımız sınıfın yapıcı metodunun string tipte bir parametreye sahip olmasını sağladık.

```
public Hata_sinifim(string msg) : base(msg)
 {
 }
```

Kendi hazırladığınız exception sınıfının yapıcı metodunu bu şekilde değiştirdikten sonra **throw** anahtar kelimesi ile Exception nesnesini fırlatırken sınıfın yapıcı metoduna aşağıdaki gibi mesaj olarak kullanmak istediğiniz metni parametre olarak vermelisiniz.

```
throw new Hata_sinifim("Yanlış Şifre Girdiniz");
```

Yukarıda kendi hazırladığımız hata yakalama sınıfına taban sınıf olarak **Exception** sınıfını seçtik. Kendi hazırladığınız istisnai durum sınıfları için Exception yerine **ApplicationException** sınıfını başlangıç noktası olarak seçerseniz işiniz kolaylaşacaktır.

Bu konunun üzerinde biraz durabilmek için şimdi ikinci bir Exception sınıfı hazırlayacağız. Bu sınıf 4 yapıcı metoda sahip olacak ve **ApplicationException** sınıfını mirasçısı olacaktır. Kendi hazırladığımız sınıfta 4 yapıcı metoda yer verme nedeni taban sınıf olarak seçtiğimiz ApplicationException sınıfın 4 yapıcı metoda sahip olmasıdır.

```
public partial class Hata_sinifim : ApplicationException
{
 private string mesajim;
 public string ozellik_mesajim
  {
   get { return mesajim; }
   set { mesajim = value; }
  }
 public Hata_sinifim() : base()
  {
  }
 public Hata_sinifim(string mesaj) : base(mesaj)
  {
  }
```

```
public Hata_sinifim(string mesaj,
            Exception inner) :base(mesaj, inner)
 {
 }
public Hata_sinifim(string mesaj, string mesajim) : base(mesaj)
{
  ozellik_mesajim = mesajim;
}
}
```

Ilk bu şekilde düzenlediğimiz sınıfın 2. parametresi Exception tipinde olan yapıcı metottan söz edeceğiz. Yukarıdaki sayfalarda Exception nesnesinin **InnerException** özelliğinin adı anılmıştı. Bu özellik catch bloğuna yazılan kodlar hataya neden olduğunda işlevsel olmaktadır. Bu konuda bilgi vermek için üzerinde çalıştığımız projede bu sınıftan aşağıdaki gibi yararlanalım. Tabii bu örneğin pratik bir değerinin olmadığını baştan söylemek gerek. InnerException özelliğinden söz etmek için aşağıdaki gibi 2. sınıf hazırladık.

```
public class Test_sinifi
{
public void Hatayi_uret()
 {
  try
   {
    this.Inner_uret();
   }
  catch (Exception H)
   {
    throw new Hata_sinifim("Normal hatamız", H);
   }
  }
 public void Inner_uret()
 {
  throw new Hata_sinifim("inner(önceki) hatamız");
 }
}
```

Oldukça basit olan bu sınıfı incelerseniz yapıcı metoda sahip olmayıp Hatayi_uret() adını verdiğimiz metotta try-catch bloğu bulunmaktadır. Bu örnekte işlevsellik kaygımız olsaydı try-catch bloğunda hataya neden olma ihtimali olan satırlar yazardık. Bunu yapmak yerine catch bloğunda hata üretilen veya kendi hazırladığımız exception sınıfının örneğinin alındığı **Inner_uret()** metodunu çağırdık. Bu metotta zaten hata üretilmesine rağmen catch bloğunda exception sınıfının ikinci bir örneğini aldık. Başka bir deyişle bu sınıf sayesinde 2 hatanın arka arkaya meydana gelmesi sağlanmaktadır. Şimdi ise "Test_sinifi" adını verdiğimiz bu sınıfın örneğini alacağız. Bu amaçla üzerinde çalıştığımız uygulamada aşağıdaki gibi bir hazırlık yaptık.

```
private void button1_Click(object sender, EventArgs e)
{
 Test_sinifi test_nesne = new Test_sinifi();
 try
  {
   test_nesne.Hatayi_uret();
  }
```

```
 catch (Exception hata)
  {
   MessageBox.Show(hata.InnerException.Message);
   MessageBox.Show(hata.Message);
  }
}
```

Checked ve Unchecked Deyimleri

Değişkenler hakkında bilgi verilen bölümde kısaca söz edildiği gibi kapasitesi büyük bir değişkenin içeriği daha küçük kapasiteli değişkene aktarılırken beklenmedik sonuçlarla karşılaşılmaktadır. Bu konuda bilgi vermek için aşağıda verilen kodu hazırladık.

```
byte number1 = 128;
int number2 = number1;
```

Bu kod işletildiğinde herhangi bir sorun yaşanmaz. Çünkü **byte** tipindeki değişkenin içeriği daha büyük kapasiteli **int** değişkene aktarılmaktadır. Şimdi ise bu işlemin tersini yapacağız. Yani int değişkenin içeriğini byte değişkene aktaracağız.

```
int number2 = 255;
byte number1 = number2;
```

Bu 2 satırda int tipindeki değişkene aktardığımız değer byte tipindeki değişkenin kapasitesini aşmamasına rağmen hata meydana gelir. Çünkü dönüştürme yapılmadan int değişkenin içeriğinin byte değişkene aktarılmasına derleyici izin vermez. Aşağıda verilen kodda açık tür dönüşümü yapıldığı için hata meydana gelmez ve bilgi kaybı olmaz.

```
int number2 = 255;
byte number1 = (byte) number2;
textBox1.Text = number1.ToString();
```

Bu kodda int tipindeki değişkene byte değişkenin kapasitesini aşacak bir değer aktarmış olsaydık hata mesajı alınmazdı ama yanlış ve yanıltıcı sonuçlar alırdık. Örneğin bu kodda int tipindeki değişkene 300 değeri aktarılıp kod işletilirse TextBox'a 44 yazılır.

Yukarıda açık tür dönüşümü yapıldığı için derleyici bilgi kayıplarıyla ilgilenmez. Bu tür sorunların önüne geçmek istiyorsanız **checked** anahtar kelimesinden yararlanabilirsiniz. Checked anahtar kelimesi bilgi kayıpları söz konusu olduğu zaman hata meydana getirir. Checked deyimi ile olası bilgi kayıplarının nasıl önlendiğini aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 byte number1;
 int number2 = 300;
 checked
 {
  number1 = (byte) number2;
  textBox1.Text = number1.ToString();
 }
}
```

Bu kod hatasız derlenir ama bu metot işletilirse **OverflowException** hatası meydana gelir ve programın çalışması kırılır. Programın çalışmasının kırılmasını istemiyorsanız **checked** bloğunu **try-catch** bloğunun içine alabilirsiniz. Çalışma anında formdaki ilk TextBox'a 255'ten büyük bir sayı girilip aşağıda verdiğimiz kod işletilirse hem bilgi kayıplarına izin verilmez hem de programın çalışmasının kırılması önlenmiş olur.

```
private void Form1_Click(object sender, EventArgs e)
{
 byte number1;
 int number2 = Convert.ToInt16(textBox1.Text);
 try
 {
  checked
  {
    number1 = (byte)number2;
    textBox2.Text = number1.ToString();
   }
  }
 catch (System.OverflowException Hata)
  {
   MessageBox.Show("TextBox'a girdiğiniz sayı büyük");
  }
}
```

Bazen checked bloğu içinde yapılan bazı işlemler bilgi kaybına rağmen yapılmak istenir. Bu gibi durumlarda ilgili satırlar **unchecked** bloğuna alınır. Aşağıda verilen kodda sınır aşımı nedeniyle bilgi kayıplarına neden olan satırları unchecked bloğuna aldığımız için ilk TextBox'a büyük bir sayı girilse bile hata meydana gelmez ve catch bloğu işletilmez.

```
private void Form1_Click(object sender, EventArgs e)
{
 byte number1;
 int number2 = Convert.ToInt16(textBox1.Text);
 try
  {
   checked
    {
     unchecked
      {
       number1 = (byte)sayi2;
       textBox2.Text = number1.ToString();
      }
    }
  }
 catch (System.OverflowException Hata)
  {
   MessageBox.Show("TextBox'a girdiğiniz sayı büyük");
  }
}
```

21. BÖLÜM

KOLEKSİYON SINIFLARI – ARRAYLIST, STACK, QUEUE

Koleksiyonlar birden fazla veri veya nesneyi tutabilen nesnelerdir. Bu yanıyla dizi değişkenlerle benzer işleve sahipler. .NET Framework ile birlikte programcıların koleksiyon sınıfı ihtiyaçlarını gideren **ArrayList**, **Stack**, **Queue**, **Hashtable** ve **SortedList** gibi çok sayıda sınıfı hazır olarak gelmektedir. Bu bölümde bu sınıflar ve bu sınıflar tarafından uygulanan arayüzler hakkında bilgi verilecektir.

Bu bölümde anlatılan koleksiyon sınıflarının bazılarının **Generic** versiyonları bulunmaktadır. Mevcut koleksiyon sınıfları hakkında anlatılanların kolay izlenebilmesi için generic koleksiyonlardan ayrı bir bölümde söz edilecektir. Çünkü generic yapıdaki koleksiyon sınıfları üzerinde çalışmaya başlamadan önce generic sınıflar ve generic arayüzler hakkında bilgi sahibi olmakta fayda vardır.



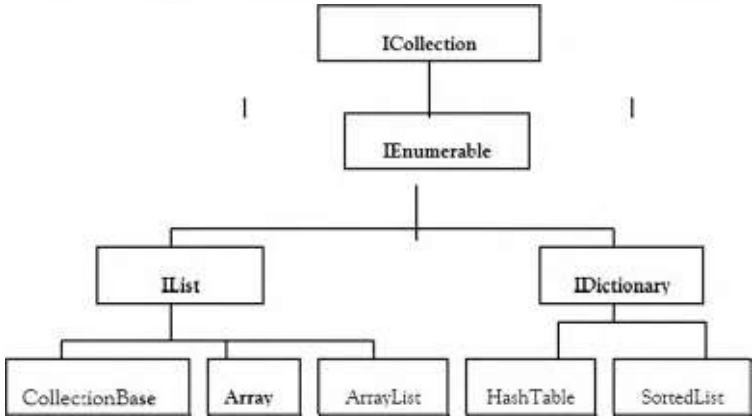
Daha önce .NET programcılığıyla az çok ilgilenmiş birisi bu bölümün başlığına bakıp, ArrayList'in nasıl hazırlanıp eleman eklendiğini biliyorum diyebilir. Elbette bu bölümde ArrayList nesnesinin nasıl hazırlanıp eleman eklendiğinden söz edilecektir. Bize sorarsanız, bu kitabı eline alan herkes bu bölümü baştan sona okumalıdır. Çünkü bu bölümde ArrayList sınıfının Add(), Insert() ve Remove() metotlarından fazlası var.

Uygulamalarınızda generic koleksiyonları kullanmayı tercih ediyor olsanız bile bu bölümü okumanız önerilir. Çünkü tekrara düşmemek için bazı konular generic koleksiyonlar için ayrıca işlenmedi. Diğer yandan uygulama geliştirmeye başlanıp konu koleksiyon sınıfı seçmeye geldiğinde önceliği generic koleksiyonlara verin diyebiliriz.

Uygulamanızda 20-30 elemanlı generic olmayan ArrayList nesnesi hazırlayıp kullanmanız halinde uygulamanızın performansının yerlerde sürünmesine neden olmayacaktır. Bu bölümün sonunda generic olmayan koleksiyon sınıflarıyla generic olanlar hakkında performans bağlamında kısa bir değerlendirme yapılacaktır.

Bu bölümde hakkında bilgi verilen **ArrayList**, **SortedList**, **Stack** ve **Queue** sınıfları **System**'deki **Collections** adlı namespace'te yer almakta ve bu sınıflar **IList**, **IEnumerator**, **IEnumerable**, **IDictionary** arayüzlerini uygulamaktadır. Konunun devamında bu arayüzler (interface) hakkında bilgi verilecektir. Bu arayüzlerin generic versiyonları hakkında bilgi ilgili bölümde verilecektir. Burada sayılan arayüzler birçok yerde karşınıza çıkacaktır. Hele hele bu arayüzlerin generic sürümleri, LINQ teknolojisi ile yapılacak sorgulamalarda çok önemlidir. Bu nedenle bu bölümde arayüzlerle ilgili olarak anlatıları önemsemek gerekir.

Anlatılanların kolay anlaşılması için bazı koleksiyon sınıflarının ve ilgili arayüzlerin hiyerarşik yapılarını resmetmekte fayda vardır. Temsili resim kolay izlensin diye bazı sınıf ve arayüzlere yer vermedik. Bu temsili resim dikkatlice incelenirse en tepedeki **IEnumerable** arayüzünün bütün koleksiyon sınıflarının ve diğer arayüzler tarafından uygulandığı görülür.



Temsili resimden fark edileceği gibi **HashTable** ve **SortedList** sınıfları **IDictionary** arayüzünü uygulamaktadır. IEnumerable arayüzünü uygulayan **ICollection** arayüzü ise **IList** ve **IDictionary** arayüzlerinin mirasçısı durumundadır. Bazı programcılar IList arayüzünü uygulayan koleksiyon sınıflarına **liste tipi koleksiyonlar** derken IDictionary arayüzünü uygulayan koleksiyonlara ise **sözlük tipi koleksiyonlar** demektedirler.

Hem ICollection hem de IEnumerable arayüzünün mirasçısı durumundaki **IList** arayüzünü liste tipi koleksiyon sınıflarından **Array**, **ArrayList** ve **CollectionBase** uygulamaktadır. Buna göre Array, ArrayList ve CollectionBase sınıfları; IList, ICollection ve IEnumerable arayüzlerinde tanımlı metot ve özelliklere sahiptir. Dolayısıyla bu arayüzler hakkında bilgi sahibi olmak, yukarıda adlarını verdiğimiz sınıflarla yapılabilinecek işlemleri bilmek demektir.

Konuya bu arayüzlerin metotlarını ve özelliklerini anlatmakla başlanabilirdi. Ancak o zaman anlatılanlar biraz zor anlaşılırdı. Konu ezbere olmasın diye **IList** arayüzü üzerinde biraz konuşacağız. Bu amaçla IList arayüzünün yapısı aşağıda verildi. ICollection ve IEnumerable arayüzlerinin mirasçısı durumundaki IList arayüzü **Add**(), **Clear**(), **IndexOf**(), **Insert**() ve **Remove**() gibi metotlara sahiptir. IList arayüzünü uygulayan liste tipi sınıflar ister istemez bu metotlara sahip olacaktır. Bu metotların nasıl kullanıldığını ileriki sayfalarda .NET Framework ile gelen koleksiyon sınıfları anlatılırken görebilirsiniz.

```
public interface IList : ICollection, IEnumerable
{
 int Add(object value);
 void Clear();
 bool Contains(object value);
 int IndexOf(object value);
 void Insert(int index, object value);
 void Remove(object value);
 void RemoveAt(int index);
 bool IsFixedSize { get; }
 bool IsReadOnly { get; }
 object this[int index] { get; set; }
}
```

Bu yapıya bakılıp konunun devamında sözü edilecek ArrayList sınıfındaki **Add()** metodunun kaynağının IList arayüzü olduğu fark edilebilir. **IList** arayüzünde ayrıca yalnızca **get** bloğuna sahip **IsFixedSize** ve **IsReadOnly** özellikleri bulunmaktadır. Generic olmayan IList'ten başka generic IList arayüzü bulunmaktadır. Generic IList arayüzünü generic koleksiyon sınıfları uygulamaktadır.

Bir koleksiyon sınıfı olan **Array** hakkında bilgiyi **Dizi Değişkenler** adlı bölümde bulabilirsiniz. Array sınıfının Add(), Remove() ve IndexOf() gibi metotları olmadığı, başka bir deyişle Array tipinde koleksiyon nesnelerinin eleman sayısı sonradan değiştirilemediği için bu bölümde sözü edilecek koleksiyon sınıflarına göre kullanılması esnek değildir.

IList arayüzünde dikkatinize sunmak istediğimiz önemli nokta, bu arayüzdeki indeksleyicidir. Bundan şu sonucu çıkarabiliriz: IList arayüzünü uygulayan koleksiyon sınıflarında elemanlara indeksle erişebilinmektedir.



Bu bölümde koleksiyon sınıfları hakkında bilgi verirken bazen konsol, bazen de Windows Form uygulamasından yararlandık. Anlatmak istediğimiz işleme uygun olarak uygulama tipini seçtik. Bu nedenle bu bölüm üzerinde çalışmaya ve anlatılanları denemeye başlamadan önce standart özelliklere sahip bir konsol uygulaması ve bir forma sahip Windows Form uygulaması hazırlamanızı öneriyoruz.

Koleksiyon Sınıfları İçin Özet Bilgi

Bu kitapta .NET Framework ile gelen çok sayıda koleksiyon sınıfı hakkında bilgi verilmeye çalışıldı. Henüz yolun başında olan programcıların önüne bu kadar seçenek konulunca hangi şartlarda hangi koleksiyonu seçip kullanmalarının uygun olacağı konusunda tereddüde düşüyorlar. Bu nedenle bölümün başında kısa bir özet geçmeye karar verdik.

Array Sınıfı: Dizi değişkenler adlı bölümde ayrıntılarıyla anlatıldığı üzere hazırlamak istediğiniz koleksiyonda tutmak istediğiniz elemanların sayısı önceden belli ise tercihinizi dizi değişkenden yana yapabilirsiniz. Çünkü dizi değişkenlerde sonradan eleman ekleme ve çıkarma işlemleri zahmetlidir.

ArrayList Sınıfı: Dizi değişkenlerin yetersiz olduğu noktada akla ilk gelmesi gereken sınıftır. Genel amaçlı bir liste sınıfı olan ArrayList'lerde elemanlara dizi değişkenlerde olduğu gibi indeksle erişilebilmektedir. ArrayList nesnelerine eleman ekleme, çıkarma ve arama işlemleri son derece kolaydır. Özetle, elinizdeki veriler dizi değişkenlerde tutulmaya uygun olmasına rağmen dizi değişkenlerin sizi sınırladığını düşünüyorsanız başvurmanız gereken sınıflardan birisi ArrayList olabilir.

Stack Sınıfı: Şimdiye kadar anlatılardan fark edilmiş olacağı gibi koleksiyon kullanmak demek birden fazla bilgiyi bellekte organize etmektir. Koleksiyon halinde organize etmek istediğiniz bilgiler üst üste duran tabaklar gibiyse veya sürekli olarak en son eklenen eleman üzerinde işlem yapmanız gerekiyorsa yararlanmanız gereken sınıf **Stack** olabilir. Yeni elemanın en üste konulmasına, çıkarılan elemanın en üstte duran eleman olmasına "son giren ilk çıkar (Last In First Out-LIFO)" şeklinde özetlenmektedir.

Queue Sınıfı: Elinizdeki verileri ilk giren ilk çıkar prensibine göre organize etmek istiyorsanız bu sınıftan yararlanabilirsiniz. Günlük hayatta sık karşılaşılan bilet kuyrukları işlev bakımından Queue sınıfına çok benzemektedir.

HashTable Sınıfı: Elinizde **anahtar-değer** şeklinde düzenlenmiş bilgiler varsa HashTable sınıfından yararlanarak bellekte tutabilirsiniz

ArrayList Sınıfı İle Koleksiyon Hazırlamak

ArrayList sınıfı üzerinde konuşmaya başlamadan önce, bu sınıfı kullanma ihtiyacıyla ilgili olarak maddeler halinde bir durum tespiti yapmaya çalışacağız:

- Dizi değişkenin eleman sayısını çalışma anında değiştirmeye gerek duyuyorsanız,
- Kodlama sırasında tanımlamak istediğiniz dizideki eleman sayısı hakkında elinizde net bilgi yoksa,
- Çalışma anında diziden eleman silmeniz gerekiyorsa,
- Gelişen şartlara göre dizinin herhangi bir konumuna yeni eleman eklemek istiyorsanız,
- Dizide sıklıkla eleman aramak durumunda kalıyorsanız,

ArrayList sınıfından yararlanıp liste tipi koleksiyon hazırlamanız şart olmuştur. Her ne kadar burada ArrayList sınıfına hangi durumlarda başvurmanız gerektiğinden söz ediyorsak ta bu maddeleri aynı şekilde **List** sınıfı için de yazabiliriz.

ArrayList sınıfı sayesinde değişik tipte bilgi aktarılabilinen ve çalışma anında eleman sayısı otomatik olarak değişebilinen, dizi değişken gibi işlev gören koleksiyon adı verilen veri yapıları hazırlanabilmektedir. Yukarıda işaret edildiği üzere ArrayList sınıfı **IList**, **ICollection** ve **IEnumerable** arayüzlerini uygulamaktadır. Aşağıda verilen satır ile "Takimlar" adında bir ArrayList nesnesi, yani liste tipi bir koleksiyon nesnesi hazırlandık.

```
ArrayList takimlar = new ArrayList();
```

ArrayList sınıfı **System.Collections**'da bulunduğu için, ya bu namespace **using** deyimi ile sınıfların aranacağı namespace listesine dahil edilmeli ya da her seferinde sınıf adından önce bu namespace'in adı yazılmalıdır. Visual Studio ile hazırlanan projelerde **System.Collections.Generic**'te bulunan sınıflar için otomatik olarak using işlemi yapıldığından generic koleksiyonlar için using anahtar kelimesinin kullanılması gerekmiyor.

ArrayList tipinde koleksiyon nesnesi hazırlanırken kapasitesi belirtilmediği zaman bunun 0 olduğu kabul edilmektedir. Kapasitesi belirtilmeden hazırlanan ArrayList nesnesine ilk eleman eklendiğinde kapasite 4 yapılır.

Belli sayıda elemana sahip ArrayList nesnesi hazırlarken kapasiteyi yapıcı metoda parametre olarak verebilirsiniz. Koleksiyonun mevcut kapasitesi **Capacity** özelliğinde tutulmaktadır. Aşağıda verilen satırlar bir metoda yazılıp işletildiğinde ekrana 10 yazılır.

```
ArrayList takimlar = new ArrayList(10);
Console.WriteLine(takimlar.Capacity);
```



ArrayList nesnesinin mevcut kapasitesi yeterli gelmediğinde geri planda kapasite iki katına çıkarılırken elbette sisteme bir yük gelecektir. Bize sorarsanız bunu kafayı takmamak gerekir.

ArrayList sınıfının yukarıda sözü edilenlerden başka yapıcı metodunun 3. bir sürümü daha bulunmaktadır. Yapıcı metodun 3. sürümü, başka bir koleksiyondan yola çıkılarak ArrayList nesnesi hazırlanmak istendiğinde kullanılmaktadır. **ICollection** arayüzünü uygulamış bir sınıftan yola çıkılarak hazırlanmış bir nesneyi parametre olarak alan yapıcı metodun bu sürümünün nasıl kullanıldığını aşağıda görebilirsiniz.

```
ArrayList takimlar = new ArrayList(10);
ArrayList liste = new ArrayList(takimlar);
```

Bu 2 satırda önce kapasitesi 10 olan bir ArrayList nesnesi hazırladık. Ardından bu ArrayList nesnesini ArrayList sınıfının yapıcı metoduna parametre olarak verip ikinci bir ArrayList nesnesi hazırladık. Bu durumda 2. ArrayList nesnesinin kapasitesi yine 10 olur.

ArrayList nesnesine eleman eklerken **Add()** metodu kullanılmaktadır. Object tipinde bir parametreye sahip Add() metodu her kullanıldığında ArrayList nesnesinin dahili koleksiyonun sonuna yeni bir eleman eklenir ve Add() metoduna parametre olarak verilen bilgi bu elemana aktarılır. Add() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
Static void Main(string[] args)
{
 ArrayList takimlar = new ArrayList(10);
 takimlar.Add("Fenerbahçe");
 takimlar.Add("Galatasaray");
 takimlar.Add("Beşiktaş");
 takimlar.Add("Gençlerbirliği");
 Console.WriteLine(takimlar[0]);
 Console.WriteLine(takimlar[1]);
 Console.WriteLine(takimlar[2]);
}
```

Add() metodu geriye ArrayList nesnesine eklenen elemanın sıra numarasını göndermektedir. Burada, eklediğimiz elemanların sıra numaralarına gerek duymadığımız için yakalamadık.

Yukarıda verilen koddan fark edilmiş olacağı üzere ArrayList nesnesinin hangi elemanın içeriği elde edilmek isteniyorsa o elemanın sıra numarası köşeli parantezlerin içine yazılmalıdır. Çünkü ArrayList nesnesinin elemanları indekslenmektedir. Bu özelliği başka bir cümle ifade etmek gerekirse, ArrayList koleksiyonun elemanlarına dizi değişkenlerde olduğu gibi indeksleyici ile de erişilebilmektedir.



ArrayList nesneleri üzerinde işlem yaparken, sınıfın kendi metotlarından başka LINQ ile gelen birkaç genişletme metodundan yararlanmak mümkündür. LINQ ile gelen bu metotlardan generic koleksiyon sınıflarının anlatıldığı bölümde söz edilecektir.

Add() metodu ile ArrayList nesnesine eklenen elemanlar gerçekte object tipindeki dahili dizi değişkene aktarılmaktadır. Bizler bu dahili koleksiyon üzerinde doğrudan işlem yapamı-

yoruz. ArrayList sınıfının yapıcı metodu parametresiz kullanılıp kapasite belirtilmediği zaman ArrayList nesnesi içinde **Items** koleksiyonu aşağıdaki gibi hazırlanıyor.

```
this.Items = new object[];
```

ArrayList sınıfından yararlanılarak hazırlanan koleksiyona farklı tipte bilgiler aktarılabilinir. Başka bir deyişle ArrayList sınıfı sayesinde hazırlanan koleksiyonların elemanları **object** tipindedir. Aşağıda verilen kodda ArrayList nesnesine farklı tipte bilgiler aktardık.

```
public static void Main()
{
 ArrayList dizi = new ArrayList(10);
 dizi.Add("Fenerbahçe");
 dizi.Add(1907);
 dizi.Add(true);
 dizi.Add(541.345);
}
```

Add() metodu kullanılarak ArrayList nesnesine yeni bir eleman eklendiğinde bu elemanın tipi object değilse geri planda bu eleman object tipine çevrilip ArayList'in dahili koleksiyonuna öyle eklenir. ArrayList nesnesinden okuma yapılırken yine dönüştürme işlemine gerek duyulur. ArrayList nesnesinin mevcut elemanlarından birisi okunduğunda tipi object olacaktır. Okunan bu bilgi hangi tipe aktarmaya uygunsa ona göre dönüştürme yapılır. Bu da performans kaybına neden olacaktır.

Hazırlamak istediğiniz ArrayList nesnesine aktarılacak elemanları dizi değişkenlerde olduğu gibi tanımlama satırında süslü parantezlerin arasına yazabilirsiniz. Koleksiyon nesnelerine tanımlama satırında eleman ekleme imkanı C# 3.0 ile birlikte geldi. Aşağıda veriler satır bir metoda yazılıp işletilirse hazırlanan ArrayList nesnesi 3 elemana sahip olur.

```
ArrayList takimlar = new ArrayList()
{
  "Fenerbahçe","Beşiktaş", "Galatasaray"
};
```

Yeni elemanı koleksiyonun sonu yerine herhangi bir konumuna eklemek istiyorsanız Add() yerine **Insert**() metodunu kullanmalısınız. Insert() metodu 2 parametreye sahiptir. İlk parametrede elemanın koleksiyon içindeki sıra numarası, 2. parametrede ise elemanın içeriği verilmektedir. Insert() metodunun nasıl kullanıldığını aşağıda görebilirsiniz. Insert() metodu Add() metodundan farklı olarak geriye bilgi göndermemektedir.

```
public static void Main()
{
 ArrayList takimlar = new ArrayList(10);
 takimlar.Add("Fenerbahçe");
 takimlar.Add("Beşiktaş");
 takimlar.Add("Trabzonspor");
 takimlar.Insert(2, "Galatasaray");
}
```

Burada kullanılan **Insert()** metoduna 2. parametre olarak verilen bilgi, 3. elemanın içeriği olur ve Insert() metodu kullanılmadan önce 3. sırada bulunan eleman ise 4. eleman olur. Insert() metodu ile ArrayList nesnesine eleman eklerken dikkatli olunmalıdır. Örneğin bu kod aşağıdaki gibi düzenlenirse çalışma anında hata meydana gelir.

```
ArrayList takimlar = new ArrayList(10);
takimlar.Add("Fenerbahçe");
takimlar.Add("Beşiktaş");
takimlar.Add("Trabzonspor");
takimlar.Insert(4, "Galatasaray"); // hata meydana gelir
```

Çünkü burada "takimlar" adını verdiğimiz ArrayList nesnesi 3 elemana sahip iken indisi 4 (5. eleman) olacak bir elemanı nesneye eklemek istiyoruz. Bu nedenle Insert() metodunun 1. parametresi olarak kullanılan sayısal değer ArrayList nesnesinin mevcut eleman sayısından fazla olamaz. Aşağıdaki gibi kullanılan Insert() metodu işlev bakımından Add() metodundan farklı olmaz. Çünkü ArrayList nesnesi 3 elemana sahipken Insert() ile 4. sıraya yeni bir eleman eklenmektedir.

```
ArrayList takimlar = new ArrayList(10);
takimlar.Add("Fenerbahçe");
takimlar.Add("Beşiktaş");
takimlar.Add("Trabzonspor");
takimlar.Insert(3, "Galatasaray");
```

Mevcut bir koleksiyonun içeriğini ArrayList nesnesinin herhangi bir yerine eklemek istiyorsanız **InsertRange()** metodunu kullanabilirsiniz. Bu metodun nasıl kullanıldığını anlatmak için projenin formuna 2 ListBox yerleştirdik.



Çalışma anında "Aktar" düğmesi tıklandığında ilk ListBox'ın elemanları işaret edilecek konumundan itibaren ArrayList nesnesine eklenecek. En son olarak ArrayList'in bütün elemanları 2. ListBox'a aktarılacak. Bu işlemi yapacak kodu aşağıda görebilirsiniz.

```
private void Aktar_Click(object sender, EventArgs e)
{
 ArrayList iller = new ArrayList();
 iller.Add("İstanbul");
 iller.Add("İzmir");
 iller.InsertRange(1, listBox1.Items);
 int sayi = iller.Count;
 for (int i = 0; i < sayi; i++)
  {
    string eleman = iller[i].ToString();
```

```
            listBox2.Items.Add(eleman);
        }
    }
```

Bu kodda ArrayList nesnesi hazırlandıktan sonra Add() metodu ile 2 eleman eklendi. Ardından ilk ListBox'ın elemanları **InsertRange()** metodu ile ArrayList'e eklendi. InsertRange() metoduna 1. parametre olarak "1" verildiği için eleman ekleme işlemi 1. elemanın arkasına yapılır. AddRange() ve InsertRange() metotları **ICollection** arayüzünü uygulamış sınıflardan yola çıkılarak hazırlanmış nesneleri parametre olarak alabilmektedir.

Tanımlayıp eleman eklediğiniz ArrayList nesnesinin mevcut elemanlarının içeriğini indeksleyiciden yararlanarak değiştirebilirsiniz. Aşağıda verilen kodda ArrayList'e Add() metodu ile eklenen 3 elemanın içerikleri sonradan değiştirilmektedir.

```
ArrayList iller = new ArrayList();
iller.Add("Ankara");
iller.Add("İstanbul");
iller.Add("İzmir");
iller[0] = "Bursa";
iller[1] = "Konya";
iller[2] = "Gaziantep";
```

ArrayList'in bazı elemanlarını başka bir ArrayList nesnesinin elemanları ile değiştirecekseniz **SetRange()** metodunu kullanabilirsiniz. Aşağıda verilen kod işletildiğinde "buyuksehir" adı verilen nesnenin elemanları "iller" nesnesinin mevcut elemanlarına aktarılır.

```
ArrayList iller = new ArrayList();
ArrayList buyuksehir = new ArrayList();
iller.Add("Bursa");
iller.Add("Konya");
iller.Add("Denizli");
iller.Add("Gaziantep");
buyuksehir.Add("İstanbul");
buyuksehir.Add("Ankara");
buyuksehir.Add("İzmir");
iller.SetRange(1, buyuksehir);
```

Burada SetRange() metoduna 1. parametre olarak 1 verildiği için değiştirme işlemi 2. elemandan itibaren yapılır. SetRange() metoduna 2. parametre olarak verilen ArrayList'in eleman sayısı diğerinden fazlaysa hata meydana gelir.

ArrayList sınıfının yapıcı metoduna parametre olarak verilen ve ICollection arayüzünü uygulamış herhangi bir sınıftan yola çıkılarak hazırlanmış koleksiyon nesnesinin elemanları otomatik olarak yeni ArrayList'e aktarılmaktadır. Bu konuda bilgi vermek için aşağıdaki gibi bir dizi değişken tanımlayıp elemanlarına bilgi aktardık.

```
string[] iller = new string[3];
iller[0] = "Adana";
iller[1] = "Ankara";
iller[2] = "Antalya";
```

Array sınıfı **ICollection** arayüzünü uygulayan bir sınıf olduğu için ArrayList nesnesi hazırlarken bu dizi değişkenden yararlanabiliriz. Aşağıda verilen satırlar metot olarak düzenlenip işletildiğinde ListBox'a dizi değişkenin 3 elemanın içeriği aktarılır.

```
string[] iller = new string[3];
iller[0] = "Adana";
iller[1] = "Ankara";
iller[2] = "Antalya";
ArrayList Liste = new ArrayList(iller);
foreach (string il in Liste)
  listBox1.Items.Add(il);
```

ArrayList'te Arama İşlemleri

Dizi değişkenlerin anlatıldığı bölümde sözü edilen **BinarySearch, IndexOf, LastIndexOf ve Contains** metotları IList arayüzü kaynaklıdır. ArrayList sınıfı IList arayüzünü uyguladığı için, arama yaparken bu metotları kullanabilirsiniz. ArrayList sınıfının **Contains**() metodu parametre olarak aldığı bilgiyi arar. Bulursa geriye true, bulamazsa false gönderir.

```
bool sonuc = iller.Contains("Ankara");
if (sonuc == true)
  MessageBox.Show("Bu eleman mevcut");
```

Dolayısıyla Contains() metodu ilgilenilen elemanın olup olmadığı araştırılmak istendiği zaman kullanılır. ArrayList nesnesinde ilgilendiğiniz elemanın sıra numarasını öğrenmek istiyorsanız, başka bir deyişle aradığınız elemanı elde etmek istiyorsanız **IndexOf()** metodunu kullanabilirsiniz. IndexOf() metodu kendisine parametre olarak verilen bilgiyi ArrayList'te bulamazsa geriye -1 göndermektedir.

```
int sira = iller.IndexOf("Bursa");
if (sira > 0)
  MessageBox.Show(iller[sira]);
```

Arama işlemini ArrayList nesnesinin bütün elemanlarını kapsayacak şekilde yapmak istemiyorsanız aramanın başlamasını istediğiniz elemanın sıra numarasını IndexOf() metoduna 2. parametre olarak vermelisiniz. **IndexOf()** metodu aşağıdaki gibi kullanılırsa arama 3. elemandan itibaren yapılır.

```
private void Arama_Click(object sender, EventArgs e)
{
 ArrayList iller = new ArrayList();
 iller.Add("İstanbul");
 iller.Add("Ankara");
 iller.Add("İzmir");
 iller.Add("Adana");
 int sira = iller.IndexOf("Ankara", 2); // sira -1 olur.
}
```

Arama işlemini belli sayıda elemanda yapmak istiyorsanız aramaya konu edilecek eleman sayısını 3. parametrede belirtmeniz gerekir. Aramayı ArrayList nesnesinin yalnızca 2. ve 3. elemanlarında yapmak isteseydik IndexOf() metodunu aşağıdaki gibi düzenlerdik.

```
int sira = iller.IndexOf("İzmir", 1, 2);
```

ArrayList nesnesinde içeriği aynı olan birden fazla eleman varsa ve söz konusu bilgiye sahip son elemanın sıra numarasını öğrenmek istiyorsanız **LastIndexOf()** metodunu kullanabilirsiniz. Aşağıda verilen kodda ArrayList nesnesi içinde "Adana" içeriğine sahip 2 eleman bulunmaktadır. Aramayı LastIndexOf() metodu ile yaptığımız için geriye 3 gönderilir.

```
ArrayList iller = new ArrayList();
iller.Add("İstanbul");
iller.Add("Adana");
iller.Add("İzmir");
iller.Add("Adana");
int sira = iller.LastIndexOf("Adana");  // sira 3 olur.
```

Aramayı LastIndexOf yerine IndexOf() metodu ile yapmış olsaydık bu kez içeriği "Adana" olan son elemanın sıra numarası yerine ilk elemanın sıra numarası geriye gönderilirdi.

ArrayList nesnesinde arama yapılırken kullanılan **BinarySearch()** metodu ile nasıl arama yapıldığını göstermek için aşağıda verilen kodu yazdık. BinarySearch() metodu ile arama yapabilmek için ArrayList nesnesinin elemanlarının sıralı olması gerekiyor. Bu nedenle BinarySearch() metodunu kullanmadan önce **Sort()** metodu ile sıralama yaptık. Aşağıda verilen kodda Sort() metoduna yer vermemiş olsaydık hata meydana gelirdi.

```
ArrayList iller = new ArrayList();
iller.Add("İstanbul");
iller.Add("Adana");
iller.Add("İzmir");
iller.Add("Ankara");
iller.Sort();
int sonuc = iller.BinarySearch("Ankara");
```

Bu şekilde kullanılan BinarySearch() metodu kendisine parametre olarak verilen elemanı ArrayList'te arar. Bulursa geriye elemanın sıra nosunu, bulamazsa geriye sıfırdan küçük bir değer gönderir. Sıra numarası öğrenilen eleman indeksleyiciden yararlanılıp elde edilebilir.

ArrayList'in Eleman Sayısını Öğrenmek ve Eleman Silmek

Insert() veya Add() metodu ile ArrayList'e eleman eklemeden önce mevcut elemanların sayısı öğrenilmek veya ArrayList sınıfı aynı içeriği sahip birden fazla elemanın olmasına izin verdiği için eleman eklenmeden önce mevcut olup olmadığı araştırılmak istenebilir.

```
ArrayList liste = new ArrayList(10);
liste.Add("Fenerhabçe");
liste.Add("Beşiktaş");
liste.Add("Trabzonspor");
int sayi = liste.Count; / sayi 3 olur.
```

ArrayList nesnesinin eleman sayısı **Count** özelliğinde tutulmaktadır. ArrayList nesnesinin kapasitesini eleman sayısı ile karıştırmamak gerekir. Burada ArrayList nesnesini hazırlarken

kapasitenin 10 olmasını sağladık. Bu sırada Add() metodu ile nesneye 10'dan fazla eleman eklenmesi halinde kapasite otomatik olarak iki katına çıkar.

ArrayList nesnesinin kapasitesini eleman sayısına eşitlemek istiyorsanız **TrimToSize()** metodunu kullanmalısınız. Aşağıda verilen satırlar bir metoda yazılıp işletilirse kapasite 3 olur. Çünkü TrimToSize() metodunu kullandığımız sırada bu nesne 3 elemana sahipti.

```
ArrayList liste = new ArrayList(10);
liste.Add("Fenerhabçe");
liste.Add("Beşiktaş");
liste.Add("Trafzonspor");
liste.TrimToSize();
int kapasite = liste.Capacity); // kapasite 3 olur.
```

ArrayList nesnesinden eleman silmek istiyorsanız **RemoveAt()** metodunu kullanabilirsiniz. Silinecek elemanın sıra numarası bu metoda parametre olarak verilmelidir. RemoveAt() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
ArrayList liste = new ArrayList(3);
liste.Add("Fenerbahçe");
liste.Add("Beşiktaş");
liste.Add("Trabzonspor");
liste.RemoveAt(1);
int sayi = liste.Count; // sayi 2 olur.
```

Bu kodda Add() metodu ile koleksiyona 2 eleman eklendikten sonra **RemoveAt()** metodu ile 2. eleman silinmektedir. Dolayısıyla silme işlemi sonrası listenin 3. elemanı 2. elemanın yerini alacaktır. Bu nedenle silme sonrası "liste[2]" ile işaret edilen bir eleman olmayacaktır.

ArrayList nesnesinden eleman silme işlemini sıra numarası yerine elemanın içeriğine göre yapmak istiyorsanız RemoveAt() yerine **Remove()** metodunu kullanabilirsiniz. Bu metot aşağıdaki gibi kullanıldığında ilgili ArrayList nesnesinden belirtilen eleman silinir.

```
liste.Remove("Beşiktaş");
```

Remove() metoduna parametre olarak verilen eleman yoksa hata meydana gelmez.Yani olmayan bir elemanı silmek isterseniz program bir tepkide bulunmaz. Bu nedenle silinmek istenen eleman önce aranabilir.

ArrayList tipindeki koleksiyonun bütün elemanlarını silmek istiyorsanız **Clear()** metodunu kullanabilirsiniz. Belli sayıdaki ve konumdaki elemanlarını silmek istiyorsanız **RemoveRange** metodu var. Bu metot aşağıdaki gibi kullanıldığında ArrayList nesnesinin 2. elemanından başlamak üzere 2 eleman silinir.

```
liste.RemoveRange(1, 2);
```

2 parametreye sahip RemoveRange() metodunun ilk parametresi ile silinecek ilk elemanın sıra nosu işaret edilmektedir. 2. parametre ile silinecek eleman sayısı belirtilmektedir. Remove() veya RemoveAt() metoduyla yapılan eleman silmenin kapasiteye bir etkisi yoktur.

Eleman Sayısı Sabit ArrayList Nesnesi Hazırlamak

Eleman sayısı sabit ArrayList nesnesi hazırlamak istiyorsanız statik **FixedSize**() metodundan yararlanabilirsiniz. Eleman sayısı değişmeyen nesne hazırlayabilmek için öncelikle kapasitesi belli ArrayList nesnesi hazırlayıp bunu FixedSize metoduna parametre olarak vermelisiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 ArrayList iller = new ArrayList(4);
 iller.Add("Adana");
 iller.Add("Ankara");
 iller.Add("istanbul");
 ArrayList sabit_dizi = ArrayList.FixedSize(iller);
}
```

Bu kodda önce "iller" adın verdiğimiz ve kapasitesi 4 olan bir ArrayList nesnesi hazırlayıp 4 ilin adını eleman olarak aktardık. Devamında bu 4 elemanlı ArrayList nesnesinden yola çıkıp **FixedSize**() metodu ile sabit elemanlı başka bir ArrayList nesnesi hazırladık. Bu şartlarda her iki nesne de aynı elemanlara sahiptir.

Statik **FixedSize**() metodu sayesinde hazırlanan ArrayList nesnesine yeni eleman eklenemez ve mevcut elemanlar silinemez. Ancak istenirse mevcut elemanlarda değişiklik yapılabilir. Mevcut elemanlarda değişiklik yapılırken indeksleyiciden yararlanılır.

İlgilendiğiniz ArrayList nesnesinin eleman sayısının sabitlenip sabitlenmediğini öğrenmek istiyorsanız **IsFixedSize** özelliğine bakabilirsiniz. İlgili ArrayList nesnesinin eleman sayısı sabitlenmiş ise bu özellik true olmaktadır. Bu özellikten aşağıdaki gibi yararlanılabilir.

```
private void ekle_Click(object sender, EventArgs e)
{
 bool sonuc = sabit_dizi.IsFixedSize;
 if (sonuc)
   MessageBox.Show("Bu ArrayList'e eleman eklenemez");
 else
   sabit_dizi.Add("Mersin");
}
```

ArrayList sınıfı **FixedSize**() metodundan başka **ReadOnly**() adında statik bir metoda sahiptir. Bu metottan yararlanarak eldeki bir ArrayList nesnesinden yola çıkıp mevcut elemanlarının içeriği değiştirilemeyen ve yeni eleman eklenemeyen başka bir nesne hazırlamak mümkündür. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
ArrayList iller = new ArrayList();
iller.Add("Adana");
iller.Add("Ankara");
iller.Add("İstanbul");
ArrayList sabit_dizi = ArrayList.ReadOnly(iller);
```

FixedSize() metodu sayesinde hazırlanan ArrayList'in mevcut elemanların değiştirilmesine izin verilirken ReadOnly() metodu ile hazırlananda sadece okuma yapılabilir. İlgilendiğiniz ArrayList'in ReadOnly olup olmadığını öğrenirken **IsReadOnly** özelliğine bakabilirsiniz.

ArrayList Sınıfının Sort ve Reverse() Metotları

ArrayList'te elemanlar eklendikleri sırada duruyorlar. Elemanlara indeksleyici ile tek tek erişildiğinde sıralı olup olmaması önemli değildir. Mevcut elemanların sıralı bir şekilde durmasını istiyorsanız **Sort**() metodundan yararlanabilirsiniz.

```
ArrayList gunler = new ArrayList();
gunler.Add("Pazar")
gunler.Add("Pazartesi")
gunler.Add("Salı")
gunler.Sort();
```

Burada Sort() metodunu anlatmak için hazırladığımız ArrayList nesnesinde standart veri tiplerini kullandık. ArrayList'e kendi hazırladığımız bir tipte nesneler aktarmış olsaydık Sort() metodu bu şekilde sıralama yapamazdı. ArrayList nesnesinin elemanlarının sırasını tersine çevirmek, başka bir deyişle ilk elemanın son, son elemanın ise ilk eleman yapılmasını istiyorsanız **Reverse**() metodunu kullanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
ArrayList iller = new ArrayList();
iller.Add("İstanbul");
iller.Add("Ankara");
iller.Add("İzmir");
iller.Add("Adana");
iller.Reverse();
```

ArrayList sınıfının Reverse() metodu parametresiz kullanıldığında bütün elemanların sırasını tersine çevirmektedir. Reverse işleminin bütün elemanlar yerine belirli sayıda elemana uygulanmasını istiyorsanız başlangıç elemanını ve yerleri değiştirilecek eleman sayısını parametre olarak vermelisiniz. Reverse() metodu aşağıdaki gibi kullanıldığında söz konusu nesnenin 2. elemanından başlamak üzere 2 eleman reverse işlemine tabi tutulur.

```
iller.Reverse(2, 2);
```

Mevcut bir ArrayList nesnesi ile aynı içeriğe sahip 2. bir ArrayList nesnesi hazırlamak istiyorsanız **Clone**() metodunu kullanabilirsiniz.

```
ArrayList liste2 = (ArrayList)liste1.Clone();
```

GetEnumerator() Metodu

ArrayList sınıfı IEnumerator arayüzünü uyguladığı için bu arayüzde bulunan **GetEnumerator**() metodu ile elemanları numaralı bir listeye alıp bazı ek işlemleri yapabilirsiniz. Aşağıda verilen satır sayesinde ArrayList tipindeki koleksiyonun bütün elemanları IEnumerator tipindeki numaralayıcıya aktarılır.

```
IEnumerator numaralayici = liste.GetEnumerator();
```

Her ne kadar burada, GetEnumerator metoduyla ilgili olarak ArrayList nesnesindeki bütün elemanları bir numaralayıcıya alıyor desek bile gerçekte bu şekilde hazırlanan IEnumerator nesnesi ArrayList'in elemanlarına erişimde kullanılan bir araç işlevini görmektedir.

Şimdi Windows Application Forms şablonu ile bir proje hazırlayıp IEnumerator tipindeki listenin üzerinde **MoveNext()** metodu ile dolaşacağız. Üzerine gidilen elemanın içeriğini TextBox'a aktaracağız. Bu amaçla aşağıda verilen kodu hazırladık.

```
ArrayList liste;
IEnumerator numaralayici;
private void Form1_Load(object sender, EventArgs e)
{
 liste = new ArrayList(3);
 liste.Add("Hard Disk");
 liste.Add("Yazıcı");
 liste.Add("Ekran");
 numaralayici = liste.GetEnumerator();
 numaralayici.MoveNext();
 textBox1.Text = numaralayici.Current.ToString();
}
```

Bu kod sayesinde numaralayıcının ilk elemanın içeriği TextBox'a aktarılır. Forma bir düğme yerleştirip **MoveNext()** metodunu tekrar işleterek koleksiyondaki bir sonraki elemanın üzerine gidip bu elemanın içeriğini aşağıda yapıldığı gibi elde edebilirsiniz.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 numaralayici.MoveNext();
 textBox1.Text = numaralayici.Current.ToString();
}
```

Numaralayıcının en son elemanın üzerinde iken bu kod işletilirse 2. satır hataya neden olur. En son elemanın üzerinde iken MoveNext() metodu işletildiğinde hareket başarılı olamayacağı için geriye false gönderilir. Bu nedenle MoveNext() metodunun geriye gönderdiği bilgi aşağıdaki gibi kontrol edilip olası hataların önüne geçilebilir.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 bool sonuc = numaralayici.MoveNext();
 if (sonuc == true)
   textBox1.Text = numaralayici.Current.ToString();
 else
   MessageBox.Show("Koleksiyonun son elemanın üzerindesiniz");
}
```

Numaralayıcının üzerinde olunan elemanın içeriği **Current** özelliği ile elde edilmektedir. **GetEnumerator()** metodu ile hazırlanan numaralayıcının ilk elemanın üzerine gidilmek istendiğinde **Reset()** metodu kullanılmalıdır.

```
numaralayici.Reset();
```

GetEnumerator() metodu ile hazırlanan koleksiyonun elemanlarının nasıl kullanıldığına örnek olması için forma 2 ListBox yerleştirip aşağıda verilen kodu hazırladık. Bu kod sayesinde ilk ListBox'ın mevcut içeriği önce ArrayList nesnesine, oradan **GetEnumerator()** metodu ile hazırlanan enum tipine aktarılır.

```
private void Aktar_Click(object sender, EventArgs e)
{
 ArrayList takimlar = new ArrayList();
 foreach (string eleman in listBox1.Items)
   takimlar.Add(eleman);
 IEnumerator numaralayici = takimlar.GetEnumerator();
 while(Koleksiyon.MoveNext())
 {
  listBox2.Items.Add(numaralayici.Current.ToString());
  }
}
```

Yukarıdaki sayfalarda konuyu biraz uzatmış olmakla birlikte ArrayList sınıfının önemli metot ve özelliklerinden söz ettik. Burada öncelikli amacımız, koleksiyon sınıfları hakkında temel bilgiye sahip olmanıza önayak olmaktır. Bunu yaparken ArrayList sınıfından yararlandık.

C#'ın ilk sürümüyle uygulama geliştirenler için ArrayList en çok kullanılan koleksiyon sınıfı iken 2.0 sürümünden itibaren programcılar ArrayList ile benzer işleve sahip generic desteği olan **List** sınıfını tercih etmeye başladılar.

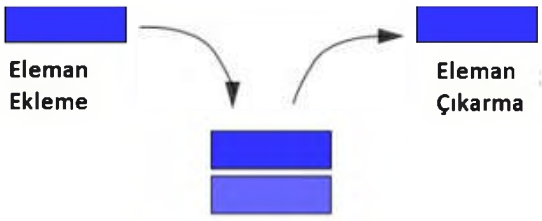
Stack Sınıfı

.NET Framework ile gelen Stack sınıfından söz etmeden önce stack adı verilen veri yapısından kısaca söz edeceğiz. Stack veri yapısında hem eleman ekleme hem de çıkarma işlemleri sürekli en üstten yapılmaktadır. Başka bir deyişle stack veri yapısına yeni bir eleman eklendiği zaman bu eleman en üstte durur. Stack veri yapısından bir eleman çıkarılmak istendiğinde bu eleman, en son eklenen eleman olur. Yeni elemanın en üste konulmasına, çıkarılan elemanın en üstte duran eleman olması son giren ilk çıkar (Last In First Out-LIFO) veya ilk giren son çıkar (First-in-Last-out FILO) şeklinde özetlenmektedir.



Bu bölümde öncelikli amacımız .NET Framework ile gelen koleksiyon sınıfları hakkında bilgi verip nasıl kullanıldığını göstermektir. Bu nedenle stack ve queue gibi veri yapıları hakkında teorik bilgi vermeyi tercih etmedik. .NET Framework ile gelen Stack ve Queue gibi sınıfları kullanmakla yetinmeyip benzer amaçlı sınıflar hazırlamak isteyenlerin veri yapıları konusunda kendilerini kavramsal bilgi ile donatmaları gerekir.

.NET Framework ile gelen Stack sınıfı ile hazırlanan nesnelerde bilgiler üst üste konulmuş tabaklar gibi organize edilmektedir. Örneğin üst üste koymuş olduğunuz 10 tabaktan 2. sine gerek duyduğunuzda üstte kalan 8 tabağı kaldırdıktan sonra ancak 2. tabağa erişebilirsiniz. Başka bir deyişle Stack tipindeki koleksiyonlar **son giren ilk çıkar** mantığına bağlı olarak işlev görmektedir. Stack sınıfının işleyiş şeklini aşağıda verilen görsel iyi özetlemektedir.



Stack sınıfı **ICollection**, **IEnumerable** ve **ICloneable** arayüzlerini uygulamaktadır. Dolayısıyla yukarıda ArrayList sınıfı hakkında konuşurken sözü edilen metotların çoğu Stack nesnesi üzerinde işlem yapılırken kullanılabilir. Elinizde "son giren ilk çıkar" prensibine göre organize edilmeye uygun bilgiler varsa Stack nesnesi hazırlayabilirsiniz.

```
Stack s1 = new Stack();
```

Stack nesnesi hazırlarken kapasiteyi yapıcı metoda parametre olarak verebilirsiniz. Stack nesnesine eleman eklemek istiyorsanız **Push**() metodunu kullanmalısınız.

```
public static void Main()
{
 Stack s1 = new Stack(4);
 s1.Push("Gençlerbirliği");
 s1.Push("Galatasaray");
 s1.Push("Beşiktaş");
 s1.Push("Trabzonspor");
}
```

Bu şartlarda Push() metodu ile bu nesneye 4'ten fazla eleman eklerseniz nesnenin kapasitesi kendiliğinden artar. Stack nesneleri object tipinde bilgi saklayabildiği için her türde bilgi aktarılabilir. Stack sınıfının generic sürümünün olduğunu ve ilgili bölümde bundan söz edileceğini bilmenizi isteriz.

Stack nesnesinde tutulan elemanları elde edip kullanmak istiyorsanız **Pop**() metodunu kullanmalısınız. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Uygulama çalıştırılıp bu satır işletildiğinde Stack nesnesine Push() metodu ile en son eklenen bilgi okunur.

```
public static void Main()
{
 Stack s1 = new Stack(4);
 s1.Push("Gençlerbirliği");
 s1.Push("Galatasaray");
 s1.Push("Beşiktaş");
 s1.Push("Trabzonspor");
 object obj = S1.Pop();
}
```

Pop() metodunu arka arkaya birden fazla sayıda kullanıp Stack nesnesinin diğer elemanlarını elde edebilirsiniz. Yukarıda verilen konsol uygulaması aşağıdaki gibi düzenlenirse Stack nesnesine en son eklenen 2 elemanın içeriği konsola yazılır.

```
public static void Main()
{
 Stack s1 = new Stack(4);
 s1.Push("Gençlerbirliği");
 s1.Push("Galatasaray");
 s1.Push("Beşiktaş");
 s1.Push("Trabzonspor");
 Console.WriteLine(s1.Pop());
 Console.WriteLine(s1.Pop());
}
```

Nasıl ki Push() metodu Stack nesnesinin eleman sayısını arttırıyorsa, Pop() metodu her kullanıldığında eleman sayısı 1 azalır. Şimdi Stack nesnesinin bütün elemanlarını konsola yazdıracağız. Bütün elemanlar Pop() metodu ile okunduğunda eleman sayısı 0 olur.

```
public static void Main()
{
 Stack s1 = new Stack(4);
 s1.Push("Gençlerbirliği");
 s1.Push("Galatasaray");
 s1.Push("Beşiktaş");
 int eleman_sayi = s1.Count;
 for (int i = 0; i < eleman_sayi; i++)
   Console.WriteLine(s1.Pop());
}
```

Pop() metodu ile Stack'ten okuma yapıldığında en son eklenen elemanın içeriği geriye gönderilip bu eleman silinmektedir. Eleman okunurken silme işleminin yapılmasını istemiyorsanız Pop() yerine **Peek**() metodunu kullanmalısınız. **Peek**() metodu, Stack nesnesindeki bütün elemanları okumaktan çok stack'ın en üstünde duran eleman okunmak istendiği zaman işlevsel olmaktadır.

Stack nesnesinin bütün elemanlarını silmek istiyorsanız **Clear**() metodunu kullanabilirsiniz. Mevcut bir Stack nesnesinden yararlanarak aynı içeriğe sahip 2. bir stack nesnesi hazırlamak istiyorsanız **Clone**() metodu kullanılabilir.

```
Stack s2 = (Stack) s1.Clone();
```

Stack Nesnesinin İçeriğini ListBox ve Dizi Değişkene Aktarmak

Şimdi Visual Studio ile hazırladığımız projenin formuna bir ListBox yerleştirip ListBox'ın elemanlarını Stack nesnesine aktaracağız. Bu amaçla class bloğu içinde bir değişken tanımlayıp formun Load olayını temsil eden metotta Stack nesnesi hazırladık.

```
Stack yigin;
private void Form1_Load(object sender, EventArgs e)
{
 yigin = new Stack(5);
 yigin.Push("Gençlerbirliği");
 yigin.Push("Galatasaray");
 yigin.Push("Beşiktaş");
}
```

Stack nesnesinin bütün elemanlarını ListBox'a aktarmak için aşağıda verilen kodu yazdık. Bu kodda Stack nesnesinin elemanları Pop() metodu ile okunup ListBox'a aktarılmaktadır.

```
private void button1_Click(object sender, EventArgs e)
{
 int sayi = yigin.Count;
 for (int i = 0; i < sayi; i++)
  listBox1.Items.Add(yigin.Pop());
}
```

Benzer mantıkla Stack nesnesinin içeriğini dizi değişkene ve dizi değişkenin elemanlarını ise stack'e aktarmak mümkündür. Stack nesnesinin bütün elemanları Pop() metodu ile okunu dizi değişkene aktarıldığında Stack nesnesi boşalmaktadır. Stack sınıfının **CopyTo**() metodu ile Stack nesnesinin içeriğini kolayca dizi değişkene aktarabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] dizi;
dizi = new string[4];
S1.CopyTo(dizi, 0);
```

Stack nesnesinin içeriğini dizi değişkene kopyalarken CopyTo() yerine **ToArray**() metodunu kullanabilirsiniz. ToArray() metodunun nasıl kullanıldığını aşağıda görebilirsiniz. **ToArray**() metodu ile kopyalama yapılan dizi değişken object tipinde olmalıdır.

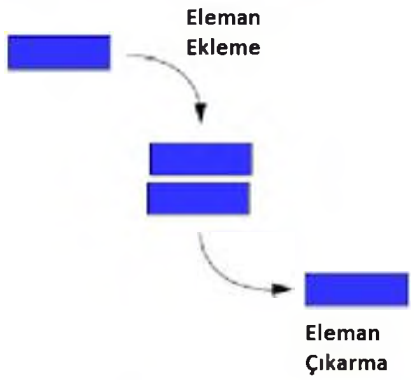
```
object[] dizi = new object[4];
dizi = S1.ToArray();
```

Yukarıdaki sayfalarda hakkında bilgi verilen ArrayList sınıfının da CopyTo() ve ToArray() metotları vardır. Bu metotlar Stack sınıfının metotlarıyla aynı işleve sahiptir.

Queue Sınıfı

Stack nesnelerinde tutulan bilgileri üst üste konulan tabaklara benzetmiştik. Başka bir deyişle bilgilerin sıralanışı ve elde edilişi **son giren ilk çıkar** prensibine dayanıyordu. Queue sınıfında ise bilgiler **ilk giren ilk çıkar** prensibi ile sıralanmaktadır. Örneğin bilet kuyruğuna girmiş kişiler söz konusu olduğunda sıraya ilk giren ilk çıkar.

Programcılıkta kuyruklar yani **ilk giren ilk çıkar** prensibi ile düzenlenmiş veri yapıları önemlidir. Bize sorarsanız burada Queue sınıfıyla ilgili olarak anlatılanlarla yetinmeyip Queue veri yapısı hakkında kavramsal bilgi edinmelisiniz.



System.Collections'de bulunan bu sınıf ICollection ve IEnumerable arayüzlerini uygulamaktadır. Dolayısıyla yukarıda hakkında bilgi verilen Stack sınıfıyla birkaç ortak metoda sahiptir. Elinizde ilk giren ilk çıkar prensibine uygun bilgiler varsa Queue tipinde koleksiyon hazırlayabilirsiniz.

```
Queue kuyruk = new Queue(10);
```

Bu satır ile kapasitesi 10 olan bir Queue nesnesi hazırlanmaktadır. Queue nesnesine eleman eklenmek istendiğinde **Enqueue**() metodu kullanılmaktadır.

```
Queue kuyruk = new Queue(10);
kuyruk.Enqueue("İlkbahar");
kuyruk.Enqueue("Yaz");
kuyruk.Enqueue("Sonbahar");
kuyruk.Enqueue("Kış");
```

Queue nesnesinden eleman çıkarma işlemi **Dequeue**() metodu ile yapılmaktadır. Aşağıda verilen konsol uygulaması derlenip çalıştırıldığında Queue nesnesine ilk dahil edilen "İlkbahar" ve "Yaz" bilgileri nesneden çıkarılıp konsola yazılır.

```
public static void Main()
{
 Queue kuyruk = new Queue(10);
 kuyruk.Enqueue("İlkbahar");
 kuyruk.Enqueue("Yaz");
 kuyruk.Enqueue("Sonbahar");
 kuyruk.Enqueue("Kış");
 Console.WriteLine(kuyruk.Dequeue());
 Console.WriteLine(kuyruk.Dequeue());
}
```

Bu şekilde **Dequeue**() metodu 2 kez işletildikten sonra Queue nesnesinin eleman sayısı 2'ye iner. Başka bir deyişle Dequeue() metodu her kullanıldığında en alttaki eleman Queue nesnesinden çıkarılıp geriye gönderilmektedir. Eleman okuma işlemi sırasında silme işleminin yapılmasını istemiyorsanız **Dequeue**() yerine **Peek**() metodunu kullanmalısınız.

Peek() metodu sürekli olarak Queue nesnesinin en alttaki elemanını okur. Örneğin yukarıda verilen örnekte Dequeue() 2 kez kullanıldıktan sonra Peek() metodu kullanılırsa Sonbahar bilgisi elde edilir. Tekrara düşmemek için Queue sınıfının Clear(), CopyTo(), ToArray() ve Contains() gibi metotlarından burada söz etmiyoruz.

Queue nesnesine aktarılan bilgileri **GetEnumerator**() metodu ile enum nesnesine alıp istediğiniz işlemleri yapabilirsiniz. Konunun anlaşılması için yeni bir proje hazırlayıp forma bir ListBox yerleştirip aşağıda verilen kodu yazdık. Bu kod işletildiğinde Queue nesnesinin eleman sayısı değişmeden bütün elemanlar elde edilip ListBox'a aktarılır.

```
private void Aktar_Click(object sender, EventArgs e)
{
 Queue kuyruk = new Queue(10);
 kuyruk.Enqueue("İlkbahar");
 kuyruk.Enqueue("Yaz");
 kuyruk.Enqueue("Sonbahar");
 System.Collections.IEnumerator liste;
 liste = kuyruk.GetEnumerator();
 while (liste.MoveNext())
   listBox1.Items.Add(liste.Current);
}
```

Hashtable Sınıfı

Hashtable sınıfı üzerine konuşmaya başlamadan önce bu kitabın **Object** sınıfı hakkında bilgi içeren bölümünden alınma 2 cümleyi buraya aktardık: ".NET uyumlu uygulamalarda hazırlanan her nesne veya tanımlanan her değişken HashCode adı verilen sayısal ve benzersiz bir koda sahiptir. Bu kod öğrenilerek söz konusu nesneye kolayca ulaşmak mümkün olmaktadır." Bu kodu öğrenmek istediğinizde Object sınıfının **GetHashCode**() metodunu kullanabilirsiniz.

Elinizde **anahtar-değer** şeklinde düzenlenmiş veriler varsa HashTable sınıfından yararlanarak bellekte tutabilirsiniz. Örneğin şehirlerarası telefon kodları, illerin plakaları ve sözlükler bu tür verilerdir. Hashtable sınıfı **IDictionary**, **ICollection**, **IEnumerable**, **ISerializable** ve **IColeneable** arayüzlerini uyguladığı için çok sayıda metoda sahiptir.

Hashtable sınıfından yararlanılarak hazırlanan veri yapılarında her eleman 2 bilgiye sahip olmaktadır. Bu bilgiler farklı tipte olabilir. Çünkü Hashtable nesnesinin elemanları **Object** tipindedir. HashTable yerine generic sınıf aradığınızda **Dictionary** sınıfını kullanabilirsiniz.

Hashtable nesnelerinin nasıl hazırlandığını ve anahtar-değer şeklinde düzenlenmiş bilgilerin Hashtable nesnesine nasıl eleman yapıldığını anlatmak için aşağıda verilen kodu hazırladık.

```
Hashtable iller = new Hashtable();
iller.Add("01", "Adana");
iller.Add("06", "Ankara");
iller.Add("07", "Antalya");
iller.Add("16", "Bursa");
iller.Add("33", "Mersin");
```

Add() metoduna ilk parametre olarak verilen bilgi elemanın anahtarını temsil etmektedir. Hashtable nesnesinde aynı anahtar bilgisine sahip birden fazla eleman olamaz. Aynı anahtarı 2. bir elemana verirseniz **ArgumentException** hatası meydana gelir. Ayrıca elemanların anahtar kısmı **null** olamaz.

Yukarıda Hashtable nesnesine eleman eklerken **Add**() metodunu kullandık. İsterseniz bu işlemi aşağıdaki gibi yapabilirsiniz. Aşağıda yapıldığı şekilde köşeli parantezler içinde parametre olarak verilen anahtar mevcut değilse eleman eklemiş olursunuz.

```
iller["01"] = "Adana";
iller["06"] = "Ankara";
iller["07"] = "Antalya";
```

Söz konusu anahtar bilgisi Hashtable nesnesinde varsa eleman eklenmeyip mevcut elemanın içeriği değişir. Buna göre aşağıda verilen satır eleman eklemeye neden olmaz. Ancak böyle bir satır Hashtable nesnesinin elemanlarının değişmesine neden olmaktadır.

```
iller["01"] = "Adanus";
```

Hashtable nesnesine Add() metodu ile eleman eklediğiniz gibi **Remove**() metodu ile eleman silebilirsiniz. Remove() metodu silinecek elemanın anahtarını parametre olarak almaktadır. Aşağıda verilen satır işletildiğinde anahtar bilgisi "34" olan eleman koleksiyondan silinir.

```
iller.Remove("34");
```

Hashtable nesnesinin bütün elemanlarını silmek için **Clear**() metodunu kullanabilirsiniz. Eleman sayısını öğrenmek istiyorsanız **Count** özelliğine bakabilirsiniz. Clear() metodunun ve Count özelliğinin nasıl kullanıldığını aşağıda görebilirsiniz.

```
iller.Clear();
int sayi = iller.Count;  // sayi 0 olur
```

Remove() metodu ile olmayan bir elemanı silmek istediğinizde size bir tepkide bulunulmaz. Silme işlemi önce elemanın varlığı **Contains**() veya **ContainsKey**() metodu ile araştırılmalıdır. Hashtable'da arama yapılırken ContainKey() ve ContainValue() metotları kullanılabilir.

```
object obj = textBox1.Text;
bool sonuc = iller.Contains(obj);
if (sonuc == true)
  iller.Remove(obj);
else
  MessageBox.Show("Bu eleman mevcut değil");
```

Yukarıda HashTable nesnesini hazırlarken HashTable sınıfının yapıcı metodunu parametresiz olarak kullandık. Bu sınıfın yapıcı metodunun aşırı yüklenmiş çok sayıda sürümü bulunmaktadır. Konuyu uzatmamak için yapıcı metodun birkaç değişik şeklini anlatmakla yetineceğiz. HashTable nesnesi aşağıdaki gibi hazırlanırsa kapasitesi 10 olarak ayarlanır.

```
Hashtable iller = new Hashtable(10);
```

Bu HashTable nesnesine 11. eleman eklenmek istendiğinde kapasite kendiliğinden iki katına çıkar. Hazırlamak istediğiniz HashTable nesnesinin kapasitesi elemanlarla tam olarak doldurulmadan arttırılmasını istiyorsanız 2 parametreli yapıcı metodu kullanmalısınız. Sözü edilen yapıcı metotla Hashtable nesnesinin nasıl hazırlandığını aşağıda görebilirsiniz.

```
float oran = 0.7f;
Hashtable iller = new Hashtable(10, oran);
```

HashTable nesnesi bu şekilde hazırlanırsa kapasitesinin %70'i kadar elemana sahip olduğunda kapasite otomatik olarak iki katına çıkarılır. Yapıcı metoda 0.1 ile 1.0 arasında değişen float bir değer 2. parametre olarak verilmezse doluluk oranının 1.0 olduğu varsayılıyor. Başka bir deyişle aşağıda verilen 2 satır aynı işleve sahiptir.

```
Hashtable iller = new Hashtable(10, 1.0f);
Hashtable iller = new Hashtable(10);
```

Hazırlamak istediğiniz HashTable nesnesinin kapasitesi konusunda mevcut bir koleksiyon nesnesinin rehberlik etmesini istiyorsanız IDictionary arayüzünü uygulamış bir nesneyi yapıcı metoda parametre olarak vermelisiniz. IDictionary arayüzünü uygulanmış bir nesneyi parametre olarak alan yapıcı metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
float oran = 0.7f;
Hashtable rehber = new Hashtable(10, oran);
Hashtable iller = new Hashtable(rehber);
```

Bu satırlar işletildiğinde "iller" adını verdiğimiz HashTable nesnesinin kapasitesi 10 ve doluluk oranı %70 olur. Kendi hazırlayıp **IDictionary** arayüzünü uyguladığınız sınıfların örneklerini yapıcı metoda parametre olarak verebilirsiniz.

Yukarıda anlatılan şekilde Hashtable nesnesi hazırlayıp anahtar-değer şeklinde düzenlenmiş bilgileri aktardıktan sonra konu ezbere ilerlemesin diye maddeler halinde ek açıklama yapmayı uygun gördük.

- HashTable tipindeki nesnelerde anahtar-değer şeklinde düzenlenmiş bilgilerin her birisine Hash Code adı verilen benzersiz bir kod verilmektedir. Programcılıkta Hashing mekanizması önemli olduğu için hakkında kavramsal bilgi edinmeniz faydalı olacaktır.
- Bu bölüm dahilinde hakkında konuşulacak **SortedList** sınıfı mevcut verileri kendi içinde, anahtar bilgilerine göre alfabetik olarak sıralamaktadır. HashTable sınıfı ise bu işlemi hash kodlarına göre yapmaktadır. Hash kodlarını kullanması nedeniyle HashTable sınıfının performansı yüksek olsa bile verilerin düzensiz olarak tutulmasına neden olmaktadır.
- Daha önceki konulardan hatırlayacağınız gibi Array ve ArrayList sınıflarından yola çıkılarak hazırlanan koleksiyonlardaki bilgilere indeks numarası ile erişilmektedir. Çünkü her iki sınıfta dahili koleksiyon indekslenmektedir. Bazen bu yöntem uygun olmayabilir. Çünkü hangi nesnenin hangi indekse karşılık geldiğini bilmemiz gerekir. Hal bu ki Hashtable koleksiyonun her elemanına istediğimiz bir ismi verip elemanlara bu isimlerle erişebiliriz. Bunun diğer adı anahtar-değer ilişkisidir

Hashtable Nesnesinin Elemanlarını ListBox'a Aktarmak

Şimdi Hashtable nesnesinin elemanlarının anahtar bilgilerinin nasıl elde edilip kullanıldığını anlatacağız. Bu amaçla forma bir ListBox yerleştirdik. Bu ListBox'a elemanların anahtarlarını aktaracağız. Bu işlemi yapacak satırlar aşağıdadır.

```
ICollection anahtar = iller.Keys;
foreach (object item in anahtar)
{
 listBox1.Items.Add(item.ToString() );
}
```

Bu satırlarda ilk olarak **ICollection** tipinde bir değişken tanımlayıp elimizdeki Hashtable nesnesinin **Keys** özelliğinin referansını bu değişkene aktardık. Devamında bu referanstan yararlanıp Hashtable koleksiyonun elemanlarını **foreach** döngüsü ile ListBox'a aktardık.

Hashtable nesnesinin elemanları okunurken elemanın anahtar bilgisi Hashtable nesnesine köşeli parantezler arasında parametre olarak verilir. Bu özellikten yararlanarak anahtarı bilinen elemanın **değerinin** nasıl elde edildiğini göstermek için forma 2. bir ListBox yerleştirip ve foreach döngüsüne aşağıda işaret edilen satırı ekledik. Bu satır sayesinde döngünün her turunda anahtar bilgisi parametre olarak verilerek elemanın değeri elde edilir.

```
ICollection anahtar = iller.Keys;
foreach (string str in anahtar)
{
 listBox1.Items.Add(str);
 listBox2.Items.Add(iller[str]);
}
```

Yukarıda **ICollection anahtar = iller.Keys;** satırı ile Hashtable nesnesinin anahtar bilgilerini içeren bir nesne hazırlamıştık. Söz konusu satırda **Keys** yerine **Values** seçeneğini kullanmış olsaydık hazırlanacak nesnede anahtar bilgileri yerine eleman metinleri olurdu. Values özelliğinin nasıl kullanıldığını anlatmak için ICollection tipinde 2. bir nesne hazırlayıp foreach döngüsü ile elemanları 2. ListBox'a aktardık.

```
private void aktar_Click(object sender, EventArgs e)
{
 ICollection anahtar = iller.Keys;
ICollection deger = iller.Values;
foreach (string str in anahtar)
  listBox1.Items.Add(str);
foreach (string str in deger)
  listBox2.Items.Add(str);
}
```

GetEnumerator() Metodu ve IDictionaryEnumerator Arayüzü

GetEnumerator() metodundan yararlanarak hem anahtar hem değer bilgilerini içeren **IDictionaryEnumerator** arayüzü tipinde bir numaralayıcı hazırlayıp bu numaralayıcı sayesinde Hashtable nesnesinin elemanları arasında dolaşabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
IDictionaryEnumerator liste;
Hashtable iller;
private void Form1_Load(object sender, EventArgs e)
{
 iller = new Hashtable();
 iller.Add("01", "Adana");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("16", "Bursa");
 iller.Add("33", "Mersin");
 liste = iller.GetEnumerator();
}
```

Bu kodda önce "liste" adında ve **IDictionaryEnumerator** arayüzü tipinde bir değişken tanımladık. Ardından **GetEnumerator()** metodu ile mevcut Hashtable nesnesinin içeriğini

bu numaralayıcıya aktardık. Bu numaralayıcıdan yararlanarak Hashtable nesnesinin anahtar ve değer bilgilerine kolayca erişebiliriz. Bu numaralayıcıdan nasıl yararlanıldığını anlatmak için forma 2 TextBox ve bir düğme yerleştirip aşağıda verilen kodu hazırladık.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 liste.MoveNext();
 textBox1.Text = liste.Key.ToString();
 textBox2.Text = liste.Value.ToString();
}
```

Bu satırlar işletildiğinde numaralayıcının bir sonraki elemanın üzerinde gidilir ve Hashtable nesnesinden alınan anahtar bilgisi ilk TextBox'a, değer bilgisi ise 2. TextBox'a aktarılır. Hashtable nesnesinin ve dolayısıyla **IDictionaryEnumerator** tipindeki koleksiyonun son elemanın üzerinde iken **MoveNext**() metodu tekrar işletilirse hata meydana gelmez ama **Key** ve **Value** özelliklerinin içerikleri okunmak istendiğinde hata meydana gelir. Bunun önüne geçmek için yukarıda verilen kodu aşağıdaki gibi düzenledik.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 if (liste.MoveNext())
  {
   textBox1.Text = liste.Key.ToString();
   textBox2.Text = liste.Value.ToString();
  }
}
```

Tekrarlamak gerekirse; önce IDictionaryEnumerator tipinde bir değişken tanımladık. Ardından anahtar-değer çifti şeklinde düzenlenmiş bilgileri içeren bir Hashtable nesnesi hazırladık. En son olarak GetEnumerator() metodundan yararlanıp Hashtable nesnesinin elemanlarını IDictionaryEnumerator tipindeki numaralayıcıya aktardık. Bu sayede HashTable nesnesinin elemanlarına erişim imkanı bulduk. **MoveNext()** ile üzerine gidilen elemanın anahtarı **Key**, değeri ise **Value** özelliğine aktarıldığı için bu bilgileri alıp kullandık.

GetEnumerator() metodu ile hazırlanıp IDictionaryEnumerator tipindeki değişkene aktarılan enum tipinden (numaralayıcıdan) geçerli anahtar ve değer bilgilerini bir seferde okumak istiyorsanız **Entry** özelliğine bakabilirsiniz. Entry özelliğinin nasıl kullanıldığını anlatmak için forma 2 ListBox yerleştirip aşağıda verilen kodu yazdık.

```
private void aktar_Click(object sender, EventArgs e)
{
 while (liste.MoveNext())
  {
   listBox1.Items.Add(liste.Entry.Key);
   listBox2.Items.Add(liste.Entry.Value);
  }
}
```

SortedList Sınıfı

Yukarıdaki sayfalarda HashTable sınıfı hakkında konuşmaya çalıştık, bu sınıf sayesinde anahtar-değer şeklinde düzenlenmiş bilgilerin liste haline getirilip üzerinde nasıl işlem yapıldığından söz ettik. SortedList ile HashTable sınıfları benzer işleve sahip oldukları için SortedList sınıfıyla yapılabilinen işlemlerden söz etmeden önce bu 2 sınıf arasındaki 2 önemli farkı dikkatinize sunacağız.

Hashtable nesnelerinin elemanlarına anahtar bilgileri ile erişilmektedir. SortedList sınıfı sayesinde hazırlanan veri yapılarındaki elemanlara hem anahtar ile hem de sıra numarası ile erişim sağlamak mümkündür. Başka bir deyişle SortedList koleksiyonuna aktarılan anahtar-değer çiftleri HashTable koleksiyonundan farklı olarak sıralı bir şekilde tutulmaktadır.

Hem HashTable hem de SortedList nesnelerine anahtar-değer şeklinde düzenlenmiş veriler eklenirken kendi içlerinde farklı sıralama yöntemi kullanıyorlar. HashTable sınıfı bu sıralama işlemini Hash kodlarına göre yaparken SortedList sınıfı ise anahtar bilgilerinin alfabetik sırasına göre yapmaktadır. Bu işlemden dolayı HashTable sınıfının performansı yüksek olmaktadır. SortedList, sınıfı Hashtable'a göre daha yavaş olsa bile IndexOfKey(), IndexOfValue(), GetByIndex, SetByIndex() metotları sayesinde indeksleme daha esnektir.

SortedList sınıfı Hashtable sınıfının uygulamış olduğu bütün arayüzleri (**IDictionary**, **ICollection**, **IEnumerable**, **ISerializable** ve **IColeneable**) uyguladığı için HashTable sınıfı ile çok sayıda ortak metot ve özelliğe sahiptir. Bu sınıf, üzerinde işlem yapılmak istenen veriler fazla değilse tercih edilebilir.

Aşağıda verilen satırlar bir metoda yazılıp işletildiğinde kapasitesi 4 olan bir SortedList nesnesi hazırlanır ve Add() metodu ile 5 eleman eklenir. 5. eleman nesneye eklenirken SortedList nesnesinin kapasitesi yetmeyeceği için 2 katına yani 8'e çıkarılır.

```
SortedList sozluk = new SortedList(4);
sozluk.Add("Screen", "Ekran");
sozluk.Add("Computer", "Bilgisayar");
sozluk.Add("Notebook", "Defter");
sozluk.Add("Printer", "Yazıcı");
sozluk.Add("Scanner", "Tarayıcı");
```

Bu şekilde SortedList nesnesine eklenen elemanlar sıralı olur. Sıra numarasına göre SortedList nesnesinden anahtar bilgisi elde etmek istiyorsanız **GetKey**() metodundan yararlanıp aşağıdaki gibi bir satır yazabilirsiniz. Bu satır sayesinde SortedList nesnesinin ilk elemanın anahtar bilgisi TextBox'a aktarılır.

```
textBox1.Text = sozluk.GetKey(0).ToString();
```

Add() metodu ile SortedList nesnesine eklenen elemanlar anahtara göre sıralanmaktadır. Bu nedenle yukarıda verilen satır işletildiğinde TextBox'a "Screen" yerine "Computer" yazılır. Sıra numarasını kullanarak elemanın anahtarı yerine kendisini elde etmek istiyorsanız **GetByIndex**() metodunu aşağıdaki gibi kullanabilirsiniz.

```
textBox2.Text = sozluk.GetByIndex(0).ToString();
```

Şimdi yukarıda hazırlayıp **Add**() metodu ile birkaç eleman aktardığımız SortedList nesnesinin anahtar ve değer bilgilerini ListBox'lara aktaracağız. Bu amaçla forma 2 ListBox yerleştirip aşağıda verilen kodu hazırladık. Bu satırlar sayesinde SortedList nesnesinin mevcut elemanların anahtar ve değerleri sıralanmış halde ListBox'lara aktarılır.

```
int sayi = sozluk.Count;
for (int i=0; i<sayi; i++)
 {
  listBox1.Items.Add(sozluk.GetKey(i).ToString());
  listBox2.Items.Add(sozluk.GetByIndex(i).ToString());
 }
```

SortedList nesnesinin elemanlarının anahtar bilgilerine erişimde kullanmak üzere bir nesne hazırlamak istiyorsanız **GetKeyList**(), elemanların değer bilgilerine erişim için bir nesne hazırlamak için **GetValueList**() metodundan yararlanabilirsiniz. Çünkü SortedList sınıfı **IList** arayüzünü uygulamaktadır. **GetKeyList**() ve **GetValueList**() metotlarının nasıl kullanıldığını aşağıda görebilirsiniz.

```
IList anahtarlar = sozluk.GetKeyList();
IList degerler = sozluk.GetValueList();
```

GetKeyList() ve GetValueList() metodu geriye **IList** tipinde nesne göndermektedir. Bu 2 satır sayesinde elde edilen nesnelerden yararlanarak HashTable nesnesinin anahtar ve değer bilgilerini elde edip istediğiniz gibi kullanabilirsiniz. Şimdi bu nesnelerden yararlanarak anahtar ve değer bilgilerini ListBox'lara aktaracağız.

```
IList anahtarlar = sozluk.GetKeyList();
IList degerler = sozluk.GetValueList();
foreach (string anahtar in anahtarlar)
     listBox1.Items.Add(anahtar);
foreach (string deger in degerler)
     listBox2.Items.Add(deger);
```

Bazen anahtar veya değer bilgisi bilinen elemanın SortedList nesnesi içindeki sıra numarası öğrenilmek istenir. Bu gibi durumlarda **IndexOfKey**() ve **IndexOfValue**() metotları kullanılır. Bu 2 metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
textBox1.Text = sozluk.IndexOfKey("Cpu").ToString();
textBox2.Text = sozluk.IndexOfValue("Yazıcı").ToString();
```

Bu 2 satır sayesinde anahtar bilgisi "Cpu" olan elemanın sıra numarası formdaki ilk TextBox'a ve değer bilgisi "Yazıcı" olan elemanın sıra numarası ise 2. TextBox'a aktarılır. Bu 2 metoda parametre olarak verilen anahtar veya değer bilgisi SortedList nesnesinde yoksa bu metotlar geriye -1 gönderirler.

SortedList sınıfının **GetValueList**() metodu ile hazırlanan nesnenin üzerinden koleksiyona eleman ekleyebilir, mevcut elemanları silebilir ve arama yapabilirsiniz. SortedList nesnesinde sıra numarasını bildiğiniz elemanın değer bilgisini öğrenirken **GetByIndex**() metodunu kullanmak gerektiğini biliyorsunuz. Sıra numarasını bildiğiniz bir elemanın değer bilgisini değiştirmek istiyorsanız **SetByIndex**() metodunu kullanabilirsiniz.

```
private void Bul_Click(object sender, EventArgs e)
{
 textBox1.Text = sozluk.GetByIndex(2).ToString();
 sozluk.SetByIndex(2, "Laptop");
 textBox2.Text = sozluk.GetByIndex(2).ToString();
}
```

Burada **GetByIndex**() metoduna parametre olarak 2 verildiği için geriye 3. elemanın değer bilgisi gönderilir. Devamında SortedList nesnesinin 3. elemanın değer bilgisi **SetByIndex**() ile metodu ile değiştirilmektedir.

SortedList sınıfıyla ilgili olarak şunlara vurgu yaparak konuyu tamamlayacağız: SortedList nesnesine eleman eklendiğinde veya mevcut elemanlardan birisi çıkarıldığında mevcut elemanlar sıralama işlemine tabi tutulur. Bu da SortrdList sınıfını yukarıda hakkında bilgi verilen sınıflara göre düşük performanslı yapar. Programcılara, güncellemenin fazla olmadığı ve az elemanlı listeler hazırlamak istendiklerinde bu sınıfa başvurmaları önerilmektedir.

OrderedDictionary Sınıfı

Yukarıdaki sayfalardan hatırlayacağınız gibi **HashTable** nesnesi içinde saklanan anahtar-değer bilgilerine anahtar bilgisi ile erişim sağlanabiliniyordu. Değer bilgisinden yola çıkıp elemanın anahtarını elde etmek için **GetEnumerator**() metodu ile enum tipi hazırlamak gerekiyordu. SortedList nesnelerinde ise anahtar-değer'den meydana gelen elemanlar otomatik olarak sıralanıyordu. Bu nedenle SortedList nesnesine eklenen elemanın ekleme anındaki sıra numarasına gerek duyulması halinde SortedList sınıfı yeterli olmamaya başlar.

Bu sorunu aşmak istiyorsanız **OrderedDictionary** sınıfından yararlanabilirsiniz. **System.Collections.Specialized**'de yer alan OrderedDictionary sınıfından yararlanırken bu namespace'in kaynaklarından yararlanmak istediğinizi **using** deyimi ile belirtebilirsiniz. Bu sınıfın HashTable'dan farkını göstermek için aşağıdaki gibi bir nesne hazırlayıp 4. sırada eklenen elemanın değer bilgisini TextBox'a aktarmak istedik.

```
private void Form1_Load(object sender, EventArgs e)
{
 OrderedDictionary iller = new OrderedDictionary();
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("16", "Bursa");
 iller.Add("01", "Adana");
 iller.Add("33", "Mersin");
textBox1.Text = iller[3].ToString();
}
```

Tahmin edebileceğiniz gibi bu kod işletilirse TextBox'a "Adana" yazılır. "iller" adını verdiğimiz bu koleksiyon HashTable sınıfından yola çıkılarak hazırlasaydık **GetByIndex**() metodu ile koleksiyonun 3. elemanın değer bilgisi okunduğunda bu değer "Adana" olmayıp "Bursa" olurdu. OrderedDictionary nesnesinin indeksleyicisine ilgili elemanın sıra numarası yerine anahtarını parametre olarak verip değer bilgisini elde etmek mümkündür.

```
textBox1.Text = iller["01"].ToString();
```

Generic Olan-Olmayan Koleksiyonlar

C# 2.0 kullanıma sunulduğundan bu yana generic koleksiyonların tercih edilmesi öneriliyor. Bizim de önerimiz bu yöndedir. Yine de generic olanla olmayan koleksiyonların performansı bağlamında birkaç cümle yazmakta fayda var. Henüz hakkında bilgi vermediğimiz bazı generic koleksiyon sınıflarının burada örneklerini alıyor olsak bile bunun üzerinde durmanızı istemeyiz. Aşağıda verilen kodda önce 3 özelliğe sahip bir sınıf hazırladık.

```
public class Kitaplar
{
 public string kitapAdi { get; set; }
 public string yazari { get; set; }
 public int sayfaSayi{ get; set; }
}
```

Bu şekilde sınıf hazırladıktan sonra uygulamanın Main() metodu içinde ArrayList ve List nesneleri hazırladık. Bu koleksiyon nesnelerine "Kitaplar" tipinde bilgileri aktaracağız. Koda yaptığımız eklemeyi aşağıda görebilirsiniz.

```
public class Kitaplar
{
 public string kitapAdi { get; set; }
 public string yazari { get; set; }
 public int sayfaSayi{ get; set; }
}
class Program
{
 static void Main(string[] args)
 {
  ArrayList liste1 = new ArrayList();
  List<Kitaplar> liste2 = new List<Kitaplar>();
  Kitaplar kitap1 = new Kitaplar()
  {
    kitapAdi = "Java 8",
    yazari = "Memik Yanık",
    sayfaSayi = 640
   };
   liste1.Add(kitap1);
   liste2.Add(kitap1);
  }
}
```

Konu uzamasında diye "Kitaplar" tipinde sadece bir nesne hazırladık ve bu nesneyi hem ArrayList nesnesine hem de List nesnesine Add() metodu ile ekledik. Şimdi soruyoruz: Elimizde 100 ya da 1000 kitap olsaydı ve bu kitapların her birisi için ayrı "Kitaplar" nesnesi hazırlasaydık sizce hangi koleksiyon nesnesi daha performanslı olurdu, hangisi daha az bellek tüketirdi? Bu sorunun cevabını merak eden arkadaşlar testler yapabilir. Bizim tahminimiz böyle bir işlem söz konusu olduğunda ArrayList ile List sınıflarının birbirlerine performans bağlamında kayda değer bir üstünlükleri yoktur. Çünkü saklanan "Kitaplar" tipinde nesnelerdir. Konunun diğer ayrıntılarını ortaya çıkarmak için yukarıda verdiğimiz kodu aşağıdaki gibi değiştirdik. Bu kez koleksiyon nesnelerimize integer bilgileri aktarıyoruz.

```
static void Main(string[] args)
{
 ArrayList liste1 = new ArrayList();
 liste1.Add(2015);
 liste1.Add(2016);
 List<int> liste2 = new List<int>();
 liste2.Add(2015);
 liste2.Add(2016);
}
```

Her iki koleksiyon nesnemize ikişer yıl aktardık. Bir for döngüsü hazırlayıp milyonlarca integer bilgiyi List ve ArrayList nesnemize aktarabilirdik. Bu şartlarda generic olanla olmayan arasında performans ve bellek kullanımı nasıl mı olurdu? Tahminimize göre List nesnesi yaklaşık 4 kat daha hızlı olurdu ve dörtte biri kadar bellek kullanırdı.

Liste tipi koleksiyonda int, long, double, byte gibi value type tutuyorsanız, generic olan koleksiyonları kullanın önerimizin asıl nedeni bu performans farkı değildir. Asıl neden, ArrayList'in tip güvenli olmamasıdır. Yani her türlü bilgiyi kabul etmesidir. Bu da hatalara açık bir durumdur. Bu son cümlemizi somutlaştırmak için aşağıda verilen kodu yazdık.

```
class Program
{
 static void Main(string[] args)
 {
  ArrayList liste1 = new ArrayList();
  liste1.Add("Fenerbahçe");
  liste1.Add(1903);
  liste1.Add("Galatasaray");
  foreach (string item in liste1)
  {
   Console.WriteLine(item);
  }
  Console.ReadKey();
 }
}
```

Bu kodu yazarken Visual Studio hiçbir uyarıda bulunmaz. Derleme sırasında herhangi bir hata mesajı alınmaz. Uygulama çalıştırıldığında foreach döngüsünün ilk turunda konsola "Fenerbahçe" yazılır. Ne zaman ki sıra ArrayList nesnesinin 2. elemanını okumaya gelir, hata meydana gelir. Bu kodda ArrayList yerine generic koleksiyon kullansaydık böyle bir hatayı yapmazdık. Çünkü Visual Studio en başından bizi uyarıp hata rapor ederdi.

```
static void Main(string[] args)
{
    List<string> liste1  = new List<string>();
    liste1.Add("Fenerbahçe");
    liste1.Add(1903);
    liste1.Add("Galatasaray");
    foreach (string item in liste1)
    {
        Console.WriteLine(item);
    }
    Console.ReadKey();
}
```

IEnumerable ve IEnumerator Arayüzleri

Yukarıdaki sayfalarda sözü edilen koleksiyon sınıflarının hepsi **IEnumerable** arayüzünü uyguladığı için hepsi **GetEnumerator()** metoduna sahiptir. foreach döngüsü ile koleksiyonun elemanları üzerinde ileri doğru hareket edebilmenin temelinde IEnumerable arayüzü vardır. Bu nedenle bu arayüz çok önemlidir, .NET programcısının eli ayağı gibidir. Bu arayüzün orijinal yapısı aşağıdaki gibidir.

```
public interface IEnumerable
{
 public IEnumerator GetEnumerator();
}
```

Bu arayüzün GetEnumerator() metodu geriye **IEnumerator** tipinde nesne göndermektedir. .NET Framework ile gelen bütün koleksiyon sınıfları IEnumerable arayüzünü uyguladığı için **GetEnumerator()** metodu ile IEnumerable nesnesi hazırlamak mümkündür. Yukarıdaki sayfalarda bazı koleksiyon sınıflarının IEnumerator arayüzü kaynaklı **MoveNext()** metodunu kullandık. IEnumerator arayüzünün orijinal yapısı aşağıdadır.

```
public interface IEnumerator
{
 public bool MoveNext();
 public void Reset();
 public object Current {get;}
}
```

Bu arayüz, **MoveNext()** ve **Reset()** adında 2 metoda sahiptir. MoveNext() metodu, GetEnumerator() metodu ile hazırlanan listedeki yani numaralayıcıdaki bir sonraki elemanın üzerine gitmeye neden olurken, Reset() metodu ilk elemana gidilmesini sağlıyor. **Current** özelliği sayesinde üzerinde bulunulan elemanın içeriği elde edilebilmektedir. Tahmin edeceğiniz gibi hem IEnumerable hem de IEnumerator arayüzünün generic sürümleri vardır. Bu arayüzlerin generic sürümleri üzerinde daha sonra durulacaktır.

IEnumerable ve IEnumerator arayüzlerinden kısaca söz ettikten sonra aşağıdaki gibi bir ArrayList nesnesi hazırlayıp dahili koleksiyona Add() metoduyla 4 eleman ekledik. Amacımız bu arayüzlerin işlevinden söz etmektir.

```
private void button1_Click_1(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
}
```

Yukarıdaki sayfalarda anlatılanlardan bildiğiniz gibi ArrayList sınıfı IEnumerable ve IEnumerator arayüzlerini uyguladığı için foreach desteğine sahiptir. Yani bu koda ekleme yapıp foreach döngüsü ile bu ArrayList nesnesinin elemanları üzerinde ileri doğru hareket edilip elemanlar elde edilebilir. Hazırladığımız foreach döngüsünü aşağıda görebilirsiniz.

```
private void button1_Click_1(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 foreach (string eleman in liste)
 {
  string takim = eleman.ToUpper();
  listBox1.Items.Add(takim);
 }
}
```

Bu foreach döngüsünde sizin için yeni bir şey olmasa bile birkaç ayrıntıdan söz etmek istiyoruz. Yoksa IEnumerator ve IEnumerator arayüzlerinin buradaki işlevi görülemeyebilir.

- Yukarıda IEnumerator arayüzünün **Current** özelliğinin object tipinde olduğu işaret edilmişken burada foreach'e ait parantezin içinde **string** tipini kullandık. Bu nedenle derleyici geri planda bir dönüştürme yapıp string bilgi içeren object değişkenin içeriğini string tipe dönüştürdü. Foreach döngüsü içinde kullanılacak tipin derleyici tarafından seçilmesini isteyenler **var** anahtar kelimesinden yararlanabilirler.

- Burada IEnumerator arayüzü kaynaklı **Current** özelliğinden söz etme nedenimiz, foreach döngüsünün her turunda bu özelliğin okunuyor olmasıdır.

- ArrayList sınıfı dahili koleksiyonundaki elemanları indekslediği için burada for döngüsünü kullanabilirdik. Foreach döngüsünden söz ederken for hakkındaki bilginizden yararlanmak için yukarıda verilen kodun for halini hazırladık.

```
private void button1_Click_1(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 for (int i = 0; i < liste.Count; i++)
  {
   string takim = liste[i].ToString();
   listBox1.Items.Add(takim.ToUpper());
  }
}
```

Şimdiye kadar defalarca kullandığınız for döngülerinde for anahtar kelimesi tarafından yapılan tek şey, döngünün her turunda burada "i" adı verilen döngü kontrol değişkenin içeriğinin bir arttırılmasıdır. Döngünün içinde ne yapıldığıyla ilgilenilmez. Bu for döngüsünün içinde ArrayList sınıfının dahili koleksiyonun indekslenmiş olmasından yararlanıp koleksiyonun elemanlarını elde ettik.

Bu anlatılanlara göre for anahtar kelimesi kullanılıp döngü hazırlandığında geri planda herhangi bir işlem yapılmazken foreach'te bir çok işlem yapılmaktadır. Programın işletimi foreach döngüsüne geldiğinde geri planda; şu işlemler yapılmaktadır.

- IEnumerable arayüzü kaynaklı GetEnumerator() metodu işletilerek ArrayList nesnesinin dahili koleksiyonun elemanları IEnumerator tipindeki bir numaralayıcıya alınıyor.
- IEnumerator arayüzü kaynaklı **Reset**() metodu işletilerek koleksiyonun (GetEnumerator metodu ile hazırlanan numaralı listenin) başına, yani ilk elemanın önüne gidilir.
- Ardından **MoveNext**() metodu işletilerek ilk elemanın üzerine gidilir. Bu elemanın içeriği **Current** özelliğine aktarılıp geriye gönderilir. Böylece döngünün bir turu tamamlanmış olur.
- Bir sonraki turda Reset() metodu işletilmez. Ayrıca MoveNext() metodu işletilmeden önce koleksiyonun henüz üzerine gidilmemiş elemanı olup olmadığı kontrol edilir.
- MoveNext() metodu ile ileri doğru hareket edilip koleksiyonun sonuna ulaşıldığında ise foreach döngüsünden çıkılır.

Bu söylenenleri tek cümle ile özetlemek gerekirse, foreach döngüsü, sınıfın dahili koleksiyonu üzerinde ileri doğru hareket ederken Reset(), MoveNext(), GetEnumerator() metodundan ve Current özelliğinden yararlanmaktadır.

Şimdi konu üzerinde düşünmenizi sağlamak için bu metot ve özellikten (MoveNext metodu, Curtent özelliği) kendimiz yararlanıp koleksiyonun elemanları üzerinde ileri doğru hareket edeceğiz. Yani foreach döngüsünü kullanmayacağız. Bu amaçla hazırladığımız kod aşağıdadır.

```
private void aktar_Click(object sender, EventArgs e)
{
 ArrayList liste = new ArrayList();
 liste.Add("Fenerbahçe");
 liste.Add("Beşiktaş");
 liste.Add("Trabzonspor");
 liste.Add("Galatasaray");
 IEnumerator numarali_liste = liste.GetEnumerator();
 int sayi = liste.Count;
 for (int i = 0; i < sayi; i++)
 {
  numarali_liste.MoveNext();
  string eleman = numarali_liste.Current.ToString();
  listBox1.Items.Add(eleman);
 }
}
```

Bu kodda önce GetEnumerator() metodundan yararlanıp ArrayList nesnesinin dahili koleksiyondaki bütün elemanları IEnumerator tipindeki nesneye yani numaralı listeye aktardık. Bu işlemden sonra IEnumerator arayüzü kaynaklı Reset() ve MoveNext() metodu ile

Current özelliğinden yararlanabiliriz. Devamında bir for döngüsü kurup IEnumerator tipindeki nesnede tutulan (daha doğrusu bu nesne sayesinde erişilen) elemanların üzerinde MoveNext() metodu ile ilerleyip her turda Current özelliğinin içeriğini okuyup elde ettik.

Burada for yerine foreach döngüsü kursaydık bu işlemleri geri planda derleyici yapardı. Bize göre foreach döngüsünün varlığının programcılara 2 katkısı var: Aynı işlem için daha az kod yazmaları ve koleksiyonlar, IEnumerator ve IEnumerable arayüzleri hakkında bilgi sahibi olmadan liste tipi verilerdeki elemanlar üzerinde ileri doğru hareket edilebilmeleri.



Yukarıda ifade edildiği gibi IEnumerable arayüzü bir metoda, IEnumerator arayüzü ise 3 metoda sahiptir. Bu arayüzü uygulayan her sınıfta bu metotlarla farklı işlemler yapılıyor olabilir. Örneğin Hashtable sınıfında GetEnumerator() metodu için yazılan kodlar ArrayList'teki GetEnumerator() için yazılanlardan farklı olmalıdır.

IEnumerable ve IEnumerator Arayüzlerini Sınıflara Uygulamak

Yukarıdaki sayfalarda ArrayList, Hashtable ve SortedList gibi sınıflardan yararlanarak koleksiyon nesneleri hazırlayıp kullandık. Saydığımız bu sınıfların hepsi IEnumerable arayüzünü uyguladığı için GetEnumerator() metoduna sahiptirler. GetEnumerator() metodu ile hazırlanan **IEnumerator** tipindeki numaralayıcı sayesinde ilgili nesnenin elemanları üzerinde **foreach** döngüsü ile dolaşmak mümkün oluyordu.

Şimdi kendimiz bir koleksiyon sınıfı hazırlayacağız. Bu sayede bu arayüzlerin sınıflara nasıl uygulandığı ve bu arayüzdeki metotların nasıl hazırlanıp kullanıldığı konusu üzerinde düşünmüş olacağız.



Sadece .NET Framework ile gelen koleksiyon sınıflarının kullanımıyla ilgiliyim ve kendi koleksiyon sınıfımı henüz yazma noktasında değilim diyenler bu bölümün kalan sayfalarını daha sonra okumak üzere atlayıp bir sonraki bölüme geçebilirler.

Konunun kolay izlenmesini sağlamak için hazırlayacağımız koleksiyon sınıfının basit olmasını tercih ettik. Hazırlayacağımız projede formu temsil eden sınıftan başka "Firma" ve "Firmalar" adında 2 sınıf olacak. Her sınıf için ayrı bir CS dosyası hazırlayacağız. "Firma" adını verdiğimiz sınıfı aşağıda görebilirsiniz. Bu sınıf yalnızca string tipinde public bir alana sahip olacaktır. Kod kısa olsun diye bu alan için özellik hazırlamadık.

```
namespace WindowsApp1
{
 public class Firma
 {
   public string Firma_adi;
   public Firma(string F)
   {
    this.Firma_adi = F;
   }
 }
}
```

"Firma" adını verdiğimiz sınıfı bu şekilde hazırladıktan sonra **Add New Items** diyalog kutusunu tekrar ekrana getirip "Firmalar" adında 2. bir sınıf daha hazırlayıp projeye dahil ettik. Bu sınıfa uygulayacağımız IEnumerable ve IEnumerator arayüzleri **System.Collections**'da tanımlı oldukları için bu namespace'ten yararlanmak istediğimizi **using** deyimi ile belirttik.

```
namespace WindowsApp1
{
 class Firmalar : IEnumerator, IEnumerable
  {
  }
}
```

Bu sınıfın IEnumerator ve IEnumerable arayüzlerinde tanımlı metotlara sahip olmasını istediğimiz için bu arayüzleri sınıfa uyguladık. "Firmalar" sınıfının işlevsel olmasını hedeflemediğimiz için sınıfta tanımlanan dizi değişkene eleman eklemeyi sınıfın yapıcı metodunda yaptık. "Firmalar" sınıfı ayrıca "sira" adında bir özelliğe sahip olacak.

Bu sınıfın yapıcı metodunda yukarıda hazırladığımız "Firma" sınıfı tipinde ve 4 elemana sahip bir dizi değişken tanımladık. Ayrıca bu dizi değişkenin elemanlarına bilgi aktarma işlemini yapıcı metotta yaptık.

```
class Firmalar : IEnumerator, IEnumerable
{
 public Firma[] Liste;
 private int sira = -1;
 public Firmalar()
  {
   Liste = new Firma[4];
   Liste[0] = new Firma("Erpa İletişim");
   Liste[1] = new Firma("Kont Bilişim");
   Liste[2] = new Firma("Koyuncu Bilgisayar");
   Liste[3] = new Firma("Arena Bilgisayar");
  }
}
```

Bu sınıfta "Ekle" adında bir metoda yer verip koleksiyona eleman ekleme işlemini çalışma anında yapabilirdik. Bunu yapmama nedenlerinden birisi şimdilik IEnumerator ve IEnumerable arayüzleriyle ilgileniyor olmamızdır.

Firmalar adını verdiğimiz sınıfımızın bu halinde sizin için yeni bir şey içermiyor olmasına rağmen maddeler halinde üzerinde biraz konuşacağız.:

- Bu sınıftan önce "Firma" adını verdiğimiz bir sınıf hazırladık. Dolayısıyla elimizde her birisi ayrı bir firma hakkında bilgi içerecek çok sayıda "Firma" tipinde nesne olacak. Bu nesneler hazırlanırken doğal olarak veritabanından bilgi çekilecektir.
- "Firmalar" adını verdiğimiz 2. sınıfta ise "Firma" tipindeki nesnelerimiz üzerinde işlem yapılacaktır. Burada yapmak istediğimiz işlem, firma listesi üzerinde ileri doğru hareket edip her firmayla ilgili bilgilere erişebilmektedir. Firma listesinden söz

edildiğine göre "Firmalar" sınıfımız dahili bir koleksiyona sahip olacaktır. Bu koleksiyon "Firma" tipindeki nesneleri tutacaktır.

- Bu amaçla "Firmalar" sınıfında "Firma" tipinde 4 elemanlı bir dizi değişken tanımladık. Gerçek hayatta bu dizi değişkenin eleman sayısı veritabanından çekilecek bilgilerin sayısına göre ayarlanırdı. Burada işin kolayına kaçıp eleman sayısını 4 olarak sabitledik. Yapıcı metot dahilinde 4 adet "Firma" nesnesi hazırlayıp "Liste" adını verdiğimiz dahili koleksiyona ekledik.

Yapıcı metodu hazırladıktan sonra sıra IEnumerator ve IEnumerable arayüzlerinin gereklerini yerine getirmeye geldi. Çünkü sınıfa arayüz uygulamak demek, sınıfın arayüzdeki metot ve özelliklere sahip olacağını kabul etmek demektir. Bu nedenle "Firmalar" adını verdiğimiz sınıfımız, IEnumerable arayüzündeki GetEnumerator() metoduna ve IEnumerator'deki **MoveNext**(), **Reset**() metotlarına ve **Current** özelliğine sahip olmalıdır.

Başka bir deyişle "Firmalar" adını verdiğimiz sınıf MoveNext(), Reset(), GetEnumerator() metotlarına ve Current özelliğine sahip olmalıdır. Reset() metodu koleksiyondaki ilk elemanın üzerine gitmede kullanıldığı için bu metotta "Firmalar" sınıfının "sira" özelliğine -1 aktarmakla yetindik. Geçerli kaydı elde etmeye yarayan **Current** özelliği ise yalnızca **get** bloğuna sahip olacak. IEnumerator ve IEnumerable arayüzlerini uygulayan "Firmalar" sınıfının tümü aşağıda verildi.

```
class Firmalar : IEnumerator, IEnumerable
{
 public Firma[] Liste;
 private int sira = -1;
 public Firmalar()
 {
   Liste = new Firma[4];
   Liste[0] = new Firma("Erpa İletişim");
   Liste[1] = new Firma("Kont Bilişim");
   Liste[2] = new Firma("Koyuncu Bilgisayar");
   Liste[3] = new Firma("Arena Bilgisayar");
 }
 public void Reset()
  {
   sira = -1;
  }
 public bool MoveNext()
 {
  if (sira < Liste.Length - 1)
   {
    sira++;
    return true;
   }
   else
    {
     return false;
    }
  }
  public object Current
  {
```

```
    get
    {
      return Liste[sira];
    }
  }
  public IEnumerator GetEnumerator()
  {
   return (IEnumerator)this;
  }
}
```

Burada "Firmalar" adını verdiğimiz sınıfın örneğini hazırlayıp dahili koleksiyonun elemanları üzerinde foreach döngüsü sayesinde ileri doğru hareket edip elemanlar tek tek elde edilirken geri planda işletilen metotları hazırladık. Bu metotların hepsi foreach döngüsü tarafından kullanılacaktır.

Bu hazırlıklardan sonra projenin ilk formunun Load olayını temsil eden metodunu aşağıdaki gibi düzenleyerek firma adlarını ListBox'a aktardık. Kendi hazırladığımız sınıfa IEnumerator ve IEnumerable arayüzlerini uygulamış olmasaydık foreach döngüsü ile koleksiyonun elemanları üzerinde ileri doğru hareket edemezdik.

```
private void Form1_Load(object sender, EventArgs e)
{
 Firmalar nesne = new Firmalar();
 foreach (Firma F in nesne)
   listBox1.Items.Add(F.Firma_adi.ToString());
}
```

Aşağıda verilen ekran görüntüsünü hazırladığımız projeyi çalıştırdıktan sonra aldık.



GetEnumerator() metodu hakkında bilgi vermek için ListBox'ı silip yerine bir TextBox ve "Sonraki" başlıklı bir düğme yerleştirdik. Proje çalıştırıldığında ilk firmanın adı TextBox'a yazılacak. Bu amaçla formun Load olayıyla ilgili metodu aşağıdaki gibi düzenledik.

```
Firmalar nesne = new Firmalar();
IEnumerator numarali_liste;
Firma F;
private void Form1_Load(object sender, EventArgs e)
{
 numarali_liste = nesne.GetEnumerator();
 numarali_liste.MoveNext();
 F = (Firma) numarali_liste.Current;
 textBox1.Text = F.Firma_adi.ToString();
}
```

Değişken tanımlamayı metodun dışında yaptığımız için bu nesneleri başka metotlarda kullanabiliriz. TextBox'ta ilk firmanın adı görüntülenirken "Sonraki" düğmesi tıklandığında bir sonraki firma adının TextBox'a yazılmasını sağlamak için aşağıda verilen kodu hazırladık.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 numarali_liste.MoveNext();
 F = (Firma) numarali_liste.Current;
 textBox1.Text = F.Firma_adi.ToString();
}
```

Bu kitapta şimdiye kadar birkaç kez ilgili sınıfa IEnumerable ve IEnumerator arayüzlerinin mutlaka uygulanması gerektiğinden söz ettik. Şimdi bu 2 arayüzü yukarıda verdiğimiz sınıfımıza uygulamaktan vazgeçeceğiz. Ancak sınıfımızdaki metotlara dokunmayacağız. İşte sınıfımızın son hali.

```
class Firmalar
{
  public Firma[] Liste;
  private int sira = -1;
  public Firmalar()
  {
   Liste = new Firma[4];
   Liste[0] = new Firma("Erpa İletişim");
   Liste[1] = new Firma("Kont Bilişim");
   Liste[2] = new Firma("Koyuncu Bilgisayar");
   Liste[3] = new Firma("Arena Bilgisayar");
  }
  public void Reset()
  {
    sira = -1;
  }
  public object Current
  {
   get
    {
     return Liste[sira];
    }
   }
   public bool MoveNext()
   {
    if (sira < Liste.Length - 1)
    {
     sira++;
     return true;
    }
    else
    {
     return false;
    }
   }
   public IEnumerator GetEnumerator()
   {
    return Liste.GetEnumerator();
   }
}
```

Bu şartlarda bile foreach döngüsü çalışır. Çünkü foreach döngüsünün kullanacağı metotlar aynen duruyorlar. Çünkü foreach'in istediği metotlardır. Yoksa class'ın IEnumerable ve IEnumerator arayüzlerini uygulayıp uygulamadıkları kontrol edilmiyor.

ICollection Arayüzü

Yukarıdaki sayfalarda hazırladığımız sınıfa sadece IEnumerator ve IEnumerable arayüzünü uygulamıştık. Çünkü amacımız, sınıfta yer vereceğimiz dahili koleksiyonun elemanları üzerinde foreach döngüsü yardımıyla ileriye doğru hareket etmekti. Başka bir deyişle sınıfımıza foreach döngüsü desteği vermekle yetinmiştik.

Sınıftaki dahili koleksiyona eleman eklerken, silerken veya değiştirirken kullanabileceğimiz metodumuz yoktu. Bu metotları hazırlayıp kendimiz isim verseydik kodumuz standart dışı olurdu. Bu nedenle hazırladığınız sınıfa ek beceriler kazandırmayı düşündüğünüzde amacınıza hizmet edecek .NET Framework ile gelen bir interface veya abstract bir class'ın olup olmadığını araştırmak yerinde bir davranış olacaktır.

Kendi koleksiyon sınıflarını hazırlamak isteyenlerin mutlaka ilgilenmeleri gereken arayüzlerden birisi ICollection'dır. Yukarıdaki sayfalarda hakkında bilgi verilen **ArrayList**, **HashTable**, **SortedList** gibi koleksiyon sınıfları ICollection arayüzünü uyguladıklarına göre bu arayüz önemlidir. **ICollection** arayüzünün orijinal yapısı aşağıda verildi.

```
public interface ICollection : IEnumerable
{
 void CopyTo(Array array, int index);
 int Count { get; }
 bool IsSynchronized { get; }
 object SyncRoot { get; }
}
```

IEnumerable arayüzünü uygulayan bu arayüz **CopyTo()** adında bir metoda, **Count**, **IsSynchronized** ve **SyncRoot** adında 3 özelliğe sahiptir. Daha önceki konulardan bildiğiniz gibi CopyTo() metodu koleksiyonların içeriği dizi değişkene kopyalanırken kullanılmaktadır.

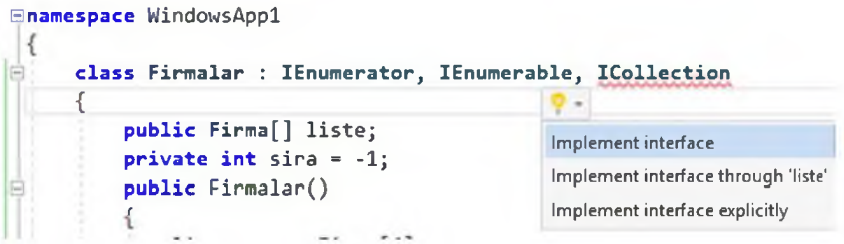
Count özelliğinde ise koleksiyonun eleman sayısı tutulmaktadır. Bu bölümün ilk sayfasında verilen temsili resimde görebileceğiniz gibi ICollection arayüzünü .NET Framework ile gelen bütün kcleksiyon sınıfları uygulamaktadır. Koleksiyonun senkronize edilip edilmediğini bool tipindeki ve yalnızca get bloğuna sahip **IsSynchronized** özelliğine bakarak öğrenebilirsiniz.

Yukarıda hazırladığımız koleksiyonun bir sonraki elemanın üzerine gitmek için **MoveNext**() metodunu kullandık ve üzerinde bulunulan elemanın içeriğini elde etmek için **Current** özelliğinden yararlandık. MoveNext() metodu ve **Current** özelliği **IEnumerator** arayüzü kaynaklıdır.

Yukarıda "Firmalar" adını verip sınıfta dahili koleksiyona sahip olmasını sağladığımız sınıfa şimdi ICollection arayüzünü de uygulayacağız. Bu sayede sınıfımızdaki dahili koleksiyonun eleman sayısı öğrenebileceğiz ve istersek elemanları bir dizi değişkene kopyalayabileceğiz. Bu amaçla "Firmalar" sınıfımızla ilgili ilk satırı aşağıdaki gibi düzenledik.

```
class Firmalar : IEnumerator, IEnumerable, ICollection
{
```

Burada yapıldığı gibi sınıfın deklarasyon satırına ICollection arayüzünün adı yazılırsa, henüz bu arayüzün gerekleri yerine getirilmediği için Visual Studio hata rapor eder. Arayüzün gereklerini yerine getirirken Visual Studio'da yardım alabileceğinizi biliyorsunuz. Henüz gerekleri yerine getirilmediği için altı kırmızı çizgi ile çizilen arayüzün üzerine fare işareti götürülüp aşağıda verilen menünün açılmasını sağlayıp **implement interface** komutunu verirseniz Visual Studio, ilgili arayüzdeki metot ve özelliklerin içi boş hallerini hazırlar.



Bu komutu verdiğinizde Visual Studio akıllı davranıp olmayan metot ve özellikler için hazırlık yapıyor. Yani daha önce ilgili interface'in gereklerini yerine getirmek üzere hazırladığınız metotlara hiç dokunmuyor. Visual Studio'nun ICollection arayüzü için sınıfımızda yaptığı hazırlıkları aşağıda görebilirsiniz.

```
public int Count => throw new NotImplementedException();
public object SyncRoot => throw new NotImplementedException();
public bool IsSynchronized => throw new NotImplementedException();
public void CopyTo(Array array, int index)
{
 throw new NotImplementedException();
}
```

Visual Studio tarafından hazırlanan bu satırlara dikkat edilirse özelliklerde lambda "=>" operatörüne yer verildiğini görürsünüz. Bu hali bu sınıfın CopyTo() metodu veya Count özelliği kullanılmak istenirse hata meydana gelir ve ortama **NotImplementedException** tipinde bir nesne fırlatılır. ICollection arayüzünden gelen Count özelliğinden ve CopyTo() metodun yararlanmakla yetinmek istediğimiz için Visual Studio tarafından hazırlanıp sınıfımıza dahil edilen satırları aşağıdaki gibi düzenledik.

```
public int Count => Liste.Length;
public object SyncRoot => throw new NotImplementedException();
public bool IsSynchronized => throw new NotImplementedException();
public void CopyTo(Array array, int index)
{
 for (int i = 0; i < Count; i++)
  {
   array.SetValue(liste[i].firma_adi, i);
  }
}
```

Visual Studio tarafından hazırlanan satırlarda Count özelliğini ayarlarken lambda operatörünü aynen koruduk ve sınıfın dahili koleksiyonun eleman sayısını aktarmakla yetindik. CopyTo() metodunda ise for döngüsü hazırlayıp dahili koleksiyonun bütün elemanlarını metodun parametresi olan Array tipindeki nesneye ekledik. Bunu yaparken Array sınıfının **SetValue**() metodunu kullandık.

Bu hazırlıktan sonra dahili koleksiyona sahip sınıfımızdaki (Firmalar) **Count** özelliğinden ve CopyTo() metodundan yararlanabilirsiniz. Bu özellik ve metottan nasıl yararlanıldığına örnek olması için üzerinde çalıştığımız Windows Forms uygulamasının formuna bir düğme ve ListBox yerleştirip aşağıda verilen kodu yazdık.

```
private void button2_Click(object sender, EventArgs e)
{
 Firmalar f = new Firmalar();
 object[] dizi = new object[f.Count];
 f.CopyTo(dizi, 0);
 for (int i = 0; i < dizi.Length; i++)
 {
  listBox2.Items.Add(dizi[i]);
 }
}
```

Burada önce "Firmalar" tipinde bir nesne hazırladık. Yukarıdaki sayfalarda bu sınıfı hazırlarken dahili koleksiyona eleman eklemeyi yapıcı metotta yaptığımız için burada "f" adını verdiğimiz nesnenin dahili koleksiyon elemanlara sahip olacaktır. Devamında, sınıfımızın dahili koleksiyonuyla aynı eleman sayısına sahip bir dizi değişken tanımladık. Çünkü yukarıda hazırladığımız **CopyTo**() metodu parametre olarak object tipinde bir dizi değişkeni istemektedir. Gerisinde sizin için yeni bir şey yoktur.

CollectionBase Sınıfı

Arayüzleri anlatırken öncelliği bu bölümün ilk sayfasında verilen temsili resimde üstte yer alan arayüzlere verdik. Şimdi sırada **CollectionBase** sınıfı var. Bu sınıfın yapısının bir kısmı aşağıda verildi. Bu yapıda özellikle IList tipindeki **List** ve ArrayList tipindeki **InnerList** özelliklerini dikkatinize sunmak istiyoruz.

```
public abstract class CollectionBase: List,
              ICollection, IEnumerable
{
 prutected CollectionBase();
 protected CollectionBase(int capacity);
 [ComVisible(false)]
 public int Capacity { get; set; }
 public int Count { get; }
 protected ArrayList InnerList { get; }
 protected IList List { get; }
 public void Clear();
 public IEnumerator GetEnumerator();
}
```

CollectionBase sınıfı **IList**, **ICollection** ve **IEnumerable** arayüzlerini uygulamaktadır. Bu sınıfa, özellikle kendi koleksiyon sınıflarınızı hazırlamak istediğinizde gerek duyarsınız.

Hazırlamak istediğiniz koleksiyon sınıfınızı **CollectionBase** sınıfının mirasçısı yaparsanız yazmanız gereken kod kısalır.

CollectionBase sınıfı abstract bir sınıf olduğu için örneği alınamaz. Bu sınıfın örneği alınabilseydi hemen bu sınıftan yola çıkıp nesne hazırlar sonra da kullanırdık. Buna imkan olmadığı için bu sınıfı hazırlayacağımız sınıfın mirasçısı olmasını sağlayacağız. Bu amaçla aşağıdaki gibi "Araba" adında bir sınıf hazırladık. Bu sınıfın gerek duyduğumuz sayıda örneğini alıp kendi hazırlayacağımız koleksiyonda saklayacağız.

```
public class Araba
{
 private string marka_model;
 private int motor_hacmi;
 public string ozellik_marka
 {
  get { return marka_model;}
  set { marka_model = value;}
 }
 public int ozellik_motor_hacmi
 {
  get { return motor_hacmi; }
  set { motor_hacmi = value; }
 }
 public Araba(string m_model, int m_hacmi)
 {
  ozellik_marka = m_model;
  ozellik_motor_hacmi = m_hacmi;
 }
}
```

Kod uzamasın diye bu sınıfı kısıtlı tuttuk. Bu sınıf 2 alana ve özelliğe sahiptir. Ayrıca bu sınıfın yapıcı metodu 2 parametreye sahiptir. Şimdi sırada **CollectionBase** sınıfının özellik ve metotlarını miras alacak koleksiyon sınıfımızı hazırlamak var. Yukarıda belirtildiği gibi CollectionBase sınıfı **IList**, **IEnumerable** ve **ICollection** arayüzlerini uyguladığı için işimiz nispeten kolaydır. Bu amaçla hazırladığımız sınıfı aşağıda görebilirsiniz.

```
public class Arabalar : CollectionBase
{
  public int Add(Araba araba)
  {
   return (this.List.Add(araba));
  }
  public int IndexOf(Araba araba)
  {
    return (List.IndexOf(araba));
  }
  public void Remove(Araba araba)
  {
    List.Remove(araba);
  }
  public bool Contains(Araba araba)
  {
   return (List.Contains(araba));
  }
```

```
    public Araba this[int index]
    {
     get
      {
       return ((Araba)List[index]);
      }
     set
      {
       List[index] = value;
      }
    }
}
```

Bu kodu incelerseniz **List** adında bir özelliğin veya koleksiyonun kullanıldığını fark edersiniz. Bu koddaki metotlar **CollectionBase** sınıfından miras alınan metotlardır. Koleksiyon sınıfınızdaki metotlara Türkçe adlar verebilirsiniz. CollectionBase sınıfında **IList** arayüzü kaynaklı birkaç metot daha olmasına rağmen kod uzamasın diye bu metotlardan yararlanmadık.Bu şekilde koleksiyon sınıfımızı hazırladıktan sonra sırada bu sınıfı kullanmak var. Bu amaçla Windows Form uygulamasının ilk formunu aşağıdaki gibi düzenledik.



Formun Load olayını temsil eden metodu aşağıdaki gibi düzenleyerek hazırladığımız sınıf tipinde bir nesnesin hazırlanmasını sağladık. Hazırladığımız nesneyi "Form1" sınıfının her yerinde kullanabilmek için değişken tanımlamayı metodun dışında yaptık.

```
Arabalar araba_liste;
private void Form1_Load(object sender, EventArgs e)
{
 araba_liste = new Arabalar();
}
```

Burada yapılan ArrayList nesnesi hazırlamaktan pek farklı değildir. Kullanıcı koleksiyona aktarmak istediği arabanın marka modelini ve motor hacmini TextBox'lara yazıp "Ekle" düğmesini tıklayacak. "Ekle" düğmesi için hazırladığımız kod aşağıdadır.

```
private void Ekle_Click(object sender, EventArgs e)
{
 Araba araba = new Araba(textBox1.Text,
               Convert.ToInt32(textBox2.Text));
 araba_liste.Add(araba);
 textBox1.Clear();
 textBox2.Clear();
}
```

Bu kodda TextBox'ların içeriklerine göre "Arabalar" sınıfının örneği hazırlanıp **Add**() metodu ile kendi koleksiyonumuza ekledik. Hemen hatırlatmak gerekirse buradaki Add()

metodu kendi hazırladığımız sınıftaki Add() metodudur. Diğer yandan bu Add() metodu içinde **IList** koleksiyonundan gelen Add() metodu kullanılmaktadır. Bu bölümün ilk sayfalarına bakılırsa IList arayüzünün **Add**, **Insert**, **Remove** gibi metotlara sahip olduğu görülür.

"Ekle" düğmesi için hazırlanan bu kod istenilen sayıda işletilip koleksiyona istenilen sayıda araba bilgisi eleman olarak eklendikten sonra "Listele" düğmesi tıklanıp koleksiyonun mevcut içeriği ListBox'larda listelenebilir. "Listele" düğmesi için yazılan kod aşağıda verildi.

```
private void Listele_Click(object sender, EventArgs e)
{
 listBox1.Items.Clear();
 listBox2.Items.Clear();
 foreach (Araba a in araba_liste)
 {
  listBox1.Items.Add(a.ozellik_marka);
  listBox2.Items.Add(a.ozellik_motor_hacmi.ToString());
 }
}
```

Koleksiyonun mevcut elemanlardan birisi silinmek istendiğinde söz konusu elemanın yani arabanın marka modeli ilk TextBox'a yazılıp "Sil" düğmesi tıklanacak. Bu düğme tıklandığı zaman kendi hazırladığımız sınıfının Remove() metodunu aşağıdaki gibi kullandık.

```
private void Sil_Click(object sender, EventArgs e)
{
 foreach (Araba a in araba_liste)
  if (a.ozellik_marka == textBox1.Text)
  {
   araba_liste.Remove(a);
   break;
  }
}
```

IDictionary Arayüzü

Yukarıdaki sayfalarda anahtar-değer çifti şeklinde düzenlenmiş bilgiler organize edilirken kullanılan **Hashtable** ve **SortedList** sınıfları hakkında yazılanları hatırlarsanız bu 2 sınıf **Keys** ve **Values** özelliklerine; **Add**, **Clear**, **Remove**, **Contains** ve **IDictionaryEnumerator** metotlarına sahipti. Bu özellik ve metotların kaynağı **IDictionary** arayüzüdür. Bu arayüzün yapısını aşağıda görebilirsiniz.

```
public interface IDictionary : ICollection, IEnumerable
{
 bool IsFixedSize { get; }
 bool IsReadOnly { get; }
 ICollection Keys { get; }
 ICollection Values { get; }
 object this[object key] { get; set; }
 void Add(object key, object value);
 void Clear();
```

```
 bool Contains(object key);
 IDictionaryEnumerator GetEnumerator();
 void Remove(object key);
}
```

Bu arayüzdeki metot ve özelliklerin nasıl kullanıldığı konusunda bilgi Hashtable ve SortedList sınıflarının anlatıldığı sayfalarda verildiği için tekrara düşmemek için bu arayüzün özellik ve metotları tek tek anlatılmadı. Bu yapı incelenirse **IDictionaryEnumerator** numaralayıcısının GetEnumerator() metodu kaynaklı olduğu fark edilir.

Hazırlamak istediğiniz sınıfa IDictionary arayüzünü uygulamak istiyorsanız temel sınıf olarak **DictionaryBase**'i almanız size zaman kazandıracaktır. Çünkü bu sınıf IDictionary, IEnumeable ve ICollection arayüzlerini uyguladığı için bu arayüzlerdeki metotları ve özellikleri kendi sınıfınızda hazırlamak zorunda kalmıyorsunuz. **DictionaryBase** sınıfı abstract bir sınıf olduğu için örneği alınamaz.

```
public abstract class DictionaryBase : IDictionary, ICollection,
IEnumerable
{
 protected DictionaryBase();
 public int Count { get; }
 protected Hashtable InnerHashtable { get; }
 protected IDictionary Dictionary { get; }
 public void Clear();
 public void CopyTo(Array array, int index);
 public IDictionaryEnumerator GetEnumerator();
 protected virtual void OnClear();
 protected virtual void OnClearComplete();
 protected virtual object OnGet(object key, object currentValue);
 protected virtual void OnInsert(object key, object value);
 protected virtual void OnInsertComplete(object key, object value);
 protected virtual void OnRemove(object key, object value);
 protected virtual void OnRemoveComplete(object key, object value);
 protected virtual void OnSet(object key, object oldValue,
       object newValue);
 protected virtual void OnSetComplete(object key, object
       oldValue, object newValue);
 protected virtual void OnValidate(object key, object value);
}
```

Şimdi yukarıda CollectionBase sınıfından yola çıkıp kendi koleksiyon sınıfımızı hazırladığımız gibi DictionaryBase sınıfından yola çıkıp anahtar-değer çiftini saklayabilen bir koleksiyon sınıfı hazırlayacağız.

Yukarıdaki sayfalarda CollectionBase sınıfını anlatmak için hazırladığımız örnekte **IList** arayüzünden miras olarak gelen **List** koleksiyonunda birden fazla bilgiyi saklayabilmek için önce 2 özelliğe sahip bir sınıf hazırlamış ve bu sınıfın örneklerini kendi hazırladığımız sınıftaki List koleksiyonunda saklamıştık. Ancak **DictionaryBase** taban sınıf olarak alınıp yeni bir sınıf hazırlandığında IDictionary kaynaklı **Dictionary** adlı koleksiyonda 2 bilgiyi yani anahtar-değer çiftini saklamak mümkün olduğu için ayrıca bir class hazırlamaya gerek duymayıp direk kendi koleksiyon sınıfımızı hazırladık.

```
public class Iller : DictionaryBase
{
  public void Ekle(string plaka, string sehir)
  {
    Dictionary.Add(plaka, sehir);
  }
  public void Uzaklastir(string str)
  {
    Dictionary.Remove(str);
  }
  public Iller this[int index]
  {
   get
    {
      return ((Iller)Dictionary[index]);
    }
    set
    {
      Dictionary[index] = value;
     }
    }
   public IEnumerator Enum_plakalar()
   {
     return Dictionary.Keys.GetEnumerator();
   }
   public IEnumerator Enum_iller()
   {
     return Dictionary.Values.GetEnumerator();
   }
 }
```

Kod uzamasın diye bu sınıfı kısıtlı tuttuk. Kendi sınıfımızdaki "Ekle" metoduyla IDictionary arayüzü kaynaklı koleksiyona yine bu arayüzden miras alınan Add() metodu ile eleman eklerken Uzaklastir() adını verdiğimiz metotta ise **Remove**() metodunu kullanıp Dictionary koleksiyonundan işaret edilen elemanı sildik.

Bu koleksiyonun mevcut elemanlarının anahtar ve değer bilgilerini enum olarak elde edebilmek için **GetEnumerator**() metodunun kullanıldığı Enum_plakalar() ve Enum_iller() adında 2 metot hazırladık. DictionaryBase sınıfı mirasçısı sınıfımızı hazırladıktan sonra üzerinde çalıştığımız projenin formunu aşağıdaki gibi şekilde düzenledik. Kullanıcı koleksiyona bir eleman olarak eklemek istediği anahtar(plaka) ve değer bilgilerini TextBox'lara yazıp "Ekle" düğmesini tıklayacaktır.



Yukarıda işaret edildiği gibi IDictionary arayüzünden miras alınan **Dictionary** koleksiyonu anahtar-değer çiftini saklamak üzere düzenlendiği için Add() metodu 2 parametreye sahiptir. Formun Load olayını temsil eden metotta kendi hazırladığımız sınıfın örneğini aldıktan sonra "Ekle" düşmesi için aşağıda verilen kodu yazdık.

```
Iller il_liste;
private void Form1_Load(object sender, EventArgs e)
{
 il_liste = new Iller();
}
private void Ekle_Click(object sender, EventArgs e)
{
 il_liste.Ekle(textBox1.Text, textBox2.Text);
 textBox1.Clear();
 textBox2.Clear();
}
```

Bu şekilde hazırladığımız sınıfın örneğini alıp istenen sayıda ilin plaka ve adını koleksiyona eleman olarak ekledikten sonra "Listele" düğmesi için aşağıda verilen kodu yazdık.

```
private void Listele_Click(object sender, EventArgs e)
{
 listBox1.Items.Clear();
 listBox2.Items.Clear();
 IEnumerator liste_plaka = il_liste.Enum_plakalar();
 IEnumerator liste_il = il_liste.Enum_iller();
 int sayi = il_liste.Count;
 for (int i=0; i<sayi; i++)
  {
   liste_plaka.MoveNext();
   listBox1.Items.Add(liste_plaka.Current.ToString());
   liste_il.MoveNext();
   listBox2.Items.Add(liste_il.Current.ToString());
  }
}
```

Bu kodda kendi hazırladığımız sınıftaki IEnumerator tipinde numaralayıcı geri gönderen 2 metottan yararlanıp koleksiyona eklediğimiz illerin plaka ve adlarını bu enum tiplerine aldıktan sonra ListBox'lara aktardık. Geriye bir tek "Sil" düğmesi için yazmamız gereken kod kaldı. Kısıtlı özelliklere sahip bu örneği hazırlamadaki amaç IDictionary arayüzü ve DictionaryBase sınıfı hakkında kısaca bilgi vermektir.

```
private void Sil_Click(object sender, EventArgs e)
{
 IEnumerator liste_plaka = il_liste.Enum_plakalar();
 int sayi = il_liste.Count;
 for (int i = 0; i < sayi; i++)
  {
   liste_plaka.MoveNext();
   if (liste_plaka.Current.ToString() == textBox1.Text)
    {
     il_liste.Uzaklastir(textBox1.Text);
     break;
    }
  }
}
```

22. BÖLÜM

GENERIC SINIFLAR, METOTLAR, INTERFACE'ler

C# 2.0 ile C# diline getirilen en önemli yenilik generic yapıların desteklenmesiydi. Generic yapılar (class'lar, interface'ler, metotlar, delegate'ler) C# programcılığını köklü bir değişikliğe uğratmıştı. Tabi 2.0 sürümünden bu yana geçen 12 yılda 5 yeni C# sürümü release edildi. Dolayısıyla eski programcılar bile C# 2.0 öncesini unuttular.

Şimdilerde kimse C# 1.0 ile C#'a merhaba demediğine göre sanki generic desteği C#'a yeni gelmiş gibi yaklaşım göstermemek gerekir diyoruz. Buna rağmen C# kaynaklarında genel alışkanlıkla henüz yolun başında olan programcılara konular kolay anlaşılsın diye başlangıçta generic yapılardan söz edilmez. Sanki C#'ın generic desteği yokmuş gibi davranılır. Bu kitapta da aynı yaklaşımı benimsedik. Ancak bilmelisiniz ki generic yapılar her yerde karşınıza çıkacak, generic yapıları iyi kavramadığınız sürece zorluklar hep sizi bekleyecektir.

Bu bölümde amacımız, C# dilinde hem generic olan ne varsa anlatmak hem de sizlere generic yapıları kullanma alışkanlığı kazandırmaktır. Bu alışkanlığı kazandırmanın yolunun generic yapıların hepsi hakkında yeteri ayrıntıda bilgiye sahip olmaktan geçtiğini tahmin ederseniz. Örneğin generic sınıf ve metotların nasıl hazırlandığını bilmek yetmez. Ayrıca generic delegate ve generic interface hakkında bilgi sahibi olmak gerekir.

Koleksiyonların anlatıldığı bölümde .NET Framework ile gelen koleksiyon sınıfları ayrıntılı bir şekilde incelendi. C# 2.0'dan itibaren daha önce sözü edilen koleksiyon sınıflarının **Generic** versiyonları kullanıma sunuldu. Koleksiyonlar adlı bölümde sözü edilen sınıflar **System.Collections**'da bulunurken, generic desteği olan koleksiyon sınıfları ve bazı arayüzler **System.Collections.Generic**'te yer almaktadır. NET Framework ile gelen bazı generic koleksiyon sınıfları hakkında bilgi bir sonraki bölümde verilecektir.

Konuları bu şekilde sıralarken şöyle düşündük: Henüz generic sınıflar hakkında bilgi edinmeyen birisine generic koleksiyonlar anlatıldığı zaman anlamada zorluk yaşanması ihtimal dahilindedir. Diğer yandan generic koleksiyonlar hakkında bilgi edinmeye başlamadan önce generic olmayan koleksiyonlar hakkında bir şeyler okumak, generic olmayan koleksiyonları kullanmak sizi generic koleksiyonlara hazırlayacaktır.

Bir sonraki bölümde generic koleksiyonları, hem generic sınıfları hem de koleksiyonları bilenlere anlatıyor olacağız. Başka bir deyişle, bu kitabın bir önceki bölümünü dikkatlice okuduysanız koleksiyon sınıfları hakkında yeterince kendinizi donattınız demektir. Bu bölümde ise generic sınıflar hakkında genel bilgi edineceksiniz. Bu sayede bir sonraki bölümde generic koleksiyonlar hakkında yazılanları ilk okumada anlayabileceksiniz.



Bu bölümü hazırlarken, bu kitabın "Koleksiyon Sınıfları" adlı bir önceki bölümünü okuduğunuzu varsaydık. Bazı arkadaşlar ArrayList sınıfını uygulamalarımda kullanmayacağım, neden ArrayList'in anlatıldığı sayfaları okuyup zaman kaybedeyim ki diye sorabilirler. Bize sorarsanız C# programlama dilini öğrenme sürecinde böyle ayrımlar yapmamak gerekir.

Tek cümle ile tanımlamak gerekirse, tipi sonradan (nesne hazırlanırken) belirlenen alanlar içeren sınıflara **generic sınıf** adı verilmektedir. Generic olmayan sınıfların alanlarında

saklanacak veya üzerinde işlem yapılacak bilgilerin tipi kodlama sırasında belirtilir. Generic sınıflarda ise kullanılacak bilgilerin tipi, söz konusu sınıftan yola çıkıp nesne hazırlanırken belirtilir. Tabi, ihtiyaç olması halinde aldıkları parametrelerin ve geriye gönderdikleri bilginin tipinin nesne hazırlanırken belirlendiği metotlar hazırlamak mümkündür.

Generic sınıflar üzerinde konuşmaya başlamak üzere önce aşağıdaki gibi **int** tipinde bir alana sahip bir sınıf hazırladık. Devamında bu sınıfın örneğini hazırlayıp dizi değişken gibi işlev gören alanın (dahili koleksiyon) elemanlarına bilgi aktardık.

```
public class Liste
{
 public int[] dizi = new int[3];
}
public partial class Form1 : Form
{
 private void Form1_Click(object sender, EventArgs e)
  {
    Liste nesne = new Liste();
    nesne.dizi[0] = 100;
    nesne.dizi[1] = 200;
    nesne.dizi[2] = 300;
  }
}
```

Bu kod dikkatlice incelenirse 2 sınıfın olduğu ve "Liste" adı verilen ilk sınıfta **int** tipinde bir dizi değişkenin tanımlandığı; yani bu sınıfın dahili koleksiyon olarak kullanılabilinecek bir alana sahip olduğu görülebilir. "Form1" adlı sınıfta ise "Liste" adlı ilk sınıfın örneği alınıp "dizi" adı verilen alanın elemanlarına bilgi aktarılmaktadır. Bunu, yukarıda verilen tanımın ışığında yorumlarsak şunu diyebiliriz: Burada hazırlanan "Liste" adlı sınıfın "dizi" alanı generic değildir. Çünkü bu alanda tutulacak bilginin tipi kodlama sırasında belirtiliyor.

"dizi" adını verdiğimiz koleksiyonun elemanlarına int tipinden farklı bilgiler aktarılma gereği duyuluyorsa ya "Liste" adlı sınıfta 2. veya 3. bir alan hazırlanmalı ya da bu alanın tipinin **object** olması sağlanmalıdır. Değişik sakıncalardan dolayı object değişkenlerin kullanılması önerilmiyor. Zaten generic yapılar, şimdi burada saymadığımız sakıncaları gidermek için C#'a dahil edilmiştir. Konunun kolay izlenmesini sağlamak için yukarıda verilen 2 sınıfı aşağıdaki gibi düzenledik. Bu durumda sınıfımız dahili koleksiyon olarak kullanılabilinecek 2 alana sahip oldu.

```
public class Liste
{
 public int[] dizi_int = new int[3];
 public string[] dizi_str = new string[3];
}
public partial class Form1 : Form
{
 private void Form1_Click(object sender, EventArgs e)
 {
   Liste nesne = new Liste();
   nesne.dizi_int[0] = 100;
   nesne.dizi_int[1] = 200;
   nesne.dizi_int[2] = 300;
   nesne.dizi_str[0] = "Beşiktaş";
```

```
        nesne.dizi_str[1] = "Galatasaray";
        nesne.dizi_str[2] = "Fenerbahçe";
      }
    }
```

Bu değişiklikten sonra "Liste" sınıfının örneklerinde "dizi_str" adını verdiğimiz 2. alanından yararlanıp string bilgileri saklamak mümkün olur. Bu şekilde her tip için ayrı hazırlık yapmak programcı için zaman kaybıdır. Bu hali ile "Liste" sınıfı yine generic bir sınıf değildir, Çünkü sınıftaki 2 alana aktarılacak bilgiler kodlama sırasında belirleniyor.

Generic tiplerin nasıl hazırlanıp kullanıldığı üzerinde konuşmaya başlamadan önce .NET Framework ile gelen ArrayList sınıfı üzerinden birkaç cümle yazacağız. Koleksiyonlar adlı bölümde uzun uzun anlatılanlardan hatırlayacağınız gibi ArrayList nesnelerinin dahili koleksiyonu **object** tipindedir. Dolayısıyla ArrayList nesnelerine her türden bilgileri aktarmak mümkündür. Örneğin aşağıdaki gibi ArrayList nesnesi hazırlayıp değişik tipte bilgileri elemanlara aktardığınızda herhangi bir sorun çıkmaz.

```
public static void Main()
{
 ArrayList liste = new ArrayList();
 liste.Add(1903);
 liste.Add("Beşiktaş");
}
```

Bu kodda sizin için yeni olan herhangi bir şey olmasa bile üzerinde biraz konuşmak gerekiyor. ArrayList sınıfının **Add**() metodu object tipinde bir parametreye sahiptir. Burada hazırladığımız kodda Add() metodunun ilk kullanımında int tipinde bilgi parametre olarak verildiği için derleyici devreye girip burada kapalı bir dönüştürme (boxing işlemi) yapıp int bilgiyi object tipine dönüştürüp öyle ArrayList nesnesinin dahili koleksiyona ekler. Benzer durumla Add() metodunun burada 2. kez kullanıldığı satırda da karşılaşılır. Add() metoduna string bilgi parametre olarak verildiği için geri planda bu bilgi object tipe dönüştürülür.

Generic olmayan ArrayList sınıfının şimdilik karşımıza çıkan 2 dezavantajı var:

- Add() metodunun her kullanılmasında boxing işlemi yapıldığı için performans kaybı yaşanır. Tabi bu performans kaybının kullanıcı tarafından hissedilmesi için binlerce kaydın bir seferde ArrayList nesnesine aktarılması gerekir.
- Yukarıda ArrayList nesnesini string bilgileri liste halinde bellekte tutmak üzere hazırlamışken programcı dalgınlıkla int bilgiyi Add() metoduna parametre olarak vermesine rağmen buna ne Visual Studio'nun ne de derleyicinin bir itirazı olur. Yani ArrayList gibi object tipinde bilgileri kabul eden sınıflar tip güvenli değildir. Programcının gerekli kontrolleri yapması, tedbir alması gerekiyor.

Yukarıda hazırladığımız ArrayList nesnesi üzerinde konuşmaya devam etmek üzere verdiğimiz koda birkaç satır ekleyip mevcut elemanları indeksleyici yardımıyla okuduk.

```
public static void Main()
{
 ArrayList liste = new ArrayList();
 liste.Add(1903);
 liste.Add("Beşiktaş");
```

```
 int eleman1 = Convert.ToInt32(liste[0]);
 string eleman2 = liste[1].ToString();
}
```

ArrayList nesnesinde tutulan bilgiler object tipinde olduğu için okuduğumuz her bilgiyi ilgili tipe dönüştürdük. Elbette burada yapılan dönüştürme işlemleri programcılar için zor işlemler olmasa bile hataya açık işlemlerdir.

Bu durumda Microsoft firması .NET Framework için sınıf kitaplığını hazırlarken sadece int ve string bilgilere yönelik ayrı ArrayList sınıfları hazırlayabilirdi. Bu da firma için çok zor bir işlem olmazdı. Ancak o zaman benzer işlevlere sahip çok sayıda sınıf söz konusu olurdu ki bu istenmez. Çünkü benzer amaçlı bir sınıfta sonradan akla gelen bir değişikliği çok sayıda sınıfta gerçekleştirmek gerekir ki bu kodun bakım maliyetini arttırır.

Buna rağmen zamanında C# programlama dilinin ilk sürümü kullanılırken çok sayıda programcı tiplendirilmiş koleksiyon adı verilen "tiplere özel" kendi koleksiyon sınıflarını geliştirdiler. Aradan çok zaman geçmeden C#'a generic tip desteği gelince tiplere özel koleksiyon sınıfları hazırlama ihtiyacı ortadan kalktı.

Generic Class Hazırlamak

Generic sınıflardan yararlanılmadığı zaman yaşanacak sorunlardan biraz söz ettikten sonra tekrar bir tanımlama yapacağız: "NET Framework 2.0 ile birlikte desteklenmeye başlanan **Generic** sınıflar sayesinde sınıflardaki alanların, özelliklerin ve metotların kullanacakları tipleri sınıfın örneği hazırlarken belirlemek mümkün olmaktadır."

Kendi generic sınıflarınızı nasıl hazırlayabileceğinizden söz etmeden önce konunun kolay anlaşılmasına katkı olsun diye bir sonraki bölümde ayrıntılı bir şekilde incelenecek "List" sınıfından kısaca söz edip List nesnesi hazırlayıp bilgi aktaracağız. Aşağıda verilen kodda List tipinde 2 nesne hazırladık. "List<int>" gibi yazılan yani diamond operatöre sahip tiplere **constructed type** denilmektedir.

```
public static void Main()
{
 List<string> liste1 = new List<string>();
 List<int> liste2 = new List<int>();
}
```

Bu 2 nesnenin ilkinde string tipte elemanlara yer verilebilinirken ikincisinde int tipinde bilgiler tutulabilir. Yani List sınıfının **Add**() metodu aşağıdaki gibi kullanılırsa hata meydana gelir. Çünkü new anahtar kelimesi ile "liste1" adını verdiğimiz nesneyi hazırlarken koleksiyonda tutulacak elemanların string olacağını beyan etmişken burada int tipindeki bilgiler eleman olarak nesneye eklenmek istenmektedir.

```
public static void Main()
{
 List<string> liste1 = new List<string>();
 List<int> liste2 = new List<int>();
 liste1.Add(1907); // hata meydana gelir.
}
```

Burada örneğini aldığımız "List" sınıfı generic bir sınıf olduğu için nesne hazırlama işlemi sırasında programcıya tip seçme imkanı verilmektedir. Yani programcı" List" sınıfını istediği veri türleri için kullanabilmektedir. Ancak her generic sınıf List gibi değildir; seçilebilinecek veri türleri bakımından bazı kısıtlamalar söz konusu olabilmektedir. İleriki sayfalarda bu konudan söz edilecektir.

Bu şekilde generic sınıfları gözümüzün önünde biraz canlandırdıktan sonra şimdi yukarıda "Liste" adını verdiğimiz sınıfın generic olmasını sağlayacağız. Konuyu adım adım anlatmak için önce sınıfın yalnızca gövdesini verdik.

```
public class Liste<T>
 {
 }
```

Şimdiye kadar hazırladığımız sınıflardan farklı olarak sınıf adının sonuna <T> ekledik. Diamond adı verilen bu operatörün "< >" arasına yazılan "T" harfi "tip" anlamına gelmektedir. "T" yerine başka bir harf veya kelimeyi kullanabilirsiniz. Ancak "T" harfi C# programcıları tarafından yaygın olarak kullanılmaktadır.

Sınıf adından sonra küçüktür-büyüktür işaretlerinin arasına yazılan "T" harfi sınıf dahilinde kullanılacak tipin, sınıfın örneği hazırlanırken belirleneceğini ve bu tipin sınıf dahilinde "T" harfi ile temsil edileceğini işaret etmektedir. Bu bilgilerin ışığında "Liste" adını verdiğimiz bu sınıfın aşağıda verilen halini incelerseniz konu nispeten aydınlanmış olur.

```
public class Liste<T>
{
 public T[] dizi = new T[3];
}
```

"Liste" adını verdiğimiz sınıfın yeni halinde dizi değişken tanımladık ama dizi değişkenin tipini alıştığınız şekilde (int, string, byte, char vs) değilde "T" harfi ile işaret ettik. Buna göre bu sınıfın örneği hazırlanırken "T" yerine hangi tip kullanılırsa dizi değişken o tipte olacaktır. Bu dizi değişkende "int "bilgileri saklamak üzere bu sınıfın örneğini aşağıdaki gibi hazırladık.

```
public partial class Form1 : Form
{
 private void Form1_Click(object sender, EventArgs e)
  {
    Liste<int> nesne1 = new Liste<int>();
    nesne1.dizi[0] = 100;
    nesne1.dizi[1] = 200;
    nesne1.dizi[2] = 300;
  }
}
```

Yukarıda sınıf adının yanına "<T>" yazmıştık. Başka bir sınıfın içinde bu sınıfın örneğini hazırlarken sınıf adının yanına T'nin yerini almasını istediğimiz tipi yazdık. Sınıfın örneğini hazırlarken sınıf adından sonra "<int>" yazdığımız için nerede "T" kullanıldıysa söz konusu tipin "int" olduğu varsayılır.

Şimdi bu sınıfın ikinci bir örneğini hazırlayacağız ve bu kez sınıftaki "dizi" adını verdiğimiz dahili koleksiyonunda string bilgileri saklayacağız. Başka bir deyişle, sınıfı kodlarken nerede "T" harfine yer verdiysek nesne hazırlarken oraya **string** yazacağız.

```
private void Form1_Click(object sender, EventArgs e)
{
 Liste<int> nesne1 = new Liste<int>();
 nesne1.dizi[0] = 100;
 nesne1.dizi[1] = 200;
 nesne1.dizi[2] = 300;
 Liste<string> nesne2 = new Liste<string>();
 nesne2.dizi[0] = "Beşiktaş";
 nesne2.dizi[1] = "Galatasaray";
 nesne2.dizi[2] = "Fenerbahçe";
 textBox1.Text = nesne1.dizi[0].ToString();
 textBox2.Text = nesne2.dizi[0];
}
```

Yukarıda hazırladığımız sınıfta bir tek generic tip vardı. Gerek duyulması halinde Generic sınıfın adından sonra birden fazla generic tipe yer verilebilir. Bunu örneklemek için yukarıda verilen generic sınıfı aşağıdaki gibi değiştirdik.

```
public class Liste<T1, T2>
{
 public T1[] dizi_1 = new T1[3];
 public T2[] dizi_2 = new T2[3];
}
```

Bu sınıfta "T1" ve "T2" adı verilen tipler sınıftan nesne hazırlanırken yapıcı metoda ait parantezlerden önce "<>" arasına yazılmalıdır. Bu sınıfın örneğini nasıl alıp kullandığımızı aşağıda görebilirsiniz. T1'in yerine int, T2'nin yerine ise string kullandık.

```
Liste<int, string> nesne = new Liste<int, string>();
nesne.dizi_1[0] = 100;
nesne.dizi_1[1] = 200;
nesne.dizi_1[2] = 300;
nesne.dizi_2[0] = "Fenerbahçe";
nesne.dizi_2[1] = "Galatasaray";
nesne.dizi_2[2] = "Beşiktaş";
```

Dolayısıyla burada "dizi_1" adını verdiğimiz dizi değişkenin tipi int olacaktır. "Liste" tipindeki başka bir nesnede bu dizi değişkenin tipi farklı olabilir. Programcı tip sınırlaması yapmadıysa, nesne hazırlanırken T1 ve T2'nin yerine istenen tipin adı yazılabilir.

Aynı Ada, Farklı Sayıda Generic Tipe Sahip Sınıflar

Nasıl ki sınıflarda aynı isme sahip birden fazla metot hazırlayabiliyorsak, generic tip sayıları farklı olmak şartıyla aynı ada sahip birden fazla generic sınıf hazırlanabilir. Bu amaçla aynı namespace içinde aşağıdaki gibi aynı ada sahip 2 sınıf hazırladık.

```
public class Array1<T>
{
 public T alan1;
}
```

```
public class Array1<T1, T2>
{
 public T1 alan1;
 public T2 alan2;
}
```

Bu 2 sınıf aynı ada sahip olsalar bile farklı sayıda generic tipe sahip oldukları için derleyici overload mantığıyla Bu şekilde iki sınıf hazırlanmasına itiraz etmez. Programcı burada "Array1" adını verdiğimiz sınıf tipinde nesne hazırlarken bir generic tip kullandığında ilk sınıf tipinde, 2 generic tip kullanılmak istendiğinde ikinci sınıf tipinde nesne hazırlanır. Bu sınıflar birbirinden farklı işlemler yapıyor olabilir. Bu sınıflardan yola çıkarak nasıl nesne hazırladığımızı aşağıda görebilirsiniz.

```
public static void Main()
{
 Array1<int> nesne1 = new Array1<int>();
 Array1<int, int> nesne2 = new Array1<int, int>();
}
```

Derleyici aynı ada sahip birden fazla sınıfın hazırlanmasına izin verse bile özel bir neden yoksa bunu önermiyoruz. Çünkü sınıfın adı ile işlevi arasında bir yakınlığın olması gerek. Tabi, programcı tek başına uygulama geliştiriyorsa ve muhtemel isimlendirme sorunlarını önemsemiyorsa birden fazla sınıfa aynı adı verebilir.

Örnek Generic Koleksiyon Sınıfı

Generic sınıflarla ilgili bazı ayrıntılardan henüz söz etmemiş olsak bile konunun özü sanırız anlaşıldı. Generic sınıfları kullanma alışkanlığı kazanmanıza katkı olması için şimdi bir **Collection** sınıfı hazırlayacağız. .NET Framework ile birlikte hemen her işlem için kullanılabilinecek koleksiyon sınıfı geldiği için programcıların kendi koleksiyon sınıflarını hazırlama ihtiyacı azdır.

Hazırlayacağımız bu sınıfın birisi generic, diğeri "int" tipinde 2 alanı olacak. Bu sınıfın yapıcı metodu ile dahili koleksiyon olarak işlev görecek nesne hazırlanmakta ve bu nesnenin eleman sayısını tutacak alan sıfırlanmaktadır.

Aşağıda verilen sınıfta generic tip desteği hariç sizin için yeni bir şey yoktur. Bu sınıfı kendi bilgisayarınızda kodlayıp denemenizi öneriyoruz. Hatta bu sınıfa sonradan yeni beceriler kazandırmayı düşünebilirsiniz. Örneğin sınıf dahilinde hazırlanan generic koleksiyonun kapasitesinin yapıcı metot aracılığıyla seçilmesini sağlayabilirsiniz.

```
public class MyStack <T>
{
 T[] dizi; // generic alan
 int index;
 public MyStack()
 {
  dizi = new T[4];
  index = 0;
 }
}
```

Bu sınıfın Push() ve Pop() adında 2 metodu olacak. Push() metodu koleksiyona yeni eleman ekleyip sayısını bir artırırken, Pop() metodu koleksiyonun en son eklenen elemanı koleksiyondan okuyup geriye gönderdikten sonra silecektir.

Başlangıçta bu sınıftaki dahili koleksiyonun kapasitesinin 4 olduğu varsayılacaktır. Koleksiyona 5. eleman eklenmek istendiğinde ise kapasite ikiye katlanıp 8'e çıkarılacaktır. "MyStack" adını verdiğimiz sınıfın dahili koleksiyonuna eleman eklerken kullanacağımız Push() metodu hazırlanmış hali aşağıda verildi.

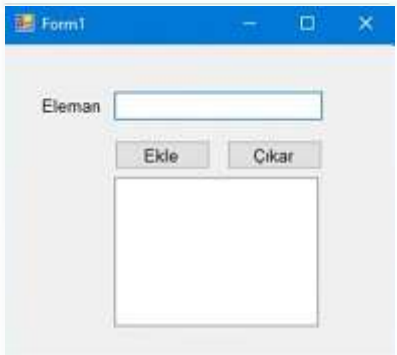
```
public class MyStack<T>
{
 T[] dizi;
 int index;
 public MyStack()
 {
  dizi = new T[4];
  index = 0;
 }
 public void Push(T eleman)
 {
  dizi[index] = eleman;
  index++;
  if (index >= dizi.Length)
  {
   T[] temp_dizi = new T[dizi.Length * 2];
   dizi.CopyTo(temp_dizi, 0);
   dizi = temp_dizi;
  }
 }
}
```

İlk olarak bu sınıfın **Push**() metodu üzerinde biraz duracağız. Daha önce ArrayList sınıfı hakkında bilgi verilen bölümde belirtildiği gibi ArrayList sınıfı ile hazırlanan koleksiyonların başlangıç kapasiteleri 0 olarak seçiliyordu. Kullanıcı koleksiyona ilk elemanı eklediğinde kapasite 4 yapılıyordu. Sonra yeni elemanlar eklenip kapasite eleman sayısını karşılamadığında koleksiyonun mevcut kapasitesi 2 kat arttırılıyordu.

Yukarıda hazırladığımız koleksiyon sınıfında başlangıç kapasitesini 4 olarak ayarladık. İsteseydik ArrayList sınıfında olduğu gibi başlangıç kapasitesini 0 olarak ayarlayıp ilk eleman eklendiğinde 4'e çıkarabilirdik. Bu hali ile koleksiyona ilk 4 eleman eklenirken geri planda şunlar gerçekleşir:

- Yeni eleman "dizi" adını verdiğimiz dahili koleksiyonda "index" değişkenin işaret ettiği konuma yerleştirilir ve "index" değişkenin içeriği 1 arttırılır. "index" değişkenin başlangıç içeriğini sınıfın yapıcı metodunda 0 olarak ayarladık.
- Bu arttırmadan sonra "index" değişkenin içeriği başlangıçta 4 olan kapasiteyi aştığında mevcut kapasitenin iki katı sayıda elemana sahip yeni bir dizi değişken tanımlanır.
- Ardından mevcut elemanları içeren ve burada "dizi" adını verdiğimiz dizinin elemanlarını Array sınıfının **CopyTo**() metodu ile geçici bir dizi değişkene aktardık. Devamında geçici dizi değişkeni asıl diziye aktardık.

Henüz sadece eleman eklerken kullanabileceğimiz **Push**() metodunu hazırladığımız bu sınıfı denemek için projenin formunu aşağıdaki gibi düzenledik.



Proje çalıştırıldığında yukarıda hazırlayıp "MyStack" adını verdiğimiz generic sınıf tipinde bir nesne hazırlanacak. Bu sınıftan yola çıkarak hazırladığımız nesnede string bilgileri saklamak istediğimiz için sınıfın örneğini hazırlarken generic tipin yerine "string" yazdık.

```
public MyStack<string> strStack;
private void Form1_Load(object sender, EventArgs e)
{
 strStack = new MyStack<string>();
}
```

Sınıfın örneğini başka metotlarda kullanmak istediğimiz için değişken tanımlama işlemini class bloğu içinde yani formun Load olayını temsil eden metodun dışında yaptık. Bu koleksiyona eleman eklemek üzere "Ekle" düğmesi için aşağıda verilen kodu yazdık.

```
private void Ekle_Click(object sender, EventArgs e)
{
 strStack.Push(textBox1.Text);
 listBox1.Items.Add(textBox1.Text);
 textBox1.Text = "";
}
```

Koleksiyona eklenmek istenen elemanın içeriği TextBox'a yazılıp 'Ekle' düğmesi tıklandıkça yeni eleman eklenmiş olur. Form'un Load olayıyla ilgili metotta "MyStack" adını verdiğimiz sınıf tipinde nesne hazırlarken sınıfın dahili koleksiyonda string bilgileri tutmak istediğimizi beyan ettik. Bu koleksiyonda isteseydik int veya başka tipte bilgileri saklayabilirdik.

Koleksiyona eklenen elemanları görmek için forma bir ListBox nesnesi yerleştircik. Yeni elemanı yani burada textBox1'e yazılan bilgiyi Push() metodu ile koleksiyona ekledikten sonra ayrıca ListBox'a aktardık. Bu sayede yazdığımız koleksiyon sınıfını test etme imkanı bulmuş olacağız.

Bu şekilde koleksiyona eleman ekleme işlemini tamamladıktan sonra sıra koleksiyondan eleman çıkarmaya geldi. Sınıfımızda hazırlayacağımız Pop() metodu, sınıf dahilinde hazırlanan dahili koleksiyonun (dizi değişkenin) en son elemanı geriye gönderip eleman sayısını 1 azaltacaktır. "MyStack" sınıfımızın Pop() metodu hazırlanmış hali aşağıda verildi.

```
public class MyStack<T>
{
 T[] dizi; // generic alan
 int index;
 public MyStack()
 {
  dizi = new T[4];
  index = 0;
 }
 public void Push(T eleman)
 {
  dizi[index] = eleman;
  index++;
  if (index >= dizi.Length)
  {
   T[] temp_dizi = new T[dizi.Length * 2];
   dizi.CopyTo(temp_dizi, 0);
   dizi = temp_dizi;
  }
 }
 public T Pop()
 {
  --index;
  T eleman = dizi[index];
  return eleman;
 }
}
```

Pop() metodunda ilk olarak "index" değişkenin içeriği bir azaltılarak en son eklenen elemanın ulaşılmaz olması sağlanmaktır. Devamında en son eleman dahili koleksiyondan yani dizi değişkenden okunup geriye gönderilmektedir.

"MyStack" adını verdiğimiz sınıfın örneğine eklenen en son elemanı çıkarıp elde etmek üzere Windows Forms uygulamamızda "Çıkar" adını verdiğimiz düğmenin "Click" olayını temsil eden metotta **Pop()** metodunu kullandık. Pop() metodunun her işletilmesi sonucu MyStack tipindeki nesnemizin dahili koleksiyondan eleman silindiğini anında görmek için bu elemanı ayrıca formdaki ListBox nesnesinden sildik.

```
private void Cikar_Click(object sender, EventArgs e)
{
 listBox1.Items.Remove(strStack.Pop());
}
```

Koleksiyonumuz, dolayısıyla formdaki ListBox nesnesi herhangi bir elemana sahip değilken "Çıkar" düğmesi tıklanıp bu metot işletilirse uygulama hata üretir. Bunun önüne geçmek için Pop() metodunda değişiklik yapmak gerekir.

```
public T Pop()
{
 if (index > 0)
  {
   --index ;
```

```
        T eleman = dizi[index];
        return eleman;
       }
      else
       {
        T eleman = dizi[index];
        return eleman;
       }
    }
```

Bu sınıf üzerinde biraz çalışıp daha işlevsel yapabilirsiniz. Diğer yandan .NET Framework ile birlikte her türlü işlem için koleksiyon sınıfları geldiği için böyle bir sınıf üzerinde çalışmanın işlevi, generic sınıfları tanımak ve kullanma alışkanlığı edinmek olur.

İndeksleyici Desteği Olan Generic Koleksiyon Sınıfı

Yukarıda hazırladığımız koleksiyon sınıfı "son giren ilk çıkar (Last In First Out-LIFO)" veya "İlk giren son çıkar (First-in-Last-out FILO)" ilkesine göre işlev görüyordu. Sınıfımızın bu ilkeye uygun olmasını sağlamak için elemanları hep en sona ekledik, eleman çıkarırken de hep en sonda duranı çıkardık. Şimdi hazırlayacağımız generic sınıf ise liste türü bir sınıf olacak.

Bu generic sınıfın yukarıda verilenden farklarından birisi 2 yapıcı metoda sahip olmasıdır. Diğer fark ise bu sınıfın indeksleyici desteğinin olmasıdır. İndeksleyici sayesinde dizi değişkenlerde olduğu gibi istenen eleman köşeli parantezlerden yararlanılarak elde edilir. Bu sınıfta yer verdiğimiz indeksleyici ile ilgili satırlara dikkatlice bakmanızı öneriyoruz. Çünkü bu sınıfta asıl amacımız daha önce hakkında bilgi verilen indeksleyicilerin generic tiplerle nasıl kullanıldığını göstermektir.

```
public class Liste<T>
{
 private T[] elemanlar;
 public int eleman_no;
 public Liste()
 {
  elemanlar = new T[16];
 }
 public Liste(int sayi)
 {
  elemanlar = new T[sayi];
 }
 public void Ekle(T eleman)
 {
  elemanlar[eleman_no] = eleman;
  eleman_no++;
  if (eleman_no >= elemanlar.Length)
  {
   T[] temp_dizi = new T[elemanlar.Length * 2];
   elemanlar.CopyTo(temp_dizi, 0);
   elemanlar = temp_dizi;
  }
 }
```

```
 public T this[int index]
 {
  get {return elemanlar[index];}
  set {elemanlar[index] = value;}
 }
}
```

Bu sınıfı sıfırdan kendimiz hazırladığımız için bize kolaylık sağlayacak .NET Framework ile gelen mevcut sınıf ve interface'lerden yararlanma yoluna gitmedik. Bir sonraki bölümde bu arayüz ve sınıflardan söz edilecektir. Çünkü bu bölümde asıl amacımız sizlere generic tipleri kullanma alışkanlığı kazandırmaktır.

Bu sınıfın ilk yapıcı metodu, nesne hazırlanırken kapasite seçilmediği zaman kullanılır ve kapasitenin 16 olduğu varsayılır. İkinci yapıcı metot ise sınıfın örneği alınırken kapasite belirtildiği zaman işletilir. Yine yukarıdaki sayfalarda yaptığımız gibi eleman ekleme işlemi sonucu mevcut kapasite yetmediği zaman mevcut kapasitenin 2 katı kadar elemana sahip olması sağlanırken yeni bir dahili koleksiyon hazırlanıyor ve Array sınıfının **CopyTo**() metodu ile eski elemanlar yeni diziye aktarılıyor. En son olarak yeni 2 katı kadar elemana sahip yeni dizi değişkenin referansı eski diziye aktarılıyor.

Hazırladığımız koleksiyon sınıfında sadece eleman ekleme ve listeleme işlemleri yapılabiliniyor. Array sınıfının metotları kullanılıp bu sınıfının dahili koleksiyonda yani dizi değişkende eleman arama, değiştirme ve silme işlemleri yapılabilir.

Bu sınıfa indeksleyici desteği aşağıda verilen satırlar ile sağlanmaktadır.İndeksleyiciler adlı bölümde hazırladıklarımızdan tek farkı geriye gönderilen bilgi tipinin generic olmasıdır.

```
public T this[int index]
{
 get
  {
   return elemanlar[index];
  }
 set
  {
   elemanlar[index] = value;
  }
}
```

Bu sınıfın örneğini alıp eleman eklemek için Visual Studio ile hazırladığımız projenin formuna bir TextBox yerleştirip aşağıda verilen kodu yazdık. Bu kod sayesinde proje çalıştırıldığında dahili koleksiyonun başlangıç kapasitesi 2 olan bir nesne hazırlanır.

```
public Liste<string> liste1;
private void Form1_Load(object sender, EventArgs e)
{
 liste1 = new Liste<string>(2);
}
private void ekle_Click(object sender, EventArgs e)
{
 liste1.Ekle(textBox1.Text);
 textBox1.Text = "";
}
```

Hazırladığımız bu sınıfın örneğini alırken yapıcı metoda 2'yi parametre olarak verdik. Dolayısıyla sınıf dahilindeki koleksiyona 3. eleman eklenmek istendiğinde kapasite ikiye katlanıp 4 yapılır.

Koleksiyona eklenen elemanlara indeksleyici sayesinde nasıl eriştiğimizi göstermek için forma bir ListBox yerleştirip aşağıda verilen kodu yazdık. Formdaki "Listele" adını verdiğimiz düğme tıklandığında işletilecek metoda dikkat ederseniz mevcut elemanların içeriği elde edilirken indeksleyiciden yararlanıldığını görürsümüz.

```
private void Listele_Click(object sender, EventArgs e)
{
 for (int i=0; i<liste1.eleman_no; i++)
   listBox1.Items.Add(liste1[i]);
}
```

Generic Sınıflarda Metotlar

Yukarıdaki sayfalarda hazırladığımız generic sınıflarda generic alan, generic özellik ve indeksleyicilere yer verdik. Bildiğiniz gibi class'larda ayrıca metotlar, arayüzler, delegate'ler olabilmektedir. Metotların parametrelerinde ve geri dönüş bilgilerinde generic tipler kullanılabilir.

Şimdi yukarıda verdiğimiz basit generic sınıfın generic parametreli bir metoda sahip olmasını istiyoruz. Bu örneğin pratik değerinin olmadığını belirtmek isteriz. Çünkü buradaki amacımız, sınıftaki metoda odaklanmanızı sağlamaktır.

```
public class Liste<T>
{
 public T[] dizi;
 public Liste(int sayi)
  {
   dizi = new T[sayi];
  }
 public void Ekle(int index, T eleman)
  {
   dizi[index] = eleman;
  }
}
```

Bu sınıfın "Ekle" adını verdiğimiz metodun 2. parametresi generic'tir. Başka bir deyişle 2. parametrenin tipi sınıftan yola çıkılarak nesne hazırlandığı zaman belirtilecektir. Sınıfın yapıcı metodunda ise "dizi" adını verdiğimiz dahili koleksiyon hazırlanmaktadır. Bu sınıftan yola çıkılıp nasıl nesne hazırlayıp kullandığımızı aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 Liste<string> nesne = new Liste<string>(10);
 nesne.Ekle(0, "Fenerbahçe");
 nesne.Ekle(1, "Beşiktaş");
 Liste<int> nesne1 = new Liste<int>(10);
 nesne1.Ekle(0, 1907);
 nesne1.Ekle(1, 1904);
}
```

Yukarıda verilen örnekte "Ekle()" metodunun 2. parametresinin sınıfın generic tipi ile aynı olmasını sağladık. Aslında burada generic sınıflar bağlamında yeni bir şey yoktur. Çünkü yukarıdaki sayfalarda hazırladığımız generic sınıflarda generic tipleri metotlarda zaten kullandık. Konu üzerinde düşünmenizi sağlamak amacıyla aynı konuyu değişik şekillerde önümüze getiriyoruz. Aşağıda generic sınıftaki metotlarla ilgili 2. bir örnek hazırladık.

```
public class Islemler<T>
{
 public int SonucInt;
 public string SonucStr;
 public void Toplama(T Par1, T Par2)
 {
  Type tip = typeof(T);
  if (tip.Name=="Int32")
    SonucInt = Convert.ToInt16(Par1) + Convert.ToInt16(Par2);
  if (tip.Name=="String")
    SonucStr = Convert.ToString(Par1) + Convert.ToString(Par2);
 }
}
```

"Islemler" adını verdiğimiz bu sınıf 2 alana ve Toplama() adında bir metoda sahiptir. Bu metoda parametre olarak verilen bilgiler **int** tipinde ise toplanmakta, string tipte ise birleştirilmektedir. Metodun işletilmesi sonucu elde edilen bilgiyi geriye göndermek yerine **public** değişkenlere aktardık.

Burada sizin için yeni olan generic sınıftan yola çıkıp nesne hazırlanırken kullanılan tipi öğrenip ona göre işlem yapılmasıdır. Nesne hazırlanırken seçilen tipi **typeof** operatörü ile öğrendik. "Islemler" adını verdiğimiz bu sınıfı aşağıdaki gibi kullandık.

```
private void Form1_Click(object sender, EventArgs e)
{
 Islemler<int> nesne1 = new islemler<int>();
 nesne1.Toplama(11, 22);
 MessageBox.Show(nesne1.SonucInt.ToString());
 Islemler<string> nesne2 = new islemler<string>();
 nesne2.Toplama("Fener", "bahçe");
 MessageBox.Show(nesne2.SonucStr);
}
```

Yukarıda "Islemler" sınıfında Toplama() adı verilen metot her ne kadar generic parametreye sahip olsa bile generic bir metot değildir. Çünkü bir metodun içinde, sınıf tanımlama satırında sınıf adından sonra yazılan generic tiplerin kullanılıyor olması o metodun generic olduğunu göstermez. Başka bir deyişle şimdiye kadar generic tipleri sınıf seviyesinde tanımlayıp kullandık. Generic tipler metot seviyesinde de tanımlanabilir. Yani geliştirdiğiniz sınıfın değil, içerisindeki metotların generic olmasını istiyorsanız bu mümkündür.

Şimdi yukarıda verdiğimiz "Islemler" adlı sınıftaki metodun generic olmasını sağlayacağız. Bu amaçla class adından sonraki T'yi silip metot adından sonra kullandık. Sınıf adından sonraki <> operatörlerini silmeyebilirdik. Silme nedenimiz sınıfımızın kısıtlı olmasıdır. Yoksa hem sınıfımız hem de metodumuz generic olabilirdi.

```
public class Islemler
{
 public int SonucInt;
 public string SonucStr;
 public void Toplama<T>(T Par1, T Par2)
  {
    Type tip = typeof(T);
    if (tip.Name =="Int32")
      SonucInt = Convert.ToInt16(Par1) + Convert.ToInt16(Par2);
    if (tip.Name == "String")
      SonucStr = Convert.ToString(Par1) + Convert.ToString(Par2);
  }
}
```

Burada "Islemler" adını verdiğimiz sınıfımız generic olmadığından dolayı sınıftan yola çıkarken nesne hazırlanırken diamond (<>) operatöre yer verilmez. Bu şekilde hazırlanan metot aşağıdaki gibi kullanılabilir. Aşağıda verilen kodda "Islemler" adını verdiğimiz sınıfın 2 kez örneğini alıp **Toplama**() adlı metodu kullandık.

```
private void Form1_Click(object sender, EventArgs e)
{
 Islemler nesne1 = new islemler();
 nesne1.Toplama<int>(11, 22);
 MessageBox.Show(nesne1.Sonuc_int.ToString());

 Islemler nesne2 = new islemler();
 nesne2.Toplama<string>("Fener", "bahçe");
 MessageBox.Show(nesne2.Sonuc_str);
}
```

Generic metotlara örnek arıyorsanız .NET Framework ile gelen **Enum** sınıfının **TryParse**() metoduna bakabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Burada generic tıp olarak kullandığımız "iller" kendi hazırladığımız bir enum tipidir.

```
static void Main(string[] args)
{
 Console.Write("Aranacak ilin adı veya plakası :" );
 string il = Console.ReadLine();
 iller ilPlaka;
 bool sonuc = Enum.TryParse<iller>(il, out ilPlaka);
 Console.ReadLine();
}
```

Akla şu soru gelmiş olabilir: Yukarıda "Islemler" adını verdiğimiz sınıftaki generic metot **static** olsaydı nasıl kullanılırdı? Başka bir deyişle generic metodun yer aldığı class'ın örneği alınmadan metot nasıl kullanılırdı? Bu soruya cevap olması için yukarıda verilen sınıfı aşağıdaki gibi düzenledik.

```
public class Islemler
{
 public static int Sonuc_int;
 public static string Sonuc_str;
 public static void Toplama<T>(T Par1, T Par2)
 {
```

```
    Type tip = typeof(T);
    if (tip.Name == "Int32")
      SonucInt = Convert.ToInt16(Par1)+ Convert.ToInt16(Par2);
    if (tip.Name == "String")
      SonucStr=Convert.ToString(Par1)+ Convert.ToString(Par2);
    }
}
```

Bu şekilde düzenlenen sınıfın örneği alınmadan generic static metotların nasıl işletildiğini aşağıda görebilirsiniz. "Toplama" adını verdiğimiz static metot işletilerek elde edilen bilgilerin aktarıldığı değişkenleri de static yaptık.

```
private void Form1_Click(object sender, EventArgs e)
{
 Ilemler.Toplama<int>(11, 22);
 MessageBox.Show(islemler.Sonuc_int.ToString());
 Ilemler.Toplama<string>("Fener", "bahçe");
 MessageBox.Show(islemler.Sonuc_str);
}
```

Toplama() metodumuz generic bir tipe sahip olduğu için metot adında sonra, parantezlerden önce <> operatörüne yer verip generic tipin yerine istediğimiz tipi yazdık. Yukarıda generic metotlara örnek olsun diye kullandığımız Enum sınıfının TryParse() metodunun static bir metot olduğu dikkatinizden kaçmamıştır.

Windows Forms Application Üzerinde Generic Metotlar

Generic metotlar konusu biraz karışık olduğu için konuyu uzatmak pahasına başka bir örnek hazırladık. Bu amaçla üzerinde çalıştığımız Windows Forms uygulamasının formuna 2 ListBox yerleştirip string ve int tipinde 2 dizi değişken tanımladık.

```
string[] diziStr = new string[3];
int[] diziInt = new int[3];
private void Form1_Load(object sender, EventArgs e)
{
}
```

Ardından uygulama çalıştırıldığı zaman otomatik olarak işletilen Form1_Load() metodunda bu dizi değişkenlere aşağıda yapıldığı gibi bilgiler aktardık.

```
string[] diziStr = new string[3];
int[] diziInt = new int[3];
private void Form1_Load(object sender, EventArgs e)
{
 diziStr[0] = "Fenerbahçe";
 diziStr[1] = "Beşiktaş";
 diziStr[2] = "Galatasaray";
 diziInt[0] = 2011;
 diziInt[1] = 2012;
 diziInt[2] = 2013;
}
```

Burada generic metotlarla ilgili henüz herhangi bir şey yoktur. Şimdi öyle bir generic metot hazırlayacağız ki, bu metot kendisine ister int, ister string ister başka bir tipte dizi değişken

parametre olarak verilsin, bu dizi değişkenin elemanlarını ListBox'a aktaracak. Başka bir deyişle bu metodun generic olması gerekecek. Metodu, üzerinde çalıştığımız Windows Forms uygulamasının "Form1" adlı sınıfı dahilinde hazırladık.

```
string[] diziStr = new string[3];
int[] diziInt = new int[3];
private void Form1_Load(object sender, EventArgs e)
{
 diziStr[0] = "Fenerbahçe";
 diziStr[1] = "Beşiktaş";
 diziStr[2] = "Galatasaray";
 diziInt[0] = 2011;
 diziInt[1] = 2012;
 diziInt[2] = 2013;
}
static void Listele<T>(T[] dizi, ListBox liste_kutu)
{
 foreach (T eleman in dizi)
   liste_kutu.Items.Add(eleman);
}
```

Dizi değişkenin içeriğinin aktarılacağı ListBox'ı metot çağrılırken belirtmek için generic metodun 2 parametreye sahip olmasını sağladık. Metodun ilk parametresi generic iken ikincisi **ListBox** tipindedir. Bu metodu aşağıdaki gibi işletip istediğiniz tipteki dizi değişkenin içeriğini ListBox'lara aktarabilirsiniz.

```
private void Aktar_Click(object sender, EventArgs e)
{
 Listele(diziStr, listBox1);
 Listele(diziInt, listBox2);
}
```

Listele() adını verdiğimiz generic metot **void** olduğu için hazırlanması ve kullanılması kolaydır. Void olmayan generic metotlar hakkında bilgi vermek amacıyla aşağıdaki gibi bir generic metot hazırladık. Bu metot void olmayıp generic tipe uygun olarak geriye bilgi göndermektedir. Yani bu metodun geriye göndereceği bilginin tipi de generic'tir.

```
class Islemler
{
 public T Getir<T>()
 {
  if (typeof(T) == typeof(int))
    return (T)(object)2017;
  if (typeof(T) == typeof(double))
     return (T)(object)9.81;
  if (typeof(T)== typeof(string))
     return (T)(object) "Fenerbahçe";
   return default(T);
  }
}
```

Bu sınıfın örneği alınıp generic metot işletilirken **typeof** operatörü sayesinde seçilen tipin int, double veya string olup olmadığı araştırılmaktadır. Bu metot aşağıdaki gibi işletilirse geriye "Fenerbahçe" bilgisini gönderir.

```
private void button1_Click(object sender, EventArgs e)
{
 Islemler nesne = new Islemler();
 string takim = nesne.Getir<string>();
}
```

Bu 2 satırın yorumu şudur: Burada Getir() adını verdiğimiz metotta generic tip olarak string kullanıldığında geriye string bilgi gönderilir. Başka bir deyişle bu metot aşağıdaki gibi işletilmek istenirse Visual Studio anında hatalı bir durum olduğunu işaret eder.

```
Islemler nesne = new Islemler();
double sayi = nesne.Getir<string>();
```

Getir() metodunu dikkatlice incelerseniz generic tip olarak string, double ve int için hazırlık yapıldığını görürsünüz. Bu 3 tipten farklı bir tip kullanıldığında hata meydana gelmemesi için **default** anahtar kelimesinden yararlandık. Bu anahtar kelime sayesinde geriye o tipin varsayılan değeri gönderilir. Örneğin byte tipinin varsayılan değeri 0 olduğu için aşağıda verilen kod işletildiğinde TextBox'a 0 yazılır.

```
private void button1_Click_1(object sender, EventArgs e)
{
 Islemler nesne = new Islemler();
 byte sayi = nesne.Getir<byte>();
 textBox1.Text = sayi.ToString();
}
```

Konu üzerinde düşünmeniz için Getir() adını verdiğimiz generic metodu bir de aşağıdaki gibi işletmeyi denedik. List sınıfının varsayılan değeri null olduğu için Getir() metodu şekilde işletildiğinde geriye null gönderilir.

```
private void button1_Click_1(object sender, EventArgs e)
{
 Islemler nesne = new Islemler();
 List<int> liste = nesne.Getir<List<int>>();
}
```

Şimdi Getir() adını verdiğimiz metotta değişiklik yapıp ayrıca bir parametreye sahip olmasını sağlayacağız. Başka bir deyişle bu metot kendisine parametre olarak verilen bilgiyi geriye gönderecektir. Metodun yeni haline bakarsanız parametre olarak verilen bilgiyi aynen geriye gönderdiğini tespit edebilirsiniz.

```
public T Getir<T>(T degisken)
{
 if (typeof(T) == typeof(int))
   return (T)degisken;
 if (typeof(T) == typeof(double))
   return (T)degisken;
 if (typeof(T)== typeof(string))
    return (T)degisken;
 return default(T);
}
```

Metot bu haliyle çağrıldığında kendisine parametre olarak verilen bilgi veya değişkenin içeriği T'ye dönüştürülüp öyle geriye gönderilmektedir. Aslında burada dönüştürme yapılmasa bile sorun çıkmaz. Bu nedenle aşağıda verilen metot hataya neden olmaz.

```
public T Getir<T>(T degisken)
{
 return degisken;
}
```

Bu metotta **typeof** operatörü ile parametrenin tipi araştırılmadığı için "return default(T);" satırına yer vermedik. void olmayan generic metotların özelliklerini ortaya çıkarmak için yukarıda verilen "Getir" metodunu aşağıdaki gibi düzenledik. Bu metot, parametre olarak verilen bilgi string ise bilginin büyük harfe çevrilmiş halini geriye göndermektedir.

```
public T Getir<T>(T degisken)
{
 if (typeof(T) == typeof(string))
  {
   string str = (string)(object)degisken;
   return (T)(object) str.ToUpper();
  }
 return default(T);
}
```

Bu kodda generic parametreyi önce **object** tipe sonra string tipe dönüştürüp string değişkene aktardık. Devamında bu string değişkenin içeriğine **ToUpper**() metodunu uygulayıp büyük harfe çevrilmiş halini önce object'e sonra generic tipe dönüştürüp geriye gönderdik.

Generic Metotlara Sınırlama Koymak

Generic tip içeren metotlarda kullanılacak tipler sonradan belli olduğu için tedbir alınmazsa sorunlar yaşanabilir. Yaşanması muhtemel sorunları daha kod yazılırken Visual Studio işaret eder. Örneğin int tipindeki 2 sayıdan hangisinin büyük olduğunu öğrenmek üzere ">" operatörü kullanılabilinirken birbirleriyle karşılaştırılan veriler string ise hata rapor edilir.

Bu ve benzeri noktaları ortaya çıkarmak veya generic metotlar hakkında bilgi vermeye devam etmek üzere aşağıda verilen sınıfı hazırladık. Bu sınıftaki "Kucuk" metodu parametre olarak aldığı 2 bilgiden küçük olanı geriye göndermektedir.

```
public class Islemler
{
 public T Kucuk<T>(T sayi1, T sayi2)
 {
  if (sayi1 < sayi2)
   {
    return sayi1;
   }
  return sayi2;
 }
}
```

Bu sınıftaki metodu incelediğinizde sizin için anlaşılmaz olmadığını hemen fark edersiniz. Ne var ki bu sınıfın örneğini alıp bu metodu işletmek istediğinizde hata mesajı alırsınız. Çünkü

derleyici "<" operatörü ile "T" adı verilen generic bilgileri birbirleri ile karşılaştıramayabilir. Daha doğrusu bu operatöre parametre olarak verilecek tiplerin karşılaştırılamaz olması ihtimal dahilindedir. Derleyici işi şansa bırakmayıp derlemeyi ret etmektedir.

Bu sorunun çözümü, bu metoda **IComparable** arayüzünü uygulamak ve metoda parametre olarak verilen 2 bilgiden küçük olanı araştırırken **CompareTo**() metodunu kullanmaktır. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Bu şekilde hazırlanan metot ile istenilen tipteki 2 bilgi bir biri ile karşılaştırılabilir.

```
public class Islemler
{
 public T Kucuk<T>(T sayi1, T sayi2) where T : IComparable
  {
   if (sayi1.CompareTo(sayi2) < 0)
    {
     return sayi1;
    }
  return sayi2;
 }
}
```

Burada **where** anahtar kelimesiyle gerçekte yapılan şudur: Derleyiciye diyoruz ki, bu sınıfın generic metodunda kullanacak tipin **IComparable** arayüzünü uygulamasını şart koşuyorum. O tip bu arayüzü uygulamadıysa bu metodun kullanılmasına izin verme.

Burada **where** deyimi ile generic tipe **IComparable** arayüzünü uygulama şartını getirdiğimiz için metodun istenen yerinde **CompareTo**() metodu ile karşılaştırma yapılabilir. Aynı sonucu aşağıda verilen kod ile almak mümkündür.

```
public class Islemler
{
 public T Kucuk<T>(T sayi1, T sayi2)
 {
  if (((IComparable)sayi1).CompareTo(sayi2) < 0)
   {
    return sayi1;
   }
  return sayi2;
 }
}
```

Bu şekilde düzenlenen sınıfın örneğini nasıl alıp Kucuk() adını verdiğimiz metodu nasıl kullandığımızı aşağıda görebilirsiniz.

```
private void button1_Click_1(object sender, EventArgs e)
{
 Islemler nesne = new Islemler();
 int sayi1 = 100;
 int sayi2 = 200;
 int sonuc = nesne.Kucuk<int>(sayi1, sayi2);
}
```

Generic Class'larda Miras

Generic olan veya olmayan her sınıf, generic bir sınıfın mirasçısı olabilir. Generic sınıflarda miras konusu üzerinde çalışmaya başlamak üzere aşağıdaki gibi kısıtlı özelliklere sahip bir sınıf hazırladık. Bu generic sınıfta, tipi sonra belirlenebilinen ve dahili koleksiyon olarak kullanılabilinecek 3 elemanlı bir alan bulunmaktadır.

```
public class Liste<T>
{
 public T[] dizi = new T[3];
}
```

Şimdi başka bir sınıfın bu sınıfın mirasçısı olmasını sağlayacağız. Generic sınıflarda miras söz konusu olduğunda generic sınıfın hangi tipteki versiyonuna gerek duyulduğu baştan belirtilmelidir.

```
public class Takimlar : Liste<int>
{
}
```

Burada "Takimlar" adını verdiğimiz sınıf, generic bir sınıf olan "Liste" sınıfındaki tipin "int" olarak seçilmiş halini miras almaktadır. Başka bir deyişle generic sınıfın mirasçısı "Takimlar" adını verdiğimiz sınıfın örneği alınırken "<>" arasında "T" harfinin yerini alacak herhangi bir tipi işaret etmek gerekmeyecektir.

```
private void button1_Click(object sender, EventArgs e)
{
 Takimlar nesne = new Takimlar();
 nesne.dizi[0] = 100;
 nesne.dizi[1] = 200;
 nesne.dizi[2] = 300;
}
```

"Takimlar" adını verdiğimiz sınıfı aşağıdaki gibi düzenleseydik "Liste" adlı generic sınıftaki alanda bu kez string bilgiler tutulabilinirdi.

```
public class Takimlar : Liste<string>
{
}
```

Şimdiye kadar generic sınıflarda mirasla ilgili pek fazla bir şey yapmamış olmamıza rağmen konunun karışma ihtimaline karşı birkaç cümle ekleyeceğiz:

- Elimizde generic bir tipte sahip bir sınıf vardı.
- Sonra bu generic sınıfımızı başka bir sınıfın mirasçısı yapmak istedik.
- Generic bir sınıf base class olarak kullanılmak istendiğinde generic tipin yerine istenen tipin baştan yazılması gerekiyor. Dolayısıyla generic sınıftaki generic tipin yerini normal tip alıyor.

Generic sınıflarda miras konusu bu kadarla sınırlı değildir. Bir diğer ayrıntıya değinmeden önce C# programcılarının "Liste<T>" şeklinde yazılan generic sınıfları "open-constructed

generic" ve Liste<int> şeklindeki sınıfları ise "closed-constructed generic" olarak adlandırdıklarını belirtmek gerekiyor. Bu tanımlamaya göre yukarıda normal bir sınıfı closed-constructed bir sınıfın mirasçısı yaptık.

Generic bir sınıf normal bir sınıfın base class'ı olabildiği gibi generic tip içeren sınıfın da base class'ı olabilir. Bu nedenle aşağıdaki gibi düzenlenen sınıflar hataya neden olmazlar.

```
public class Liste<T>
{
 public T[] dizi = new T[3];
}
public class Takimlar<T1> : Liste<string>
{
 public T1 alan1;
 public T1 alan2;
 public Takimlar(T1 a1, T1 a2)
 {
  alan1 = a1;
  alan2 = a2;
 }
}
```

Yukarıda derived class olarak kullandığımız "Takimlar" sınıfımız bu kez generic bir tipe ve bu generic tipten 2 alana sahiptir. Sınıfımızın ayrıca 2 parametreli bir yapıcı metodu bulunmaktadır. "Liste" adını verdiğimiz sınıfı hazırlarken generic tipe sahip olmasını sağlarken, base class olarak kullanırken generic tipin yerinde built-in tiplerden string'i seçtik.

Şimdi bu sınıf tipinde nesne hazırlayacağız. Bu generic tipe sahip olduğu için diamond operatörlerin arasına bir tip adı yazmak gerekecek. Ayrıca sınıfın yapıcı metodu, seçilen generic tiple aynı tipte 2 parametreye sahiptir.

```
static void Main(string[] args)
{
 Takimlar<int> nesne = new Takimlar<int>(100, 200);
 nesne.dizi[0] = "Fenerbahçe";
 nesne.dizi[1] = "Galatasaray";
 nesne.dizi[2] = "Beşiktaş";
 Console.WriteLine(nesne.alan1);
}
```

Şimdi generic sınıflarda miras konusundaki başka bir ayrıntıdan söz edebilmek için hem derived hem de base class'ınızın open-constructed generic olmasını sağlayacağız. Bu amaçla yukarıda verdiğimiz sınıfları aşağıdaki gibi düzenledik.

```
public class Liste<T>
{
 public T[] dizi = new T[3];
}
public class Takimlar<T1> : Liste<T2>
{
 public T1 alan1;
 public T1 alan2;
```

```
 public Takimlar(T1 a1, T1 a2)
 {
  alan1 = a1;
  alan2 = a2;
 }
}
```

Sizin için bir yenilik içermeyen bu 2 sınıfı yakında incelerseniz her şeyin normal olduğunu düşünebilirsiniz. Ancak C# derleyicisi burada derived class olarak kullandığımız "Takimlar" sınıfında hata rapor eder. Çünkü, derleyici hem base hem de derived class'ta kullanılan generic tipin aynı olmasını istemektedir. Bu nedenle derived class'ı aşağıdaki gibi düzenlemeliyiz.

```
public class Takimlar<T1> : Liste<T1>
{
 public T1 alan1;
 public T1 alan2;
 public Takimlar(T1 a1, T1 a2)
 {
  alan1 = a1;
  alan2 = a2;
 }
}
```

Çünkü derived class tipinde nesne hazırlanırken sadece bir generic tip kullanabiliriz. Bu durumda hem base hem de derived class'ın generic tiplerinin aynı olması şart olmaktadır. "Takımlar" sınıfımız tipinde nesne hazırlanırken yapıcı metot 2 parametreye sahip olsa bile tipleri aynıdır. Bu şartlarda "Takimlar" tipinde nesne hazırlamak istemiş olsaydık yine aşağıdaki gibi satır yazardık.

```
Takimlar<string> nesne1 = new Takimlar<string>("Adana", "Ankara");
```

Muhtemelen bu noktada aklınıza şöyle bir soru gelmiştir: Ya base class'ta 2 generic tip olsaydı. Bu soruya cevap aramak için yukarıda verdiğimiz ve base class olarak kullandığımız "Liste" adlı sınıfımızı aşağıdaki gibi düzenledik.

```
public class Liste<T1, T2>
{
 public T1[] dizi1 = new T1[3];
 public T2[] dizi2 = new T2[3];
}
```

Bu şartlarda bu sınıfı base class olarak kullanacak class'ta en azından 2 generic tip olmalıdır. Çünkü derived class'ın örneği alındığında en az 2 generic tipe yer verilebilmelidir ki base class'tan gelen özellik ve metotlar kullanılabilinsin. Bu nedenle derived class aşağıdaki gibi düzenlenirse derleyici hata rapor etmez.

```
public class Liste<T1, T2>
{
 public T1[] dizi1 = new T1[3];
 public T2[] dizi2 = new T2[3];
}
```

```
public class Takimlar<T, TT> : Liste<T, TT>
{
}
```

Derived class'ta generic tipler için kullanılan karakterlerin önemli olmadığını işaret etmek için farklı olmasını sağladık. Son bir ayrıntıdan söz etmek için derived class'ın base class'tan gelenden ayrı olarak kendisinin de bir generic tipe sahip olmasını sağlayacağız.

```
public class Liste<T1, T2>
{
 public T1[] dizi1 = new T1[3];
 public T2[] dizi2 = new T2[3];
}
public class Takimlar<T, TT, TTT> : Liste<TT, TTT>
{
 public T alan1;
 public Takimlar(T a)
 {
  alan1 = a;
 }
}
```

Konunun diğer ayrıntılarını ortaya çıkarmak için hazırladığımız bu 2 sınıfı yakında incelerseniz bura "T" adı verilen generic tipin derived class'a ait olduğu, TT ve TTT'nin ise base class'ın generic tip taleplerini karşılamak için kullanıldığını fark edersiniz. Derived class'ın örneğini nasıl aldığımızı aşağıda görebilirsiniz.

```
public static void Main()
{
 Takimlar<int,int,string> nesne =new Takimlar<int,int,string>(100);
 Console.WriteLine(nesne.alan1);
}
```

Generic Interface'ler

Generic interface'ler normal interface'lerden pek farklı değildir. Generic interface'lerin nasıl kullanıldığını anlatmak için aşağıdaki gibi basit bir interface hazırladık.

```
interface IArayuz<T>
{
 T Topla(T x, T y);
}
```

Bu interface, 2 parametreye sahip **Topla**() adında bir metoda sahiptir. Generic interface'lerin normal interface'lerden farkı, sınıfa uygulandıklarında tip veya tiplerin işaret edilmesi gerektiğidir. Bu interface'i kendi sınıfımıza nasıl uygulayıp gereğini nasıl yerine getirdiğimizi aşağıda görebilirsiniz.

```
class Matematik : IArayuz<int>
{
 public int Topla(int sayi1, int sayi2)
  {
   return sayi1 + sayi2;
  }
}
```

Bu interface'i sınıfa uygularken tip olarak int'i seçtiğimiz için interface'in gereklerini yerine getirmek için sınıfta hazırlamamız gereken metodun int tipinde 2 parametreye sahip olması ve geriye int bilgi göndermesi gerekir. Şimdi bu arayüzü başka bir sınıfa uygulayacağız.

```
class Karakterler : IArayuz<string>
{
 public string Topla(string str1, string str2)
  {
   return str1 + str2;
  }
}
```

Bu şekilde "IArayuz" adını verdiğimiz generic interface'in uygulandığı sınıfta hazırlanması gereken **Topla**() metodu ile string bilgiler birleştirilebilir. "Karakterler" adını verdiğimiz bu sınıftaki Topla() metodunu nasıl kullandığımızı aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 Karakterler islem = new Karakterler();
 this.Text = islem.Topla("Fener", "bahçe");
}
```

Generic interface'leri anlatmak üzere yukarıda hazırladığımız derived class'lar generic değildir. Generic interface uyguladığınız sınıf generic olabilir. Yani yukarıda generic interface uyguladığımız "Karakterler" adlı sınıf aşağıdaki gibi düzenlenmiş olabilir. Yukarıda Generic Sınıflarda Miras başlığı altında yazdıklarımız generic interface için de geçerli olduğu için mükerrer anlatım olmasın diye burada kesiyoruz.

```
class Karakterler<T> : IArayuz<string>
{
 public string Topla(string str1, string str2)
 {
  return str1 + str2;
 }
}
```

Generic Sınıflarda Kullanılacak Tipleri Sınırlamak

.NET Framework ile gelen her tip generic sınıfların tip parametresinin yerinde kullanılabilmektedir. Ancak bu bazen sorunlara neden olur. Örneğin hazırladığınız generic sınıf dahilinde sadece sayısal bilgiler üzerinde işlem yapılabiliniyor olabilir. Bu durumda yapılması gereken şudur: Generic sınıftan yola çıkıp nesne hazırlanırken <> işaretlerinin (diamond operatörü) arasına yazılabilinecek tipleri sınırlamaktır.

Bu işlemin nasıl yapıldığını anlatmak amacıyla önce generic bir sınıf hazırlayacağız. Bu sınıfta kodladığımız metot, yapıcı metodu aracılığıyla gönderilen dizi değişkenin içeriğini toplayıp geriye gönderecektir. Konuyu adım adım anlatmak için önce sınıfın bir kısmını verdik.

```
public class Islemler<T>
{
 private T[] sayilar;
```

```
 public Islemler(T[] dizi)
 {
  sayilar = dizi;
 }
}
```

Burada "T" adını verdiğimiz bir generic parametreye sahip bu sınıf yukarıda verilen generic sınıflardan farklı olarak yapıcı metoda sahiptir. Yapıcı metot aracılığıyla gönderilecek "dizi" nesnesi sınıfta tanımlanan generic diziye aktarılıyor. Bu anlattıklarımızı somutlamak amacıyla üzerinde çalıştığımız uygulamada Main() metoduna sahip sınıfa geçip bu generic sınıfı aşağıdaki gibi kullandık.

```
class Program
{
 public static void Main()
 {
  int[] intDizi = { 4, 7, 9, 20, 25 };
  Islemler<int> nesne = new Islemler<int>(intDizi);
 }
}
```

Hazırladığımız generic sınıftan yola çıkarak nesne hazırlamadan önce int tipinde bir dizi değişken tanımladık. Generic sınıf yardımıyla üzerinde işlem yapmak istediğimiz bilgiler int tipte olduğu için generic sınıftan yola çıkıp nesne hazırlarken diamond operatörün içine **int** yazdık. Şimdi sırada generic sınıfta "toplamBul()" adını verdiğimiz metodu hazırlamak var. Dizideki bütün elemanları toplamak istediğimiz için "for" döngüsü hazırladık.

```
public class Islemler<T>
{
 private T[] sayilar;
 public Islemler(T[] dizi)
 {
  sayilar = dizi;
 }
 public int ToplamBul()
 {
  int toplam = 0;
  for (int i = 0; i < sayilar.Length; i++)
   {
     toplam += Convert.ToInt32(sayilar[i]);
   }
   return toplam;
 }
}
```

Herhangi bir hatalı durumun olmadığı bu sınıf değişik tipler için kullanıldığında hata üretmesi ihtimal dahilindedir. Çünkü C# derleyicisi bu sınıfın yapıcı metoduna parametre olarak verilecek dizinin tipi konusunda herhangi bir bilgiye sahip değildir. Bu nedenle Convert sınıfının **ToInt32**() metodu her zaman çalışmayabilir. Çünkü burada ToplamBul() adını verdiğimiz metot int tipinde bilgi içeren dizi değişkenler üzerinde işlem yapacakmış gibi düzenlenmiştir. Örneğin üzerinde çalıştığımız uygulamanın Main() metodunu aşağıdaki gibi düzenlersek kod yazımı sırasında Visual Studio herhangi bir hata rapor etmezken uygulama çalıştırıldığında hata meydana gelir.

```
static void Main(string[] args)
{
 string[] strDizi = { "Adana", "Mersin", "Ankara", "İzmir" };
 Islemler<string> nesne = new Islemler<string>(strDizi);
 string sonuc = nesne.toplamBul().ToString();
 Console.WriteLine(sonuc);
 Console.ReadKey();
}
```

Yukarıda "Islemler" adını verdiğimiz generic sınıfı hazırlarken generic tipin struct kaynaklı olmasını istemiş olsaydık Visual Studio burada yaptığımız hatayı anında işaret ederdi. **where** anahtar kelimesi ile birlikte struct'ın nasıl kullanıldığını aşağıda görebilirsiniz.

```
public class Islemler<T> where T : struct
{
```

Generic metotların anlatıldığı sayfalardan hatırlayacağınız gibi generic tipler için kısıtlama koyma işlemi **where** anahtar kelimesi ile yapılmaktadır. Where anahtar kelimesinden sonra önce generic tipi temsil eden karakter(ler) sonra ilgili kısıtlama yazılmaktadır. Generic tipleri kısıtlarken sıklıkla karşılaşılan durumları burada sıralayacağız. Örneğin generic tip olarak yalnızca default yapıcı metoda sahip bir sınıfın kullanılabilinmesini istiyorsanız where anahtar kelimesi ile birlikte **new**() metodunu kullanabilirsiniz. Bu söylediğimizi örneklemek için aşağıda verilen sınıfı hazırladık.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
 public Tarih()
 {
  DateTime Bugun = DateTime.Today;
  Gun = Bugun.Day;
  Ay = Bugun.Month;
  Yil = Bugun.Year;
 }
}
```

3 public alana ve default yapıcı metoda sahip bu sınıfı daha önce yapıcı metotlar hakkında konuşurken vermiştik. Şimdi bu sınıfı bir generic sınıfta generic tip olarak kullanacağız. Amacımız generic sınıflarda tip kısıtlamalarından söz etmek olduğu için hazırladığımız sınıfın işleviyle ilgilenmedik.

```
public class Islemler<T> where T : new()
{
 public Islemler()
  {
  }
}
```

Bu sınıfta where anahtar kelimesi ile birlikte new() metoduna yer verdiğimiz için T yerine kullanılacak sınıfın yapıcı metodunun herhangi bir parametreye sahip olmaması veya yapıcı metodun hazırlanmamış olması gerekir. Bu nedenle bu sınıfın örneği aşağıdaki gibi alınırsa derleyici itiraz etmez.

```
class Program
{
 static void Main(string[] args)
 {
  Islemler<Tarih> islem = new Islemler<Tarih>();
  Console.ReadKey();
 }
}
```

Bu şartlarda gidip "Tarih" adını verdiğimiz sınıfta değişiklik yapıp yapıcı metodunun bir parametreye sahip olmasını sağlarsak derleyici hata rapor eder. Çünkü generic sınıfımızı hazırlarken where ile birlikte new() metoduna yer verdik.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
 public Tarih(int g)
 {
  DateTime Bugun = DateTime.Today;
  Gun = g;
  Ay = Bugun.Month;
  Yil = Bugun.Year;
 }
}
```

Bu durumda programcı ya generic sınıfında new() metodu ile ileri sürdüğü şartı değiştirecek veya "Tarih" sınıfının yapıcı metodunun parametresiz olmasını sağlayacak. Burada Tarih adını verdiğimiz sınıf yerine Tarih adında bir Struct hazırlamış olsaydık derleyici herhangi bir itirazda bulunmazdı. Çünkü struct'larda yapıcı metoda verildiğinde bütün alanların ilk değerlerin aktarılması şartı vardır.

Yukarıda new() metoduyla ilgili olarak söylenenlerde bakılırsa bu uygulamanın derleme hatasına neden olacağı düşünülebilir. Ancak derleyicinin hu koda bir itirazı olmaz. Çünkü struct'a ait yapıcı metodunun parametreli olması bir zorunluluktur. Generic tiplerle ilgili bir ayrıntıdan söz etmek üzere yukarıda Islemler adını verdiğimiz struct'ta tanımlama satırını aşağıdaki gibi düzenledik.

```
public class Islemler<T> where T : new(), struct
{
 public Islemler()
 {
 }
}
```

Generic sınıfımızda where anahtar kelimesinden sonra new() metoduna ve ardından struct'a yer vermekle 2 farklı hatanın meydana gelmesine neden olduk. Birinci hata şudur: new() metodu başka kısıtlama ile birlikte kullanılacaksa en sona yazılmalıdır. Bu nedenle bu generic sınıfla ilgili ilk hatayı gidermek üzere aşağıdaki gibi düzenledik.

```
public class Islemler<T> where T : struct, new()
{
 public Islemler()
  {
  }
}
```

Bu düzenlemeden sonra kodumuzdaki hataların sayısı bire indi. Aslında derleyicinin bu şartlarda verdiği hatayı yukarıda işaret etmiştik. Derleyici diyor ki kısıtlama olarak struct ve new'i birlikte kullanamazsınız. Konu üzerinde düşünmenizi sağlamak için Tarih adını verdiğimiz struct'ı tekrar class olarak düzenledik. Burada yaptığımız düzenlemeye derleyicinin bir itirazı olmaz.

```
public class Tarih
{
 public int Gun;
 public int Ay;
 public int Yil;
}
class Program
{
 static void Main(string[] args)
 {
  Islemler<Tarih> islem = new Islemler<Tarih>();
  Console.ReadKey();
 }
}
public class Islemler<T> where T : class, new()
{
 public Islemler()
 {
 }
}
```

Generic Delegate

Bu kitabın delegate'ler hakkında bilgi verilen bölümde örnek olması için hazırlayıp delegate'ler aracılığıyla işlettiğimiz metotlar generic değildi. Yani metodun parametrelerinin ve geri dönüş bilgisinin tipi kodlama sırasında belirtiliyordu. Dolayısıyla söz konusu metotları gerektiğinde temsil edip işletilmesinde aracılık edecek delagate'ler generic değildi.

Hem generic tipleri hem de delagete'leri bilen birisi için generic delegate hazırlayıp kullanmak zor değildir. Buna rağmen konuyu temelden alacağız. Bu amaçla ilgili bölümün başında hazırladığımız delegate tipini buraya aldık.

```
public delegate int TemsilciI1(int x, int y);
```

Bu delegate tipindeki nesnelerin int tipinde 2 parametreye sahip olup geriye int tipinde bilgi gönderen metotları temsil edebildiğini biliyorsunuz. Hazırlayacağımız generic delegate'in biraz işlevsel olmasını sağlamak için bu delegate tipinin bir de double veriler üzerinde işlem yapan halini hazırladık.

```
delegate double Temsilci2(double x, double y);
```

Parametre sayısı bakımından birbirine benzer olup farklı tipte veriler üzerinde işlem yapan birden fazla delegate hazırlamak istemiyorsanız generic delegate hazırlayabilirsiniz. Bu konuda bilgi vermek için aşağıdaki gibi bir generic delegate hazırladık.

```
public delegate T Temsilci<T>(T x, T y);
```

Bu delegate parametre yapısı bakımından yukarıda verilenlere benzemektedir. Başka bir deyişle, bu delegate 2 parametreye sahip olan ve geriye parametrelerle aynı tipte bilgi gönderen metotları temsil edebilir. Şimdi bu delegate'in örneğini alıp bir metodu temsil etmesini sağlayacağız.

```
Temsilci<int> intTemsilci = new Temsilci<int>(Topla);
```

Bu satır sayesinde hazırlayıp "intTemsilci" adını verdiğimiz delegate nesnesi int tipinde 2 parametreye sahip metotları temsil edebilir. Generic delegate'lerin nasıl kullanıldığını anlatmak için aşağıdaki gibi basit bir metot hazırladık.

```
public int Topla(int x, int y)
{
 return (x + y);
}
```

Şimdi bu metodu yukarıda hazırladığımız generic delegate aracılığıyla işleteceğiz. Bu amaçla forma 3 TextBox yerleştirip aşağıda verilen kodu yazdık.

```
public delegate T Temsilci<T>(T x, T y);
private void Form1_Click(object sender, EventArgs e)
{
 Temsilci<int> intTemsilci = new Temsilci<int>(Topla);
 int sayi1, sayi2, sayi3;
 sayi1 = Convert.ToInt32(textBox1.Text);
 sayi2 = Convert.ToInt32(textBox2.Text);
 sayi3 = intTemsilci(sayi1, sayi2);
 textBox3.Text = sayi3.ToString();
}
```

Şimdi bu generic delegate'in 2 string bilgiyi birleştirip geriye gönderen bir metodu temsil etmesini sağlayacağız. Bu amaçla "Birlestir" adında bir metot hazırladık ve yukarıda verilen kodu aşağıdaki gibi düzenledik.

```
public delegate T Temsilci<T>(T x, T y);
private void Form1_Click(object sender, EventArgs e)
{
 Temsilci<string> strTemsilci=new Temsilci<string>(Birlestir);
 textBox3.Text = strTemsilci(textBox1.Text, textBox2.Text);
}

public string Birlestir(string str1, string str2)
{
 return (str1 + str2);
}
```

23. BÖLÜM

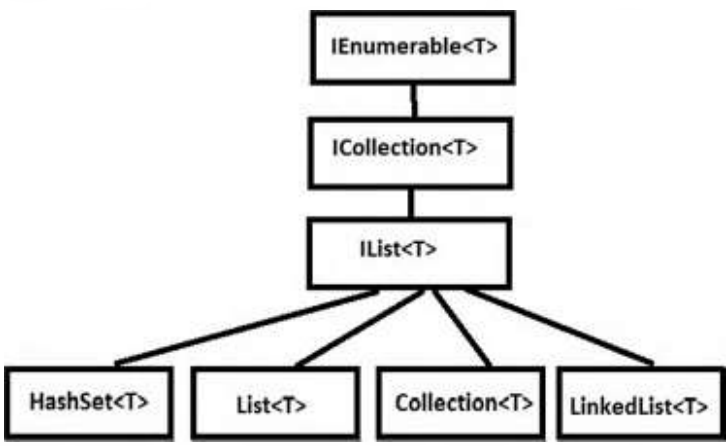
GENERIC KOLLEKSİYON SINIFLARI

Koleksiyonların anlatıldığı bölümde .NET Framework ile gelen koleksiyon sınıfları ayrıntılı bir şekilde incelendi. C# 2.0'dan itibaren daha önce sözü edilen koleksiyon sınıflarının **Generic** versiyonları kullanıma sunulmuştu. Koleksiyonlar adlı bölümde sözü edilen sınıflar **System.Collections**'da bulunurken generic desteği olan sınıf ve arayüzler **System.Collections.Generic**'te yer almaktadır. Bu bölümde .NET Framework ile gelen bazı generic koleksiyon sınıfları hakkında bilgi verilecektir.

List Sınıfı

C# kodu yazanlara liste tipe koleksiyona ihtiyaçları olduklarında ArrayList yerine List adlı generic sınıfı kullanmaları önerilmektedir. Bu bölümde List sınıfından kısaca söz edeceğiz. Çünkü List sınıfı birçok yönden ArrayList'e benzemektedir. ArrayList hakkında az çok bilgi edinmiş birisi List sınıfına adapte olmakta zorluk çekmeyecektir.

List sınıf, IList<T>, ICollection<T>, IEnumerable<T>, IList; ICollection ve IEnumerable arayüzlerini uyguladığı için çok sayıda metot ve özelliğe sahiptir. List sınıfının uyguladığı interface'leri resmetmek için aşağıda verilen hiyeraşik yapıyı hazırladık.



Visual Studio ile hazırlanan projelerdeki formların yani Form'u base class olarak kullanan sınıfların **using** bloğunda **System.Collections.Generic**'deki sınıflardan yararlanma isteği otomatik olarak işaret edilmektedir. Bu nedenle sınıf adından önce namespace adını yazmadan List sınıfından yararlanmak mümkündür.

```
List<string> liste1 = new List<string>();
```

List nesnelerinde hangi tipte bilgi saklanmak isteniyorsa, nesneyi hazırlarken o tipin belirtilmesi gerekmektedir. Yukarıda hazırladığımız List nesnesinde **string** bilgiler tutulabilir. Bu List nesnesinde string yerine integer bilgileri tutmak istemiş olsaydık nesne hazırlama işlemini aşağıdaki gibi yapardık.

```
List<int> liste1 = new List<int>();
```

Bu List nesnesi hazırlanırken kapasitesi belirtilmediği için başlangıçta 0'dır. Sonra **Add()** metodu ile ilk eleman eklendiğinde kapasite 4 olur. 5. eleman eklenmek istendiğinde mevcut kapasite yetmeyeceği için otomatik olarak 8 yapılır.

ArrayList ve diğer koleksiyon sınıflarında olduğu gibi List nesnesinin başlangıç kapasitesi yapıcı metot aracılığı ile belirtilebilir. List nesnesi aşağıdaki gibi hazırlanırsa dahili koleksiyonun başlangıç kapasitesi 16 olarak ayarlanır.

```
List<string> liste1 = new List<string>(16);
```

Bu satırı şöyle yorumlamak mümkündür: "liste1" adını verdiğimiz nesnede dahili koleksiyon olarak kullanılmak üzere string tipinde ve 16 elemanlı bir dizi değişken tanımlanır. Bu List nesnesine eklenen elemanlar dahili koleksiyonda tutulur, okumalar oradan yapılır. Ne zaman ki 16 sayısı yetmez, geri planda kapasitesi 32 olan yeni bir dahili koleksiyon hazırlanır.

List sınıfının aşırı yüklenmiş 3 farklı yapıcı metodu bulunmaktadır. Yapıcı metot parametresiz kullanıldığında koleksiyonun kapasitesi 0 olarak kabul edilmektedir. Yapıcı metodun 3. sürümü, başka bir koleksiyondan yola çıkılarak List nesnesi hazırlanmak istendiğinde kullanılmaktadır. **IEnumerable** arayüzünü uygulamış bir sınıftan yola çıkılarak hazırlanmış bir nesnenin yapıcı metoda parametre olarak nasıl verildiğini aşağıda görebilirsiniz.

```
List<int> liste1 = new List<int>(8);
List<int> liste2 = new List<int>(liste1);
```

Burada "liste2" nesnesini hazırlarken başka bir List nesnesinden yola çıktık. Dizi değişkenler ve diğer koleksiyon sınıfları **IEnumerable** arayüzünü uygulamış oldukları için List sınıfının yapıcı metoduna parametre olarak verilebilir. Aşağıda verilen kod işletildiğinde List nesnesine dizi değişkenin içeriği aktarılmış olunur.

```
private void Form1_Load(object sender, EventArgs e)
{
 int[] dizi = new int[3];
 dizi[0] = 2014;
 dizi[1] = 2015;
 dizi[2] = 2016;
 List<int> liste = new List<int>(dizi);
}
```

Bu List nesnesini hazırlarken "<>" operatörünün arasına "int" yazdığımız için yapıcı metoda parametre olarak verdiğimiz dizi değişkenin int tipinde olması şarttır. List nesnelerinde tutulan elemanlar dizi değişkenler gibi indekslendiği için istenen elemanın sıra numarası köşeli parantezin içine yazılıp mevcut elemanların içeriği elde edilebilir.

Yukarıdaki sayfada işaret edildiği üzere List sınıfı **ICollection** arayüzünü uygulamaktadır. Dolayısıyla bu arayüz kaynaklı metotları List nesnesi üzerinde işlem yaparken kullanabilir. Hazırlamak istediğiniz nesnede sadece ICollection arayüzünden gelen metotların kullanıma sunulmasını istiyorsanız List nesnesinin yapıcı metodunun geriye gönderdiği nesne referansını ICollection tipindeki değişkene aşağıda yapıldığı gibi aktarmanız yeterlidir..

```
ICollection<string> liste = new List<string>();
```

List Nesnesine Eleman Ekleme, Okuma ve Arama

Şimdi yukarıda hazırladığımız string tipteki List nesnesine birkaç eleman ekleyeceğiz. Tahmin edeceğiniz gibi konu koleksiyona eleman ekleme, silme veya arama olduğunda List nesneleri ArrayList nesnelerine çok benzemektedir.

```
private void Form1_Load(object sender, EventArgs e)
{
 List<string> liste1 = new List<string>();
 liste1.Add("Fenerbahçe");
 liste1.Add("Galatasaray");
 liste1.Add("Beşiktaş");
}
```

String tipte bilgiler aktarılmak üzere bu şekilde hazırlanan **List** koleksiyonuna integer veya başka tipte bilgiler aktarılmak istenirse hata meydana gelir. List koleksiyonun tipini Object olarak seçmeniz halinde performans ve işlev olarak **ArrayList**'ten pek farkı kalmaz.

```
List<object> liste1 = new List<object>();
```

Hazırlamak istediğiniz List nesnesine eleman olarak aktarılacak bilgiler kodlama sırasında belli ise nesne hazırlama satırında ayrıca elemanları aktarabilirsiniz. Aşağıda verilen satır bir metoda yazılıp işletilirse List nesnesi 3 elemana sahip olur.

```
List<string> iller =new List<string>() {"Adana","Ankara","İzmir"};
```

Daha sonra bu List nesnesine **Add**() metodu ile başka elemanlar eklenebilir. Dizi değişkenler hakkında bilgi verilen bölümden hatırlayacağınız gibi, C# programlama dilinin ilk sürümünden itibaren dizi değişkenlere bu şekilde ilk değer aktarma imkanı varken koleksiyonlara bu şekilde eleman ekleme imkanı programcılara C# 3.0 ile birlikte sunuldu. Bu şekilde hem koleksiyon hazırlanmasına hem de ilk değer olarak istenen sayıda elemanın eklenmesine **Collection Initializers** adı verilmektedir.

ArrayList sınıfı hakkında yazılanlardan hatırlayacağınız gibi eleman ekleme işlemini **Insert**(), **InsertRange**() ve **AddRange**() metotları ile yapmak mümkündür. Insert() metodu sayesinde List nesnesinin istenen yerine eleman eklenebilir. **InsertRange()** metodu ile başka bir generic koleksiyonun veya bir dizi değişkenin elemanları List koleksiyonuna eklenebilir. AddRange() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
string[] dizi = new string[3];
dizi[0] = "Fenerbahçe";
dizi[1] = "Galatasaray";
dizi[2] = "Beşiktaş";
List<string> liste1 = new List<string>();
liste1.AddRange(dizi);
```

List nesnesinin elemanlarını okuma yani içeriği elde etme işlemi ArrayList koleksiyonundan farklı değildir. Çünkü bu sınıfın indeksleyicisi aynı şekilde çalışmaktadır. Aşağıda verilen satırlar bir metoda yazılıp işletildiğinde List nesnesinin elemanları ListBox'a aktarılır.

```
List<string> liste1 = new List<string>();
liste1.Add("Fenerbahçe");
liste1.Add("Galatasaray");
liste1.Add("Beşiktaş");
listBox1.Items.Add(liste1[0]);
listBox1.Items.Add(liste1[1]);
listBox1.Items.Add(liste1[2]);
```

List koleksiyonun elemanlarını elde ederken **foreach** döngüsünden yararlanmak mümkündür. Başka bir deyişle mevcut bir List koleksiyonun elemanları üzerinde hareket etmek dizi değişkenin elemanları üzerinde dolaşmaktan farklı değildir.

```
List<string> liste1 = new List<string>();
liste1.Add("Fenerbahçe");
liste1.Add("Galatasaray");
liste1.Add("Beşiktaş");
foreach (string takim in liste1)
   listBox1.Items.Add(takim);
```

ArrayList koleksiyonlarında nasıl arama yapılıyorsa aynı şekilde List koleksiyonunda arama yapılabilmektedir. Hatırlayacağınız gibi **Contains**() metodu kendisine parametre olarak verilen elemanın koleksiyonda olup olmadığını araştırıyordu. **IndexOf**() metodu ise söz konusu elemanı koleksiyonda arayıp bulduğunda geriye sıra numarasını gönderiyordu. **IndexOf**() metodu kendisine parametre olarak verilen elemanı koleksiyonda bulamadığında geriye -1 göndermektedir. IndexOf() metodunun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void Ara_Click(object sender, EventArgs e)
{
 int sira = Liste1.IndexOf("Beşiktaş");
 if (sira == -1)
   MessageBox.Show("Aradığınız eleman koleksiyonda yok");
 else
   this.Text = sira.ToString();
}
```

LINQ İle Gelen Metotlarla List Nesnesi Üzerinde İşlem Yapmak

Yukarıdaki sayfalarda C# 2.0'dan itibaren programcıların istifadesine sunulan List sınıfının bazı metotlarını kullanarak arama yaptık. Bu metotların bazıları ArrayList sınıfı hakkında bilgi verilirken anlatıldığı için tekrara düşmemek için burada üzerinde durulmadı. ArrayList'te olmayıp List nesnesinde arama yaparken kullanabileceğimiz bazı metotlar bulunmaktadır. **Exists**() bu metotlardan birisidir ve imzası aşağıdaki gibidir.

```
public bool Exists(Predicate<T> match);
```

Exists() metodunun imzasına baktığımızda .NET Framework ile gelen **Predicate** adlı delegate'i parametre olarak alıp geriye bool bilgi gönderdiğini görürsünüz. Bu imzanın açıklaması şudur: C# 3.0 ile gelen bir genişletme metodu olan Exitsts() aracılığıyla geriye bool bilgi gönderen metotlar hazırlayıp kullanabilirsiniz. Dolayısıyla Exists() metodu ile yapılacak işlem, programcının yazacağı metoda bağlıdır. Bu metodun nasıl kullanıldığını en basit şekliyle anlatmak için önce aşağıdaki gibi bir List nesnesi hazırladık.

```
List<int> yillar;
private void Form1_Load(object sender, EventArgs e)
{
 yillar = new List<int>();
 yillar.Add(2013);
 yillar.Add(2014);
 yillar.Add(2015);
 yillar.Add(2016);
}
```

Exists() metodunu üzerinde test edeceğimiz List nesnesini hazırladıktan sonra bu metodun gerek duyacağı Predicate delegate için aşağıdaki gibi bir metot hazırladık.

```
bool Varmi(int i)
{
 return yillar.Contains(i);
}
```

Bu metot parametre olarak aldığı bilgiyi Contains() metodu sayesinde yukarıda hazırladığımız List nesnesinde arayacaktır. Bulursa geriye true, yoksa false gönderecektir. En son olarak formdaki düğme tıklandığında işletilecek metodu aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
 Predicate<int> varDelege = Varmi;
 bool sonuc = varDelege(2014);
 if (sonuc)
   MessageBox.Show("Söz konusu yıl listede var");
}
```

Gördüğünüz gibi yaptığımız işlem son derece basit olmasına rağmen oldukça fazla kod yazdık. Burada yaptığımız işlemin açıklaması bir sonraki konu başlığında verileceği için tekrara düşmemek için bu kodun üzerinde konuşmadık. Bir sonraki sayfada **RemoveAll**() metodu hakkında yazılanları okuduğunuzda burada yapılan işlem sizin için anlaşılmaz olmayacaktır.

Diğer yandan Exists() gibi genişletme metotlarına lambda ifadelerini parametre olarak verip işlemi basitleştirmek mümkündür. Burada yaptığımız işlemi lambda ifadesi olarak yapmak amacıyla yukarıda hazırlayıp Varmi() adını verdiğimiz metodu silip button1 nesnesi tıklandığında işletilecek kodu aşağıdaki gibi düzenledik. Burada yapılan işlemler hakkında ayrıntılı bilgiyi lambda ifadelerin anlatıldığı bölümde bulabilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 bool sonuc = yillar.Exists(yil => yillar.Contains(2014));
 if (sonuc)
   MessageBox.Show("Söz konusu yıl listede var");
}
```

List nesnelerinde arama yaparken LINQ ile gelen birkaç genişletme metodundan yararlanmak mümkündür. Bu metotlar System.Enumerable sınıfında kodlanmış olmasına rağmen IEnumerable<> arayüzünü genişletmektedir. List nesneleri üzerinde işlem yaparken bu genişletme metotlarından yaralanmak istiyorsanız using bloğunda "using System.Linq;" satırına yer vermeniz gerekir. Bu metotlarla LINQ sorguları ve Lambda ifadeleri hakkında yeteri kadar bilgi edindikten sonra ilgilenmeniz önerilir.

List Nesnesinden Eleman Silme

List koleksiyonundan eleman silinirken ArrayList sınıfından bildiğiniz **Remove**, **RemoveAt** ve **RemoveRange** metotları kullanılmaktadır. Bu metotların kullanımı farklı olmadığı için burada tekrar üzerinde durmuyoruz. Diğer yandan List nesneleri üzerinde işlem yaparken C# diline LINQ sorguları için eklenen metotları ve operatörleri kullanmak mümkündür. Bu metotlar hakkında konuşmayı LINQ konulu bölüme bıraktığımız için burada üzerinde durmayacağız. Yine de örnek olsun diye RemoveAll() metodundan söz edeceğiz. Bu amaçla yukarıda hazırladığımız List nesnesini aşağıdaki gibi düzenledik.

```
List<int> liste = new List<int>();
liste.Add(1903);
liste.Add(1905);
liste.Add(1907);
liste.Add(1967);
```

Şimdi RemoveAll() metodunu öyle bir kullanacağız ki kuruluş yılı 1907'ye eşit veya daha büyük olanları List nesnesinden sileceğiz. Önce Visual Studio'dan yararlanıp RemoveAll() metodunun orijinal haline baktık.

```
public int RemoveAll(Predicate<T> match);
```

Bu kitabın Predicate delegate'ler hakkında bilgi verilen bölümünü okumayanlara bu metot imzası bir şey ifade etmez. Bu nedenle RemoveAll() metodunun imzası hakkında kısa bir açıklama yapacağız.

- Bu metodun parametresi bool bir sonuc üreten bir delegate'tir.
- Predicate, bu amaçla kullanılabilinecek ve .NET Framework ile birlikte gelen bir delegate'tir.
- Bu nedenle bool sonuç üreten bir metot hazırlayıp bu metodun Predicate<> delegate tarafından temsil edilmesini sağlayıp RemoveAll() metoduna parametre olarak verebiliriz.
- Bize gerekli olan metot, List nesnesinin ilgili elemanı 1907'ye eşit veya büyükse geriye true, yoksa false gönderen bir metottur.

Bu amaçla hazırladığımız metot aşağıdadır. List nesnemiz int tipinde bilgiler içerdiği için Silinsinmi() adını verdiğimiz metodun int tipinde bir parametreye sahip olmasını sağladık.

```
static bool Silinsinmi(int i)
{
 return i >= 1907;
}
```

List nesnesinin söz konusu elemanı 1907'den küçük ise geriye false gönderilecektir. Bu metot, .NET Framework ile gelen ve sık kullanılan Predicate delegate'in yapısına uygundur. List nesnesinden istediğimiz şartlara uyan bütün elemanları RemoveAll() metodu ile silerken kullanacağımız metodu hazırladıktan sonra şimdi bu metodun Predicate tipinde bir nesne ile temsil edilmesini sağlayacağız. Bu amaçla aşağıdaki gibi bir satır hazırladık.

```
private void button2_Click(object sender, EventArgs e)
{
 List<int> liste = new List<int>();
 liste.Add(1903);
 liste.Add(1905);
 liste.Add(1907);
 liste.Add(1967);
 Predicate<int> silDelege = Silinsinmi;
}
static bool Silinsinmi(int i)
{
 return i >= 1907;
}
```

Bu andan itibaren Silinsinmi() adını verdiğimiz metodu burada "silDelege" adını verdiğimiz delegate nesnesi temsil ediyor. Geriye bir tek bu delegate nesnesini List sınıfının RemoveAll() metoduna parametre olarak verip istenen elemanları silmek kaldı. Bu metodu nasıl kullandığımızı aşağıda görebilirsiniz.

```
private void button2_Click(object sender, EventArgs e)
{
 List<int> liste = new List<int>();
 liste.Add(1903);
 liste.Add(1905);
 liste.Add(1907);
 liste.Add(1967);
 Predicate<int> silDelege = Silinsinmi;
 liste.RemoveAll(silDelege);
}
```

Yukarıda işaret edildiği üzere RemoveAll() metodu bool sonuç üreten bir metodu temsil eden bir delegate'i parametre olarak almaktadır. RemoveAll() metodu işletildiğinde ise söz konusu List nesnesinin her elemanın içeriğini tek tek burada Silinsinmi() adını verdiğimiz metoda gönderiyor. Metottan geriye o eleman için true gönderildiğinde o elemanı List nesnesinden silmektedir. Predicate delegate tarafından temsil edilen metot geçerli eleman için geriye false gönderdiğinde ise RemoveAll() o elemana dokunmuyor.

Şimdi yukarıda RemoveAll() metodu ile ilgili olarak yaptığımız işlemleri özetleyeceğiz:

- List nesnesinden eleman silerken kullanmak üzere geriye bool sonuç gönderen bir metot hazırladık.
- Bu metodun .NET Framework ile gelen Predicate tipinde bir delegate nesnesi tarafından temsil edilmesini sağladık.
- Bu delegate nesnesini RemoveAll() metoduna parametre olarak verip istediğimiz elemanları List nesnesinden sildik.

List Sınıfının GetEnumerator() Metodu

List nesnesinin elemanlarını bir numaralayıcıya almak istiyorsanız **GetEnumerator**() metodunu kullanabilirsiniz. Aşağıda verilen kod işletildiğinde List nesnesinin elemanları numaralayıcıya alınır ve ilk elemanın içeriği TextBox'a aktarılır.

```
List<string> liste1;
IEnumerator<string> Enum_nesne;
private void Form1_Load(object sender, EventArgs e)
{
 liste1 = new List<string>();
 liste1.Add("Fenerbahçe");
 liste1.Add("Galatasaray");
 liste1.Add("Beşiktaş");
 Enum_nesne = liste1.GetEnumerator();
 Enum_nesne.MoveNext();
 textBox1.Text = Enum_nesne.Current.ToString();
}
```

Burada kullanılan GetEnumerator() metodu, ArrayList sınıfının GetEnumerator() metodundan farklı olarak geriye generic bir numaralayıcı göndermektedir. İlgilendiğimiz List nesnesi string bilgiler içerdiği için IEnumerator tipinde değişken tanımlarken bunu gözettik.

GetEnumerator() metodu ile yapılan işlemi bahane edip araya girip bir ayrıntıyı dikkatinize sunacağız. Bu amaçla generic IEnumerable<> arayüzünün orijinal halini buraya alacağız. Sadece bir metoda sahip bu arayüzü incelerseniz generic olmayan IEnumerable arayüzünün mirasçısı olduğunu fark ederseniz. Dolayısıyla IEnumerable arayüzünün generic olmayan hali hakkında bilgi sahibi olmakta fayda vardır.

```
public interface IEnumerable<out T> : IEnumerable
{
 IEnumerator<T> GetEnumerator();
}
```

ConvertAll() Metodu

Bazen List nesnesinin bütün elemanları başka tipteki List nesnesine aktarılmak istenir. Bu işlem için kullanabileceğimiz ConvertAll() metodu hakkında bilgi vermek için int tipinde bir List nesnesi hazırlayıp birkaç eleman ekledik.

```
List<int> yillar = new List<int>();
yillar.Add(2015);
yillar.Add(2016);
yillar.Add(2017);
```

Basit bir döngü ile bu List nesnesinin elemanlarını tek tek okuyup istenen tipe dönüştürüp başka bir List nesnesine eleman olarak eklemek mümkün olmakla birlikte amacımız List sınıfının **ConvertAll**() metodu hakkında bilgi vermektir. Bu amaçla önce aşağıdaki gibi "Cevir" adında bir metot hazırladık. Bu metot kendisine parametre olarak verilen int bilgiyi Convert sınıfının **ToString**() metodu ile string tipi dönüştürüp geriye göndermektedir.

```
public static string Cevir(int i)
{
 return Convert.ToString(i);
}
```

Dönüştürme yapacak metodu hazırladıktan sonra **ConvertAll()** metodunu aşağıdaki gibi kullanıp int tipindeki List nesnesinden yola çıkıp String tipte List nesnesi hazırlanabilir.

```
private void Form1_Load(object sender, EventArgs e)
{
 List<int> yillar = new List<int>();
 yillar.Add(2005);
 yillar.Add(2006);
 yillar.Add(2007);
 List<string> liste = yillar.ConvertAll(
              new Converter<int,string> (Cevir));
}
```

Bu şekilde kullanılan ConvertAll() metodu List nesnesinin her elemanı için otomatik olarak işletilmektedir. ConvertAll() metodu içinde kullanılan **Converter** delegate nesnesi mevcut ve yeni List nesnesinin tipini parametre olarak almaktadır.

ForEach() Metodu

List sınıfının, foreach döngüsü ile benzer işleve sahip **ForEach**() adında bir metodu vardır. Bu metot her eleman için mevcut bir metodu otomatik olarak işletebilmektedir. Yani List nesnesinin bütün elemanlarına aynı işlemi uygulamak istediğinizde bu metot işinize yarayacaktır. List sınıfının **ForEach**() metodunun nasıl kullanıldığına örnek olması amacıyla **Ekle**() adında bir metot hazırladık.

```
private void Ekle(string str)
{
 string eleman = str.ToUpper();
 listBox1.Items.Add(eleman);
}
```

Ekle() adını verdiğimiz bu metot parametre olarak aldığı string bilgiyi büyük harfe çevirip formdaki ListBox nesnesine eleman olarak eklemektedir. Başka bir anlatımla, elimizdeki List nesnesinin bütün elemanlarını sıra ile elde edip büyük harfe dönüştüreceğiz. Devamında elemanın büyük harfe çevrilmiş halini formdaki ListBox nesnesine ekleyeceğiz. List sınıfının ForEach() metodu her elemana uygulanacak işlem için gerek duyduğu metodu hazırladıktan sonra aşağıdaki gibi kullandık.

```
private void button1_Click(object sender, EventArgs e)
{
 List<string> aylar = new List<string>();
 aylar.Add("ocak");
 aylar.Add("şubat");
 aylar.Add("mart");
 aylar.ForEach(Ekle);
}
```

Burada hazırlayıp Ekle() adını verdiğimiz metodu List sınıfının ForEach() metoduna parametre olarak verdiğimiz için List nesnesi kaç elemana sahipse her eleman için Ekle() metodu sanki bir döngü hazırlanmış gibi işletilir. Bu işlem sırasında elimizdeki List nesnesinin elemanları ilk hallerini korumaktadır. Bu sırada ForEach() metodundan haberimiz olmasaydı aynı işlem için aşağıdaki gibi kod yazabilirdik.

```
private void button1_Click(object sender, EventArgs e)
{
 List<string> aylar = new List<string>();
 aylar.Add("ocak");
 aylar.Add("şubat");
 aylar.Add("mart");
 foreach (string item in aylar)
  {
   string str = item.ToUpper();
   listBox1.Items.Add(str);
  }
}
```

Önce metot hazırlayıp sonra bu metodu List sınıfının ForEach() metoduna parametre vermek için yazılan kodlarla burada yapılan işlem için yazılan kodlar birlikte değerlendirilirse ForEach() metodunun önemli katkısının olmadığını düşünebilirsiniz. Burada Ekle() adını verdiğimiz metotta yaptığımız işlem foreach döngüsü içinde yapılmaya uygun olduğu için ForEach() metodu için olmazsa da olur diyebilirsiniz.

Şimdi konu üzerinde düşünmenize imkan sağlamak üzere yukarıda verdiğimiz kodda değişiklik yapacağız. Yapacağımız değişiklik sonucu yukarıda hazırladığımız Ekle() metodu işletildiğinde mevcut list nesnesinin elemanlarının büyük harfe çevrilmiş halleri başka bir List nesnesine eklenecektir.

```
List<string> AYLAR;
private void button1_Click(object sender, EventArgs e)
{
 AYLAR = new List<string>();
 List<string> aylar = new List<string>();
 aylar.Add("ocak");
 aylar.Add("şubat");
 aylar.Add("mart");
 aylar.ForEach(Ekle);
 foreach (string item in AYLAR)
 {
  listBox1.Items.Add(item);
 }
}
private void Ekle(string str)
{
 string eleman = str.ToUpper();
 AYLAR.Add(eleman);
}
```

Şimdiye kadar örnek olsun diye hazırladığımız List nesnesinin dahili koleksiyonunda.NET Framework ile gelen build-in tipleri sakladık. Programcılar sıklıkla List nesnelerinde kendi hazırladıkları tiplerin örneklerini saklamaya ihtiyaç duyarlar.

Generic Stack Sınıfı

Koleksiyonlar adlı bölümde generic olmayan **Stack** sınıfı hakkında bilgi verilmişti. Stack sınıfı **son giren ilk çıkar** ilkesine göre işlev görüyordu ve bu sınıftan yararlanılarak hazırlanan koleksiyonlarda object tipinde bilgiler tutulabiliniyordu.

.NET Framework 2.0'dan itibaren Stack sınıfının generic versiyonu verilmeye başlandığı için değişik tipte bilgileri eleman olarak saklayabilen tip güvenli koleksiyon nesneleri hazırlamak mümkün hale geldi. IEnumerable<T>, ICollection ve Enumerable arayüzlerini uygulayan bu sınıf tipinde nesnelerin nasıl hazırladığını aşağıda görebilirsiniz.

```
Stack<string> takimlar = new Stack<string>();
takimlar.Push("Fenerbahçe");
takimlar.Push("Başiktaş");
takimlar.Push("Galatasaray");
takimlar.Push("Trabzonspor");
```

Generic Stack nesnesini hazırlarken kapasitesini belirtebilirsiniz. Bu satırlar bir metoda yazılıp işletildiğinde 4 elemanlı ve string tipte bilgiler içeren bir Stack nesnesi hazırlanır. Push() metodu ile koleksiyona en son eklenen eleman en üstte durur.

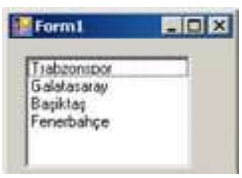
Stack nesnesini hazırlarken mevcut bir stack nesnesinden veya dizi değişkenden, başka bir deyişle enumerable olan herhangi bir nesneden yola çıkabilirsiniz. Aşağıda verilen kodda önce bir Stack nesnesi hazırlayıp Push() metodu ile 3 elemana sahip olmasını sağladık. Ardından bu stack nesnesinden yola çıkıp ikinci bir stack nesnesi hazırladık.

```
Stack<int> yillar = new Stack<int>();
yillar.Push(2006);
yillar.Push(2007);
yillar.Push(2008);
Stack<int> Yigin = new Stack<int>(yillar);
this.Text = Yigin.Pop().ToString();
```

Stack nesnesinden okuma yapılırken **Pop**() metodu kullanılmaktadır. Pop() metodu her kullanıldığında eleman sayısının bir azaldığını unutmamak gerekir. Bunu göstermek için forma bir ListBox yerleştirip aşağıda verilen kodu hazırladık.

```
Stack<string> takimlar = new Stack<string>();
takimlar.Push("Fenerbahçe");
takimlar.Push("Başiktaş");
takimlar.Push("Galatasaray");
takimlar.Push("Trabzonspor");
int sayi = takimlar.Count;
for (int i = 0; i < sayi; i++)
  listBox1.Items.Add(takimlar.Pop());
```

Bu kod işletildikten sonra Stack nesnesinin eleman sayısı 0 olur. Pop() metodu sürekli olarak en üstteki elemanı çıkarıp geriye gönderdiği için aşağıdaki gibi bir sonuç alınır.



Generic olmayan Stack sınıfı hakkında yazılanlardan hatırlayacağınız gibi **Peek**() metodu stack'ın en üstünde duran eleman elde edilmek istendiğinde kullanılmaktadır. Stack koleksiyonunda arama yapılırken **Contains**() metodunun kullanıldığını biliyorsunuz.

Stack nesnesinin elemanlarını bir dizi değişkene aktarmak istiyorsanız **ToArray**() metodunu kullanabilirsiniz. ToArray() metodunu uygulayacağınız stack koleksiyonu hangi tipte bilgi içeriyorsa önceden o tipte bir dizi değişken tanımlamanız gerekir.

```
string[] dizi = takimlar.ToArray();
```

Stack nesnesinin elemanları dizi değişkene aktarılırken stack nesnesine en son eklenen eleman dizi değişkenin ilk elemanı yapılır. Stack nesnesinin elemanlarını **List** nesnesine aktarmak istiyorsanız **ToList**() metodunu kullanabilirsiniz. Bu sayede mevcut bir Stack nesnesinin elemanlarını List nesnesine aktarıp bu elemanlar üzerinde istediğiniz işlemi yapma imkanını elde etmiş olursunuz.

```
Stack<int> yillar = new Stack<int>();
yillar.Push(2006);
yillar.Push(2007);
yillar.Push(2008);
List<int> liste;
liste = yillar.ToList<int>(); //using System.Linq gerektirir
this.Text = liste[0].ToString(); // 2008 elde edilir
```

Burada kullandığımız **ToList**() metodu, LINQ sorgularında kullanmak üzere **Enumerable** sınıfında kodlanan bir genişletme metodudur. Dolayısıyla System.Linq'de yer alan sınıflardan yararlanma isteği using bloğunda belirtilmediyse ToList() metodu bu şekilde kullanılamaz.

Generic Queue Sınıfı

Generic Queue sınıfı generic olmayan Quene'den pek farklı değildir. İlgili bölümden hatırlayacağınız gibi Queue sınıfı **ilk giren ilk çıkar** prensibine göre işlev görmektedir.

```
Queue<int> yillar = new Queue<int>();
yillar.Enqueue(2005);
yillar.Enqueue(2006);
yillar.Enqueue(2007);
yillar.Enqueue(2008);
int sayi = yillar.Count;
for (int i = 0; i < sayi; i++)
  listBox1.Items.Add(yillar.Dequeue());
```

Queue koleksiyonuna eleman ekleme **Enqueue**() ile yapılırken eleman çıkarma **Dequeue**() metodu ile yapılmaktadır. Yukarıda verilen kodda Enqueue() metodunu 4 kez kullanarak koleksiyonun 4 elemana sahip olmasını sağladık. Devamında for döngüsü içinde **Dequeue**() metodunu 4 kez kullandığımız için for döngüsünden çıkıldığı noktada Queue nesnesinin eleman sayısı 0 olur.

Diğer yandan mevcut bir dizi değişkenden yola çıkıp generic Queue nesnesi hazırlamak mümkündür. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Bu kodda int tipindeki dizi değişkeni Queue sınıfının yapıcı metoduna parametre olarak verdiğimiz için bu dizi değişkenin bütün elemanları generic Queue nesnesine eleman yapılır.

```
int[] dizi = new int[3];
dizi[0] = 2009;
```

```
dizi[1] = 2010;
dizi[2] = 2011;
Queue<int> yillar = new Queue<int>(dizi);
int sayi = yillar.Count;
for (int i = 0; i < sayi; i++)
  listBox1.Items.Add(yillar.Dequeue());
```

Generic Queue koleksiyonun elemanlarının hepsini aynı tipteki dizi değişkene aktarmak istiyorsanız **CopyTo**() metodunu kullanabilirsiniz. CopyTo() metodu mevcut bir dizi değişkeni ve dizi değişkenin eleman aktarılacak boyutunu parametre olarak almaktadır.

```
Queue<int> yillar = new Queue<int>();
yillar.Enqueue(2010);
yillar.Enqueue(2011);
yillar.Enqueue(2012);
int[] dizi = new int[yillar.Count];
yillar.CopyTo(dizi, 0);
```

Queue koleksiyonun elemanlarının hepsini bir numaralayıcıya yani Enum tipine aktarmak istiyorsanız **GetEnumerator**() metodundan yararlanabilirsiniz. Bu metottan yararlanıp generic Queue koleksiyonun elemanlarını enum nesnesine aktardıktan sonra **MoveNext**() metodu ile enum üzerinde ileri doğru hareket edebilirsiniz.

```
IEnumerator<int> enum_tipi;
enum_tipi = yillar.GetEnumerator();
enum_tipi.MoveNext();
this.Text = enum_tipi.Current.ToString();
```

Yukarıda Stack ve Queue koleksiyonlarının generic versiyonlarını hazırlarken kapasitelerini belirtmedik. Stack ve Queue nesnelerini hazırlarken yapıcı metoda tercih ettiğiniz kapasiteyi parametre olarak vermezseniz bunun 0 olduğu varsayılır. Eleman eklendikçe kapasite yetersizliği söz konusu olduğunda otomatik olarak 2 katına çıkarılır.

Queue nesnesinin kapasitesini eleman sayısına eşitlemek istiyorsanız **TrimExcess**() metodunu kullanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
yillar.TrimExcess();
```

Queue nesnesinin ilk elemanını elde etmek istiyorsanız **First**() metodunu kullanabilirsiniz. First() metodu uygulandığı Queue nesnesinin ilk elemanını geriye göndermektedir.

```
Queue<int> yillar = new Queue<int>(3);
yillar.Enqueue(2010);
yillar.Enqueue(2011);
int ilk = yillar.First(); // ilk değişkeni 2010 olur
```

Queue nesnesinin son elemanını elde etmek istiyorsanız **Last**() metodunu kullanabilirsiniz.

```
Queue<int> yillar = new Queue<int>(3);
yillar.Enqueue(2010);
yillar.Enqueue(2011);
int son = yillar.Last(); // son değişkeni 2011 olur
```

Generic Queue nesnesinin tipini sayısal olarak seçtiyseniz(int, double vs..) **Max**() ve **Min**() metotlarından yararlanıp en küçük ve en büyük değere sahip elemanını elde edebilirsiniz.

```
Queue<int> yillar = new Queue<int>();
yillar.Enqueue(2009);
yillar.Enqueue(2010);
yillar.Enqueue(2011);
int kucuk = yillar.Min(); // kucuk değişkeni 2009 olur
int buyuk = yillar.Max(); // buyuk değişkeni 2011 olur
```

Dictionary ve SortedDictionary Sınıfları

Generic olmayan koleksiyon sınıflarının anlatıldığı bölümden hatırlayacağınız gibi anahtar-değer çifti şeklinde düzenlenmiş bilgileri koleksiyon şeklinde organize edip bellekte tutma gereği duyulduğunda **Hashtable** sınıfından yararlanılıyordu. Generic bir sınıf olan Dictionary sınıfı Hashtable ile benzer işleve sahiptir.

```
Dictionary<string,string> iller = new Dictionary<string, string>();
iller.Add("01", "Adana");
iller.Add("06", "Ankara");
iller.Add("07", "Antalya");
iller.Add("34", "İstanbul");
```

Dictionary sınıfı ile hazırlanan koleksiyonun her elemanı 2 bilgiden meydana geldiği için nesne hazırlanırken <> arasında 2 tip adı yazmak gerekmektedir. Koleksiyonda aynı **key** değerine sahip birden fazla elemanın olmasına izin verilmez. Eleman ekleme işlemini **Add**() metodu yerine aşağıdaki gibi yapmak mümkündür.

```
Dictionary<string,string> iller = new Dictionary<string, string>();
iller["01"] = "Adana";
iller["06"] = "Ankara";
iller["07"] = "Antalya";
iller["34"] = "İstanbul";
```

Dictionary koleksiyonundan eleman silmek için **Remove**() metodu kullanılabilir. Bu metot silinmek istenen elemanın anahtarını parametre olarak almaktadır. Dictionary koleksiyonun bütün elemanları silinmek istendiğinde **Clear**() metodu kullanılmaktadır.

Dictionary koleksiyonundaki elemanların anahtar bilgilerini elde etmek istiyorsanız **Keys** özelliğinden yararlanabilirsiniz. Bu özelliğin nasıl kullanıldığını anlatmak için forma bir ListBox yerleştirip aşağıda verilen kodu yazdık.

```
ICollection<string> anahtarlar;
anahtarlar = iller.Keys;
foreach (string plaka in anahtarlar)
 {
  listBox1.Items.Add(plaka);
 }
```

Dictionary koleksiyonundaki elemanların "değer" bilgilerini elde etmek istiyorsanız **Values** özelliğinden yararlanabilirsiniz. Bu özellikten nasıl yararlanıldığını anlatmak için forma 2. bir ListBox yerleştirip yukarıda verilen koda ekleme yaptık.

```
ICollection<string> anahtarlar = iller.Keys;
ICollection<string> degerler = iller.Values;
foreach (string plaka in anahtarlar)
 {
  listBox1.Items.Add(plaka);
 }
foreach (string il in degerler)
 {
  listBox2.Items.Add(il);
 }
```

Dictionary koleksiyonunda arama yapmak istiyorsanız **ContainsKey**() metodunu kullanabilirsiniz. ContainsKey() metodu aranmak istenen elemanın anahtar bilgisini parametre olarak almaktadır. Kendisine parametre olarak verilen anahtara sahip bir eleman varsa geriye true yoksa false göndermektedir. Arama işlemini "değer" bilgisine göre yapmak istiyorsanız **ContainsValue**() metodunu kullanmalısınız.

```
bool sonuc = iller.ContainsKey(textBox1.Text);
if (!sonuc)
  MessageBox.Show("Aradığınız eleman bulunamadı");
else
  textBox2.Text = iller[textBox1.Text];
```

Dictionary koleksiyonundaki elemanları bir numaralayıcıya(enum nesnesine) almak istiyorsanız **GetEnumerator()** metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz. Aşağıda verilen satırlar sayesinde Dictionary koleksiyonun ilk elemanın **key** bilgisi ilk TextBox'a, **value** bilgisi ise 2. TextBox'a aktarılır.

```
IDictionaryEnumerator Liste = iller.GetEnumerator();
Liste.MoveNext();
textBox1.Text = Liste.Key.ToString();
textBox2.Text = Liste.Value.ToString();
```

MoveNext() metodu her işletildiğinde **IDictionaryEnumerator** nesnesindeki bir sonraki elemanın üzerine gidilir. Bu nedenle aşağıdaki gibi kod yazıp MoveNext() metodu ile enum nesnesi üzerinde ileri doğru hareket edebilirsiniz.

```
private void Sonraki_Click(object sender, EventArgs e)
{
 if (Liste.MoveNext())
  {
   textBox1.Text = Liste.Key.ToString();
   textBox2.Text = Liste.Value.ToString();
  }
}
```

Yukarıda örnek olması için hazırladığımız Dictionary nesnesine aktarılan anahtar ve değer bilgilerinin ikisi de string tipteydi. Koleksiyona aktarmak istediğiniz elemanların anahtar ve değer bilgilerinin istediğiniz tipte olmasını sağlayabilirsiniz. Örneğin aşağıda verilen generic koleksiyonun anahtar bilgisi string iken değer bilgisi double tiptedir.

```
Dictionary<string,double> Oyuncular;
Oyuncular = new Dictionary<string, double>();
```

```
Oyuncular.Add("Mehmet Okur", 2.10);
Oyuncular.Add("İbrahim Kutluay", 1.97);
Oyuncular.Add("Hidayat Türkoğlu", 2.05);
```

Yukarıda kısaca hakkında bilgi verilen generic Dictionary koleksiyonunda elemanlara anahtar bilgisi ile erişilmektedir. Tabii bütün elemanları **GetEnumerator()** metodu ile bir numaralayıcıya yani enum nesnesine alıp elemanlara daha esnek bir şekilde erişim sağlamak mümkündür. Anahtar-değer şeklinde bilgiler içeren koleksiyonda elemanlarının sıralı bir şekilde durmasını istiyorsanız **SortedDictionary** sınıfından yararlanabilirsiniz.

```
private void Form1_Load(object sender, EventArgs e)
{
 SortedDictionary<string, string> sozluk;
 sozluk = new SortedDictionary<string,string>();
 sozluk.Add("Screen", "Ekran");
 sozluk.Add("Cpu", "işlemci");
 sozluk.Add("Ram", "Bellek");
}
```

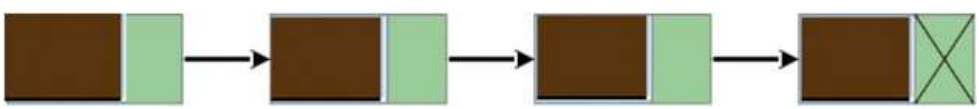
SortedDictionary nesnesine elemanlar hangi sırada eklenirse eklensin otomatik olarak anahtar bilgisine göre sıralanır. Anahtarını bildiğiniz elemanın değer bilgisini kolayca öğrenebilirsiniz. Ancak aşağıda yapıldığı gibi olmayan bir anahtar bilgisi SortedDictionary nesnesinin indeksleyicisine parametre olarak verilirse hata meydana gelir.

```
textBox1.Text = sozluk["Ram"];
```

Bu nedenle generic SortedDictionary nesnesinin indeksleyicisine parametre olarak verilmek istenen anahtarın koleksiyonda varlığı **ContainsKey()** metodu ile araştırılabilir.

LinkedList Sınıfı

LinkedList'lere günlük hayattan bir örnek vermek gerekirse birbirine bağlanmış tren vagonları akla gelebilir. Vagon şeklinde düzenlenmiş yapıyı aşağıdaki gibi resmetmek mümkündür. Burada vagon olarak adlandırmış olduğumuz 2 parçadan meydana gelen yapıya programcılar node (düğüm) adını veriyorlar.

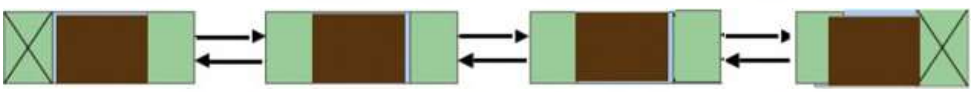


Vagon örneğinden devam edersek, birbirine bağlanmış bu vagonların her birisi 2 bölmeden meydana gelmektedir. Her düğümün ilk bölmesine ilgilenilen veri, ikinci bölmeye ise bir sonraki düğümün adresi yerleştirilmektedir. Yani ilk düğümün 2. bölmesinde 2. düğümün adresi olmaktadır. 2. düğümün 2. bölmesine ise 3. düğümün adresi konulmaktadır.

Bağlı listeler hakkında bilgi vermek için kullandığımız bu temsili resimde toplam 4 düğüme yer verdik. Son düğüme dikkat ederseniz 2. parçasına adres bilgisi yok anlamında "X" işareti konulmuştur. Çünkü düğümlerin 2. parçasında bir sonraki düğümün adresi olmaktadır. Dolayısıyla son düğümün 2. bölmesinde adres bilgisi olmaz. Ne zaman ki bağlı listedeki düğüm sayısı 5 yapılırsa 4. düğümün ikinci bölmesine bir sonraki düğümün adresi konulur.

Verilen temsili resimden de fark etmiş olacağınız gibi düğümler arası hareket ilk düğümden son düğüme doğrudur. Düğümler arası dolaşımın sadece ileri doğru olabildiği bağlı listelere **Singly Linked List** (tek yönlü bağlı liste) adı verilmektedir.

Bağlı liste üzerinde geriye doğru hareket istenildiğinde her düğümde 3 bölmeye ihtiyaç olmaktadır. Düğümüm ortasında bağlı listede tutulmak istenen veri olurken sol bölmesinde bir önceki düğümün, sağ bölmesinde ise bir sonraki düğümün adresi olmaktadır. Dolayısıyla ilk düğümün sol bölmesinde ve son düğümün sağ bölmesinde bilgi olmaz. Bu yapıyı aşağıdaki gibi resmetmek mümkündür. Bu şekilde düğmeler arasında 2 yönlü dolaşıma izin veren Linked List'lere Doubly Linked List (çift yönlü bağlı liste)adı verilmektedir.



Linked List'lerin en önemli özelliği sonradan listenin istenilen konumuna yeni düğümlerin yerleştirilebilir olmasıdır. Örneğin bu temsili resimde 2. ile 3. düğümün arasında 5. bir düğüm eklendiğinde 2. düğümün sağ ve 3. düğümün sol bölmesindeki adres bilgileri değiştirilir.

C# kodu yazarak bu şekilde çalışacak veri yapısını sıfırdan hazırlamak mümkündür. .NET Framework ile birlikte Double Linked List işlevi görecek **LinkedList** adında bir sınıf geldiği için elinizdeki verileri bağlı liste şeklinde düzenlenmek istediğinizde bu sınıfı kullanabilirsiniz.

Linked List'ler hakkında bu şekilde biraz genel bilgi verdikten sonra şimdi .NET Framework ile gelen LinkedList sınıfından kısaca söz edeceğiz. Bu sınıfın sadece generic sürümü vardır. System.Collections.Generic'te bulunan LinkedList sınıfı tipinde nesne hazırlamak yukarıdaki sayfalarda hakkında bilgi verilen generic List<> sınıfından farklı değildir.

```
static void Main(string[] args)
{
 LinkedList<string> liste = new LinkedList<string>();
}
```

En başta belirtelim ki LinkedList sınıfı ile yapılabilinecek işlemlerin neredeyse tamamı List sınıfıyla yapılabilmektedir. Aynı yorum LinkedList sınıfı için de yapılabilir. Yani List sınıfıyla yapılan işlemlerin hemen hepsi LinkedList ile yapılabilmektedir. Yani hem List hem de LinkedList genel amaçlı koleksiyon sınıflarıdır denilebilir. Buna rağmen ne zaman List, ne zaman LinkedList sınıfı kullanılmalıdır sorusuna cevap verilmelidir.

Konu listelerden okuma olduğunda List'in daha hızlı olduğu ifade edilmektedir. Diğer yandan yapılan işlem, listelerden eleman silme veya araya eleman ekleme olduğunda ise LinkedList daha performanslıdır.

Benzer şekilde LinkedList sınıfıyla hazırlanan koleksiyonda, elemanları listenin ortasına bir yere ekleme işlemi oldukça hızlıdır. Böyle bir durumda, önceki elemanın Next referansı ve bir sonraki elemanın Previous referansı yeni eklenen elemanı gösterecek şekilde güncellen-

mektedir. List<T> ve ArrayList'de ise yeni bir eleman eklendiğinde takip eden bütün elemanların hareket ettirilmesi gerekmektedir. Bu ise performansı düşürecektir.

Düğüm eklemedeki bu hız farkından dolayı sağlanan avantaj, okuma işlemi sırasında dezavantaja dönüşebilmektedir. Çünkü istenilen elemanlara erişim liste üzerinde sırasıyla hareket edilerek sağlanmaktadır. Bu performans kaybına neden olmaktadır.

List'ten farklı olarak LinkedList nesnesi hazırlarken yapıcı metoda kapasite değeri parametre olarak verilemiyor. Çünkü LinkedList nesnesi bağlı liste olarak hazırladığı için kapasite özelliğine gerek yoktur. Yukarıdaki sayfalarda işaret edildiği üzere LinkedList nesnesi çok amaçlı olarak kullanılabildiği için eleman ekleme işlemini kullanım amacına göre yapmak gerek. Bu LinkedList nesnesi stack olarak kullanmak istediğimiz için eleman eklerken AddLast() metodundan yararlandık.

```
static void Main(string[] args)
{
 LinkedList<string> liste = new LinkedList<string>();
 liste.AddLast("Galatasaray");
 liste.AddLast("Trabzonspor");
 liste.AddLast("Beşiktaş");
 liste.AddLast("Fenerbahçe");
 Console.ReadKey();
}
```

Bu kodda eleman eklemeyi isteseydik **AddFirst**() metodu ile yapabilirdik. Stack olarak kullanmak üzere hazırladığımız LinkedList nesnesinin elemanların içeriğini elde edip kullanmak istiyorsanız Last() metodunu kullanabilirsiniz.

```
string eleman = liste.Last();
System.Console.WriteLine(eleman);
```

Bu satırlar işletilirse, Last() metoduyla LinkedList nesnesine en son eklenen bilgi (Fenerbahçe) ekrana yazılır. Last() metodu, Stack sınıfının Pop() metodundan farklı olarak okuduğu elemanı listeden silmez. Aşağıda yapıldığı gibi Last() metodundan hemen sonra **RemoveLast**() metodunu kullanarak aynı sonucu alabilirsiniz.

```
static void Main(string[] args)
{
 LinkedList<string> liste = new LinkedList<string>();
 liste.AddLast("Galatasaray");
 liste.AddLast("Trabzonspor");
 liste.AddLast("Beşiktaş");
 liste.AddLast("Fenerbahçe");
 string eleman = liste.Last();
 System.Console.WriteLine(eleman);
 liste.RemoveLast();
 eleman = liste.Last();
 System.Console.WriteLine(eleman);
 Console.ReadKey();
}
```

24. BÖLÜM

PATTERN MATCHING

C# 7.0 ile birlikte kullanım alanı oldukça geniş Pattern Matching (Desen Eşleştirme) adında bir yenilik programcılara sunuldu. Kullanım alanına bağlı olarak Pattern Matching'e farklı isimler veriliyor: Type Pattern, Constant Pattern, Var Pattern, Wildcard Pattern, Switch Pattern gibi. Bu pattern'ler zamanla programcıların bazı alışkanlıklarını değiştirmeye aday olduğu için önemseyip ayrıntılı bir şekilde inceleyip üzerine konuşup örnekler verdik.

En başta belirtelim ki, C# 7.0 ile kullanıma sunulan eşleştirme desenlerini yani Pattern Matching'leri kullanmazsanız, bunlara kayıtsız kalırsanız çok şeyi kaçırmış olmazsanız. Sizi etkileyen, söz konusu edilecek işlem için sadece birkaç satır daha uzun kod yazmak olur. Bize fikrimizi sorarsanız bu desenlere kayıtsız kalmayın deriz.



Önceleri Pattern Matching ile gelen yenilikleri ilgili konu dahilinde anlatmayı düşündük. Yani var'la ilgili pattern matching'i var anahtar kelimesi hakkında bilgi verirken, switch blokları ilgili pattern'den bu kitabın switch anahtar kelimesinin anlatıldığı bölümde söz edecektik. Sonra bu kararımızı değiştirip C# 7.0 ile gelen Pattern Matching'leri ayrı bir bölüm altında incelemeye karar verdik. Pattern matching'leri ayrı bir bölüm halinde inceleyince bazı tekrarlar kaçınılmaz oldu. Bu tekrarlar, hem unutulan konuların hatırlanması hem de ilk anlatıldığında anlaşılmamış konuların başka bir bakış açısıyla anlatıldığında anlaşılması ihtimal dahilinde olduğundan bizce faydalıdır.

is Operatörü ve Type Pattern

Bu bölümde C# 7.0 ile gelen patternler hakkında bilgi verirken kullanım alanlarına odaklanmayı tercih ettik. Yani yapmak istediğimiz işleme uygun pattern seçimi yaptık. İlk olarak "is" operatörü ile yapılan tip dönüştürmeye gelen yeniliklerden söz edeceğiz. Bunu yaparken ayrıca "as" ve "is" operatörleri ile yapılan tip kontrolü ve dönüştürme işlemlerini size hatırlatmış olacağız. Çünkü bu kitabın "Operatörler" adlı bölümünde "as" ve "is" operatörlerinin bütün kullanım şekillerinden söz edilip örnek verilmişti. C# 7.0 ile birlikte "is" operatörüne, Pattern Matching adı verilen yenilik sayesinde yeni beceriler kazandırıldı. Bu sayede kodun daha okunur ve kısa olması sağlandı.

Type Pattern adı verilip "is" operatörü ile kullanılan pattern'den söz etmek için bu kitabın "Operatörler" adlı bölümünde verilen kodu buraya aldık. **is** operatörünün işlevinin anlaşılması için hazırladığımız bu kodun işlevsel olması veya işlevini görebilmeniz için üzerinde çalıştığınız Windows Forms uygulamasının formuna daha önceden birkaç TextBox yerleştirmek gerekmektedir. Çünkü bu kod işletildiğinde forma yerleştirilen bütün TextBox'ların içeriklerini temizleyecektir. Formun üzerinde TextBox yokken bu kod işletilirse herhangi bir etkisi olmaz.

```
private void button1_Click_1(object sender, EventArgs e)
{
 int sayi = this.Controls.Count;
 for (int i = 0; i < sayi; i++)
```

```
        {
          if (this.Controls[i] is TextBox)
          {
            TextBox edit = this.Controls[i] as TextBox;
            edit.Clear();
          }
        }
      }
```

Bu kodda ilk olarak formdaki nesnelerin sayısını öğrendik. Forma tasarım veya çalışma anında yerleştirilen her nesne formun **Controls** koleksiyonuna eleman olarak eklenmektedir. Dolayısıyla bu koleksiyonun eleman sayısı formun üzerine yerleştirilmiş nesnelerin sayısı olmaktadır. Controls koleksiyonuyla ilgili olarak bilinmesi gereken önemli bir nokta var: Controls koleksiyonunda tutulan nesneler **Control** tipindedir.

Controls koleksiyonun **Count** özelliğine bakıp elemanların sayısını öğrendikten sonra sıra ile for döngüsü dahilinde Controls koleksiyonundaki bütün elemanların üzerinde ileri doğru hareket ederek dolaştık. Formun üzerindeki ilgili nesnenin TextBox tipine dönüşüme uygun olup olmadığını "is" operatörü ile öğrendikten sonra bu nesneyi dönüştürüp referansını TextBox tipindeki değişkene aktardık. En son olarak bu TextBox'ın içeriğini TextBox sınıfının **Clear()** metodu ile temizledik.

Şimdi bu kodu inceleyip gördüklerimizi maddeler halinde sıralayacağız: Bunu yaparken sizlerin bu kitabın Operatörler adlı bölümünde "is" ve "as" hakkında yazılanları okuduğunuzu varsayıyoruz.

- Önce formun üzerindeki nesnelerin sayısı Form sınıfının **Controls** koleksiyonun Count özelliğine bakılarak öğrenilmektedir.
- Count özelliğinin içeriğine bağlı olarak for döngüsü kurularak formun üzerindeki bütün nesneler tek tek inceleniyor. Formun üzerine yerleştirilen her nesnenin Form sınıfının Controls koleksiyonuna üye yapıldığını biliyorsunuz.
- Control koleksiyonundan okunan nesnenin TextBox tipine dönüşüme uygun olup olmadığı "is" operatörü ile araştırılmaktadır. Çünkü Form sınıfının Controls koleksiyonu Control tipindedir. Bu koleksiyon, TextBox dahil olmak üzere her türlü nesneyi eleman olarak tutabilmektedir.
- if bloğuna ait parantezin içinde "is" operatörü ile kontrol edilen nesne TextBox tipine dönüşüme uygun ise hemen orada TextBox tipinde bir değişken tanımlanıp Controls koleksiyonun o elemanı "as" operatörü ile TextBox tipinde dönüştürülüp tanımlanan değişkene aktarılıyor.
- En son olarak TextBox sınıfının **Clear()** metodu ile ilgili TextBox nesnesinin içeriği temizlenmektedir.

Burada yapılan dönüştürme işlemini "as" operatörü ile yapmak zorunda değilsiniz. Çünkü if bloğunun içinde iken Controls koleksiyonun söz konusu elemanın TextBox tipinde bir nesne içerdiğini ve TextBox tipine dönüşmeye uygun olduğunu biliyoruz. Buradaki amacımız "is" operatörüne C# 7.0 ile kazandırılan yeni becerilerden söz etmek olduğu için kontrolü "is" ile, dönüştürmeyi "as" ile yaptık.

Bu şekilde "is" ve "as" operatörlerinin nasıl birlikte kullanıldığını hatırladıktan sonra C# 7.0 ile gelen ve Pattern Matching adı verilen yeniliklerden yararlanıp yukarıda verdiğimiz kodu yeniden yazacağız.Aşağıda verilen kod işlev bakımından yukarıda verilip hakkında açıklama yapılan koddan farksızdır. Formun üzerindeki bütün TextBox'ların içeriğini temizlemek amacıyla hazırlanan bu kodu dikkatlice incelemenizi ve yukarıda verilen ilk hali ile karşılaştırmanızı istiyoruz.

```
private void button1_Click_1(object sender, EventArgs e)
{
 int sayi = this.Controls.Count;
 for (int i = 0; i < sayi; i++)
  {
   if (this.Controls[i] is TextBox edit)
     {
      edit.Clear();
     }
   }
 }
```

Karşılaştırma yaptığınızda ilk göreceğinizi sandığımız fark şudur: if bloğunda "as" operatörünün kullanıldığı satır kaldırılmış. Çünkü buna artık gerek yoktur, "is" operatörü hem kontrol işlemini hem de dönüştürmeyi tek başına yapabilmektedir. C# 7.0 ile gelen yeniliğin özü şu satırdadır;

```
if (this.Controls[i] is TextBox edit)
```

For döngüsü içinde kullandığımız bu satırda yapılan şudur:
Form sınıfının Controls koleksiyonun söz konusu elemanın içeriği "is" operatörü ile kontrol edilerek TextBox tipine dönüşüme uygun olup olmadığına bakılıyor. Controls koleksiyonun sıradaki elemanı yani Control tipindeki nesne TextBox tipine dönüşüme uygun değilse (Button veya ListBox tipinde olabilir) koleksiyondaki bir sonraki elemana geçiliyor. Kontrol edilen nesne TextBox tipine dönüşebilecekse, Controls koleksiyonun ilgili elemanından yola çıkılıp hemen orada TextBox tipinde bir nesne hazırlanıyor.

Burada bu nesnenin referansını tutacak değişkene "edit" adını verdik. TextBox tipinde nesne hazırlama işlemi otomatik olarak yapıldığı için yukarıda bu işlemi yaptığımız satırı sildik. Başka bir deyişle yukarıda verdiğimiz satırdaki "is" operatörü şu işlemleri gerçekleştirdi:

- Soluna yazılan nesnenin sağına yazılan nesneye dönüşüme uygun olup olmadığını araştırdı.
- Bu araştırmanın sonucu negatif ise yani Controls koleksiyonun ilgili sıradaki elemanı TextBox tipine dönüşüme uygun değilse if deyimi ile kurulan bloğa girilmeyip aynı araştırma Controls koleksiyonun bir sonraki elemanı için yapılır.
- is operatörü ile yapılan karşılaştırmanın sonucu true ise, yani Controls koleksiyonun o sırada kontrol edilen elemanı TextBox tipine dönüşüme uygunsa,

- if deyimine ait parantezin içine TextBox tipinden sonra, TextBox nesnesini tutabilecek bir değişken adı yazılmışsa,
- "is" operatörü tekrar devreye girer ve Controls koleksiyonun ilgili elemanın içeriğini TextBox tipine dönüştürüp referansını adı yazılan değişkene aktarır.

Farkındayız son cümle maddeler haline getirilip bölünmüş olsa bile oldukça uzun oldu. Şimdi bu cümlede anlatılmak istenileni farklı kelimelerle tekrar anlatacağız.

if deyimine ait parantezin içinde "is" operatörü ile yapılan karşılaştırmada bir tip eşlenmesi varsa yani "is" operatörünün soluna yazılan nesne ile sağına yazılan tip arasında bir eşleşme varsa is operatörünün pattern matching becerisi devreye girmektedir.

if deyimine ait parantezin içine bir değişkenin adını yazmamış olsaydık yani "is" operatörünü aşağıdaki gibi kullansaydık "is" operatörü kendisine C# 7.0 ile kazandırılan Type Pattern becerisini sergilemeyip sadece tip kontrolünü yapardı.

```
if (this.Controls[i] is TextBox)
```

Yukarıdaki sayfalarda Type Pattern'ler hakkında bilgi vermeyi pratik değeri olan bir işlem üzerinde anlatmayı tercih ettiğimiz için ayrıca Form sınıfının **Controls** koleksiyonundan söz etmemiz gerekti. Type Pattern konusunu kapatmayıp şimdi daha basit işlemlerde kullanmayı deneyeceğiz. Bu amaçla Microsoft firmasına ait sayfalardan yararlanıp aşağıdaki gibi kısıtlı özelliklere sahip 2 sınıf hazırladık. "Dikdortgen" ve "Kare" adını verdiğimiz bu sınıfların instance'larını kontrol edip ona göre işlem yapmak amacıyla (alanlarını bulmak) bir metot hazırlayacağız.

```
public class Kare
{
 public double Kenar { get; }
 public Kare(double uzunluk)
 {
  Kenar = uzunluk;
 }
}
public class Dikdortgen
{
 public double Uzunluk { get; }
 public double Genislik { get; }
 public Dikdortgen(double length, double height)
 {
  Uzunluk = length;
  Genislik = height;
 }
}
```

Bu metot **object** tipinde bir parametreye sahip olacak. Çünkü amacımız "is" operatöründen söz etmektir. Yani tip dönüştürmeye ihtiyacımız olsun istiyoruz. AlanBul() adını verdiğimiz bu metodu Main() metoduna sahip "Program" sınıfında hazırladık.

```
public static double AlanBul(object sekil)
{
 if (sekil is Kare)
 {
  Kare k = sekil as Kare;
  double alan = k.Kenar * k.Kenar;
  return alan;
 }
 if(sekil is Dikdortgen)
 {
  Dikdortgen d = sekil as Dikdortgen;
  double alan = d.Genislik * d.Uzunluk;
  return alan;
 }
 throw new ArgumentException(message: "şekil uygun değil");
}
```

Sizin için herhangi bir yenilik içermeyen bu metottaki ilk "if" ile, parametre olarak gönderilen nesnenin "Kare" tipine dönüşmeye uygun olup olmadığı "is" operatörü ile araştırılmaktadır. Bu araştırmanın sonucu true ise bir sonraki satırda "as" operatörü ile object tipindeki bu nesne "Kare" tipine dönüştürülmektedir. Devamında "Kare" nesnesinin alanı bulunup geriye gönderilmektedir.

AlanBul() metoduna gönderilen nesne "Kare" tipinde değilse bu kez 2. if'e ait parantezin içinde, gönderilen nesnenin "Dikdortgen" tipine dönüştürmeye uygun olup olmadığı araştırılıyor. "is" operatörü ile yapılan bu karşılaştırmanın sonucu true ise gönderilen nesne as operatörü ile Dikdortgen tipine dönüştürülmektedir.

AlanBul() metoduna parametre olarak gönderilen nesne Kare ve Dikdortgen tipine dönüştürülmeye uygun değilse throw anahtar kelimesi ile ortama **ArgumentException** nesnesi fırlatılmaktadır. Kod uzamasın diye bu Exception nesnesini yakalamak için hazırlık yapmadık.

Anlatılan şekilde AlanBul() adını verdiğimiz metodumuzu hazırladıktan sonra üzerinde çalıştığımız konsol uygulamasının Main() metodunu aşağıdaki gibi düzenledik.Main() metodunda önce Kare tipinde bir nesne hazırladık. Kare sınıfının yapıcı metodu int tipinde bir parametreye sahip olduğu için alanını bulmak istediğimiz karenin kenar uzunluğunu yapıcı metoda parametre olarak verdik.Devamında hazırladığımız Kare tipindeki nesneyi AlanBul() metoduna parametre olarak verdik.

```
static void Main(string[] args)
{
 Kare sekil1 = new Kare(10);
 double alan = AlanBul(sekil1);
 Console.WriteLine(alan);
}
```

İstenirse ayrıca "Dikdortgen" tipinde bir nesne hazırlanıp AlanBul() metoduna parametre olarak verilebilir. Burada asıl anlatmak istediğimiz AlanBul() metodunda "is" ve "as"

operatörleri ile yapılan dönüştürme işlemleridir. Bu metotta C# 7.0 ile gelen type pattern imkanından yararlanmak istemiş olsaydık aşağıdaki gibi düzenlerdik.

```
public static double AlanBul(object sekil)
{
 if (sekil is Kare k)
  {
   double alan = k.Kenar * k.Kenar;
   return alan;
  }
 if (sekil is Dikdortgen d)
  {
   double alan = d.Genislik * d.Uzunluk;
   return alan;
  }
 throw new ArgumentException(message: "şekil uygun değil");
}
```

Bu düzenleme sonucu aynı işlem için as operatörünü kullanmamız gerekmedi ve kodumuz 2 satır kısaldı. Burada AlanBul() adını verdiğimiz metodun type pattern'den yararlanılmış hali mi daha okunur yoksa as operatöründen yararlanılan hali mi? Bu soruya cevap vermek bizce kolay değil. Tahminimiz ise şudur: Programcılar kodun kısa olmasını sevdikleri için type pattern gibi yeniliklerin tutma ihtimali yüksektir.

Yukarıdaki sayfalarda type pattern'ler üzerine konuşurken .NET Framework ile gelen int, byte, string ve long gibi built-in tipleri kullanmadık. Type patternlerin bu tiplerle ilişkini göstermek için aşağıdaki gibi object tipinde bir parametreye sahip bir metot hazırladık.

```
public static void Kontrol(object nesne)
{
 if (nesne is byte)
  {
   Console.WriteLine("Nesnemiz byte tipindedir");
   byte b = (byte) nesne;
   Console.WriteLine(b);
  }
 if (nesne is int)
  {
    Console.WriteLine("Nesnemiz int tipindedir");
    int i = (int)nesne;
    Console.WriteLine(i);
  }
 if (nesne is string)
  {
   Console.WriteLine("Nesnemiz string tipindedir");
   string str = nesne as string;
   Console.WriteLine(str);
  }
}
```

Bu metotta sizin için yeni bir şey yoktur. Bu metotla ilgili olarak hatırlatmak istediğimiz önemli nokta şudur: String bilgi içeren object tipindeki nesneyi string'e dönüştürmek amacıyla as operatörünü kullanırken, byte ve int bilgi içeren nesneleri dönüştürürken casting operatöründen yararlandık. Çünkü "as" operatörünün value type desteği yoktur.

Kontrol() adını verdiğimiz bu metotta C# 7.0 ile gelen type pattern'lerden yararlanmak istemiş olsaydık aşağıdaki gibi düzenlerdik. Bu sayede kodumuz her if bloğunda bir satır kısalırdı.

```
public static void Kontrol(object nesne)
{
 if (nesne is byte b)
 {
  Console.WriteLine("Nesnemiz byte tipindedir");
  Console.WriteLine(b);
 }
 if (nesne is int i)
 {
  Console.WriteLine("Nesnemiz int tipindedir");
  Console.WriteLine(i);
 }
 if (nesne is string str)
 {
  Console.WriteLine("Nesnemiz string tipindedir");
  Console.WriteLine(str);
 }
}
```

Burada kullanılan type patternlerin nasıl işlevsel olduğu veya nasıl devreye girdiği belki henüz anlaşılmamıştır diye şu satırı mercek altına alacağız:

```
"if (nesne is byte b)"
```

Bu satırda if deyimine ait parantezlerdeki ilk 3 ifadenin işlevini Operatörler adlı bölümden biliyoruz. "is" operatörü, soluna yazılan nesne veya değişkeni sağına yazılan tip ile karşılaştırmaktadır. Karşılaştırmanın sonucuna göre geriye bool bir bilgi göndermektedir. C# 7.0 ile birlikte "is" operatörü, soluna ve sağına yazılanları bir desen olarak değerlendirmektedir.

Bu desende yani "is" operatörünün soluna yazılanla sağına yazılan arasında bir eşleşme varsa, operatörün sağına yazılan tipte bir nesne hazırlanmaktadır. Parantezin arasına böyle değişken adı yazılmışsa o sırada (bizim örnekte byte 'b' var) bir değişken tanımlanıp hazırlanan nesnenin referansı bu değişkene aktarılmaktadır. Özetlemek gerekirse, "if (nesne is byte b)" satırı ile;

- Burada adı "nesne" olan nesne is operatörü tarafından byte tipi ile karşılaştırılmaktadır.
- Object tipindeki nesne, byte tipine dönüşüme uygunsa is operatörü bu dönüşümü yapıp byte tipinde bir değişken hazırlamaktadır.

- Programcı değişken adı olarak kullanmak üzere is operatörünün sağındaki tipin sağına bir değişken adı yazmışsa, hazırlanan nesne bu değişkene aktarılmaktadır.

Yukarıda hazırlayıp Kontrol() adını verdiğimiz metodu kullanmak istediğinizde aşağıdaki gibi kod yazılabilir.,

```
public static void Main()
{
 object obj1 = 127;
 object obj2 = 1907;
 object obj3 = "Adana";
 Kontrol(obj1);
 Kontrol(obj2);
 Kontrol(obj3);
}
```

Constant Pattern'ler

Bu bölümün başında ifade edildiği gibi C# 7.0 ile kullanıma sunulan ve Pattern Matching adı verilen yenilik aslında birden fazla yeniliğe verilen ortak bir addır. Yukarıdaki sayfalarda tipler üzerinde işlem ve dönüştürme yaparken yararlanılan Type Pattern'ler üzerinde durduk.

Şimdi sabit bilgiler üzerinde işlem yapılırken kullanabileceğiniz ve **Constant Pattern** adı verilen pattern'ler üzerinde duracağız. Bu pattern'de başrolde yine "is" operatörü var. Bu pattern, sabit bir bilgi is operatörü ile kontrol edildiğinde işlevsel olmaktadır. Bu amaçla aşağıdaki gibi bir kod yazdık.

```
class Program
{
 static void Main(string[] args)
 {
  object takim = "Adana Demirspor";
  if (takim is null)
  {
   Console.WriteLine("takim değişkeni null");
  }
  else
  {
   Console.WriteLine(takim);
  }
 }
}
```

İlk bakışta herhangi bir farklılığı olmayan bu kodu kısa bir süre inceledikten sonra 7.0 öncesi bir C# derleyicisi ile derlemeyi deneyin. Bu metni hazırlarken elimizde Visual Studio 2008 kurulu bir bilgisayar olduğundan bu kodu C# 3.0 ile derlemek isteyince hata mesajı aldık. Çünkü C# dilinin 7.0'dan önceki versiyonlarında "is" operatörünü bu şekilde kullanmak mümkün değildi.

Bu kodda "if" deyimine ait parantezin içinde is operatörü ile "takim" değişkenin null olup olmadığı araştırılmaktadır. Bu kodu aşağıdaki gibi düzenlemiş olsaydık yukarıda üzerinde çalıştığımız type pattern'i kullanmış olurduk.

```
static void Main(string[] args)
{
 object takim = "Adana Demirspor";
 if (takim is string str)
 {
  Console.WriteLine(str);
 }
}
```

Bu başlık altında constant pattern hakkında bilgi vermek istediğimiz için aşağıdaki gibi birkaç satır yazdık. Bu kodun C# 7.0 öncesi versiyonlarda derlenmeyeceğini biliyorsunuz. Önceki C# versiyonlarında "is" operatörün sağına sadece tipleri yazıp kontrol edebiliyorduk.

```
class Program
{
 static void Main(string[] args)
  {
   object sehir = "Adana";
   if (sehir is "Adana")
    {
      Console.WriteLine("Adana'da sıcağı yaşıyorsunuz");
    }
  }
}
```

Burada içeriğini "is" operatörü ile kontrol ettiğimiz object tipindeki değişkenin içeriği baştan bellidir. Konu üzerinde düşünmenizi sağlamak için bu kodu aşağıdaki gibi düzenledik.

```
static void Main(string[] args)
{
 object sehir = Console.ReadLine();
 if (sehir is "01")
 {
  Console.WriteLine("Adana'nın plakası 01");
 }
 if (sehir is "Güney")
  {
   Console.WriteLine("Adana Güney Bölgesindedir");
  }
 }
}
```

Var Pattern

Bu kitabın var anahtar kelimesi hakkında bilgi verilen sayfalarında ifade edildiği gibi var ile tanımlanan değişkenlerin tipi derleme sırasında belirlenmektedir. Değişkene ilk değer olarak hangi tipte bilgi aktarıldıysa derleyici değişkenin tipini ona göre seçmektedir.

Bu konuda Değişkenler adlı bölümde bilgi verilmesine rağmen burada var anahtar kelimesi ile nasıl değişken tanımlandığından kısaca söz edeceğiz. Örneğin aşağıdaki gibi bir satır yazıldığında burada "str" adını verdiğimiz değişkenin tipi string olur. Çünkü ilk değer olarak string bilgi aktardık.

```
var str = "Fenerbahçe";
```

Daha sonra bu değişkene string'den farklı tipte bilgi aktarılmak istendiğinde hata meydana gelir. Var anahtar kelimesi ile tanımlanmış bir değişkenin hangi tipte bilgi içerdiğini öğrenmek istiyorsanız **GetType()** metodundan yararlanabilirsiniz. **var** anahtar kelimesi ile değişken tanımlama işlemi aşağıdaki gibi yapıldığında değişkenin tipi Int32 olarak ayarlanır.

```
var sayi = 255;
textBox1.Text = sayi.GetType().ToString();
```

Var anahtar kelimesi ile tanımlanan değişkenlere **Implicitly Typed Local Variables** adı verilmektedir. Dolayısıyla var anahtar kelimesi class bloğunda kullanılamaz. Var anahtar kelimesi hakkında bu girişi yaptıktan sonra gelelim var ile kullanılan pattern'e. Önce en basit şekliyle bu pattern'i kullanacağız. Bu amaçla aşağıda verilen kodu yazdık.

```
public class Program
{
 static void Main(string[] args)
  {
   var oyuncu = "Ömer Onan";
   bool sonuc = (oyuncu is var str);
   Type tip = str.GetType();
   Console.WriteLine($"{tip.Name} = {str}");
  }
}
```

Bu kodda ilk olarak var anahtar kelimesini kullanarak "oyuncu" adında bir değişken tanımlayıp string bilgi aktardık. Devamında Pattern Matching'in olmazsa olmazı "is" operatörünü kullandık. Bu operatörü şimdiye kadar hep if deyimine ait parantezin içinde kullanmıştık. Bu kez farklılık olsun diye veya bu operatörü kullanırken if'ten yararlanmanın şart olmadığını göstermek için bool sonuc üreten bu operatörü parantezlerin içine aldık. Parantezlerin içinde üretilen bool değeri bir değişkene aktardık. Burada tanımlanan bool değişkene her şartta true aktarılır. Burada asıl üzerinde durmamız gereken parantezin içine yazılanlardır:

```
bool sonuc = (oyuncu is var str);
```

Burada düz mantıkla anlaşılmayan değişik bir durum var. Derleyici burada "is" operatörünün soluna yazılan değişkenin içeriğinden yola çıkarak var anahtar kelimesi ile yeni bir değişken tanımlıyor. Dolayısıyla burada "str" adını verdiğimiz değişkene ilk değer olarak string bilgi aktarılıp string tipte olması sağlanır. Devamında GetType() metodu ile bu değişkenin tipini araştırdık. Aşağıda verilen ekran görüntüsünü bu kodu çalıştırdıktan sonra aldık.



Her ne kadar var pattern'i anlatmak için hazırladığımız kodda string bilgi aktardığımız değişkeni var ile tanımlamış olsak bile buna gerek yoktur. Yani bu değişkeni aşağıda yapıldığı gibi tanımlasaydık değişen bir şey olmazdı.

```
static void Main(string[] args)
{
 string oyuncu = "Ömer Onan";
 bool sonuc = (oyuncu is var str);
 Type tip = str.GetType();
 Console.WriteLine($"{tip.Name} = {str}");
}
```

Hazırladığımız bu kodda var pattern'i işini yapmış olsa bile sanki kullanılmasaydı da olurdu gibi duruyor. Bu nedenle ikinci bir örnek hazırlayacağız. Örneğin işlevsel olmasını sağlamak için önce bir metot hazırlayacağız. Bu metoda değişik tipte bilgiler gönderebilmek için **object** tipinde bir parametreye sahip olmasını sağlayacağız.

```
static void TipBul(object obj)
{
 if (obj is var str)
  {
   Type tip = str.GetType();
   Console.WriteLine($" {tip.Name} = {str}");
  }
}
```

Object tipinde bir parametreye sahip bu void metot incelenirse parametre olarak verilen bilginin is operatörüne parametre olarak verildiği görülür. Operatörün sağında var anahtar kelimesi olduğu için derleyici bunu "var pattern" olarak değerlendirecektir. Dolayısıyla object değişkenin içeriğine bakarak burada "str" adını verdiğimiz değişken için tip seçimini yapacaktır. Gerisi zaten bildiğiniz şeyler. Bu metodu işletmek için üzerinde çalıştığımız uygulamasının Main() metodunu aşağıdaki gibi düzenledik.

```
static void Main(string[] args)
{
 string oyuncu = "Ömer Onan";
 TipBul(oyuncu);
}
```

Bu kod işletildiğinde TipBul() adını verdiğimiz metottaki var pattern'î sayesinde string tipte tipte bir değişken tanımlanır. Bu metodun işlevini net olarak göstermek için TipBul() metodunu farklı tipte bilgilerle tekrar işlettik.

```
static void Main(string[] args)
{
 string oyuncu = "Ömer Onan";
 int yil = 2017;
 double pi = 3.14;
 DateTime tarih = DateTime.Now;
 TipBul(oyuncu);
 TipBul(yil);
 TipBul(pi);
 TipBul(tarih);
}
```

Bu kodu aşağıda verilen ekran görüntüsü ile birlikte değerlendirirseniz var pattern'in işlevi net olarak görülebilir.



Switch Pattern

İlgili bölümde switch –case bloklarıyla ilgili sıklıkla karşılaşılan durumlardan söz edildi. Şimdi C# 7.0 ile gelen ve **Pattern Matching** adı verilen yeniliğin switch –case bloklarına olan etkisinden söz edeceğiz.

C# 7.0 ile programcılara Switch deyimleriyle birlikte pattern kullanma imkanı sonuldu. Bu sayede blok dahilinde istenen tipler üzerinden kontrol yapılabilinmektedir. Örneğin bir enum üzerinden kontrol yapılabilinir. Önceliği switch deyimi ile kullanılan pattern'lerin en basit haline verip aşağıdaki gibi bir kod yazdık.

```
class Program
{
 static void Main(string[] args)
 {
  object secim = null;
  Console.Write("Gün Seçimiziniz :");
  secim = Convert.ToInt32(Console.ReadLine());
  switch (secim)
  {
  }
 }
}
```

Main() metodu içinde hazırladığımız kodda switch anahtar kelimesine yer vermiş olsak bile henüz switch'le ilgili bir işlem yapmadık. Bu kodda ilk dikkat etmenizi istediğimiz object tipinde değişken tanımlayıp switch'te kontrol değişkeni olarak kullanmış olmamızdır. Switch

blokları içinde integer bilgilerle ilgili bir pattern hazırlamak istediğimiz için konsoldan girilen bilgiyi burada "secim" adını verdiğimiz değişkene aktarmadan Convert sınıfının ToInt32() metodundan yararlanıp int tipine dönüştürdük. Konsoldan girilen bilginin int olmasını bir nevi sağlamış olsak bile kontrol değişkenimiz yine **object** tipindedir.

Tabi burada konsoldan int tipine uygun bilgi girilmezse hata meydana gelir. Şimdi bu koda ekleme yapıp case blokları hazırlayacağız. Dikkatinizi hazırladığımız kodun yeni halinde case anahtar kelimesi ile birlikte kullanılan pattern'e çekmek istiyoruz.

```
class Program
{
 static void Main(string[] args)
  {
   object secim = null;
   Console.Write("Gün Seçiminiz :");
   secim = Convert.ToInt32(Console.ReadLine());
   switch (secim)
   {
    case int gun when gun == 1:
      Console.WriteLine("Pazartesi");
      break;
    case int gun when gun == 2:
      Console.WriteLine("Salı");
      break;
    case int gun when gun == 3:
      Console.WriteLine("Çarşamba");
      break;
   }
  }
}
```

Bu kodda asıl anlatmak istediğimiz switch-case bloklarında kullanılabilinen patterm'ler olduğu için pattern içeren satırlardan birisini buraya aldık. Çünkü bu satırı mercek altına alacağız.

```
case int gun when gun == 1:
```

Bu satırda ve öncesinde yapılan işlemler üzerinde maddeler halinde konuşacağız:

- Switch için kullandığımız kontrol değişkeni burada object tipinde olsa bile bu şart değildir. Tipi object seçmemizin nedeni pattern sayesinde switch'lerde her türlü bilginin, nesnenin kullanılabileceğine vurgu yapmaktır.
- case bloklarında kontrol değişkenin içeriğinin int olduğunu düşündüğümüz için case'ten hemen sonra int anahtar kelimesine yer verdik. Tabi bu anahtar kelimenin hemen yanına bir değişken adı yazdık.
- Visual Studio ile kod yazarken imlecin "case int gun" sonunda olduğunu varsayalım. Çünkü bu anı anlatacağız. Derleyici tam bu noktada int tipinde bir değişken tanımlar ve kontrol değişkenin (burada secim) güncel içeriğini bu değişkene

aktarır. Tabi kontrol değişkenin içeriğinde int'e uygun bilgi olmasaydı hata meydana gelirdi.

- Gerçekte pattern burada bitiyor. Yukarıda verdiğimiz kodda when anahtar kelimesine yer vermiş olsak bile when'in kullanılması şart değildir. Tabi bu durumda her şartta ilk case geçerli olacağı için bu kodu yazmak anlamlı olmazdı.
- Kontrol değişkenin içeriği burada "gun" adını verdiğimiz değişkene aktarıldıktan sonra when anahtar kelimesi ile bu değişkenin içeriği kontrol edilmektedir. Eğer bu kontrol işleminin sonucu true ise o case bloğundaki satırlar işletilmektedir.
- when ile yapılan karşılaştırma false olursa bir sonraki case bloğuna geçilir ve switch pattern tekrar devreye girer.
- Bu switch pattern'de when deyimi ile birlikte "==" operatörünü kullandık. Sizler bool sonuc üreten başka operatörleri kullanabilir veya karşılaştırmalar yapabilirsiniz.

Şimdiye kadar switch pattern'ler hakkında yazılanlara bakılırsa, gayet esnek switch-case blokları hazırlamak mümkündür. Switch pattern'lere ikinci bir örnek olması için yukarıda "var pattern'leri anlatırken hazırlayıp TipBul() adını verdiğimiz metodu switch pattern'lere uyarlayacağız. Bu metodun yeni hali aşağıda verildi.

```
static void TipBul(object obj)
{
 switch(obj)
 {
  case string str:
    Type tip = str.GetType();
    Console.WriteLine($" {tip.Name} = {str}");
    break;
  case int i:
    tip = i.GetType();
    Console.WriteLine($" {tip.Name} = {i}");
    break;
  case double d:
    tip = d.GetType();
    Console.WriteLine($" {tip.Name} = {d}");
    break;
  case DateTime simdi:
    tip = simdi.GetType();
    Console.WriteLine($" {tip.Name} = {simdi}");
    break;
 }
}
```

Bu metodun var pattern'leri anlatmak amacıyla hazırladığımız hali aynı işlemi yapmasına rağmen çok daha kısaydı. Buradaki amacımız en basit şekli ile switch pattern'i anlatmak olduğu için bu işlem için switch case uygun mu değil mi bakmadık.

Burada hazırladığımız TipBull() metodumuz ve dolayısıyla switch'te kullandığımız kontrol değişkeni object tipindedir. Bu metotta hazırladığımız case bloklarına baktığınızda yukarıda-

kinden farklı olarak her case'te farklı bir pattern'in kullanıldığını ve **when** anahtar kelimesinden yararlanılmadığını fark edersiniz. Örnek olması için aşağıda verilen case'de kullanılan patterm üzerine biraz konuşacağız.

```
"case string str:"
```

Kodun işletimi bu satıra geldiğinde derleyici switch anahtar kelimesine ait parantezin içine yazılan object değişkenin içeriğinin string tipte olup olmadığını kontrol eder. Bu değişken string tipte bilgi içeriyorsa pattern'de bir işleşme olduğuna karar verilip burada "str" adını verdiğimiz değişken tanımlanır ve kontrol değişkenin içeriği bu değişkene aktarılır.

Bu case'te pattern eşleşmesinden sonra sadece değişken hazırlamakla yetindik. İsteseydik when anahtar kelimesine yer verip ayrıca case bloğuna yazılan satırların işletilmesini bir veya birden fazla koşula bağlayabilirdik. When anahtar kelimesini nasıl kullandığımızı aşağıda görebilirsiniz.

```
switch(obj)
{
 case string str when str.Length > 15:
   Type tip = str.GetType();
   Console.WriteLine($" {tip.Name} = {str}");
   break;
```

Şimdi sırada switch pattern'lerle birlikte enum tiplerinden yararlanmak var. Bu amaçla üzerinde çalıştığımız kod dosyasında aşağıdaki gibi bir enun tipi hazırladık.

```
public enum Renkler
{
 Beyaz = 1,
 Yeşil = 2,
 Mavi = 3,
 Sarı = 4,
 Siyah = 5
}
```

Bu şekilde enum hazırladıktan sonra üzerinde çalıştığımız konsol uygulamasının Main() metodunu aşağıdaki gibi düzenleyerek bu enum'dan yola çıkılarak rastgele bir rengin seçilmesini sağladık.

```
static void Main(string[] args)
{
 Random rastgele = new Random();
 int sira = rastgele.Next(1, 5);
 Renkler renk = (Renkler)sira;
}
```

Uygulamanın her çalıştırılmasında burada "renk" adını verdiğimiz "Renkler" tipindeki değişkene farklı bir bilgi aktarılabilir.Şimdi bu koda ekleme yapıp Renkler tipindeki

değişkeni switch anahtar kelimesine parametre olarak vereceğiz. Bu sayede enum tiplerinin switch pattern'lerle nasıl kullanıldığını görmüş olacağız.

```
static void Main(string[] args)
{
 Random rastgele = new Random();
 int sira = rastgele.Next(1, 5);
 Renkler renk = (Renkler)sira;
 switch (renk)
 {
  case Renkler.Beyaz:
      Console.WriteLine("Bahtına beyaz renk çıktı");
      break;
   case Renkler.Mavi:
      Console.WriteLine("Bahtına beyaz çıktı");
      break;
   case Renkler.Sarı:
      Console.WriteLine("Bahtına sari çıktı");
      break;
      Console.ReadLine();
 }
}
```

Bu switch pattern örneğinde switch anahtar kelimesine parametre olarak verilen enum bilgisi case'den sonra yazılan ilk enum bilgisi ile karşılaştırılır. Bir eşleşme varsa o case bloğundaki satırlar işletilir. Aynı karşılaştırma diğer case blokları için de yapılır.

25. BÖLÜM

LINQ SORGULARINA HAZIRLIK

C# 3.0 ile kullanıma sunulan LINQ sorgularına altyapı olması için C# diline çck sayıda yenilik kazandırılmıştı. Bu bölümde sizleri LINQ'e hazırlayacak konuların üzerinde durulacaktır. Her ne kadar bu bölümde sözü edilecek anahtar kelimeler, yenilikler vs LINQ'e destek için C# diline kazandırılmış olsalar bile LINQ harici kullanımları önemlidir.

var Anahtar Kelimesi İle Değişken Tanımlamak

C# 3.0'dan itibaren **var** anahtar kelimesi ile değişkenin tipi işaret edilmeden tanımlama yapma imkanı programcılara sunuldu. Var anahtar kelimesi ile tanımlanan değişkenlere **Implicitly Typed Local Variables** adı verilmektedir. Var ile yapılan değişken tanımlama işlemine Tip Tahmini (Type Inference) adı verilmektedir. Var anahtar kelimesi LINQ ile yapılan sorgularda sıklıkla kullanıldığı için önemlidir.

var anahtar kelimesi ile tanımlanan değişkenlerin geçerlilik alanın, içinde tanımlandıkları metot ile sınırlı olduklarını bilmek gerekir. Yani var anahtar kelimesiyle class bloğu içinde değişken tanımlanamaz. Bunu, var anahtar kelimesi ile tanımlanan değişkenlere verilen addan anlamak mümkündür.

Normalde C# kodu dahilinde değişken tanımlanmak istendiği zaman önce değişkenin tipi, sonra değişkene verilmek istenen ad yazılmaktadır. var anahtar kelimesi ile tanımlanan değişkenin tipi, değişkene aktarılan bilgiye göre derleyici tarafından derleme anında seçilmektedir. **var** ile değişkenlerin nasıl tanımlandığını adım adım anlatmak üzere önce aşağıdaki gibi bir satır yazmakla yetindik.

```
private void button1_Click(object sender, EventArgs e)
{
 var str;  // hata meydana gelir.
}
```

Bu satırda **var** anahtar kelimesi ile tipi henüz belli olmayan bir değişken tanımladık. Bu sırada proje çalıştırılırsa bu satır hataya neden olur. Çünkü derleme sırasında C# derleyicisi bu değişkene aktarılmak istenen bilginin tipini bilmek ister. Bu nedenle "var" anahtar kelimesi ile tanımlanan değişkene metot dahilinde hangi tipte bilgi aktarılması düşünülüyorsa o tipte bir başlangıç değeri aktarmak gerekiyor.

Yukarıda var anahtar kelimesi ile tanımladığımız değişkenin hataya neden olmaması için ilk değer olarak boşluk aktardık.

```
private void button1_Click(object sender, EventArgs e)
{
 var str = "";
}
```

Bu andan itibaren bu değişken string tipte bir değişkendir ve string'den farklı tipte bilgilerin aktarılmasına izin verilmez. Bu nedenle aşağıdaki gibi bir satır hataya neden olur. Çünkü C#

derleyicisi "var" anahtar kelimesi ile tanımlanan değişkenin tipinin "ilk" aktarılan bilgi ile aynı olmasını sağlamakta ve geri planda ona göre IL kodu hazırlamaktadır.

```
private void button1_Click(object sender, EventArgs e)
{
 var str = "";
 str = 2017; // hata meydana gelir
}
```

Burada var deyimi ile tanımladığımız değişkene ilk değer olarak boşluk atadığımız için derleyici bu değişkenin tipini string olarak seçti. Dolayısıyla string tipindeki değişkene int bilginin aktarılmasına derleyici izin vermez. Bu değişkene sonradan string bilgi aktarmış olsaydık hata meydana gelmezdi.

Var anahtar kelimesi ile tanımlanan değişkenlerin tipi derleme sırasında belirlenip ona göre IL kodu hazırlandığı için object değişkenlerdeki gibi performans kaybı söz konusu değildir. Başka bir deyişle "var" bir değişken tipi olmayıp değişken tanımlama şeklidir. Başka bir anlatımla, var anahtar kelimesiyle değişken tanımlamakla derleyiciye diyoruz ki, tanımlamak istediğim değişkenin tipi konusunda şu anda net bilgiye sahip değilim, aktarılacak bilgiye bakarak derleme sırasında sen tahmin et. Derleyici bu tahmini doğru olarak yapmaktadır.

Var anahtar kelimesi ile tanımlanmış bir değişkenin hangi tipte bilgi içerdiğini öğrenmek istiyorsanız **GetType()** metodundan yararlanabilirsiniz. Bu metodun nasıl kullanıldığını aşağıda görebilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 var str = "Fenerbahçe";
 textBox1.Text = str.GetType().ToString();
}
```

Bu kod işletildiğinde TextBox'a "System.String" yazılır. **var** anahtar kelimesi ile tanımlanan değişkenlerin özelliklerini anlatmaya devam etmek için bu kodu aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
 var sayi = 255;
 textBox1.Text = sayi.GetType().ToString();
}
```

Var anahtar kelimesi ile tanımlanan bu değişkene, **byte** tipindeki değişkenler tarafından saklanabilinecek kadar küçük bir değer aktarılmasına rağmen C# derleyicisi bu değişkenin tipini **Int32** olarak seçer.

var ile tanımlanan değişkene aşağıdaki gibi kesirli bir sayı ilk değer olarak aktarılırsa değişkenin tipi derleme sırasında **Single** olarak seçilir. Burada "number" adını verdiğimiz değişkene aktarılan ondalıklı sayının sonuna "f" eklememiş olsaydık değişkenin tipinin **Double** olduğuna karar verilirdi.

```
private void button1_Click(object sender, EventArgs e)
{
 var number = 0.18f;
 textBox1.Text=number.GetType().ToString();
}
```

Yukarıda ifade edildiği üzere var anahtar kelimesi ile tanımlanan değişkene sayısal bir bilgi aktarıldığında değişkenin tipi int olarak ayarlanmaktadır. Var anahtar kelimesi ile tanımlanan değişkene int tipinin sınırlarını aşacak büyüklükte bir değer aktarılırsa değişkenin tipi "uint" yani işaretsiz int olarak ayarlanır.

```
private void button1_Click(object sender, EventArgs e)
{
 var number = 2147483649;
 textBox1.Text = number.GetType().ToString();
}
```

Burada "number" adını verdiğimiz değişkene daha büyük bir değer aktarsaydık değişken için seçilen tip long olurdu. Derleyicinin tip seçimine müdahale etmek istiyorsanız "l" gibi "sonek" karakterlerden yararlanabilirsiniz. Aşağıda verdiğimiz değişken tanımlama örneğinde "number" değişkenin tipi long olur.

```
private void button1_Click(object sender, EventArgs e)
{
 var number = 255l;
 textBox1.Text = number.GetType().ToString();
}
```

Var anahtar kelimesi ile tanımladığınız değişkene ilk değer olarak tek bir karakteri aktararak değişkenin tipinin derleme sırasında **char** olarak seçilmesini sağlayabilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 var kar = 'A'; // veya kar = (char)65;
 textBox1.Text = kar.GetType().ToString();
}
```

Var anahtar kelimesi ile tanımlanan değişkenlere ilk değer olarak **null** aktarılamaz. Mevcut bir değişkenin içeriği, var ile tanımlanmak istenen değişkene ilk değer olarak aktarılabilinir. Bu nedenle aşağıda verilen değişken tanımlama işlemi doğrudur.

```
string takim = "Fenerbahçe";
var str = takim;
textBox1.Text=str.GetType().ToString(); //TextBox'a System.String yazılır
```

Şimdiye kadar var anahtar kelimesi hakkında bilgi verirken hep .NET Framework ile gelen hazır tipleri kullandık. Gerçekte var anahtar kelimesi, programcının hazırladığı tiplerin aktarılacağı değişkenler tanımlanmak istendiğinde işlevsel olmaktadır. Bu konu havada kalmasın diye üzerinde çalıştığımız uygulamadaki "Form1" sınıfının hazırlandığı kod dosyası içinde aşağıdaki gibi bir sınıf hazırladık.

```
class Takim
{
 public string takim_adi;
 public string baskan;
 public int kurulus;
}
```

Devamında bu sınıfın örneğini alıp var anahtar kelimesi ile tanımladığımız değişkene aktardık. Burada "t" adını verdiğimiz değişkene "Takim" tipinde bilgi aktardığımız için değişkenin tipi "Takim" olarak ayarlanır.

```
private void button1_Click(object sender, EventArgs e)
{
 var t = new Takim();
 t.takim_adi = "Fenerbahçe";
 t.baskan = "Aziz Yıldırım";
 t.kurulus = 1907;
}
```

Her ne kadar yukarıda birkaç sayfa boyunca **var** anahtar kelimesi ile değişkenlerin nasıl tanımlandığı anlatılmış olunsa bile var anahtar kelimesi ile değişken tanımlama işlemine daha çok isimsiz tipler hazırlandığı zaman gerek duyulmaktadır. C# 3.0 ile gelen isimsiz tipler ise daha çok LINQ tekniği ile sorgulama yapılırken kullanılmaktadır. Bu kitapta LINQ ile sorgulamaların nasıl yapıldığı bölümden önce LINQ için altyapı işlevi gören bütün kavram ve C# özelliklerinden söz edilecektir. LINQ sorgulama tekniğinin asıl veritabanları üzerinde işlem yapılmak istendiğinde işlevsel olduğunu bilmekte fayda vardır.

İsimsiz Tipler – Anonymous Types

Anonymous Types yani isimsiz tipler üzerinde konuşmaya başlamadan önce yukarıda hakkında konuşulan "var" anahtar kelimesini hatırlamak gerekiyor. **var** anahtar kelimesi ile tanımlanan değişkene hangi tipte bilgi aktarılırsa derleyici geri planda değişkene aktarılan bilginin tipine göre değişkeni derleme sırasında hazırlamaktadır. Örneğin derleme sırasında, aşağıda verilen koddaki ilk değişkenin **string**, ikincinin ise **int** tipinde olduğuna hükmedilir.

```
private void Form1_Click(object sender, EventArgs e)
{
 var takim = "Fenerbahçe";
 var yil = 2017;
}
```

Bu kodu kısaca yorumlamak gerekirse, C# derleyicisi "yil" adı verilen değişkenin System.Int32 yapısından (struct) yola çıkılarak hazırlandığını varsayar ve System.Int32 yapısının örneğini alır. Başka bir anlatımla bu kodda C# derleyicisine dedik ki, "elimdeki bu bilgiye uygun bir tipi sen seç". Ancak bazen tanımlamak istediğimiz değişken veya hazırlamak istediğimiz nesneye uygun tip veya class elinizde olmayabilir. Bu durumda programcı gerek duyduğu (genellikle sadece alanlara sahip) class'ı hazırlayıp kullanır.



Daha önce C# hakkında az çok bilgi edinmiş birisi isimsiz tiplerin anlatıldığı bu sayfalara şöyle bir baktığında konuyu gereksiz yere uzattığımızı düşünebilir. Bu uzatmanın nedeni şudur: LINQ ile sorgulama yapabilmek için dile kazandırılan yenilikler iyice kavramadan LINQ'i ve ilgili konuları anlamak zordur.

Sonradan gerek duyulan class veya tipe, geliştirilen uygulamanın başka yerlerinde gerek duyulması ihtimali düşükse, tek kullanımlık class hazırlamak mantıklıdır. Tek kullanımlık class'ları hazırlayıp sonra da unutmak programcı için zaman kaybıdır.

C# 3.0 sürümüyle birlikte basit ve tek kullanımlık class'ların hazırlanmasında programcılara sunulan isimsiz tipler (Anonymous Types) sayesinde kolaylık sağlandı. İsimsiz tiplerin nasıl hazırlanıp kullanıldığını anlatmak amacıyla aşağıdaki gibi bir class hazırladık.

```
public class Rehber
{
 public string Firma;
 public string Telefon;
 public Rehber(string par1, string par2)
 {
  Firma = par1;
  Telefon = par2;
 }
}
```

Bu şekilde sadece 2 public alana sahip class hazırlamak yerine **Struct** hazırlamak daha anlamlı olabilirdi. Şimdi bu class'ın örneğini alırken işletilen yapıcı metot aracılığıyla nesnenin alanlarına bilgi aktaracağız.

```
private void Form1_Click(object sender, EventArgs e)
{
 Rehber kayit = new Rehber("Seçkin Yayıncılık", "0-312-435 3030");
 textBox1.Text = kayit.Firma;
 textBox2.Text = kayit.Telefon;
}
```

Burada sizin için yeni bir şey yoktur. Sınıfımızın yapıcı metodu string tipte 2 parametreye sahiptir. Nesne hazırlandığında bu parametrelerin içerikleri "Firma" ve "Telefon" alanlarına aktarılmaktadır. Burada hazırlayıp "Rehber" adını verdiğimiz sınıfımızın ayrıca varsayılan yapıcı metodunu hazırlamış olsaydık alanlara bilgi aktarmayı nesneyi hazırladıktan sonra yapabilirdik. Programcı olarak C# 3.0 ile desteklenen Object Initializer tekniğini kullanma alışkanlığınız varsa yukarıda yaptığımız nesne hazırlama işlemini aşağıdaki gibi yapabilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 Rehber kayit = new Rehber()
              {Firma="Seçkin Yayıncılık", Telefon="0-312-4353030"};
 textBox1.Text = kayit.Firma;
 textBox2.Text = kayit.Telefon;
}
```

Burada yapılan şudur: Mevcut bir class'ın örneği hazırlanmakta ve bu sırada alanlara ilk değerler aktarılmaktır. Bu yapılırken süslü parantezlerin arasında, olmayan bir alana bilgi aktarmaya çalışılırsa veya alana uygun bilgi aktarılmazsa kod derlenemez. Nesne hazırlarken eşitliğin sol tarafında tip adı yerine "var" anahtar kelimesini kullanabileceğinizi biliyorsunuz.

Şimdi yukarıda yaptıklarımızı özetleyelim; sadece 2 public alana sahip bir class hazırlayıp uygulamada sadece bir metodun içinde kullandık. Dolayısıyla bu class'la işimiz bitti, unutup gideceğiz. C# derleyicisi bu şekilde tek kullanımlık class'ları kullanacağınız yerde yani ilgili metodun içinde isimsiz olarak tanımlamanıza izin vermektedir. Tabi metodun içinde tanımlanan class'ları metodun dışında kullanamayız.

Şimdi "Rehber" adını verdiğimiz class'ın adını silip ondan sonra "var" anahtar kelimesi ile string tipte 2 alana sahip bir nesne hazırlayacağız. Bu amaçla hazırladığımız kod aşağıdadır.

```
private void Form1_Click(object sender, EventArgs e)
{
 var kayit = new
  {
     Firma = "Seçkin Yayıncılık",
     Telefon = "0-321-435 30 30"
  };
 textBox1.Text = kayit.Firma;
 textBox2.Text = kayit.Telefon;
}
```

"Rehber" adını verdiğimiz sınıfın adını ve **class** anahtar kelimesini silmemize rağmen sanki silmemiş gibi, "Rehber" sınıfından yola çıkarak yukarıda hazırladığımız nesne ile bire bir aynı olan bir başka nesne hazırladık. Burada yapılan işlem üzerinde maddeler halinde konuşalım:

- Burada ilk dikkatinize sunmak istediğimiz nokta, "Firma" ve "Telefon" adını verip derleyici tarafından alan olarak değerlendirilecek bilgilerin tipini yazmamış olmamızdır. Çünkü, derleyici tip tahmini bana bırakın diyor. Bu nedenle bu isimsiz tipi aşağıdaki gibi hazırlamak istemiş olsaydık hata meydana gelirdi.

```
var kayit = new
{
 string Firma = "Seçkin Yayıncılık", // hata meydana gelir
 Telefon = "0-321-435 30 30"
};
```

- **var** deyimini kullandığımız satırda **new** anahtar kelimesi kullanılmış olmasına rağmen örneği alınan bir **Class** veya **Struct** yoktur.
- **new** anahtar kelimesinden sonra mevcut tiplerden birisinin adı yazılsaydı burada yapılan işlem, yukarıda sözü edilen **object initializer** olurdu.
- Süslü parantezlerin arasında yazılan bilgilerin anahtar-değer şeklinde düzenlenmiş olması gerekir. Yoksa derleyici hata rapor eder.
- İsimsiz sınıftan yola çıkıp nesne hazırlanırken süslü parantezlerin arasına yazılan anahtar-değer çifti şeklinde düzenlenmiş bilgilerden yararlanır. Çünkü süslü parantezlerin arasına yazılan her anahtar-değer çifti bir alan olarak değerlendirilir.

- İsimsiz tipler sadece derleme işlemine kadar isimsizdirler. Derleme sırasında bizim isim vermediğimiz tiplere derleyici kendince bir isim verse de bu programcıyı ilgilendiren bir durum değildir.

İsimsiz Tipler (Anonymous Types) IDisposable interface'ini uygulamadıkları için disposable olamazlar. Hem buna gerek yoktur. Çünkü yaşam alanı zaten metodun içiyle sınırlıdır. Özetlemek gerekirse, derleyici süslü parantezlerin içine yazılanlara bakıp uygun bir tipi derleme sırasında oluşturduğu için bu şekilde hazırlanan tiplere isimsiz tipler (Anonymous Types) denilmektedir. Bu sayede basit ve başka yerlerde kullanılmayacak tipler için yeni sınıf veya yapı (struct) hazırlamak zorunda kalınmıyor. İsimsiz tiplerde sadece alanların olabileceğini, yani metotların olamayacağını fark etmiş olmalısınız. Ayrıca konu isimsiz tipler olduğunda kalıtım söz konusu olamıyor.

C# derleyicisi isimsiz tiplerin içinde başka isimsiz tiplere yer vermenize izin vermektedir. Bu söylediğimizi örneklemek için yukarıda verilen kodu aşağıdaki gibi değiştirdik.

```
private void Form1_Click(object sender, EventArgs e)
{
 var kayit = new
  {
    Firma = "Seçkin Yayıncılık",
    Telefon = "0-321-435 30 30",
    Adres = new {Sokak ="2158.Sokak No:13", Semt="Çankaya"}
  };
  textBox1.Text = kayit.Firma;
  textBox2.Text = kayit.Telefon;
  textBox3.Text = kayit.Adres.Sokak;
  textBox4.Text = kayit.Adres.Semt;
}
```

Kodun bu son halinde isimsiz tipin içinde "Adres" adında 3. bir alan bulunmaktadır. Bu alana normal bilgi yerine "Sokak" ve "Semt" adında 2 alana sahip başka bir isimsiz tip hazırlanıp aktarılmaktadır.

Var anahtar kelimesi hakkında bilgi verilen sayfalarda ifade edildiği üzere var ile tanımlanan değişkenler sadece tanımlandıkları metot içinde kullanılabilirler. Hemen akla var anahtar kelimesinden yararlanılarak hazırlanan isimsiz tipten yola çıkılarak hazırlanan nesnenin parametre yoluyla başka bir metoda gönderilmesi fikri gelir. Bu fikre uyup aşağıda verilen metodu hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 var kayit = new
        {
         firma = "Seçkin Yayıncılık",
         telefon = "0-321-435 30 30"
        };
 isimsizOku(kayit);
}
```

```
public void isimsizOku(object f)
{
  textBox1.Text = f.ToString();
}
```

Bu şekilde düzenlenen kod hatasız çalışmakla birlikte istenen sonucu vermez. Bu metotta isimsiz tipten yola çıkılarak hazırlanan nesnenin alanlarına erişim sorunludur. Yine de isimsiz tipin alanlarına erişime ihtiyacımız varsa çare Reflection sınıflarından yararlanmaktır. İsimsiz tipin alanlarına Reflection yöntemi ile nasıl ulaşıldığını aşağıda görebilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 var kayit = new
  {
   firma = "Seçkin Yayıncılık",
   telefon = "0-321-435 30 30",
   adres = "Sağlık Sokak Sıhhiye-Ankara"
  };
  Aktar(kayit);
 }
private void Aktar(object isimsiz_tip)
{
 Type tipim = isimsiz_tip.GetType();
 System.Reflection.PropertyInfo[] ozellikler=tipim.GetProperties();
 foreach (System.Reflection.PropertyInfo ozellik in ozellikler)
  {
   listBox1.Items.Add(ozellik.Name);
   listBox2.Items.Add(ozellik.GetValue(isimsiz_tip, null));
  }
}
```

Bu kodda isimsiz tipi hazırladıktan sonra **Aktar**() adını verdiğimiz metoda parametre olarak verdik. Object tipinde bir parametreye sahip Aktar() metodunda System.Reflection'da bulunan **PropertyInfo** sınıfından yararlanıp bu tipin özelliklerini öğrendik.

Devamında foreach döngüsü ile alanların adlarını formdaki ilk ListBox'a, alanların içeriklerini ise ikinci ListBox'a aktardık. Şartları zorlayıp bu şekilde isimsiz sınıf kaynaklı nesneleri başka metotlarda kullanmaya çalışmanın isimsiz sınıf hazırlama amacına ters bir yaklaşım olduğunu unutmamak gerek.

Her ne kadar yukarıda anlatılan şekilde hazırlanan tipler isimsiz olsa bile C# derleyicisi geri planda bu tiplere kendince isimler vermektedir. C# derleyicisinin verdiği ismi öğrenmek istiyorsanız aşağıdaki gibi kod yazabilirsiniz.

```
private void Form1_Click(object sender, EventArgs e)
{
 var oyuncu = new
 {
  adi = "Mirsat Türkcan",
  takimi = "Fenerbahçe",
 };
```

```
    var takim = new
    {
      adi = "Fenerbahçe",
      baskan = "Aziz Yıldırım",
    };
    textBox1.Text = oyuncu.GetType().Name;
    textBox2.Text = takim.GetType().Name;
}
```

Yukarıda anlatılan şekilde isimsiz bir tipten yola çıkılarak hazırlanan nesnenin üyeleri **readonly** oldukları için içeriklerinde değişiklik yapılamaz. Bu nedenle aşağıda verilen kod hataya neden olur.

```
private void button1_Click(object sender, EventArgs e)
{
 var oyuncu = new
  {
    adi = "Mirsat Türkcan",
    takimi = "Fenerbahçe",
  };
  oyuncu.adi = "Cedi Osman"; // Hata meydana gelir.
}
```

Anlatılanlardan tahmin etmiş olabileceğiniz gibi isimsiz tipler C# 3.0'dan itibaren desteklenen "var" anahtar kelimesi sayesinde hazırlanabilmektedir. İsimsiz tiplere daha çok LINQ tekniği ile sorgulama yapılırken gerek duyulmaktadır. Başka bir anlatımla, **Object Initializer** sayesinde **LINQ** sorgularında **Anonymous Type** üretiminin mümkün hale gelmesi sağlanmıştır ve C# 3.0 ile gelen bu yenilik, başka bir yeniliğin önünü açmıştır.

Extension Metotlar

Programcılar uygulama geliştirirken yapmak istedikleri işlemler için sınıf veya metot ararken önceliği .NET Framework ile gelen hazır sınıflara ve bu sınıfların metotlarına verirler. .NET Framework ile gelen sınıflar istediklerini yapmaya uygun değilse başkalarının veya kendilerinin daha önceden geliştirdikleri sınıflara yönelirler.

Bu sınıflar da işlerini görmüyorsa, mevcut bir sınıftan yola çıkıp yeni bir sınıf hazırlarlar. Başka bir deyişle programcılar bazen sıklıkla kullandıkları sınıfa yeni bir metot eklemeyi veya mevcut metotlardan bir veya birkaç tanesini override etmenin daha uygun olacağını düşünürler. Bu sınıf **sealed** olarak düzenlenmişse kalıtıma izin verilmez. Örneğin; sıklıkla kullandığımız String sınıfı sealed olarak düzenlenmiştir ve bu sınıftan yola çıkıp yeni bir sınıf hazırlayamayız, yeni metotlar ekleyemeyiz, mevcut metotlarını override edemeyiz.

Tam bu noktada programcıların imdadına C# 3.0 ile desteklenmeye başlanan Extension (genişletme) metotlar yetişiyor. Bu metotlar sayesinde projenizde herhangi bir sınıf için sıklıkla kullanacağınız ve proje boyunca her yerden ulaşabileceğiniz metotlar için extension metotları tercih edebilirsiniz.



Genişletme metotlarının C# diline kazandırılmasının en önemli nedenli LINQ sorgularındaki işlevidir. IEnumearable arayüzünü uygulamış koleksiyonlarda LINQ tipi sorgulama yapılırken Enumerable sınıfına sonradan eklenmiş genişletme metotlarından yararlanılmaktadır. LINQ hakkında bilgi verilen bölümde genişletme metotlarının nasıl işlevsel olduklarını görebileceksiniz. Genişletme metotlarının C#'a kazandırılmasının esas amacı LINQ olsa bile akla gelebilinecek hemen tür işlem için genişletme metotları yazılıp kullanılabilir. Bu nedenle extension metotlar önemlidir ve LINQ ile ilgilenmeseniz bile Extension metotlar çok işinize yarayacaktır.

C# 3.0 ile birlikte desteklenen bu metotların nasıl hazırlanıp kullanıldığını anlatmadan önce kısaca **Object** sınıfının **ToString**() metodundan söz edeceğiz. Bütün tipler Object sınıfından türetildiği için bütün sınıflar ToString() metoduna sahip oluyorlar. ToString() metodunu kullandığımız aşağıda verilen kod sizin için yeni bir şey içermiyor.

```
static void Main(string[] args)
{
 DateTime Tarih = DateTime.Today;
 String str = Tarih.ToString();
 Console.WriteLine(str.Substring(0, 10));
}
```

Visual Studio ile hazırladığımız konsol uygulamasında yazdığımız bu kısacık kodda dikkatinizi **ToString**() ve **Substring**() metotlarına çekmek istiyoruz. ToString() metodu Object sınıfı kaynaklı statik bir metot olduğu için bütün veri tiplerine uygulanabilir. Substring() metodu ise bildiğiniz gibi string bilgiler üzerinde işlem yapılırken kullanılmaktadır. Ayrıca hatırlamak gerekirse; değişik tipteki bilgileri string tipe dönüştürürken **Convert** sınıfının **ToString**() metodunu kullanmak mümkündür. Bu nedenle bu kod aşağıdaki gibi düzenlenirse aynı sonuç alınır.

```
public static void Main()
{
 DateTime Tarih = DateTime.Today;
 string str = Convert.ToString(Tarih);
 Console.WriteLine(str.Substring(0, 10));
}
```

Object sınıfı kaynaklı ToString() metodunu kullanmak daha kolay olduğu için bir çok programcı tarafından tercih edilmektedir. Ancak string veya başka bir tipteki bilgiyi **int** tipine dönüştürürken Convert sınıfından yararlanmak gerekmektedir. Çünkü Object sınıfının ToInt16() ve ToInt32() gibi adlara sahip metotları yoktur.

Şimdi bu uygulamanın üyesi olduğu namepace'te geçerli olmak üzere **ToInt16**() adında bir metot hazırlayıp ve bu metodu sanki Object sınıfına aitmiş gibi kullanacağız. Başka bir deyişle Object sınıfına bir metot ekleyeceğiz. Yani extension metot hazırlayacağız. Bu metot ancak hazırlandığı namespace içinde kullanılabilinen bir metot olacaktır. Yani .NET Framework ile gelen Object sınıfının orijinal hali bozulmayacak.

Konunun kolay izlenmesi için üzerinde çalıştığımız konsol uygulamasında gerek duyacağımız 2. sınıfı aynı kod dosyasında kodladık. Extension metotlar **static** bir sınıf içinde kodlanmalı-

dır ve dolayısıyla kendileri de static olmalıdırlar. Bu nedenle sonradan hazırladığımız sınıfımız statiktir.

```
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
   {
   }
 }
 public static class EkSinif
 {
  public static int ToInt16(this object obj)
   {
    return System.Int16.Parse((string)obj);
   }
 }
}
```

Bu kodda dikkatinizi "EkSinif" adını verdiğimiz sınıftaki metoda çekmek istiyoruz. Bu metodun object tipindeki parametresinde şimdiye kadar gördüğümüz metotlardan farklı olarak **this** anahtar kelimesine yer vererek **Extension** metot olmasını sağladık. Bundan şu sonucu çıkarabiliriz: İncelediğiniz kodda ilgilendiğiniz metoda ait parantezlerin içinde "this" anahtar kelimesi kullanıldıysa o metodun extension metot olduğuna hükmedebilirsiniz.

Bu metot, parametre adından önce kullanılan "this" anahtar kelimesi sayesinde **object** tipindeki verilere uygulanabilir. Çünkü metodun, önüne this anahtar kelimesi yazılan parametresi "object" tipindedir. Başka bir deyişle bu şekilde bir metot hazırlayarak bu uygulama bağlamında **Object** sınıfına metot eklemiş olduk.

Bu metot başka uygulamalarda ve namespace'lerde kullanılamaz. Hazırladığımız bu extension metodun gövdesinde kullanılan **Int16** yapısının **Parse**() metodunun işlevini daha önceki konulardan biliyorsunuz. Hazırlayıp ToInt16() adını verdiğimiz metodun geri dönüş tipini **int** olarak seçtik.

Şimdi yazdığımız bu extension metodu uygulamamızdaki diğer sınıfta kullanacağız.

```
using System;
namespace ConsoleApp1
{
 class Program
 {
  static void Main(string[] args)
  {
   string str = "2017";
   int yil = str.ToInt16();
   Console.WriteLine(yil.ToString());
  }
 }
```

```
public static class EkSinif
{
 public static int ToInt16(this object obj)
  {
   return System.Int16.Parse((string)obj);
  }
 }
}
```

Bu uygulamada önce string bir değişken tanımlayıp int tipine dönüştürülmeye uygun bilgi aktardık. Devamında kendi hazırladığımız extension metottan yararlanıp bu string değişkenin içeriğini dönüştürüp int tipindeki değişkene aktardık. **ToInt16**() adını verdiğimiz bu metodun parametresi **object** tipinde olduğu için bütün veri tiplerine uygulanabilir.

Şu an itibarıyla ilk extension metodumuzu yazıp kullandık. Şimdi yukarıda extension metotlarla ilgili olarak yaptığımız işlemleri maddeler halinde özetleyeceğiz:

- Extension metotlar **static** bir sınıf içerisinde static olarak tanımlanmalıdır. Yani genişletme metotları statik olmayan sınıfların içinde hazırlayamayız. Dolayısıyla extension metot kodlamak amacıyla ayrı sınıf hazırlamak iyi bir fikirdir.
- Extension metotları kodlarken kullanacağımız sınıf statik olarak seçildiğinde içine konulacak metotların statik olmasının şart olduğunu biliyorsunuz.
- Hazırlamak istediğiniz metotla hangi tip veya sınıfı genişletmek, başka bir anlatımla genişletme metodunu hangi sınıf için hazırlıyorsanız metodun 1. parametresi o sınıf tipinde olmalıdır. Ayrıca bu tipin önüne "this" anahtar kelimesini yazmalısınız.
- Bir sınıfı genişletmek amacıyla hazırladığınız metot birden fazla parametreye sahip olsa bile this anahtar kelimesi mutlaka 1. parametre için kullanılmalıdır. Bir metot birden fazla sınıfı genişletemeyeceğine göre metoda ait parantezin içinde this anahtar kelimesi bir kez kullanılır ve kullanma yeri açılan parantezlerin en sol tarafıdır.
- this, anahtar kelimesi hangi tipin önüne yazıldıysa genişletme metodu o sınıf için yazılmış olacaktır.
- Programcılar genellikle statik bir sınıf hazırlayıp uygulama dahilinde kullanmayı düşündükleri bütün genişletme metotlarını bu statik sınıfta topluyorlar. Tabi bir uygulama için hazırlanan genişletme metotların sayısı fazla ise bunları kategorize edip birden fazla statik sınıf arasında paylaştırmak gerekir.
- Yukarıda anlatılan şekilde yapılan genişletme işlemi sınıflara sadece metot eklenerek yapılabilmektedir.
- Kendisi için genişletme metodu hazırlamak istediğiniz sınıftaki mevcut metotları ve parametre yapılarını mutlaka bilmeniz gerekir. Çünkü sınıfta aynı imzalı bir metot zaten varsa yazdığınız genişletme metodu işletemezsiniz.
- Hazırladığınız extension metotları sadece nesneler yani instance'lar üzerinden işletilebilirsiniz. Yukarıda hazırlayıp ToInt16() adını verdiğimiz metodu işletmeden önce string tipte bir değişken tanımladık ve metodu bu değişken üzerinden işlettik.

- Mevcut bir sınıf için extension metot hazırlamaya başlamadan önce uygulama dahilinde sıklıkla kullanıp kullanmayacağınız konusunda öngörünüz nedir sorusuna cevap vermeniz yerinde olacaktır. Bizce uygulamanın değişik yerlerinde sıklıkla kullanılmayan bir metodu extension metot olarak düzenlenmenize gerek yoktur.

Yukarıda hazırladığımız extension metotta string bilgiye dönüştürme yapıp ondan sonra **Parse**() metodunu kullandığımız için bu metot int dışındaki tiplere uygulanırsa hata meydana gelir. Şimdi bu extension metotta küçük bir değişiklik yapacağız.

```
public static class EkSinif
{
 public static int ToInt16(this object obj)
  {
    return System.Convert.ToInt16(obj);
  }
}
```

Bu kez dönüştürme işlemini yaparken Int16 yapısının **Parse**() metodu yerine **Convert** sınıfının **ToInt16**() metodunu kullandığımız için bu metot aşağıda yapıldığı gibi byte veya object tipteki veriler üzerinden işletilebilir.

```
public static void Main()
{
 object nesne = 2017;
 int i = nesne.ToInt16();
 Console.WriteLine(i.ToString());
 byte b = 128;
 i = b.ToInt16();
 Console.WriteLine(i.ToString());
}
```

Yukarıda Extension metodun object tipinde bir parametreye sahip olmasını sağladığımız için sanki bu metot System.Object sınıfına eklenmiş gibi işlev görür. Bu extension metot aşağıdaki gibi düzenlenirse bu metot string bilgilere uygulanabilir.

```
class Program
{
 public static void Main()
 {
  string str = "2017";
  int yil = str.ToInt16();
  Console.WriteLine(yil.ToString());
 }
}
public static class EkSinif
{
 public static int ToInt16(this string str)
 {
  return System.Convert.ToInt16(str);
 }
}
```

Her ne kadar bu şekilde düzenlenen extension metot System.String sınıfına eklenmiş gibi işlev görse bile bu metodun etki alanı yer aldığı namespace'tir. Bu konsol uygulamasını aşağıdaki gibi düzenleyip bu 2 sınıfı ayrı namespace'lere alsaydık bu metodu ilk sınıfta kullanmak mümkün olmazdı. Bu namespace'teki extension metodu kullanmak istiyorsanız bu namespace'i **using** deyimi ile koda eklemeniz gerekir.

```
using ek_metotlar;
namespace Deneme
{
 class Program
  {
   public static void Main()
    {
     string str = "2008";
     int yil = str.ToInt16();
     Console.WriteLine(yil.ToString());
    }
 }
}
namespace ek_metotlar
{
 public static class EkSinif
  {
   public static int ToInt16(this string str)
    { return System.Convert.ToInt16(str); }
  }
}
```

Şimdi bu metodu aşırı yükleyip aynı zamanda **byte** tipindeki verilere uygulanabilmesini sağlayacağız. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
public static class EkSinif
{
 public static int ToInt16(this string str)
  {
   return System.Convert.ToInt16(str);
  }
 public static int ToInt16(this byte sayi)
  {
   return System.Convert.ToInt16(sayi);
  }
}
```

Bu şekilde düzenlenen extension metot, string tipe uygulanırsa ilk kopyası, byte tipe uygulanırsa ikinci kopyası işletilir. Sıklıkla kullandığınız genişletme metotlarının hepsini static bir sınıfta toplayıp her yeni uygulamamıza bu kod dosyasını dahil ederseniz bu genişletme metotlarınız elinizin altında olur.

Birden Fazla Parametreye Sahip Extension Metotlar

Şimdiye kadar hazırladığımız genişletme metotların hepsi bir parametreye sahipti. Bu parametrenin önüne this yazdığımız için hazırlanan genişletme metodu this ile işaret edilen

tipi temsil eden sınıfı genişletiyordu. Hazırlamak istediğiniz genişletme metodu birden fazla parametreye sahipse this anahtar kelimesini yine 1. parametrenin önünü yazmalısınız. Başka bir deyişle, genişletme metodunu hangi sınıfa eklemek istiyorsanız o sınıfı temsil eden tipi ilk parametre olarak kullanmalısınız. Dolayısıyla genişletme metotlarında this anahtar kelimesi sadece bir kez kullanılır ki o da ilk parametreden öncesidir. Birden fazla parametreye sahip genişletme metotlarına örnek olması için aşağıda verilen metodu hazırladık.

```
public static class EkSinif
{
 public static int ToInt16(this object obj)
 {
  return System.Convert.ToInt16(obj);
 }
 public static string SoldanAl(this string str, int uzunluk)
 {
  if (str.Length < uzunluk)
    return str;
  else
    return str.Substring(0, uzunluk);
 }
}
```

SoldanAl() adını verdiğimiz metot 2 parametreye sahiptir. Önüne "this" anahtar kelimesi yazılan değişkenin tipi, genişletme metodun ekleneceği sınıf olarak kullanılır. "int" tipindeki 2. parametre ise normal metotlardaki parametrelerden farklı bir işlev görmez.

Konunun iyice anlaşılması için bir de List sınıfını genişleten bir metot hazırlayacağız. Bu amaçla üzerinde çalıştığımız uygulamadaki static sınıfta aşağıdaki gibi "Uzaklatir" adında bir metot hazırladık.

```
public static class EkSinif
{
 public static IEnumerable<string> Uzaklastir(
              this List<string> liste, string cikarilacak)
 {
  foreach (string str in liste)
   {
    if (str != cikarilacak)
      yield return str;
   }
 }
}
```

Bu metot kendisine 1. parametre olarak verilen List nesnesinde, 2. parametre olarak verilen string bilgiyi arayacak. Bulursa bu elemanı silip List nesnesinin yeni halini geriye gönderecektir. Bu metodu nasıl kullandığımızı aşağıda görebilirsiniz.

```
static void Main(string[] args)
{
  List<string> superLig = new List<string>
       {"Beşiktaş", "Başakşehir", "Fenerbahçe", "Gaziantep" };
  var kalanlar = superLig.Uzaklastir("Gaziantep");
```

```
        foreach (string takim in kalanlar)
        {
          Console.WriteLine(takim);
        }
    }
```

Bu kodu dikkatlice incelerseniz, Uzaklastir() adını verdiğimiz genişletme metodumuz 2 parametreye sahip iken kullanırken sanki bir parametreye sahipmiş gibi davrandık. Çünkü genişletme metodunun önünde this anahtar kelimesi olan parametresi metot kullanılırken göz ardı ediliyor.

Kendi Sınıflarınız ve Extension Metotlar

Yukarıdaki sayfalarda extension metotlar hakkında yazılanlara bakıp bütün ayrıntılardan söz edilmiş olduğunu düşünebilirsiniz. Aslında extension metotlarla ilgili olarak yukarıda yazılanları okuyan bir programcının yapması gereken tek şey, elindeki metotlardan uygun olanları extension metot olarak düzenlemesidir. Buna rağmen konunun zihinlerde iyice netleşmesini sağlamak üzere kendimiz bir sınıf hazırlayıp bu sınıf için sonradan extension metot hazırlayacağız.

```
public class Adana
{
 public void Hakkinda()
 {
  Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
 }
}
```

Yukarıdaki sayfalarda .NET Framework ile gelen sınıflar için genişletme metodu hazırlamıştık. Şimdi ise kendi hazırladığımız bu sınıf için genişletme metodu yazacağız. Bu sınıfın source'nın elimizde olmadığını varsayıp extension metot imkanından yararlanıp bu sınıfı genişleteceğiz. Bu amaçla üzerinde çalıştığımız uygulamada statik bir sınıf hazırladık.

```
namespace ConsoleApp1
{
 public class Adana
  {
   public void Hakkinda()
    {
     Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
    }
  }
 public static class Ekler
  {
  }
}
```

İlgilenilen sınıfı genişletirken yazılacak metotları barındıracak sınıfın static olmasının şart olduğunu biliyorsunuz. Bu şekilde uygulamamızda static sınıf hazırladıktan sonra aşağıdaki gibi bir genişletme metodu hazırladık.

```
namespace ConsoleApp1
{
 public class Adana
 {
  public void Hakkinda()
   {
    Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
   }
 }
 public static class Ekler
 {
  public static void Seyhan(this Adana il)
  {
    Console.WriteLine("Seyhan, Adana'nın en büyük ilçesi");
  }
 }
}
```

Burada sizin için yeni bir şey olmamasına üzerine konuşacağımız bazı noktalar var:

- Hazırlayıp "Seyhan()" adını verdiğimiz metodun geri dönüş tipi yoktur.
- Bu metodu, "Adana" adını verdiğimiz sınıfta genişletme metodu olarak kullanmak istediğimiz için metoda ait parantezlerin arasında önce "this" sonra "Adana" yazdık.
- Bu genişletme metodu sadece bir parametreye sahiptir ve bu parametre genişletilecek sınıfı işaret etmektedir. Bu nedenle bu metot parametresiz olarak işletilir.

Önce statik sınıfımızı sonra genişletme metodumuzu hazırladıktan sonra uygulamamızda Main() metoduna sahip diğer sınıfa geçtik ve "Adana" adlı sınıfın örneğini aldık.

```
namespace ConsoleApp1
{
 class Program
  {
   static void Main(string[] args)
    {
     Adana a1 = new Adana();
    }
  }
}
```

Uygulamamızdaki "Program", "Adana" ve "Ekler" sınıflarının aynı namespace'e üye olduklarını işaret etmek için sınıfa ait kodu buraya aktarırken namespace kısmını da aldık. Bu şartlarda "Adana" tipindeki nesne üzerinden normal ve genişletme metodunu işletebiliriz.

```
class Program
{
 static void Main(string[] args)
  {
   Adana a1 = new Adana();
   a1.Hakkinda();
   a1.Seyhan(); // genişletme metodu işletildi.
  }
}
```

Burada dikkatinize sunmak istediğimiz bir ayrıntı var: Yukarıda Seyhan() adını verdiğimiz metodu hazırlarken "this" anahtar kelimesi ve genişletmek istediğimiz sınıfın adını parametre olarak kullanmıştık. Buna rağmen burada bu metodu işletirken parametre olarak bir şey vermedik ve derleyici buna itiraz etmedi. Çünkü genişletme metodu hazırlarken this anahtar kelimesinden hemen sonra tanımlanan parametre normal parametre olmayıp genişletilecek sınıfı işaret etmektir. Bu nedenledir ki sadece bir parametreye sahip genişletme metotları işletilirken parametre kullanılamaz..

Extension metotlarla ilgili başka bir noktayı dikkatinize sunmak için yukarıda hazırlayıp "Ekler" adını verdiğimiz sınıftaki genişletme metodunun adını değiştirdik.

```
public class Adana
{
 public void Hakkinda()
  {
   Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
  }
}
public static class Ekler
{
 public static void Hakkinda(this Adana il)
  {
    Console.WriteLine("Seyhan, Adana'nın en büyük ilçesi");
  }
}
```

Bu şartlarda derleyicinin nasıl bir tepki vereceğini yukarıdaki sayfalarda işaret etmiş olsak bile üzerinde durulması gereken önemli bir durumla karşı karşıyayız. C# derleyicisi mevcut bir metotla aynı imzaya sahip genişletme metodu hazırlanmasına itiraz etmiyor. Ancak bu durumda genişletme metodunu işletmek mümkün olmuyor. Çünkü derleyici önceliği sınıfın normal metoduna vermektedir. Bu sırada burada "Adana" adını verdiğimiz sınıftaki metot statik bir metot olsaydı metotları işletme tekniği farklı olmasına rağmen genişletme metodunu işletmek mümkün olmazdı.

Buradaki adıyla ifade etmek gerekirse, Hakkinda'yı genişletme metodu adı olarak kullanmakta ısrar ediyorsanız genişletme metodunun farklı bir imzaya sahip olmasını sağlamalısınız. Kendi hazırladığımız sınıftaki genişletme metodunu nasıl düzenlediğimizi, başka bir deyişle Hakkında() metodunu nasıl overload ettiğimizi aşağıda görebilirsiniz.

```
namespace ConsoleApp1
{
 public static class Ekler
  {
  public static void Hakkinda(this Adana il, string str)
   {
    System.Console.WriteLine(str);
   }
 }
}
```

Bu değişiklikten sonra ne zaman sınıfın genişletme metodunu işletmek istediğinizde string tipte bir bilgiyi parametre olarak vermelisiniz. Genişletme metodunda yaptığımız bu değişiklikten sonra bu metotların işletildiği Main() metodunu aşağıdaki gibi düzenledik.

```
class Program
{
 static void Main(string[] args)
  {
   Adana a1 = new Adana();
   a1.Hakkinda();
   a1.Hakkinda("Seyhan, Adana'nın en büyük ilçesi");
  }
}
```

Kendi hazırladığımız sınıf üzerinden genişletme metotlarını anlatmaya devam ediyoruz. Kendisi için genişletme metodu hazırladığımız sınıfın overload ettiğimiz metodu statik olsaydı ne olurdu? Değişen tek şey işletme şekli olurdu. Çünkü genişletme metodumuzun imzası farklıdır. Bu söylediğimizi netleştirmek için "Adana" adını verip kendisi için genişletme metodu yazdığımız sınıf ile genişletme metodunu içeren statik sınıfı aşağıya aldık.

```
public class Adana
{
 public static void Hakkinda()
  {
   Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
  }
 }
 public static class Ekler
 {
  public static void Hakkinda(this Adana il, string str)
  {
   System.Console.WriteLine(str);
  }
}
```

Konuyu uzatma pahasına tekrar maddeler halinde birkaç cümle yazacağız:

- Elimizde birisi normal diğeri extension olan 2 metot var. Her ikisi de static olmasına rağmen genişletme olanı normal metot gibi işletilmektedir. Yani genişletme metotlar her ne kadar hazırlanırken tanımlama satırında static anahtar kelimesine yer verilse bile nesnelerin üzerinden işletilmektedir.
- Bu metotların imzaları farklı olduğu için derleyici burada yapılanı overload olarak görüp hata rapor etmez.
- Hazırladığımız extension metodun son hali 2 parametreye sahip olsa bile işletilirken tek parametre kullanılır.
- Bu şartlarda kendisi için genişletme metodu hazırladığımız sınıftaki metodumuz static olmasına rağmen aynı ada sahip genişletme metodunun yazılmasına izin verildi. Bu iznin verilme nedeni metotların farklı imzaya sahip olmasıydı.

Genişletme metotlarıyla ilgili başka bir ayrıntıya dikkat çekmek için kendisi için genişletme metodu hazırladığımız sınıfı ve genişletme metodunu aşağıdaki gibi düzenlemiş olsaydık derleyici hata rapor ederdi. Çünkü C# derleyicisi statik sınıfları genişletmenize izin vermez.

```
public static class Adana
{
 public void Hakkinda()
  {
    Console.WriteLine("Adana, sıcak ve nemli havanın merkezi");
  }
 }
 public static class Ekler
 {
  public static void Hakkinda(this Adana il)
  {
    Console.WriteLine("Seyhan, Adana'nın en büyük ilçesi");
  }
}
```

Yukarıda hakkında konuşulan Extension metotlar ile programcılara bazı kolaylıklar sağlanmaktadır. Bu metotlar en çok LINQ ile yapılan sorgulamalarda işlevsel olmaktadırlar.

Hazırladığımız Extension Metotları Organize Etmek

Şimdi extension metotlarla ilgili kendimizce bir senaryo yazıp bu senaryonun gereklerini yerine getireceğiz. Madem genişletme metotlarını nasıl hazırlandığını öğrendik, elimizdeki metotlardan hangilerinin genişletme metodu olarak düzenlenmesi gerektiğini araştırdık. Bu araştırmaların sonucu aklımıza ilk olarak içinde bulunduğumuz ayın son ve ilk gününü öğrenmek amacıyla 2 genişletme metodu hazırlama fikri geldiğini varsayıyoruz.

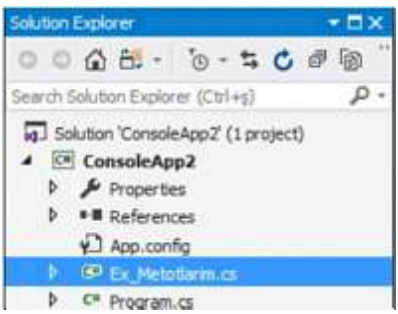
Bu metotları bundan sonra geliştirmek isteyeceğiniz uygulamalarda kullanmamız ihtimal dahilinde olduğu için aşağıdaki gibi bir kod dosyası hazırladık. Bu kod dosyasını mevcut projelerimizden bağımsız olarak diskimizde bir klasöre kaydettik.

Ek_Metotlarin.cs - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
using System;
public static class Metotlarim
{
  public static DateTime Ilkgun(this DateTime bugun)
  {
    int yil = bugun.Year;
    int ay = bugun.Month;
    DateTime ilk = new DateTime(yil, ay, 1);
    return ilk;
  }
  public static DateTime Songun(this DateTime bugun)
  {
    DateTime ilk = bugun.Ilkgun();
    DateTime son = ilk.AddMonths(1).AddDays(-1);
    return son;
  }
}
```

Ne zaman Visual Studio ile yeni bir projeye başladık o zaman Project menüsünde **Add Existing Item** komutunu verip bu kod dosyasını uygulamamıza ekledik.



İçinde extension metotlarımız olan kod dosyasını uygulamamıza ekleyip Solution Explorer penceresinde listelenmesini sağladıktan sonra ne zaman bu metotlara gerek duyduk o zaman kullandık. Bu kod dosyasında, yani "Metotlarim" adını verdiğimiz static class'taki genişletme metotlarını bu projedeki bütün class'larda kullanabiliriz.

```
class Program
{
 static void Main(string[] args)
 {
  DateTime simdi = DateTime.Now;
  DateTime ay_baslangici = simdi.Ilkgun();
  DateTime ay_bitisi = simdi.Songun();
  Console.WriteLine(ay_baslangici.ToShortDateString());
  Console.WriteLine(ay_bitisi.ToShortDateString());
 }
}
```

Yukarıda genişletme metotlarımızı içeren kod dosyasındaki static sınıfımızı herhangi bir namespace'e üye yapmamıştık. Bu sınıfın uygulamanıza dahil edilen kopyası için namespace ayarlaması yapmak iyi bir fikirdir.

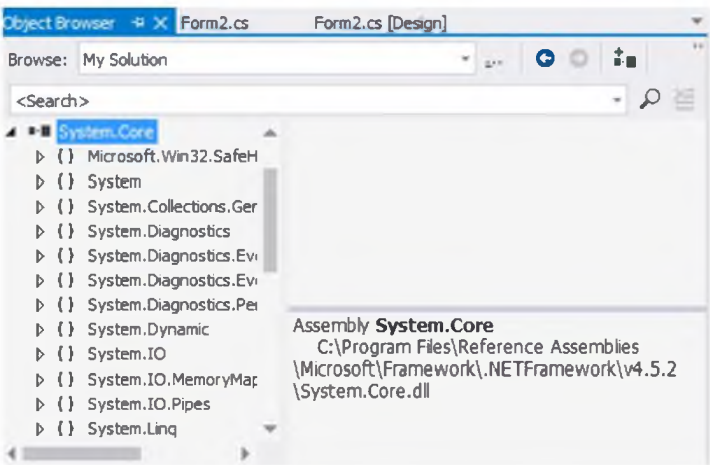
C# 3.0'dan itibaren C# dilinde yapılan birçok geliştirmede extension metotların izleri vardır. Bu nedenle bu konu iyice kavranmalıdır. Yukarıdaki sayfaları okuyup verilen kısa örnekleri denediyseniz extension metot hazırlama ve kullanma konusunda soru işaretleri kalmamış olmalıdır. Ancak konunun geri planını merak edenler için söyleyeceklerimiz henüz bitmedi.

.NET Framework ile gelen ve C# kodu içinde kullandığınız bir çok sınıfta extension metotların kullanıldığını belirtelim. Örneğin bir dizi değişken hazırlayıp değişken adının yanına nokta karakterini eklediğinizde çok sayıda metodun listelendiğini biliyorsunuz.

```
private void Form1_Load(object sender, EventArgs e)
{
    string[] takimlar = new string[10];
    takimlar.
}
```



Bu metotların bir kısmı C# 3.0 ile birlikte desteklenen genişletme metodu tekniği ile **Array** sınıfına sonradan eklenen metotlardır. Bu sırada kullandığımız Visual Studio'nun versiyonu 2003 veya 2005 olsaydı Array sınıfına eklenmiş genişletme metotları burada listelenmezdi. Visual Studio 2017 ile çalışırken Array sınıfına sonradan eklenen genişletme metotlarının listelenmesini istemiyorsanız using bloğunda "using System.Linq;" satırını silmeniz yeterlidir. İşte odaklanmak istediğimiz nokta budur: using bloğunda System'de yer alan **Linq** adlı namespace bulunan sınıflardan yaralanacağınızı belirtmediğinizde .NET Framework ile gelen çok sayıda genişletme metodundan yararlanamıyoruz.



Anonim – İsimsiz Metotlar

C# 2.0 ile gelen isimsiz metotlar delegate'lerle direk ilgilidirler. Bu nedenle bu başlık altında yazılanların delegate'ler hakkında bilgi sahibi olunduktan sonra okunması önemli olduğundan isimsiz metotları ayrı olarak düzenledik. Normalde bu başlık altında anlatılanların Metot Hazırlamak ve Kullanmak adlı bölümün sonuna eklenmesi gerekirken delegate'ler hakkında bilgi edinildikten sonra okunmasını sağlamak için buraya aldık. Diğer yandan isimsiz metotlar, lambda ifadelerine temel oluşturdukları için önemlidirler.

Bu kitapta daha önce ifade edildiği üzere delegate'ler .NET dillerine özeldir ve kavranması biraz zordur. Bu zorluğun bizce en önemli nedeni, tek başında kullanılamamalarıdır. Çünkü delegate'lerin işlevsel olabilmeleri için metotlara gerek duyarlar. Metotları, delegate olmadan işletmek mümkün olduğundan delegate'ler karşı beyinde bir bariyer oluşuyor. İsimsiz metot hazırlama delegate'lere dayandığı için bu başlık altında bir bakıma sizlere delegate'leri tekrar anlatmış olacağız.

Tek cümle ile anlatmak gerekirse, isimsiz metotlar sayesinde C# programlama dilinde metot içinde metot hazırlanabilmektedir. Programcılar isimsiz metot hazırlamayı daha çok tek kullanımlık metotlara ihtiyaç duyduklarında tercih ediyorlar.

Varsayalım ki üzerinde çalıştığınız metodu kodlarken yapılacak bir işlem için yeni bir metoda ihtiyaç duydunuz ve bu metodu başka yerde kullanma ihtimaliniz düşükse hemen üzerinde çalıştığınız metodun içinde isimsiz bir metot hazırlamak yararınıza olacaktır.

İsimsiz metotlar ile delegate'ler birlikte kullanıldıkları için konuyu bildiğiniz konular eşliğinde anlatmayı deneyeceğiz. Daha önceki konulardan bildiğiniz üzere kod yazılarak hazırlanıp forma yerleştirilen button veya diğer nesnelerin olaylarını yakalamak için .NET Framework ile gelen bir delegate olan **EventHandler**'a ekleme yapmak gerekmektedir. Bunu örneklemek için üzerinde çalıştığımız formun hazırlandığı CS dosyasına aşağıda verilen kodu yazdık.

```
Button Dugme;
private void Form1_Load(object sender, EventArgs e)
{
 Dugme = new Button();
 Dugme.Text = "Düğme1";
 Dugme.Left = 25;
 Dugme.Top = 25;
 Dugme.Click += new EventHandler(Dugme_Click);
 this.Controls.Add(Dugme);
}
private void Dugme_Click(Object sender, EventArgs e)
{
 MessageBox.Show("Düğmeyi tıkladınız");
}
```

Bu satırlar sayesinde proje çalıştırıldığında "Dugme" adında bir Button nesnesi hazırlanıp forma yerleştirilir. Burada asıl üzerinde durmak istediğimiz, aşağıda verilen satırdır. Bu satırda kendimiz delegate hazırlamak yerine .NET Framework ile gelen bir delegate olan **EventHandler**'ı kullandık.

```
Dugme.Click += new EventHandler(Dugme_Click);
```

Bu satırda gerçekte şöyle denilmektedir: Çalışma anında düğme tıklandığında, yani düğmeyle ilgili **Click** olayı meydana geldiğinde Dugme_Click() metodu işletilecektir. Başka bir anlatımla, burada Dugme adını verdiğimiz Button nesnesinin Click olayını, bu satır sayesinde Dugme_Click() metodu temsil etmeye başlar. Bu satır koda eklenip işaret edilen metot hazırlanmaz ise hata meydana gelmektedir.

C# 2.0'dan itibaren yukarıda verilen satır ile, metodu birleştirmek mümkün hale geldi. Bu işlemin nasıl yapıldığını aşağıda görebilirsiniz. Bu şekilde düzenlenen metotlara isimsiz (**anonymous**) metot denilmektedir. İsimsiz metotları EventHandler'a ekleme işlemi **delegate** anahtar kelimesi ile yapılmaktadır. Yani aşağıda verilen kodda Form1_Load() metodu içinde başka bir metot olmasına rağmen bu metodun bir adı yoktur.

```
Button Dugme;
private void Form1_Load(object sender, EventArgs e)
{
 Dugme = new Button();
 Dugme.Text = "Düğme1";
 Dugme.Left = 25;
 Dugme.Top = 25;
 this.Controls.Add(Dugme);
 Dugme.Click += delegate(object sender1, EventArgs e1)
  {
   MessageBox.Show("Düğmeyi Tıkladınız");
  };
}
```

Bu kod dikkatlice incelenirse delegate tipinin yapıcı metodunun **Object** ve **EventArgs** tipindeki parametreleri için ad seçerken Microsoft ve programcıların tercihi olan "sender" yerine "sender1" ve "e" yerine ise e1'i tercih ettik. Çünkü bu kodu zaten Form1_Load() metodu içinde hazırladık ve bu metodun parametrelerinin adı zaten "sender" ve e'dir. Parametrelere bu nedenle farklı adlar verdik.

İsimsiz metotlar genellikle yalnızca bir kez çağrılacak metotlar söz konusu olduğunda tercih edilmektedir. Burada hazırladığımız metodun adı olmadığı için başka yerden çağırıp işletmek mümkün değildir. Her ne kadar burada ismi olmayan bir metot hazırlamış olsak bile derleme işlemi sırasında derleyici geri planda bu isimsiz kod bloğuna kendince bir isim verir. Yani kodlama yaparken bir daha çağırmayacağımız için isim vermediğimiz metoda C# derleyicisi iç işleyişi gereği geri planda bir isim vermektedir.

Yukarıda isimsiz metotları anlatmak için kod yazarak forma Button nesnesi yerleştirmiştik. Ardından .NET Framework ile gelen **EventHandler** adlı delegate'ten yararlanıp düğmeyle ilgili Click olayı meydana geldiğinde işletilmek üzere isimsiz metot hazırlamıştık. Şimdi bir ayrıntıya dikkat çekmek amacıyla Button nesnesi hazırlarken Visual Studio'dan yararlanacağız. Bu amaçla üzerinde çalıştığımız Windows Forms uygulamasındaki Form1'e bir button nesnesi yerleştirdik ve Form1 sınıfına ait kodları aşağıya aldık.

```
public partial class Form1 : Form
{
 public Form1()
 {
   InitializeComponent();
   button1.Click += delegate
        {
         MessageBox.Show("Button1 tıklandı");
        };
 }
```

```
    private void Form1_Load(object sender, EventArgs e)
    {
    }
}
```

Sınıfın yapıcı metodunda forma yerleştirmiş olduğumuz "button1" adlı nesnenin Click olayın temsil etmesi amacıyla isimsiz metot hazırladık. Bu metodu hazırlarken yine **delegate** anahtar kelimesini kullandık. Dolayısıyla "button1" nesnesiyle ilgili olarak Click olayı meydana geldiğinde işletilecek isimsiz metodun object ve EventArgs tipinde 2 parametreye sahip olması zorluluğu ortadan kalktı.

Bu şekilde olaya neden olacak yani olayın tetiklenmesine aracılık edecek nesnenin parametre olarak metoda gönderilmemesi button nesneleri için önemli bir soruna neden olmaz. Ancak söz konusu nesne ListBox veya TextBox ise isimsiz metoda ilgili nesneyi parametre olarak göndermek gerekir. Konu üzerinde düşünmenizi sağlamak için yukarıda hazırladığımız isimsiz metodun 2 parametreye sahip olmasını sağladık.

```
public Form1()
{
 InitializeComponent();
 button1.Click += delegate (object sender, System.EventArgs arg)
  {
   MessageBox.Show("Button1 tıklandı");
  };
}
```

İsimsiz metodunuzu kodladığınız metodun içinde birden fazla kez kullanacaksanız veya bu metodun birden fazla düğmenin Click olayını temsil etmesini istiyorsanız mevcut delegate tipinden yola çıkıp aşağıdaki gibi delegate nesnesi hazırlamanız gerekir.

```
Button Dugme1, Dugme2;
private void Form1_Load(object sender, EventArgs e)
{
 Dugme1 = new Button();
 Dugme1.Text = "Düğme1";
 Dugme1.Left = 25;
 Dugme1.Top = 25;
 this.Controls.Add(Dugme1);
 Dugme2 = new Button();
 Dugme2.Text = "Düğme2";
 Dugme2.Left = 100;
 Dugme2.Top = 25;
 this.Controls.Add(Dugme2);
 EventHandler Temsilci = delegate(Object sender1, EventArgs e1)
  {
   Button d = (Button)sender1;
   MessageBox.Show( d.Text + " Düğmesini Tıkladınız");
  };
 Dugme1.Click += Temsilci;
 Dugme2.Click += Temsilci;
}
```

Bu kodda önce 2 Button nesnesi hazırladık ve özelliklerini ayarlayıp forma yerleştirdik. Devamında bir delegate nesnesi hazırlayıp **EventHandler** adlı delegate'e aktarıp metodu kodladık. En son olarak bu delegate'i her iki Button nesnesinin Click olayı ile ilişkilendirdik.

İsimsiz Metotlarla İlgili Ayrıntılar

Tekrar etmek gerekirse; isimsiz metotlar daha çok kod içinde yalnızca bir kez kullanılacak metotlara gerek duyulduğu zaman tercih edilmektedir. Her ne kadar burada hazırladığımız isimsiz metodu hem Dugme1 hem de Dugme2 adını verdiğimiz button nesnelerinin Click olaylarıyla ilişkilendirmiş olsak bile sadece hazırlandığı metot içinde kullandık.

Bu şekilde isimsiz metotlar üzerinde biraz konuştuktan sonra maddeler halinde bazı önemli noktaları sıralayacağız:

- İsimsiz metotlar **delegate** anahtar sözcüğüyle başlamak zorundadır. Çünkü isimsiz metot olarak değerlendirmek üzere yazılan kod satırlarını işletebilmek için delegate tipinde bir nesneye ihtiyaç vardır.
- İsimsiz metotlar normal metotlar gibi herhangi bir türde parametreye sahip olabilir. Ayrıca bu parametrelerde ref ve out anahtar kelimelerine yer verilmesine derleyici itiraz etmez.
- İsimsiz metotlarda **params** anahtar kelimesi kullanılamıyor. Çünkü isimsiz metotlar parametre yapısı önceden sabitlenmiş mevcut bir delegate ile temsil edilip çalıştırıldığı için params anahtar kelimesinin kullanılması mümkün olamamaktadır.
- İsimsiz metotlar, metot içinde hazırlanmış metotlar olsalar bile içinde bulundukları sınıftaki mevcut değişken isimleri kullanılamaz.
- İsimsiz metotlar normal bir metoda parametre olarak gönderilebilirler. Örnek olması bakımından hazırladığımız dizi değişkendeki çift sayıları ayırıp ayrı bir dizi değişkene dolduracağız. Bunu yaparken Array sınıfının **FindAll**() metodunu kullanacağız. FindAll() metodu 2 parametreye sahiptir. İlk parametrede aramaya konu dizi değişken verilirken 2. parametre olarak bir metodu almaktadır. Konumuz isimsiz metotlar olduğu için FindAll() metodunun 2. parametresini isimsiz metot olarak düzenledik. Bu işlemi nasıl yaptığımızı aşağıda görebilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] dizi = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };
 int[] ciftler = Array.FindAll(dizi, delegate (int sayi)
  {
   return (sayi % 2 == 0);
  });
 foreach (var item in ciftler)
 {
  listBox1.Items.Add(item.ToString());
 }
}
```

Delegate Hazırlayıp İsimsiz Metotlar İle Birlikte Kullanmak

Yukarıda isimsiz metotlar hakkında konuşurken .NET Framework ile gelen Button gibi kontrollerle ilgili olayları yakalarken kullanılan EventHandler'dan yararlandık. Şimdi ise kendimiz bir delegate tipi hazırlayıp isimsiz metotları yazıp işletirken kullanacağız. Bu amaçla üzerinde çalıştığımız konsol uygulamasında önce int tipinde 2 parametreye sahip metotları temsil edebilecek bir delegate hazırladık.

```
namespace Uygulama1
{
 class Program
 {
  delegate int Temsilci(int X, int Y);
  static void Main(string[] args)
   {
   }
 }
}
```

Bu şekilde class bloğu içinde hazırladığımız bu delegate tipinden, int tipinde 2 parametreye sahip metotları işletmek istediğimizde yararlanabiliriz. Yapacağımız işleme uygun bir delegate hazırladıktan sonra kendisine verilen 2 sayıyı toplayıp sonucu geriye gönderen basit bir metot hazırladık.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 public static int Topla(int x, int y)
  {
   return (x + y);
  }
 static void Main(string[] args)
  {
  }
}
```

Bu metodu, elimizdeki delegate temsil edebilir. Çünkü metodun imzası ile delegate aynı yapıya sahiptir. Bu iki hazırlıktan sonra "Temsilci" adını verdiğimiz delegate sayesinde bu metodu işletebiliriz. Bunun için "Temsilci" tipinde bir nesne hazırladık. Bu nesneyi hazırlarken ilgili metodu yapıcı metoda parametre olarak verebiliriz.

```
static void Main(string[] args)
{
 Temsilci islem = new Temsilci(Topla);
 int Sonuc = islem(100, 150); // Topla Metodu işletildi
 Console.WriteLine(Sonuc.ToString());
}
```

Elbette burada yapılan basit işlem için delegate hazırlamak gerekmiyor. Amacımız sizlere isimsiz metotları anlatmaktır. İsimsiz metotlardan söz ederken delegate kullanmak gerektiği için yukarıda en basit hali ile delegate'lerden söz ettik.

Şimdi ise yukarıda hazırlayıp delegate yardımıyla işlettiğimiz Topla() adlı metodu sileceğiz. Çünkü Topla() metodu sayesinde yaptığımız işlemi Main() metodu içinde isimsiz metot olarak yapacağız. Üzerinde çalıştığımız konsol uygulamasının Topla() adını verdiğimiz metodun silinmiş hali aşağıdadır.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
  Temsilci islem = delegate (int x, int y);
 }
}
```

Main() metodu içinde, burada "islem" adını verdiğimiz delegate nesnesini hazırladıktan sonra şimdi isimsiz bir metot hazırlayacağız. Bu amaçla Main() metodu içinde delegate nesnesi hazırlanan satırın sonunu belirten noktalı virgülü bir alt satıra alıp öncesine süslü parantezler ekledik.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
  Temsilci islem = delegate (int x, int y)
   {
   };
 }
}
```

Bu sırada süslü parantezlerin arasına yazılan her şey isimsiz metot olarak değerlendirilir. Burada hazırlanacak isimsiz metot int tipinde 2 parametreye sahip olup geriye int bilgi gönderen metotları temsil edebilen delegate ile birlikte işlev göreceği için isimsiz metodumuzu aşağıdaki gibi kodlayıp kullandık.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
  Temsilci islem = delegate (int x, int y)
   {
     return (x + y);
   };
  int sonuc = islem(100, 150); // isimsiz metot işletildi
 }
}
```

Daha önce delegate'ler hakkında yazılanların hatırlanması için aynı delegate için ikinci bir isimsiz metot hazırlayacağız. Başka bir deyişle burada "islem" adını verdiğimiz delegate

nesnesinin birden fazla isimsiz metodu temsil etmesini sağlayacağız. Yaptığımız bu değişikliğin pratik bir değerinin olmadığını bilmelisiniz.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
  {
   Temsilci islem = delegate (int x, int y)
    {
     return (x + y);
    };
    islem += delegate (int x, int y)
    {
      return (x * y);
    };
   int sonuc = islem(100, 150); // ikinci isimsiz metot işletildi
  }
}
```

Bu şartlarda burada "islem" adını verdiğimiz delegate nesnesi birden fazla isimsiz metodu temsil etmektedir. Konu üzerinde düşünmenizi sağlamak bu kodu aşağıdaki gibi düzenledik.

```
delegate int Temsilci(int X, int Y);
static void Main(string[] args)
{
 Temsilci islem = delegate (int x, int y)
  {
    return (x + y);
  };
 islem += delegate (int x, int y)
  {
   return (x * y);
  };
  object[] dizi = new object[2] {100, 200 };
  Delegate[] metotlar = islem.GetInvocationList();
  object obj = metotlar[1].DynamicInvoke(dizi);
  Console.WriteLine(obj.ToString());
}
```

Burada System'deki Delegate sınıfının **GetInvocationList**() metodunu kullanıp birden fazla metodu temsil etmesini sağladığımız "islem" adlı delegate nesnesi aracılığıyla bu metotlardan ikincisini işlettik. Bu metot işletildiğinde geriye bir delegate dizisi göndermektedir.

Object Initializer

C# 3.0 ile gelen ve Türkçede nesne ilklendirme olarak ifade edilen Object Initializer imkanından bu kitapta şimdiye kadar verilen örnek kodlarda zaman zaman yararlandık ve birkaç yerde sözünü ettik. Object Initializer hakkında derli toplu bilgi edinmek isteyenlerin olabileceğini düşünerek konuyu ayrı bir başlık altında incelemeyi uygun gördük. Bu sayede

yapıcı metotlarla ilgili bilgilerimizi tazelemiş olacağız. İlk olarak aşağıdaki gibi 2 özelliğe sahip bir class hazırladık. Bu class'ın yapıcı metodu yoktur.

```
public class Takim
{
 public string Adi { get; set; }
 public int Kurulus { get; set; }
}
```

Dolayısıyla bu sınıftan yola çıkıp new anahtar kelimesiyle nesne hazırlandığında varsayılan yapıcı metot devreye gireceği için özelliklere söz konusu tipler için varsayılan değerler aktarılır. Bu nedenle bu sınıf tipinde nesne hazırlanıp özelliklere aşağıdaki gibi erişildiğinde ilk TextBox'a boşluk, ikincisine 0 aktarılır.

```
private void Form1_Load(object sender, EventArgs e)
{
 Takim t1 = new Takim();
 textBox1.Text = t1.Adi;
 textBox2.Text = t1.Kurulus.ToString();
}
```

Bu şekilde kendisi için yapıcı metot hazırlanmamış sınıfların örnekleri alındıktan sonra alan veya özelliklere aşağıda yapıldığı gibi bilgi aktarıldığını biliyorsunuz.

```
private void Form1_Load(object sender, EventArgs e)
{
 Takim t1 = new Takim();
 t1.Adi = "Trabzonspor";
 t1.Kurulus = 1967;
 textBox1.Text = t1.Adi;
 textBox2.Text = t1.Kurulus.ToString();
}
```

Konunun diğer ayrıntılarından söz edebilmek için bu sınıfın aşağıdaki gibi yapıcı metoda sahip olmasını sağladık.

```
public class Takim
{
 public string Adi { get; set; }
 public int Kurulus { get; set; }
 public Takim(string par1, int par2)
 {
  this.Adi = par1;
  this.Kurulus = par2;
 }
}
```

Bu hazırlıktan sonra derleyicinin varsayılan yapıcı metot desteğinin bittiğini biliyorsunuz. Bu nedenle yukarıda yaptığımız gibi nesne hazırlayamayız. Bu şartlarda "Takim" tipinde nesneyi aşağıdaki gibi hazırlamak zorunda kalırız.

```
private void Form1_Load(object sender, EventArgs e)
{
 Takim t1 = new Takim("Trabzonspor", 1967);
 textBox1.Text = t1.Adi;
 textBox2.Text = t1.Kurulus.ToString();
}
```

Arada bir varsayılan yapıcı metodu kullanarak nesne hazırlamak, alanlara bilgileri sonradan kendim aktarmak istiyorum diyenler sınıflarında bir de aşağıdaki gibi herhangi bir satıra sahip olmayan metodu kodluyorlar.

```
public class Takim
{
 public string Adi { get; set; }
 public int Kurulus { get; set; }
 public Takim(string par1, int par2)
 {
  this.Adi = par1;
  this.Kurulus = par2;
 }
 public Takim()
  {
  }
}
```

Bu şartlarda yani sınıfımızın hem varsayılan hem de içeriklerini alanlara aktarılmak üzere 2 parametreye sahip yapıcı metodu varken C# 3.0'dan itibaren desteklenen Object Initializer yöntemi ile nesne hazırlayıp alan veya özelliklere bilgi aktarmak mümkündür.

Aynı teknikle C# 1.0'dan bu yana ve bütün programlama dillerinde dizi değişkenlerin aynı teknikle hazırlandığını hatırlatmak istiyoruz. Örneğin aşağıda verilen kodda 4 elemanlı bir dizi değişken tanımlanıp elemanlarına ilk değerler aktarılmaktadır.

```
string[] Gunler = new string[]
     {
       "Pazar",
       "Pazartesi",
       "Salı",
       "Çarşamba"
      };
```

Aynı teknikle kendi hazırladığımız sınıflardan yola çıkarak nesne hazırlama LINQ teknolojisine destek için C#'a dahil edildi. Object Initializer denen teknikle nasıl nesne hazırlandığını aşağıda görebilirsiniz.

```
private void Form1_Load(object sender, EventArgs e)
{
 Takim t1 = new Takim()
         {Adi = "Trabzonspor", Kurulus = 1967};
 textBox1.Text = t1.Adi;
 textBox2.Text = t1.Kurulus.ToString();
}
```

Bu şekilde nesne hazırlarken ilgili sınıfın yapıcı metotlarının olup olmamasının bir etkisi yoktur. Konu üzerinde biraz durmanızı sağlamak için yukarıda hazırladığımız sınıfın özelliklerinin sayısını üçe çıkardık ve 2. bir class hazırladık.

```
public class Takim
{
 public string Adi { get; set; }
 public int Kurulus { get; set; }
 public Baskanlar Yoneticiler { get; set; }
 public Takim(string par1, int par2)
 {
  this.Adi = par1;
  this.Kurulus = par2;
 }
 public Takim()
  {
  }
}
public class Baskanlar
{
  public string Baskan { get; set; }
  public string Yardimci { get; set; }
}
private void Form1_Load(object sender, EventArgs e)
{
 Takim t1 = new Takim()
  {
   Adi = "Trabzonspor",
   Kurulus = 1967,
   Yoneticiler = new Baskanlar
   {
    Baskan = "Muharrem Usta",
    Yardimci ="Ahmet Çubukçu",
   }
  };
 textBox1.Text = t1.Adi;
 textBox2.Text = t1.Kurulus.ToString();
 textBox3.Text = t1.Yoneticiler.Baskan;
 textBox4.Text = t1.Yoneticiler.Yardimci;
}
```

Burada "Takim" adını verdiğimiz sınıfın bir alanı "Baskanlar" tipindedir. Dolayısıyla nesne ilklendirme işlemi içinde nesne ilklendirme yapmış olduk. Nesne hazırlarken burada sözünü ettiğimiz Object Initializer tekniğini kullandığınızda derleyici geri planda önce yapıcı metodu çağırır ve sonra özelliklere değer atamasını yapar.

26. BÖLÜM

LINQ İLE SORGULAMA İŞLEMLERİ

Bu kitabın bir önceki bölümünde LINQ ile çalışmaya başlamadan önce hakkında bilgi sahibi olmanız gereken konuların üzerinde duruldu. Bazılarınızın aklına lambda operatörü, dolayısıyla lambda ifadeleri gelmiş olabilir. Lambda ifadeleri sayesinde LINQ sorgularında isimsiz metot olarak yazılması gereken kod kısalsa bile bize göre önceliğe lambda ifadelerine vermemek gerekir. Çünkü o zaman LINQ'i anlamak zorlaşmaktadır. Ne zaman ki LINQ'e hakim olunur o zaman lambda operatörüyle kod kısaltma çabasına girişilebilir.

En basit anlatımla, LINQ teknolojisi sayesinde koleksiyon şeklinde düzenlenmiş, başka bir deyişle IEnumareble arayüzünü uygulamış listelerde SQL benzeri sorgulama yapılabilmektedir. Sorgulama yapılırken birden fazla teknik ve yazım şekli söz konusu olduğu için LINQ, C# dilinin zor anlaşılan konularından birisi olmaktadır.

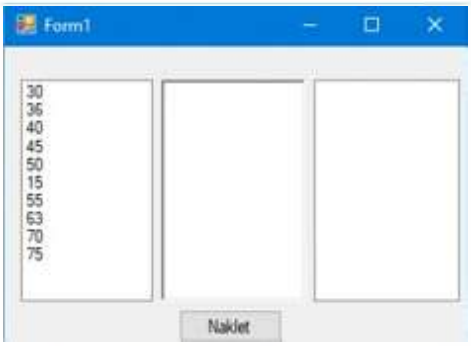
Her ne kadar yukarıda LINQ ile koleksiyon şeklinde düzenlenmiş veriler üzerinde sorgulama yapılırken kullanıldığı ifade edilmiş olunsa bile LINQ ile veritabanlarından çekilen verilerde ve XML dosyalarında sorgulama yapılabilmektedir. Bu bölümde veritabanından çekilen verilerle ve XML dosyalarıyla ilgilenmeyeceğiz.

Başka bir deyişle konu kolay izlenebilinsin diye bir veritabanından veri çekip üzerinde işlem yapmayı tercih etmedik. Bize sorarsanız bu bölümü sonuna kadar okur ve verilen örnekleri uygularsanız LINQ konusu sizin için anlaşılmaz olmaktan çıkmış olacaktır. Bu bölümde anlatılan LINQ sorgularını iyice kavramak istiyorsanız bu kitabın "var" anahtar kelimesinin anlatıldığı sayfaları, genişletme metotlarının ve isimsiz tiplerin anlatıldığı bölümde anlatılanları okuyup kavramış olmanız yararınıza olacaktır.

Bu bölümüm baş tarafında yazılanlar bir birini takip ettiği için alt başlıklara yer vermedik. Başka bir deyişle bu bölümde alt başlıksız olarak verilen ilk 10 sayfayı sonuna kadar okumanızı istiyoruz. Çünkü bu sayfalarda bilinçli olarak yapılmış tekrarlar var. Bu tekrarlar sayesinde, LINQ sorgularının anlaşılmaz olma hali ortadan kaldırılmak istenmektedir.

LINQ üzerine yazılabilinecek çok şey olduğu için başlangıçta işin teorik yanı ile ilgilenmek yerine öncelikle en basit şekli ile nasıl kullanıldığından söz edeceğiz. Ötesi, madem bu teknik .NET Framework 3.5, dolayısıyla C# 3.0 ile kullanıma sunuldu, bu teknikten yararlanmadan gelişmiş programlar yazmak mümkün olmalıdır. LINQ kullanıma sunulmadan önce programcıların elleri kolları bağlı değildi, sadece aynı işlemleri yapmak için biraz daha yoruluyorlardı. Özetlemek gerekirse, LINQ teknolojisi ile koleksiyon şeklinde düzenlenmiş bilgilerde sorgulama işlemi son derece kolay olmaktadır.

SQL kodu ile veritabanından istenen bilgiler nasıl sorgulanabiliniyorsa benzer şekilde LINQ ile C# projeleri dâhilinde koleksiyon şeklinde düzenlenmiş bilgileri sorgulamak kolay hale gelmiştir. Bu konu hakkında konuşmak vermek için Windows Forms uygulaması hazırlayıp uygulamanın formuna 2 ListBox 1 RichTextBox yerleştirip ilk ListBox'a tasarım anında rastgele sayılar aktardık.



"Naklet" adını verdiğimiz düğmeyi tıkladığımızda ilk ListBox'taki sayıların tek olanlarını 2. ListBox'a, çift olanları ise RichTextBox nesnesine aktaracağız. Bunu yaparken başlangıçta LINQ'den yararlanmayacağız. Bu amaçla hazırladığımız kodu aşağıda verdik.

```
private void Naklet_Click(object sender, EventArgs e)
{
 foreach (string aza in listBox1.Items)
  {
   int number = Convert.ToInt16(aza);
   if (number % 2 == 0)
     richTextBox1.Text = richTextBox1.Text + aza + "\n";
   else
     listBox2.Items.Add(aza);
  }
}
```

Bu kodda sizin için yeni bir şey yoktur. foreach döngüsü ile ilk ListBox'ın bütün üyelerini sırayla elde edip **int** tipine dönüştürüp üyenin tek mi çift olduğunu öğrenip tekleri formdaki 2. ListBox'a çiftleri ise RichTextBox nesnesine aktardık.

Daha önce ilgili bölümde foreach döngüleri hakkında bilgi verilirken ifade edildiği üzere foreach anahtar kelimesine 2. parametre olarak verilen koleksiyon nesnesinin ilk elemanından başlamak üzere bütün elemanları üzerinde ileri doğru hareket edilip her turda üzerine gidilen eleman foreach anahtar kelimesine 1. parametre olarak verilen değişkene aktarılıp ondan sonra istenen işlem yapılmaktadır.

C# programlama dilinin 1.0 ve 2.0 sürümlerinde elemanları üzerinde gezilen koleksiyonun tipini gözetip ona göre 1. parametre olarak verilen değişken için tip seçmek gerekiyordu. C# 3.0 ile birlikte desteklenen "var" anahtar kelimesi sayesinde programcılar isterlerse tip seçimini derleyiciye bırakabiliyorlar.

Yukarıda foreach döngüsü yardımıyla yaptığımız işlemleri LINQ deyimleri ile yapmak daha pratiktir. Peşinen söyleyelim ki burada yapılan sorgulama son derece basit olduğu için LINQ ile sorgulama yapılmayabilir. Örneğin elimizde 3 adet List nenesi bulunsaydı ve bu nesnelerde birbirleriyle ilişkili alanlar olsaydı foreach ile sorgulama yapmak karmaşık olurdu.

Konuyu adım adım anlatmak istediğimiz için yukarıda hazırladığımız uygulamadaki Button nesnesinin Click olayını temsil eden metotta değişiklik yapıp önce int tipinde bir dizi değişken tanımlayıp formdaki ilk ListBox'ın elemanlarını bu dizi değişkene aktardık.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
  {
   dizi[i] = Convert.ToInt16(listBox1.Items[i]);
  }
}
```

Bu şartlarda formdaki ilk ListBox'ın bütün üyeleri "dizi" adını verdiğimiz int tipindeki dizi değişkene aktarılmış oldu. Şimdi ise integer tipinde bilgi içeren bu dizi değişkenin bütün üyelerini LINQ tekniği ile başka bir dizi değişkene yani koleksiyona aktaracağız.

LINQ ile sorgulama yapabilmek için **using** bloğuna **using System.Linq;** yazıp bu namespace'teki sınıflardan yararlanmak istediğinizi belirtmeniz gerekiyor. LINQ ile nasıl sorgulama yapıldığını aşağıda görebilirsiniz.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
 {
  dizi[i] = Convert.ToInt16(listBox1.Items[i]);
 }
 IEnumerable<int> cift_dizi = from eleman in dizi select eleman;
}
```

Hazırladığımız LINQ sorgusu sadece bir satırdan meydana gelse bile üzerinde konuşup açıklama yapmamız gereken çok sayıda önemli nokta bulunmaktadır. Bu noktalar üzerinde maddeler halinde konuşacağız:

- LINQ ile yapılan sorgulama sonucu geriye bir **IEnumerable** nesnesi gönderildiği için eşitliğin sol tarafına generic IEnumerable nesnesi adı yazdık. Sorgulama sonucu elde edilecek yeni koleksiyonda int tipinde bilgiler olacağını bildiğimiz için "<>" işaretlerinin (diamond operatörü) arasına "int" yazdık.
- LINQ teknolojisi ile sorgulama yapmak üzere yazdığımız bu satırın bir kısmı foreach döngüsünde parantezin içine yazılanlara benzemektedir.
- LINQ ile sorgulama yapılmak üzere yazılan bu satırı dikkatlice incelerseniz **from**, **in** ve **select** deyimlerinin kullanıldığını fark ederseniz. Daha önce az çok SQL kodu yazmış olanların ilk dikkatini çekecek nokta, sanırız SQL dilinde yapılan sorgulamaların aksine select kelimesinin başa değil sona yazılmış olması olur.

- **foreach** anahtar kelimesinde olduğu gibi sorgulamaya konu edilecek dizi değişken veya koleksiyonun adı "in" deyiminden sonra yazılmaktadır.
- in'den önce yazılan ise değişken işlevi görmekte ve koleksiyonun üzerinde ileri doğru hareket edilip gezilirken o sırada değerlendirmeye alınan satır veya elemanı temsil etmektedir. Yani bu satır ile kaynak koleksiyonda (burada dizi) sorgulama yapmak üzere elemanların arasında gezilirken geçerli eleman burada "eleman" adını verdiğimiz değişkene aktarılır.
- Select deyimi ise koleksiyonun o sırada değerlendirmeye alınan eleman veya satırın, geriye gönderilip eşitliğin sol tarafına adı yazılan IEnumerable<int> nesnesine eleman olarak eklenmesi işlemini yapmaktadır.
- Burada hazırladığımız LINQ sorgusunda kullanılan select'ten deyim veya anahtar kelime söz etsek bile select'ten genellikle operatör diye söz edilmektedir. Bu adlandırmanın arka planından daha sonra söz edilecektir.

LINQ kaynaklı **select** deyiminin (operatör) sorgulama sonucu hazırlayıp geriye gönderdiği koleksiyon nesnesinin içerdiği bilgi için tip tahmini işlemi genellikle derleyiciye bırakıldığı için bu kod aşağıdaki gibi yazılabilir.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
  {
   dizi[i] = Convert.ToInt16(listBox1.Items[i]);
  }
 var cift_dizi = from eleman in dizi select eleman;
}
```

Sorgulama sonucu elde edilecek yeni generic koleksiyonda hangi tip bilgilerin olacağı kod yazımı sırasında belli ise "var" anahtar kelimesini kullanıp tip tahmini derleyiciye bırakmak yerine IEnumerable nesnesinin yanına, "<>" operatörünün arasına ilgili tipi yazabilirsiniz. LINQ ile yapılan sorgulamaların sonuçları genellikle "var" anahtar kelimesi ile tanımlanan koleksiyon nesnelerine aktarılmaktadır.

LINQ'in select deyimi, sorgulama sonucu ilgili koleksiyondan çektiği verileri IEnumerable arayüzünün generic haline doldurup geriye göndermektedir. Yukarıda sorgulamaya kaynaklık eden dizi değişken int tipinde olduğu için IEnumerable nesnesinin int tipinde bilgi içereceği yorumunu yaptık.

LINQ ile yapılan sorgulama sonucu elde edilen yeni koleksiyonun içeriği istenildiği gibi kullanılabilir. Örneğin aşağıda yapıldığı gibi bir foreach döngüsü hazırlanıp elemanlar istenildiği gibi elde edilip kullanılabilir. Burada LINQ ile hazırladığımız yeni koleksiyona "cift_dizi" adını vermiş olmakla birlikte dizide hem tekler hem de çifler bulunmaktadır. Çünkü filtreleme işlemini henüz yapmadık

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
 {
  dizi[i] = Convert.ToInt16(listBox1.Items[i]);
 }
 var cift_dizi = from eleman in dizi select eleman;
 foreach (int sayi in cift_dizi)
 {
  listBox2.Items.Add(sayi.ToString());
 }
}
```

Yukarıda anlatılan şekilde LINQ ile ilk sorgumuzu yaptıktan sonra şimdi başa dönüp konuyu başka bir pencereden ele alacağız. Yukarıda ifade edildiği gibi konu LINQ olunca başlangıçta kafası karışmayan programcı yok gibidir. Bu cümlenin arkasını doldurmaya çalışmak kolay değildir. Elbette daha önce LINQ ile hiç ilgilenmemiş, bir şey okumamış veya LINQ'i merak etmemiş birisinin yukarıda yazılanları okuduktan sonra kafasının karışması ihtimali yüksek değildir. Bu nedenle değişik açılardan konuya yaklaşarak maddeler halinde anlatmaya çalışacağız.

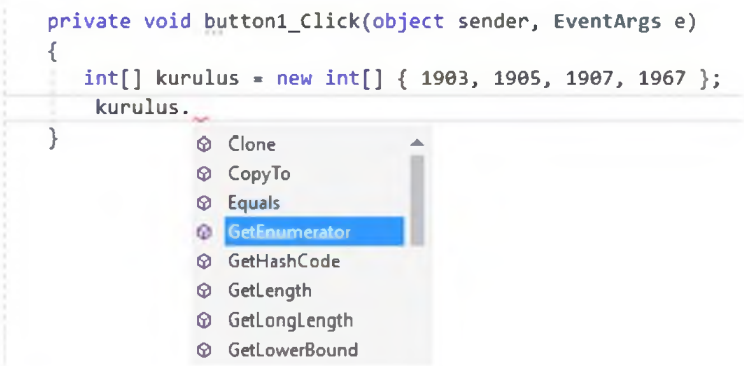
LINQ hakkında bilgi verilmek amacıyla hazırlanmış bütün kaynaklarda using bloğunda "using System.Linq" satırına yer verilmesi gerektiği söyleniyor. Ya programcı tıpkı **Console** sınıfının metotlarını kullandığı gibi sınıf adından önce namespace adını yazacağım derse ona ne cevap vereceğiz? Yani using bloğunda System'deki "Linq" adlı namespace'teki sınıfları kullanmak istiyorum demezsek LINQ'ten nasıl yararlanırız sorusunun cevabı önemlidir.

LINQ sorgulama teknolojisi üzerinde konuşmaya devam etmek için aşağıdaki gibi bir dizi değişken tanımlayıp birkaç bilgi aktardık.

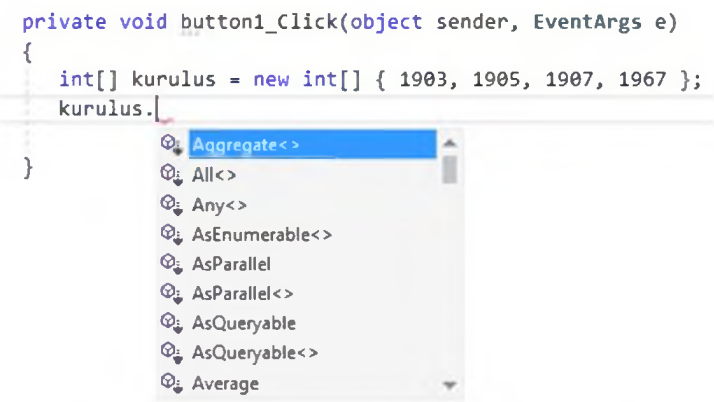
```
int[] kurulus = new int[] {1903, 1905, 1907, 1967};
```

Bu şekilde int tipinde bir değişken tanımlamakla gerçekte **Array** tipinde bir nesne hazırladık. Dolayısıyla Array sınıfının statik olmayan metotlarından istediklerimizi bu nesne yani dizi değişken üzerinden işletme imkanımız vardır.

Visual Studio'nun 2003 veya 2005 sürümleri ile çalışırken bu dizi değişkenin adının yanına nokta yazdığımızda Visual Studio aşağıdaki gibi çok sayıda metodu listeleyip istediğimizi kullanabileceğimizi işaret ederdi.



Burada listelenen metotların hepsi **Array** sınıfı kaynaklıdır. Bu sırada yararlandığımız Visual Studio, 2008 veya sonrası bir sürüm olsaydı listelenen yani bu dizi değişken üzerinden kullanıma sunulan metotların sayısı aşağıda görüleceği gibi katlanırdı.



Array nesnesi üzerinden işletebileceğimiz metotların sayısındaki artışın nedeni LINQ'tir. Bu sırada "using" bloğunda "using System.Linq" satırı olmasaydı LINQ kaynaklı metotları Array nesnesi üzerinden işletme imkanımız olmazdı ve Visual Studio bu metotları burada listelemezdi.

Bu kitabın koleksiyonların anlatıldığı bölümde işaret edildiği üzere IEnumerable arayüzü sadece **GetEnumerator**() adında bir metoda sahiptir. Bu metot IEnumerator tipinde bir nesneyi geriye gönderiyordu. IEnumerator arayüzü ise **MoveMext**(), **Reset**() ve **Current**() adında 3 metoda sahipti. Bu arayüzlerin orijinal yapıları aşağıda tekrar verildi.

```
public interface IEnumerable
{
 public IEnumerator GetEnumerator();
}
public interface IEnumerator
{
 public bool MoveNext();
 public void Reset();
 public object Current {get;}
}
```

Bu 2 arayüz, C# dilinin ilk sürümünden beri orijinal halini koruyorlar. Sonra C# 2.0 ile birlikte bu 2 arayüzün bir de generic sürümleri hazırlandı. Aşağıda bu arayüzlerin generic hallerini görebilirsiniz.

```
public interface IEnumerable<out T> : IEnumerable
{
 IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
 T Current { get; }
}
```

Bu 2 arayüzün generic hallerini dikkatlice incelerseniz 3 farklılık görürsünüz:

- Generic arayüzler, generic olmayan arayüzlerin mirasçısıdır. Dolayısıyla generic olmayanlarda ne varsa aynısı generic olanlarda da vardır.
- IEnumerable<> arayüzüne GetEnumerator() metodunun geriye generic tip içeren IEnumerator arayüzünü gönderen hali eklenmiştir.
- IEnumerator arayüzüne ise Current özelliğinin generic tipli hali eklenmiştir.

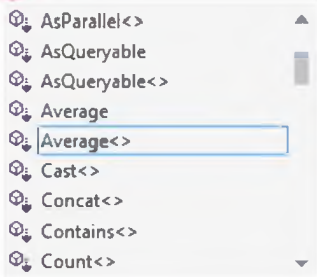
Bu interfacelerin nasıl kullanıldığını, sınıflara uygulandıklarında neler yapmak gerektiğini şimdi burada tekrar anlatmayacağız. Orijinal hallerini buraya alma nedenimiz, C# 2.0'dan C# 3.0 ve sonrasına gelirken bu interface'lerde herhangi bir değişikliğin yapılmadığını dikkatinize sunmaktır. Daha açık yazmak gerekirse, C# 3.0 ile birlikte C# diline LINQ desteği kazandırılırken IEnumerable ve IEnumerator arayüzlerinde herhangi bir değişiklik yapılmamıştır.

C# 2.0'dan bu yana IEnumerable ve IEnumerator arayüzlerinde herhangi bir değişiklik olmadığı tespitini yaptıktan sonra şimdi yukarıda verdiğimiz kodu mercek altına alacağız.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
 {
  dizi[i] = Convert.ToInt16(listBox1.Items[i]);
 }
 IEnumerable<int> cift_dizi = from eleman in dizi select eleman;
}
```

Yukarıda anlatılanlardan bildiğiniz gibi burada hazırladığımız sorguda LINQ ile gelen select operatörü, in'den sonra adı yazılan dizi değişkenin bütün elemanlarını IEnumerable tipindeki nesnede toplayıp geriye göndermektedir. Dolayısıyla burada "cift_dizi" adını verdiğimiz nesnenin üzerinden işletebileceğimiz metotlar sadece birkaç tane olmalıdır. Yani elimizde bir IEnumerable<int> nesnesi var ve bu arayüzün metot ve özelliklerinin ne olduğunu yukarıda gördük. Bu yorumun doğru olup olmadığını göstermek için Visual Studio'dan yararlanalım.

```
private void Naklet_Click(object sender, EventArgs e)
{
    int adet = listBox1.Items.Count;
    int[] dizi = new int[adet];
    for (int i = 0; i < adet; i++)
    {
        dizi[i] = Convert.ToInt16(listBox1.Items[i]);
    }
    IEnumerable<int> cift_dizi = from eleman in dizi select eleman;
    cift_dizi.
}
```



Verilen ekran görüntüsünden fark edeceğiniz gibi bu IEnumerable<int> tipindeki nesne üzerinden işletebileceğimiz çok sayıda metot bulunmaktadır. Bu metotların LINQ kaynaklı olduğunu zaten biliyorsunuz. Burada öğrenmek istediğimiz şudur: Bu metotlar önümüze nasıl geldiler? Bir sonraki başlıkta bu sorunun cevabını vermeye çalışacağız.

Yukarıda LINQ sorgulamasının yapıldığı satırda kullanılan select ve LINQ ile sorgulama yapılırken filtreleme desteği sunan "where" gibi deyimlere operatör de denilmektedir. SQL dilindeki alışkanlıktan dolayı bu kitapta "select" gibi operatörlerden bazen deyim diye söz ettik.

Select operatörü gerçekte, **Enumerable** sınıfında yer alan bir genişletme metottur. Başka bir deyişle LINQ sorgulaması yapabilmek amacıyla bu sınıfa sonradan (C# 3.0 sürümüyle birlikte) eklenen statik bir metottur. Bu sınıfın yapısının bir kısmını aşağıda görebilirsiniz.

```
public static class Enumerable

    public static TSource Aggregate<TSource>(this IEnumerable<TSource> source, Func<
    public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSour
    public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<
    public static bool All<TSource>(this IEnumerable<TSource> source, Func<TSource,
    public static bool Any<TSource>(this IEnumerable<TSource> source);
    public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource,
    public static IEnumerable<TSource> AsEnumerable<TSource>(this IEnumerable<TSourc
    public static decimal? Average<TSource>(this IEnumerable<TSource> source, Func<T
    public static decimal Average<TSource>(this IEnumerable<TSource> source, Func<TS
    public static double Average(this IEnumerable<int> source);
    public static double? Average(this IEnumerable<int?> source);
    public static double Average(this IEnumerable<long> source);
    public static double? Average(this IEnumerable<long?> source);
    public static float Average(this IEnumerable<float> source);
    public static double Average(this IEnumerable<double> source);
    public static double? Average(this IEnumerable<double?> source);
    public static decimal Average(this IEnumerable<decimal> source);
```

LINQ nedir, LINQ ile neler yapılabilir sorusuna verebileceğiniz cevap bu sınıfta gizlidir. Bu nedenle Visual Studio'dan yararlanıp veya microsoft.com'da arama yapıp bu sınıfın orijinal halini dikkatlice incelemeniz önerilir. Şimdi bu sınıfla ilgili olarak gözümüze çarpan bazı noktaları maddeler halinde sıralayacağız.

1- Bu sınıf yani C# 3.0 ile birlikte verilmeye başlanan Enumerable statik bir sınıftır. Dolayısıyla bu sınıfın bütün metotları statiktir. Yani System.Linq'de bulunan bu sınıfın hem örneği alınamaz hem de başka bir sınıfın base class'ı yapılamaz.

2- Bu sınıfta bulunan çok sayıda statik metodu tek tek incelediğinizde hepsinin imzasında "this" anahtar kelimesini görebilirsiniz. Yani Enumerable sınıfında C# 3.0 ile kullanıma sunulan çok sayıda genişletme metodu bulunmaktadır.

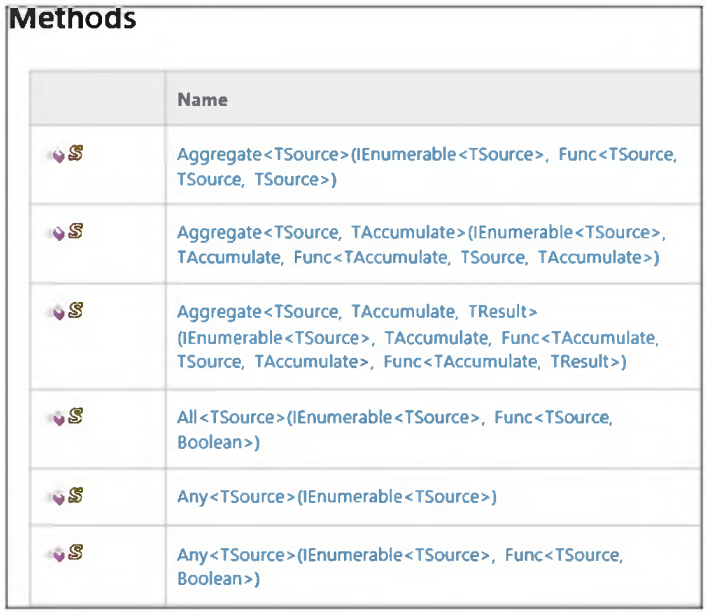
3- Bu sınıftaki metotların birkaç tanesinin imzasında "this" anahtar kelimesinden hemen sonra **IOrderedEnumerable** arayüzüne yer verilmiş olunsa bile metotların tamamına yakını IEnumerable arayüzüne yeni beceriler kazandırmaktadır. Bundan şu sonucu çıkarabiliriz: Enumerable sınıfındaki genişletme metotları sayesinde IEnumerable arayüzü, sorgulama yapılırken kullanılabilinecek çok sayıda metoda sahip olmuştur.

4- Yukarıda hazırladığımız sorguda "select" operatörünü kullanmıştık. Bu nedenle Enumerable sınıfındaki **Select**() genişletme metodunun imzasına yakından bakmakta fayda vardır. Bu metodun imzası aşağıdaki gibidir.

```
public static IEnumerable<TResult>
       Select<TSource, TResult>
              (this IEnumerable<TSource> source,
                     Func<TSource, TResult> selector);
```

Yukarıda hazırladığımız LINQ sorgusunda select'i kullanmışken burada **Select**() olarak geçmektedir. Bunun nedeni, hazırlanacak LINQ sorgu metinlerinin SQL sorgularına benzemesini sağlamaktır. Yoksa, derleyici geri planda derleme yaparken select operatörünü gördüğü yerde Select() metodunu işletmektedir.

Şimdiye kadar LINQ ilgili olarak anlatılanlardan şu sonucu çıkarmak gerekir: LINQ sorgularını anlamak ve etkin bir şekilde kullanmak bir bakıma **IEnumerable** arayüzünü anlamak demektir. Bu nedenle bu arayüzdeki metotları incelemekte fayda vardır. Aşağıda verilen ekran görüntüsünde bu arayüzde tanımlı genişletme metotlarının bazıları listelenmektedir.



Methods

	Name
S	Aggregate<TSource>(IEnumerable<TSource>, Func<TSource, TSource, TSource>)
S	Aggregate<TSource, TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate, TSource, TAccumulate>)
S	Aggregate<TSource, TAccumulate, TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate, TSource, TAccumulate>, Func<TAccumulate, TResult>)
S	All<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)
S	Any<TSource>(IEnumerable<TSource>)
S	Any<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)

Şimdi IEnumerable arayüzüyle ilgili önemli bir ayrıntıyı burada tekrar dile getireceğiz: LINQ sorgularında bu metotların yerine metotları temsil eden operatörler kullanılmaktadır. Başka bir deyişle LINQ kodu içinde select operatörü kullanıldığında derleyici geri planda aslında **Select**() metodunu kullanmaktadır. Bu nedenledir ki LINQ kodunu yazarken IEnumerable arayüzünün Enumerable sınıfı kaynaklı statik **Select**() metodu yerine "select" operatörünü kullandığımız için kodun using bloğunda "System.Linq" adlı namespace'te bulunan sınıflardan yararlanılmak istendiğini belirtmek gerekiyor.

Şimdiye kadar yaklaşık 10 sayfa boyunca LINQ'ten söz ettik. Yazılabilinecek en basit LINQ sorgusunu yazıp bir dizi değişkenin elemanlarını filtreleme yapmadan sorguladık. Ardından hazırlayıp işlettiğimiz sorgu üzerinde biraz konuştuk. Hazırladığımız tek satırlık sorgu hakkında 2 sayfa kadar açıklama yapmamıza rağmen kafalarda çok sayıda sorunun biriktiğini öngördüğümüz için konuyu temelden ele alıp bu sorulara cevap vermeye çalıştık. Bize göre, bu kitapta daha önce verilen genişletme metotları, anonim tipler, var anahtar kelimesi ve Collection Initializers hakkında bilgi içeren bölümü okuyanlar artık LINQ ile ilgili ayrıntıları öğrenmeye hazırdır.

Where Operatörü İle Filtreleme Yapmak

Yukarıdaki sayfalarda yazdığımız tek satırlık LINQ kodu sayesinde üzerinde çalıştığımız Windows Forms uygulamasının formundaki ListBox'ın bütün elemanlarını "cift_dizi" adını verdiğimiz ve int tipinde elemanlar içeren IEnumerable<> nesnesine aktarmıştık. Çünkü **select** deyiminden sonra herhangi bir şart ileri sürmedik, filtreleme yapmadık.

Şimdi yukarıdaki sayfalarda verilen koda ekleme yapıp "var" anahtar kelimesi ile tanımlayıp select deyimi ile yaptığımız sorgulama sonucu elde edilen bilgileri aktardığımız koleksiyonun elemanlarını formdaki RichTextBox'a aktaracağız. LINQ hakkında bilgi vermek amacıyla şimdiye kadar hazırladığımız kodun tamamı aşağıdadır.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
 {
  dizi[i] = Convert.ToInt16(listBox1.Items[i]);
 }
 var cift_dizi = from eleman in dizi select eleman;
 foreach (int str in cift_dizi)
 {
  richTextBox1.Text = richTextBox1.Text + str + "\n";
 }
}
```

Şimdi burada LINQ ile yapılan sorgulamanın üzerinde biraz duracağız. Amacımız konu üzerinde düşünmenizi sağlamaktır.

- Programın işletimi "select" operatörünü içeren LINQ koduna geldiğinde "in" anahtar kelimesinden sonra adı yazılan ve IEnumerable arayüzünü uygulamış koleksiyon nesnesinin bütün elemanları sırayla taranıp her turda ilgili eleman "from" operatöründen sonra adı yazılan değişkene aktarılır.
- Devamında bu değişkenin içeriği "where" ile yapılması muhtemel filtreye takılmaz ise adı select deyiminden sonra yazılan değişkene aktarılır. Bu değişkenin içeriği geriye gönderilip var anahtar kelimesi ile işaret edilen nesneye eklenir. Yani burada geri planda bir nevi foreach döngüsü hazırlanıyor.
- Burada "eleman" adını verdiğimiz değişken geri planda derleyici tarafından uygun tipte olacak şekilde tanımlanır. Konuya dikkatinizi çekmek için burada "eleman" adını verdiğimiz ve int tipinde bilgi aktarılacağını bildiğimiz değişkenin tipini klasik şekilde belirttik. Bu değişikliğe derleyicinin bir itirazı olmaz. Bize sorarsanız, bırakın bu değişkenin tipini derleyici tahmin etsin. Böylece yazacağınız LINQ kodu bir kelime kadar kısalmış olacaktır.

```
var cift_dizi = from int eleman in dizi select eleman;
```

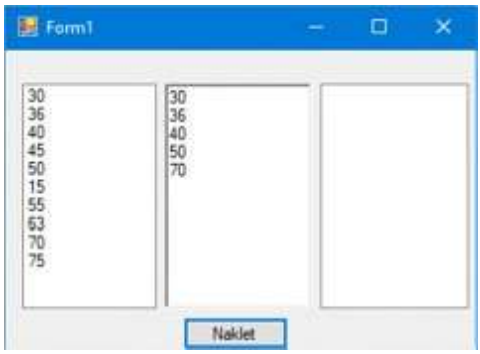
Madem "var" anahtar kelimesi ile tanımlayıp LINQ sorgusu ile bütün elemanları aktardığımız yeni koleksiyona "cift_dizi" adını verdik, sorgu satırında **where** deyimine yer verip bu koleksiyona yalnızca çift sayıların aktarılmasını sağlayacağız.

Select operatörü en sona yazılması gerektiği için select'ten önce **where** operatörüne yer verdik. Tahmin edeceğiniz gibi burada kullandığımız where operatörü SQL dilindeki WHERE deyiminden farklı bir işleve sahip değildir.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
  {
   dizi[i] = Convert.ToInt16(listBox1.Items[i]);
  }
```

```
        var cift_dizi = from eleman in dizi
                        where eleman % 2 == 0 select eleman;
        foreach (int str in cift_dizi)
        {
         richTextBox1.Text = richTextBox1.Text + str + "\n";
        }
    }
```

"dizi" adındaki koleksiyonun 2'ye tam bölünen elemanları ile ilgilendiğimiz için **where** deyimine "eleman % 2 == 0" karşılaştırmasını parametre olarak verdik. Yani "eleman" değişkenin içeriği 2'ye bölündüğünde sonuç 0 değilse filtreye takılır ve o eleman burada "cift_dizi" adını verdiğimiz IEnumerable nesnesine eklenmez. Dolayısıyla bu karşılaştırmada elenen elemanlar geriye gönderilmeyip yeni koleksiyona eklenmez. Bu şekilde düzenlenen uygulama çalıştırılıp bu kod işletilirse aşağıdaki gibi bir sonuç alınır.



Deferred Execution - Ertelenmiş Yürütme

Yukarıdaki sayfalarda ilk LINQ sorgumuzu yazıp istediğimiz sonucu aldık. Yukarıda verdiğimiz kodda LINQ sorgusunu hazırladıktan hemen sonra "cift_dizi" adını verdiğimiz IEnumerable nesnesinin içeriğine baktığımızda sadece çift sayıların olduğunu görürüz.

Tekrar niyetine burada hazırladığımız LINQ satırı üzerinde biraz konuşacağız. Konu kolay izlenebilinsin diye açıklamayı maddeler halinde vereceğiz:

- Sorgulamaya kaynaklık edecek koleksiyon "in" anahtar kelimesinden sonra yazılıyor. Burada hazırladığımız LINQ sorgusunda sorguya kaynaklık edecek bilgiler "dizi" adını verdiğimiz dizi değişkende bulunmaktadır. Bu bilgiler veritabanından alınmış olabilirdi.
- Sorgulama sırasında koleksiyonun ilk elemanından başlamak üzere bütün elemanları tek tek elde edilip "in" deyiminden önce yazılan değişkene aktarılıyor. Örneğin ilk turda dizi değişkenin ilk elemanı alınıp "eleman" adını verdiğimiz değişkene aktarılıyor. Bu değişkenin tipi derleyici tarafından belirleniyor.
- Koleksiyondan okunup "in" deyiminden önce adı yazılan değişkene aktarılan bilgi **where** deyimi ile işaret edilen kontrole tabi tutulur. Bu kontrolün sonucu true ise yani koleksiyondan okunup burada "eleman" adını verdiğimiz değişkenin içeriği

where ile belirtilen koşulu sağlıyorsa bu bilgi **select** deyiminden sonra yazılan değişkene aktarılır. Bizim burada hazırladığımız sorguda **in** deyiminden önce yazılan değişken ile select'ten sonra yazılan değişken aynıdır. Sorgulama sırasında yapılan işlemlere bağlı olarak bu 2 değişken farklı olabilmektedir.

- En son adımda select deyiminden sonra adı yazılan değişkenin içeriği geriye gönderilip sorguya ait eşitliğin sol tarafındaki koleksiyona aktarılıyor. Sorguya kaynaklık eden koleksiyonun ilgili elemanı where koşulunu sağlamadığında in deyiminden önce adı yazılan değişkenin içeriği select'ten sonra adı yazılan değişkene aktarılmaz. Dolayısıyla geriye bir şey gönderilmeyip koleksiyonun bir sonraki elemanın üzerine gidilir ve aynı işlemler bir sonraki eleman için yapılır.

LINQ ile sorgulama yaparken "in" anahtar kelimesinden sonra adını yazdığımız koleksiyondaki çift yerine tek sayılarla ilgilenseydik where ile yaptığımız karşılaştırmada değişiklik yapardık. Formdaki ilk ListBox'a girdiğimiz sayıların tek olanlarını 2. ListBox'a aktarmak üzere ikinci bir **select** sorgusu hazırladık.

```
private void Naklet_Click(object sender, EventArgs e)
{
 int adet = listBox1.Items.Count;
 int[] dizi = new int[adet];
 for (int i = 0; i < adet; i++)
  {
   dizi[i] = Convert.ToInt16(listBox1.Items[i]);
  }
 var cift_dizi = from eleman in dizi
         where eleman % 2 == 0 select eleman;
 foreach (int str in cift_dizi)
  {
   richTextBox1.Text = richTextBox1.Text + str + "\n";
  }
 var tek_dizi = from eleman in dizi
                where eleman % 2 != 0 select eleman;
 foreach (int str in tek_dizi)
  {
   listBox2.Items.Add(str);
  }
}
```

Her ne kadar yukarıda select sorgusu ile elde edilen bilgileri "var" deyimi ile tanımlanan koleksiyona aktardıysak ve var deyimini kullanmak programcılara kolaylık sağlıyor olsa bile sorgulama sonucu elde edilecek bilgilerin tipi biliniyorsa generic **IEnumerable<>** arayüzünü kullanmak mümkündür. Başka bir deyişle select operatörü ile yapılan 2 sorgulama aşağıdaki gibi yapılabilir.

```
IEnumerable<int> cift_dizi = from eleman in dizi
               where eleman % 2 == 0 select eleman;
IEnumerable<int> tek_dizi = from aza in dizi
               where aza % 2 != 0 select eleman;
```

Bu ek açıklamalardan sonra şimdi Deferred Execution - Ertelenmiş Yürütme adını verdiğimiz konu başlığı üzerinde biraz duracağız. Bu amaçla yukarıda verdiğimiz sorgunun benzerini tekrar vereceğiz. Bu kez sorgulamaya kaynaklık edecek dizi değişkene eleman ekleme işlemini doğrudan yaptık.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] dizi = { 30, 36, 45, 50, 15, 55, 63, 70, 75 };
 var tekler = from eleman in dizi
       where eleman % 2 != 0
              select eleman;
}
```

Sizin için herhangi bir yenilik içermeyen bu kod işletilip metodun sonuna ulaşıldığında burada "tekler" adını verdiğimiz IEnumerable<int> nesnesi henüz boştur. Çünkü burada LINQ sorgusunu hazırlamakla yetindik. Başka bir deyişle çalışma anında formdaki button1 düğmesi tıklanıp bu kod işletildiğinde bir LINQ sorgusu hazırlanır ancak bu sorgu işletilmeden metodun sonuna varılır. LINQ sorgularının bu özelliğine Deferred Execution - Ertelenmiş Yürütme adı verilmektedir. Hazırlanan LINQ sorgusundan ne zaman yararlanılır o zaman sorgu işletilir. Yukarıdaki sayfalarda foreach döngüsünü kullanarak sorgunun işletilmesine neden olmuştuk. LINQ sorgunu işletmek için aşağıda yapıldığı gibi sorgulama sonucu hazırlanan IEnumerable nesnesinin eleman sayısına bakılabilir.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] dizi = { 30, 36, 45, 50, 15, 55, 63, 70, 75 };
 var tekler = from eleman in dizi
       where eleman % 2 != 0
              select eleman;
 int sayi = tekler.Count();
}
```

LINQ Sorgusu İle Koleksiyondan Birden Fazla Alanı Çekmek

Yukarıda hazırladığımız LINQ sorgusunda select operatörü ile geriye gönderilip IEnumerable nesnesinde tutulan bilgilerin her elemanında sadece bir bilgi vardı. SQL deyimleri ile veritabanlarından çekilen kayıtlarda genellikle birden fazla bilgi (alan) olmaktadır. Burada bir veritabanına bağlanıp çekilen veriler üzerinde işlem yapmayı düşünmediğimiz için aşağıdaki gibi kod yazarak bir **Dictionary** nesnesi hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
}
```

Generic sürümünü seçtiğimiz Dictionary koleksiyonlarında anahtar-değer şeklinde düzenlenmiş bilgilerin tutulduğunu biliyorsunuz. C# ile gelen diğer koleksiyon sınıfları gibi Dictionary sınıfı IEnumerable arayüzünü uyguladığı için LINQ ile yapılacak sorgulamaya kaynaklık edebilir. Bu koleksiyonda sorgulama yapmak amacıyla koda ekleme yaptık.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from il in iller select il;
}
```

Bu metodun sonuna gelindiğinde elimizde, Dictionary nesnesinin bütün elemanları string tipte bilgi içeren bir IEnumerable nesnesi hazırlayacak bir sorgu vardır. Bu son cümlede ifade edildiği üzere bu metot işletilip metodun sonuna gelindiğinde burada "liste" adını verdiğimiz nesnenin içi boştur. Çünkü LINQ sorgusu ile hazırlanacak yeni koleksiyona ihtiyaç duyulup kullanılmadığı sürece sorgu işletilmez.

Şimdi hazırladığımız bu LINQ sorgusunu işletmek üzere foreach döngüsü hazırlayacağız. Döngü içinde burada "liste" adını verdiğimiz IEnumerable nesnesinden veri çekilmek istendiğinde sorgu işletilir.

```
private void button1_Click(object sender, EventArgs e)
{
  Dictionary<string, string> iller = new
              Dictionary<string, string>();
  iller.Add("01", "Adana");
  iller.Add("02", "Adıyaman");
  iller.Add("06", "Ankara");
  iller.Add("07", "Antalya");
  iller.Add("34", "İstanbul");
  iller.Add("35", "İzmir");
  var liste = from il in iller select il;
  foreach (var str in liste)
  {
   listBox1.Items.Add(str);
  }
}
```

Bu kod işletildiğinde select operatörü ile hazırlanacak IEnumerable nesnesinde, Dictionary nesnesinin anahtar ve değer bilgilerinin birleştirilmiş hali olur. Bu nedenle bu kod işletildiğinde aşağıdaki gibi bir sonuç alınır.



IEnumerable arayüzü kaynaklı metot ve özelliklerden yararlanıp anahtar ve değer bilgilerini bir birinden ayırmak mümkündür. Bu kod aşağıdaki gibi düzenlenirse istenen sonuç elde edilir. Çünkü LINQ ile yapılan sorgulama sonucu hazırlanan yeni koleksiyondaki her eleman anahtar-değer şeklinde düzenlenmiş bilgiler içermeye devam eder.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from il in iller select il;
 foreach (var str in liste)
  {
   listBox1.Items.Add(str.Key);
   listBox2.Items.Add(str.Value);
  }
}
```

Bu bölümde öncelikli amacımız LINQ hakkında adım adım bilgi vermek olduğu için LINQ ile sorgulama yapılan satırda değişiklik yapacağız. Ayrıca 2. bir LINQ sorgusu hazırlayacağız. Bu kez select operatöründen sonra geriye gönderilecek bilginin sadece anahtar ve sadece değer bilgilerinin olmasını sağlayacağız.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste1 = from plaka in iller select plaka.Key;
 var liste2 = from sehir in iller select sehir.Value;
```

```
        foreach (var str in liste1)
         {
           listBox1.Items.Add(str);
         }
        foreach (var str in liste2)
         {
          listBox2.Items.Add(str);
         }
      }
```

Bu LINQ sorgularında "iller" adını verdiğimiz koleksiyonun her elemanı "in" deyiminden önce adı yazılan yerel değişkene aktarılır. Ardından bu değişkenin içeriği select'ten sonra yazılan aynı adlı değişkene aktarılır. Çünkü bu 2 sorguda **where** deyimine yer vermediğimiz için koleksiyonun bütün elemanları sırayla okunup select'ten sonra adı yazılan değişkene aktarılır. Select'ten sonra kullanılan değişkenin adının yanına "Key" yazıldığı için geriye değişkenin bütün içeriği yerine sadece **Key** özelliğinin içeriği gönderilir. Dolayısıyla sorgulama sonucu elde edilen ilk koleksiyonda burada sorgulamaya kaynaklık eden **Dictionary** nesnesinin sadece key bilgileri olacaktır.

Benzer şekilde 2. LINQ sorgusunda select operatöründen sonra "sehir.Value" yazdığımız için sorguya kaynaklık eden Dictionary koleksiyonundaki bütün elemanların değer bilgileri geriye gönderilir. Dolayısıyla sorgulama sonucu elde edilen 2. koleksiyonda sadece değer bilgileri olur. Bu şekilde LINQ sorgusu sonucu hazırlanacak koleksiyonda yer alacak bilgileri select operatörü ile nasıl seçildiğini görmüş olduk.

Konunun arka planını iyice ortaya çıkarmak için şimdi yukarıda verdiğimiz 2 LINQ sorgusundan 2. sinde değişiklik yapacağız. Yapacağımız değişiklik sayesinde sorgulama sonucu elde edilecek koleksiyondaki il adlarının büyük harfe çevrilmesini sağlayacağız. Bu işlemi yapmadan önce yukarıda verdiğimiz 2. sorgu üzerinde biraz konuşacağız.

Bu sorguda "iller " adını verdiğimiz Dictionary koleksiyonunda alınan her eleman anahtar ve değer bilgileri birlikte önce "sehir" adını verdiğimiz yerel değişkene aktarılmaktadır. Yine hatırlatmak gerekirse burada "sehir" adını verdiğimiz değişkenin tipini seçmeyi derleyiciye bıraktık. Bu şartlarda "sehir" adını verdiğimiz değişkene aktarılan nesnede "anahtar-değer" şeklinde düzenlenmiş bilgi bulunmakta ve select deyiminden sonra "sehir.Value" yazılarak bu bilginin sadece "değer" kısmının geriye gönderilmesi sağlanmaktadır.

Konunun özeti şudur: Sorgulamaya kaynaklık eden koleksiyondan okunan eleman ile, geriye gönderilen eleman bir birinden farklı olabilmektedir. Çünkü burada "in" deyiminden önce yazılan değişken ile, select'ten sonra yazılan değişken farklıdır. Sorgulama sonucu elde edilecek yeni koleksiyondaki il adlarının büyük harfe çevirmek için ikinci LINQ sorgusunda yaptığımız değişikliği aşağıda görebilirsiniz.

```
private void button1_Click(object sender, EventArgs e)
{
  Dictionary<string, string> iller = new
              Dictionary<string, string>();
  iller.Add("01", "Adana");
```

```
        iller.Add("02", "Adıyaman");
        iller.Add("06", "Ankara");
        iller.Add("07", "Antalya");
        iller.Add("34", "İstanbul");
        iller.Add("35", "İzmir");
        var liste1 = from plaka in iller select plaka.Key;
        var liste2 = from sehir in iller
              let buyukHarf = sehir.Value.ToUpper()
                    select buyukHarf;
        foreach (var str in liste1)
        {
          listBox1.Items.Add(str);
        }
        foreach (var str in liste2)
        {
         listBox2.Items.Add(str);
        }
    }
```

Burada kullanılan "let" deyimi sayesinde ilgili koleksiyondan okunup where deyimi ile (gerekirse) filtre edilecek ve select deyimi ile geriye gönderilip yeni koleksiyona eleman olarak yazılacak bilgi artık "buyukHarf" adını verdiğimiz yerel değişkendedir. LINQ ile yapılan sorgu ifadelerinde kullanılan değişkenler için tip seçimi genellikle derleyiciye bırakıldığı için burada "buyukHarf" adını verdiğimiz değişken için tip ayarlaması yapmadık.

Dictionary koleksiyondan okunan ilk ilin adını büyük harfe çevirirken String sınıfının ToUpper() metodundan yararlandık. Söz konusu il adının büyük harfe çevrilmiş halini geriye göndermek istediğimiz için select'ten sonra bu değişkenin adını yazdık.

"select" operatörünün sağ tarafına yazılan "let" operatörü sayesinde tanımladığımız değişken yerine yine "sehir.Value" yazsaydık sorgulama sonucu hazırlanacak yeni koleksiyonda şehir adları büyük harfe çevrilmiş halleri olmazdı. Başka bir deyişle yukarıda hazırladığımız 2. LINQ sorgusunu aşağıdaki gibi düzenleseydik ToUpper() metodunun bir işlevi olmazdı.

```
var liste2 = from sehir in iller
     let buyukHarf = sehir.Value.ToUpper()
           select sehir.Value;
```

Buradaki amacımız sizlere "let" operatöründen söz etmektir. Yoksa let operatörüne yer vermeden select operatöründen sonra yazılan string tipteki değişkene **ToUpper**() metodunu uygulayıp aynı sonucu almak mümkündür.

```
var liste2 = from sehir in iller
        select sehir.Value.ToUpper();
```

LINQ Sorgularında İsimsiz Tipleri Kullanmak

Sorgulama sonucu select operatörüyle geriye birden fazla bilgi gönderilme ihtiyacı olduğunda isimsiz tiplerden yararlanılmaktadır. İsimsiz tiplerin LINQ sorguları dahilinde nasıl kullanıldığını anlatmak için yukarıda verilen LINQ sorgusunu aşağıdaki gibi düzenledik.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
       Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from sehir in iller select new { sehir };
 foreach (var str in liste)
  {
   listBox1.Items.Add(str.sehir.Key);
   listBox2.Items.Add(str.sehir.Value);
  }
}
```

Bu kod işlev bakımından yukarıda verilenlerden farklı değildir. LINQ ile sorgulama yaparken isimsiz tip tanımlamak istiyorsanız select deyiminden sonra **new** anahtar kelimesine yer vermeniz gerekir. new'den sonra yazılacak olanları süslü parantezler ile bloklamak gerekiyor. Bu LINQ sorgusunda select operatöründen sonra sadece bir bilgi yazdığımız için new anahtar kelimesini kullanmak zorunda değiliz. Yazdığımız LINQ sorgusunda select'ten sonra birden fazla değişken veya nesne adı olsaydı new'i kullanmak zorunlu olurdu.

Yazdığımız bu kodun yeni hali işletildiğinde **select** ile yapılan sorgulamada, "in" kelimesinden sonra adı yazılan koleksiyonun her elemanı için isimsiz tipin örneği alınıp geriye gönderilir. Başka bir deyişle tıpkı foreach döngüsünde olduğu gibi "in" deyiminden sonra adı yazılan bu koleksiyonun üzerinde ileri doğru hareket edilerek şartları uyan elemanlar için nesne hazırlanıp geriye gönderilir. Hazırlanan nesnenin kaynağı, new anahtar kelimesinden sonra süslü parantezlerin içine yazılanlardan yola çıkılarak hazırlanan isimsiz tiptir.

Konunun ayrıntıları ortaya çıksın diye yukarıda verdiğimiz LINQ sorgusunda sadece söz konusu şehrin plakasının yani Dictionary koleksiyondaki bilgilerin sadece anahtarın geriye gönderilmesini sağlayacağız. Bu amaçla select operatöründen sonra new anahtar kelimesiyle hazırladığımız isimsiz tipi aşağıdaki gibi düzenledik. Farklılık olsun diye projenin formuna ayrıca bir DataGridView nesnesi yerleştirip **select** operatörünün geriye gönderdiği koleksiyonun elemanlarını DataGridView'de listeledik.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
       Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from sehir in iller select new
 {
  sehir.Key
 };
```

```
    var anahtarlar = liste.ToList();
    dataGridView1.DataSource = anahtarlar;
}
```

Bu şekilde düzenlenen bu kod işletildiğinde Dictionary nesnesindeki elemanların anahtar bilgileri DataGridView'de listelenir. LINQ sorgusu sonucu hazırlanan IEnumerable nesnesinin içeriğini DataGridView'de listeleyebilmek için **ToList**() metodundan yararlandık. Bu şekilde kullanılan ToList() metodu geriye List nesnesi göndermektedir.

LINQ sorgularında select operatörü ile geriye birden fazla bilgi gönderiliyorsa isimsiz tiplerden yararlanmak zorundasınız. İsimsiz tipin hazırlandığı süslü parantezin arasına birden fazla değişken adı yazılırsa isimsiz tip birden fazla alana sahip olmuş olur. Aşağıda verdiğimiz LINQ sorgusunda Dictionary koleksiyondan okunan elemanın hem anahtar hem de diğer bilgilerin: new anahtar kelimesine ait süslü parantezlerin içine yazdığımız için yeni koleksiyonda bu 2 bilgi olur.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from sehir in iller
      select new {sehir.Key, sehir.Value };
 dataGridView1.DataSource = liste.ToList();
}
```

Türkiye'de ilk 67 ilin adları söz konusu olduğunda ister ada, ister plakaya göre sıralama yapılsın aynı sonuç alınır. Bu rağmen **orderby** operatöründen söz etmek için hazırladığımız sorguyu aşağıdaki gibi düzenledik.

Yukarıdaki sayfalarda LINQ ile sorgulama yaparken where operatörünü kullanarak select ile geriye gönderilecek verileri filtre etmiştik. "where" operatörünü hatırlatmak için yukarıda verdiğimiz LINQ sorgusunda değişiklik yapıp where operatörüne yer verdik.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string, string> iller = new
              Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from sehir in iller
    where sehir.Key[0] == '0'
```

```
            select new
            {
              sehir.Key, sehir.Value
            };
     dataGridView1.DataSource = liste.ToList();
    }
```

Tahmin edeceğiniz gibi bu kod işletildiğinde plakasının ilk hanesi 0 olan iller DataGridView nesnesinde listelenir. Burada where operatörü ile Dictionary nesnesinin ilgili elemanın anahtar bilgisinin ilk karakterinin "0" olup olmadığını araştırdık.

```
private void button1_Click(object sender, EventArgs e)
{
 Dictionary<string,string> iller =new Dictionary<string, string>();
 iller.Add("01", "Adana");
 iller.Add("02", "Adıyaman");
 iller.Add("06", "Ankara");
 iller.Add("07", "Antalya");
 iller.Add("34", "İstanbul");
 iller.Add("35", "İzmir");
 var liste = from sehir in iller
     orderby sehir.Value
        select new { sehir.Key, sehir.Value };
 var anahtarlar = liste.ToList();
 dataGridView1.DataSource = anahtarlar;
}
```

Bu LINQ sorgusunda orderby operatöründen sonra "sehir.Value" yazdığımız için sorgumuza kaynaklık eden Dictionary nesnesin çekilen elemanlar değer bilgilerine göre sıralanırlar. Sıralamanın a'dan z'ye yerine z'den a'ya (büyükten küçüğe doğru) doğru yapılmasını isteseydik **descending** anahtar kelimesini aşağıdaki gibi kullanırdık.

```
var liste = from sehir in iller
   orderby sehir.Value descending
        select new { sehir.Key, sehir.Value };
```

LINQ İle Dosyaları Listelemek

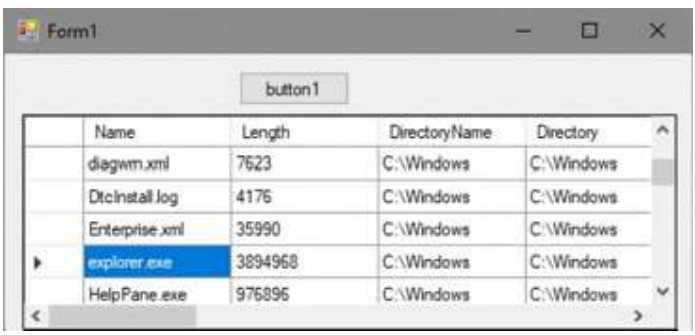
Yukarıdaki en basit şekli LINQ sorgularını nasıl hazırlayabileceğinizden söz ettik. Şimdi farklı bir örnek olması için diskimizde bulunan dosyaları LINQ yardımıyla istediğimiz gibi filtreleyip sıralayabileceğiz. Bu amaçla ilk olarak aşağıdaki gibi bir **DirectoryInfo** nesnesi ve bu nesneden yola çıkıp bir **FileInfo** nesnesi hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 DirectoryInfo klasor = new DirectoryInfo("C:\\Windows");
 FileInfo[] Dosyalar = klasor.GetFiles();
}
```

Bu 2 satır sayesinde "C:" sürücüsünün "\Windows" klasöründe bulunan her dosya için bir FileInfo nesnesi hazırlanır ve FileInfo tipindeki dizi değişkende toplanır. Her FileInfo nesnesinde ilgili dosyayla ilgili çok sayıda bilgi bulunmaktadır. Buradaki amacımız LINQ hakkında bilgi vermek olduğu için aşağıdaki gibi bir sorgu satırı hazırladık.

```
private void button1_Click(object sender, EventArgs e)
{
 DirectoryInfo klasor = new DirectoryInfo("C:\\Windows");
 FileInfo[] Dosyalar = klasor.GetFiles();
 var liste = from file in Dosyalar select file;
 dataGridView1.DataSource = liste.ToList();
}
```

Bu LINQ sorgusu işletilirse ilgili klasördeki bütün dosyalar DataGridView'de listelenir. Aşağıda verilen ekran görüntüsünü bu metodu işlettikten sonra aldık.



Bildiğiniz gibi dosyaların adları, büyüklükleri, oluşturulma tarihleri ve bulundukları yerden başka çok sayıda özelliği bulunmaktadır. DataGridView'de her dosyayla ilgili bütün bilgiler yerine sadece adlarının, oluşturulma tarihlerinin ve byte olarak büyüklükleriyle ilgilendiğimiz için hazırladığımız sorguda isimsiz tipe yer verdik.

```
private void button1_Click(object sender, EventArgs e)
{
  DirectoryInfo klasor = new DirectoryInfo("C:\\Windows");
  FileInfo[] Dosyalar = klasor.GetFiles();
  var liste = from file in Dosyalar
      select new
         { file.Name, file.CreationTime, file.Length };
  dataGridView1.DataSource = liste.ToList();
}
```

Bu şekilde düzenlenen sorgu işletildiğinde DataGridView'de her dosyayla ilgili olarak sadece dosya adı, hazırlanma tarihi ve büyüklüğü listelenir. Eğer sadece EXE uzantılı dosyalarla ilgileniyor olsaydık LINQ sorgusunu aşağıdaki gibi düzenlerdik.

```
private void button1_Click(object sender, EventArgs e)
{
 DirectoryInfo klasor = new DirectoryInfo("C:\\Windows");
 FileInfo[] Dosyalar = klasor.GetFiles();
 var liste = from file in Dosyalar
     where file.Extension == ".exe"
        select new
           { file.Name, file.CreationTime, file.Length };
 dataGridView1.DataSource = liste.ToList();
}
```

27. BÖLÜM

LAMBDA İFADELERİ

C# programlama dilini anlatma kaygısıyla hazırlanmış kaynaklarda lambda operatörü genellikle zor anlaşılan bir başka konuyla birlikte anlatılmaktadır. Daha açık bir söyleyişle LINQ sorguları konusunda kafasında çok sayıda soru bulunan birisine LINQ ile birlikte lambda operatörü anlatılmaktadır. Bu sırada okur, var anahtar kelimesi, isimsiz metotlar ve delegate'ler hakkında yeterince bilgili değilse anlatılanları anlaması zorlaşmaktadır.

Bu kitapta önce LINQ'den bağımsız olarak "var" anahtar kelimesini, isimsiz tipleri, isimsiz metotları, genişletme metotlarını, Object Initializer yeniliğini bütün ayrıntıları ile anlattık. Okurumuzun bunları iyice kavradığına emin olduktan sonra sıra LINQ sorgularına geldi. LINQ ile koleksiyon şeklinde düzenlenmiş verilerin nasıl sorgulandığını lambda operatörünü kullanmadan anlattıktan sonra nihayet sıra lambda'ya geldi. Lambda operatörlerini anlatırken aynı yöntemi uygulayıp örnek olarak basit işlemleri seçeceğiz ve lamda ifadelerini LINQ'ten bağımsız olarak anlatacağız.

En başında belirtmek isteriz ki C# uygulaması dahilinde LINQ sorgularını hazırlarken lambda operatörlerini kullanmak zorunda değilsiniz. Çünkü lambda operatörünü kullanıp lambda ifadeleriyle yapılan işlemleri şimdiye kadar öğrendiklerinizden yola çıkarak yapmanız mümkündür. C# programcıları arasında yaygın görüşe göre lambda ifadeleri sayesinde kodda kısalma olmaktadır. Size önerimiz; bu bölümde lambda operatörüne yer verilen ifadeler hakkında yazılanları dikkatlice okuduktan sonra kullanıp kullanmama konusundaki tercihinizi öyle yapmanız şeklindedir.

Lambda ifadelerinin nasıl yazılıp kullanıldığını anlatmadan önce delegate tiplerinin nasıl hazırlanıp kullanıldığını hatırlatacağız. Bu amaçla Visual Studio ile hazırladığımız konsol uygulamasında Main() metodunun olduğu sınıfta aşağıdaki gibi bir delegate tanımladık.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
  {
  }
}
```



Bu kitabın delagate'lerin anlatıldığı bölümünü henüz okumadıysanız lambda'dan önce delegate'lerle ilgilenmeniz önerilir. Ayrıca lambda ifadelerinin anlatıldığı bu bölümü okumadan önce bu kitabın isimsiz metotlarla ilgili sayfalarını okumanız önerilir. Çünkü lambda ifadelerine kısaltılmış isimsiz metotlar gözüyle bakılabilir.

Class bloğunda yazılan bu satır sayesinde geriye int tipinde değer döndüren ve int tipinde 2 parametreye sahip metotları bu delegate tipi sayesinde işletebiliriz. delegate anahtar kelimesi kullanılarak yapılan bu tip tanımlama işlemini class bloğu yerine namespace bloğu içinde yapabilirdik Bu delegate tipi ile işletebilmek amacıyla aşağıdaki gibi kendisine parametre olarak verilen 2 sayıyı çarpıp sonucu geriye gönderen bir metot hazırladık.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static int islem(int x, int y)
 {
  return x * y;
 }
 static void Main(string[] args)
  {
  }
}
```

Bu delegate aracılığıyla islem() adını verdiğimiz metodu işletmek istediğimizde "Temsilci" tipinde yani kendi hazırladığımız delegate tipinde bir nesne hazırlamamız gerektiğini biliyorsunuz. Bu işlemi yapıp sonra da metodu bu delegate nesnesi aracılığıyla nasıl işlettiğimizi aşağıda görebilirsiniz.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static int islem(int x, int y)
 {
  return x * y;
 }
 static void Main(string[] args)
 {
  var T = new Temsilci(islem);
  int sonuc = T(6, 7); // islem() metodu işletilir
  Console.WriteLine(sonuc);
 }
}
```

Bu bölümde şimdiye kadar yazdıklarımız içinde sizin için yeni bir şey yoktur. Delegate'lerin anlatıldığı bölümde buradan farklı olarak delegate nesnesi hazırlarken "var" anahtar kelimesini kullanmamıştık. Konu birazdan Lambda Expression'lara evrileceğinden var anahtar kelimesini tercih ettik.

Daha önce isimsiz metotlar hakkında yazılanlardan hatırlayacağınız gibi burada "islem()" adını verdiğimiz metodu başka yerde kullanmayı düşünmediğimiz için delegate nesnesi hazırlanan satır ile metodun kodlandığı satırları birleştirebiliriz. Bunun için ilk olarak yukarıda "islem" adını verdiğimiz metodu sildik ve uygulamamızdaki Main() metodunu aşağıdaki gibi düzenledik.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
  Temsilci T = delegate (int x, int y)
   {
    return (x * y);
   };
```

```
    int sonuc = T(10, 20);
    Console.WriteLine(sonuc);
  }
}
```

Burada yapılanlar arasında sizin için yeni bir şey yoktur. Çünkü burada yaptığımız, normal metodu isimsiz metot olarak düzenlemekten ibarettir. Bunu yaparken elimize arı olarak geçen tek şey kodun bir satır kadar kısalmasıdır.

Kodumuzun son durumunu özetlersek, elimizde int tipinde 2 parametreye sahip olup geriye int tipinde değer döndüren metotları temsil edebilecek bir delegate tipı var. Yukarıda bu delegate tipindeki nesnenin önce normal metodu, ardından isimsiz bir metodu temsil etmesini sağladık

Lambda ifadelerinin en temel işlevleri kod kısaltmak olduğuna göre burada yazdığımız kodu biraz daha kısaltmaya çalışacağız. Bunu yaparken lambda operatöründen yararlanacağız. Bu amaçla yukarıda verilen kodun isimsiz metotla ilgili kısmını sildik.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
   Temsilci T =
 }
}
```

Buradaki amacımız elimizdeki delegate'in temsil edebileceği herhangi bir metodun yapabileceği işlemi metot hazırlamadan yapabilmektir. Yani bizi isimsiz metot kesmedi, biraz daha az kod yazmaya çalışacağız.

Bu nedenle işlem yaparken yararlanacağımız delegate nesnesinin aktarılacağı değişkeni tanımladık ve "=" işaretini yazdık. Eşitliğin sağ tarafına ilk olarak üzerine işlem yapılacak bilgileri yani parametreleri yazmamız gerekiyor.

```
class Program
{
  delegate int Temsilci(int X, int Y);
  static void Main(string[] args)
   {
    Temsilci T = (int sayi1, int sayi2)
   }
}
```

Elimizdeki delegate tipi, int tipinde 2 parametreye sahip olduğu için sanki metot parametresi yazıyormuş gibi 2 değişken tanımladık. Şimdi sıra bu int değişkenler yani parametreler üzerinde işlem yapmaya geldi. Lambda ifadelerinde parametrelerle, işlem yapılan kısmı birbirinden ayırmak için lambda operatörü (=>) kullanılmaktadır. Bu nedenle satırımıza bu operatörü ekledik.

```
class Program
 {
  delegate int Temsilci(int X, int Y);
  static void Main(string[] args)
  {
   Temsilci T = (int sayi1, int sayi2) =>
  }
}
```

Şimdi sırada normal metodun gövdesi içinde yapılan işlemleri yapmak var. Yukarıda yazdığımız normal metotta, metodun parametrelerini birbirleri ile çarpıp çıkan sonucu geriye göndermiştik. Bu nedenle lambda operatörünün sağında süslü parantezleri yer verip aşağıdaki gibi düzenledik.

```
class Program
{
  delegate int Temsilci(int X, int Y);
  static void Main(string[] args)
  {
   Temsilci T = (int sayi1, int sayi2) =>
    {
     return sayi1 * sayi2;
    };
 }
}
```

Bu kodun yukarıda isimsiz metot kullanarak hazırladığımız koddan en önemli farkı **delegate** anahtar kelimesini silmiş olmamızdır. Bu şartlarda burada "T" adını verdiğimiz delegate nesnesinden yararlanıp 2 sayıyı birbirleri ile çarpabiliriz. Bu lambda ifadesinden nasıl yararlandığımızı aşağıda görebilirsiniz.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
  {
   Temsilci T = (int sayi1, int sayi2) =>
    {
     return sayi1 * sayi2;
    };
    int sonuc = T(20, 30);
    Console.WriteLine(sonuc);
  }
}
```

Bu lambda ifadesinde lambda operatöründen sonra yazılan ve normal metotlardaki gövdeyi temsil eden kısmı süslü parantezlerin içine aldık. Burada lambda operatöründen sonra yazdıklarımız kısa olduğu için süslü parantezler kullanılmayabilir. Bu nedenle bu kod aşağıdaki gibi düzenlenirse değişen bir şey olmaması gerekir diye düşünülebilir.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
 {
  Temsilci T = (int sayi1, int sayi2) => return (sayi1 * sayi2);
  int sonuc = T(20, 30);
  Console.WriteLine(sonuc);
 }
}
```

İlk bakışta normal görünen bu kod hataya neden olur. Çünkü lambda operatörünün sağ tarafında yapılan işlemin sonucu zaten geriye gönderilmektedir. Yani burada **return** anahtar kelimesini kullanmaya gerek yoktur. Bu nedenle yazdığımız lambda ifadesinde return anahtar kelimesini sildik. Ayrıca yapılan işlem üzerinde bir etkisi olmayacağı için ifadenin sağ tarafındaki parantezleri kaldırdık.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
  {
   Temsilci T = (int sayi1, int sayi2) => sayi1 * sayi2;
   int sonuc = T(20, 30);
   Console.WriteLine(sonuc);
  }
}
```

Hazırladığımız lambda ifadesinde int tipinde 2 parametre bulunmakta ve ifadede bu parametrelerin tipi işaret edilmektedir. Gerçekte derleyici delegate'e bakıp parametrelerin tipini bilebilir. Bu nedenle parametrelerin tipini işaret eden anahtar kelimeler silinebilir.

```
class Program
{
 delegate int Temsilci(int X, int Y);
 static void Main(string[] args)
  {
   Temsilci T = (sayi1, sayi2) => sayi1 * sayi2;
   int sonuc = T(20, 30);
   Console.WriteLine(sonuc);
  }
}
```

Gördüğünüz gibi yazdığımız kod oldukça kısaldı. Bu lambda ifadesini yazarken yararlandığımız delegate tipi 2 parametreye sahiptir. Lambda ifadelerinin bir özelliğinden söz etmek için mevcut delegate'te değişiklik yaptık. Delegate'in yeni halinde int tipinde bir parametre bulunmaktadır.

```
class Program
{
 delegate int Temsilci(int X);
```

```
 static void Main(string[] args)
 {
   Temsilci T = (sayi) => sayi * sayi;
   int sonuc = T(30);
   Console.WriteLine(sonuc);
 }
}
```

Lambda ifademizin yeni halinde sadece bir parametre olduğu için lambda operatörünün sol tarafına yazılan parametrenin parantez içine alınması şart değildir. Bu nedenle bu kodda parametrenin içine yazıldığı parantezler silinirse değişen bir şey olmaz.

```
class Program
{
 delegate int Temsilci(int X);
 static void Main(string[] args)
 {
  Temsilci T = sayi => sayi * sayi;
  int sonuc = T(30);
  Console.WriteLine(sonuc);
 }
}
```

Yukarıda işaret edildiği gibi lambda ifadelerinde Lambda operatörünün sağına yazılanlar birden fazla satırdan meydana gelmiyorsa süslü parantez kullanmaya gerek yoktur. Konu üzerinde düşünmenizi sağlamak için bu lambda ifadesinin satır sayısını artırmayı denedik. Yapılan işlemler gereği satır sayısını arttırınca süslü parantezlere ve return anahtar kelimesine yer vermemiz gerekti. Lambda ifadesinin bu yeni halinde tanımlanan 2 değişken delegate kaynaklı olmadığı için tiplerini belirttik.

```
class Program
{
  delegate int Temsilci(int X);
  static void Main(string[] args)
   {
    Temsilci T = sayi =>
    {
     int kareler = sayi * sayi;
     int kupler = sayi * sayi * sayi;
     return kareler + kupler;
   };
   int sonuc = T(30);
   Console.WriteLine(sonuc);
  }
}
```

Konunun iyice anlaşılması için benzer bir örnek daha yapacağız. Bu amaçla üzerinde çalıştığımız projede aşağıdaki gibi bir delegate tipi tanımladık. String tipte bir parametreye sahip ve geriye değer göndermeyen metotları temsil edebilecek bu delegate'ten yararlanıp lambda ifadesi hazırlayacağız.

```
public class Program
{
 public delegate void Mesajci(string msj);
 public static void Main(string[] args)
  {
    Console.ReadLine();
  }
}
```

Yukarıda yaptığımız gibi bu delegate tipine uygun isimsiz bir metot hazırlayıp kullanacağız. İsimsiz metodu nasıl hazırlayıp kullandığımızı aşağıda görebilirsiniz. Yazdığımız isimsiz metot kendisine parametre olarak verilen string bilgiyi konsola yazmakla yetinmektedir.

```
public class Program
{
 public delegate void Mesajci(string msj);
 public static void Main(string[] args)
 {
  Mesajci Tem = delegate (string str)
   {
     Console.WriteLine(str);
   };
  Tem("Adana Demirspor");
 }
}
```

Delegate'leri ve isimsiz metotları bilenler için burada orijinal bir işlem yapılmıyor. Şimdi bu isimsiz metodu lambda ifadesi olarak düzenleyeceğiz. Bu amaçla delegate nesnesi hazırlanan satırda eşitliğin sağ tarafını silip lambda operatörünü yazdık.

```
public class Program
{
  public delegate void Mesajci(string msj);
  public static void Main(string[] args)
  {
   Mesajci Tem =  =>
  }
}
```

Lambda operatörünün soluna, sildiğimiz isimsiz metodun parametrelerini yazmak gerektiği için üzerinde çalıştığımız uygulamanın Main() metodunu aşağıda verilen şekle dönüştürdük.

```
public static void Main(string[] args)
{
  Mesajci Tem = (string str) =>
}
```

Lambda operatörünün sağ tarafına ise sildiğimiz isimsiz metodun gövdesini temsil eden ifadeleri yazmak gerektiği için Main() metodumuz aşağıda verilen şekle dönüştü.

```
public static void Main(string[] args)
{
 Mesajci Tem = (string str)  => Console.WriteLine(str);
}
```

Bu andan itibaren elimizdeki delegate nesnesinden (burada Tem adını verdiğimiz) yararlanıp lambda ifadesini işletebiliriz. Lambda ifadesinde operatörün sağına yazdığımız parametrenin tipini belirtmek zorunda değilsiniz. Çünkü derleyici yararlandığımız delegate tipine bakıp ona göre tip seçimini yapacaktır. Bu nedenle bu lambda ifadesi aşağıdaki gibi düzenlenebilir.

```
public class Program
{
  public delegate void Mesajci(string msj);
  public static void Main(string[] args)
  {
   Mesajci Tem = (str)  => Console.WriteLine(str);
   Tem("Adana Demirspor");
  }
}
```

Burada hazırladığımız lambda ifademiz sadece bir parametreye sahip olduğu için operatörün sol tarafında parantezlere yer verilmeyebilir. Bu nedenle lambda ifademizi aşağıdaki gibi düzenleyebiliriz.

```
public delegate void Mesajci(string msj);
public static void Main(string[] args)
{
 Mesajci Tem = str => Console.WriteLine(str);
 Tem("Adana Demirspor");
}
```

Şimdiye kadar lambda ifadeleri hakkında yazılanları özetlemek gerekirse şunlar söylenebilir: Yapılmak istenen işleme uygun hazır bir delegate varsa ve yazacağınız metodu tekrar kullanmayı düşünmüyorsanız isimsiz metot imkanından yararlanıp delegate ile metot gövdesini bir araya getirebiliyoruz. Bu sayede hem az daha kod yazmış oluyoruz hem de bir daha kullanmayacağımız bir metodumuz olmamış oluyor.

İsimsiz metot yazarken .NET Framework ile hazır olarak gelen delegate'lerden yararlanabiliyoruz. İsimsiz metodun daha sadeleştirilmiş ve kısaltılmış halini kullanmak istediğinizde ise lambda operatörünü içeren ifadeler yazabilirsiniz. Şimdi yukarıda verdiğimiz lambda ifadesinde kendi hazırlayıp "Mesajci" adını verdiğimiz delegate yerine .NET Framework ile gelen **Action** adlı delegate tipinden yararlanacağız.

```
public class Program
{
 public static void Main(string[] args)
  {
   Action<string> islem = str => Console.WriteLine(str);
   islem("Adana Demirspor");
  }
}
```

Gördüğünüz gibi yapmak istediğiniz işleme uygun bir delegate tipi olduğunda yeni bir metot hazırlamak zorunda kalmadan bulunulan yerde lambda ifadesi hazırlayıp kullanabiliyoruz.

LINQ Sorgularında Metotları Kullanmak

Bu kitapta hedefimiz ilk okumada anlaşılmak olduğu için LINQ ile ilgili olarak anlatmak istediklerimizi 2 parçaya ayırdık. Çünkü LINQ sorgularını hazırlarken başvurulacak birden fazla teknik vardır. Bir teknikte yeteri bilgi edinip yazılanlar tam olarak kavranmadan ikinci teknik hakkında bilgi verildiğinde her ikisi de anlaşılmıyor.

Bu kitapta LINQ sorgularında lambda ifadelerinden yararlanmadan önce LINQ'ten bağımsız olarak lambda ifadelerinden söz edildi. Bu kitabı atlamadan okuyan birisi bu sayfaya geldiğinde hem LINQ hem de lambda ifadeleri hakkında az çok bilgiye sahiptir. Bu nedenle LINQ sorgularında artık lambda ifadelerini kullanabiliriz. LINQ adlı bölümde anlatılanları hatırlatmak amacıyla aşağıda verilen kodu yazdık.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] yillar = new int[] {2011, 2012, 2013, 2014, 2015, 2016, 2017};
 var cift_yil = from eleman in yillar
     where eleman % 2 == 0
         select eleman;
 List < int > liste = cift_yil.ToList();
 foreach (int yil in liste)
 {
  listBox1.Items.Add(yil.ToString());
 }
}
```

Bu kodda sizin için yeni bir şey yoktur. "yillar" adını verdiğimiz dizi değişkenin çift sayı içeren elemanlarını LINQ sorgusu ile elde edip "cift_yil" adını verdiğimiz dizi değişkene aktardık. Sonra bu dizi değişkenden yola çıkıp ToList() metodu ile List nesnesi hazırladık. Burada List nesnesini hazırlama amacımız List nesnesine olan ihtiyacımızdan değildir. Amacımız, kodun işletimi ToList() metodunun olduğu satıra veya bu satır olmasaydı foreach döngüsünün olduğu satıra gelmeden işletilmemesidir. Buna Deferred Execution - Ertelenmiş Yürütme denildiğini biliyorsunuz. En son olarak list nesnesinin içeriğini ListBox'a aktardık.

Daha önceki konulardan bildiğiniz gibi yukarıda verilen LINQ sorgusunda kullanılan select ve where birer LINQ operatörüdür. Bu operatörlerin her birisi bir genişletme metoduna karşılık gelmektedir. Tekrar etmek gerekirse, C# 3.0'dan itibaren Enumerable sınıfı dahilinde çok sayıda genişleme metodu hazırlanıp IEnumerable<> arayüzüne ek beceriler kazandırılmıştır. Örnek vermek gerekirse IEnumerable<> arayüzünün Where() adında bir metodu bulunmaktadır. Bu metodun imzasını aşağıda görebilirsiniz. Aşağıda verilen orijinal yapıda mutlaka görmenizi istediğimiz ayrıntı şudur: Where() metodu geriye IEnumerable<> nesnesi göndermektedir.

```
public static IEnumerable<TSource> Where<TSource>
        (this IEnumerable<TSource> source,
       Func<TSource, bool> predicate);
```

Programcılar sorgu ifadeleri içinde bu metodu kullanmak istediklerinde Where() metodunu temsil eden operatörü kullanmak durumundalar. Çünkü C# dilini geliştiren ekip, LINQ ifadelerinin SQL sorgularına benzemesini istemiştir. Bu nedenle derleyici nerede where operatörünü görürse IL kodu hazırlarken geri planda Where() metoduna çevirmektedir.

Programcı sorgulama yaparken operatör yerine (örneğin where) ilgili metodu kullanmak istiyorum dediğinde işin içine lambda operatörü girmekte, dolayısıyla yazılan sorgu ifadesi değil lambda ifadesi olmaktadır. Şimdi yukarıda "yillar" adını verdiğimiz dizi değişkendeki çift yılları çekip ayrı bir dizi değişkene alma işlemini lambda ifadesi ile yapacağız.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] yillar = new int[] {2011, 2012, 2013, 2014, 2015, 2016, 2017};
 var cift_yil = yillar.Where(yil => yil % 2 == 0);
 List <int> liste = cift_yil.ToList();
 foreach (int yil in liste)
 {
   listBox1.Items.Add(yil.ToString());
 }
}
```

Bu kod işlev bakımından yukarıda verilenden farklı değildir. Şimdi Where() metoduna odaklanıp gördüklerimizi maddeler halinde yazacağız:

- Burada "yillar" adını verdiğimiz dizi değişken IEnumerable arayüzünü uyguladığı için bu arayüzden gelen Where() ve Select() gibi metotları dizi değişkenler üzerinden işletmek mümkündür.
- Where() metodunun işletilmesine aracılık eden "yillar" adlı dizi değişken int tipinde olduğu için Where() metodu geriye bir IEnumerable<int> nesnesi gönderecektir.
- Üzerinde işlem yaptığımız dizi değişken int tipinde olduğu için Where() metodu parametre olarak "Func<int, bool> predicate" yani int tipinde bir parametreye sahip olup geriye bool değer gönderen bir delegate veya metot parametre olarak verilmelidir.
- Bu metodu kendiniz yazıp Where() metoduna parametre olarak verebilir veya yukarıda yaptığımız gibi lambda ifadesi hazırlayabilirsiniz.
- Where() metodu kendisine parametre olarak verilen lambda ifadesini burada "yillar" adını verdiğimiz dizi değişkenin her elemanı için bir kez işletir. İşletme sonucunda lambda operatörünün sağındaki karşılaştırma true ise söz konusu elemanı geriye gönderilen IEnumerable<int> nesnesine ekler.

Konu üzerinde düşünmenizi sağlamak için yukarı hazırlayıp Where() metoduna parametre olarak verdiğimiz lambda ifadesini metot olarak düzenleyeceğiz. Aşağıda verilen kod işlev olarak yukarıda verilenden farklı değildir.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] yillar = new int[] {2011, 2012, 2013, 2014, 2015, 2016, 2017 };
 IEnumerable<int> cift_yil = yillar.Where(YilKontrol);
 List <int> liste = cift_yil.ToList();
 foreach (int yil in liste)
  {
    listBox1.Items.Add(yil.ToString());
  }
}
static bool YilKontrol(int i)
{
 return i % 2 == 0;
}
```

Dikkat ettiyseniz bu kez farklılık olsun diye Where() metodunun geriye göndereceği IEnumerable nesnesinin generic tipini tahmin edilmesi işlemini var anahtar kelimesini kullanmayarak derleyiciye bırakmadık. İsteseydik burada YilKontrol() adını verdiğimiz metodu isimsiz metot olarak düzenleyebilirdik. Where() metodunun parametre olarak istediği metodu isimsiz olarak düzenlemenin nasıl yapıldığını anlatmak için Where() metodunun olduğu noktadan sonrasını sildik.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] yillar = new int[] {2011, 2012, 2013, 2014, 2015, 2016, 2017 };
 IEnumerable<int> cift_yil = yillar.Where();
}
```

Yukarıda Where() metoduna ait parantezin içine mevcut bir metodumuzun adını yazmıştık. Şimdi ise parantezin içinde delegate anahtar kelimesinden yararlanıp yukarıda YilKontrol() adını verdiğimiz metodun gövdesine yazdıklarımızı delegate anahtar kelimesinden sonra {} parantezlerin arasına yazdık.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] yillar = new int[] {2011, 2012, 2013, 2014, 2015, 2016, 2017};
 IEnumerable<int> cift_yil = yillar.Where(delegate(int i)
   {
     return i % 2 == 0;
   }
  );
}
```

Gördüğünüz gibi aynı işlemi yapmak için çok sayıda teknik bulunmaktadır. Bu da programcı adayının kafasının karışmasına, öğrendiklerini çabuk unutmasına neden olmaktadır.

Lambda İfadeleri ve LINQ Sorguları

Bir önceki bölümde en yalın hali ile LINQ sorgularından söz edildi. Bu yapılırken konu kolay anlaşılsın diye sorgu metinleri içinde genişletme metotlarının operatör hali kullanıldı ve

lambda ifadelerinden hiç söz edilmedi. Bu kitabı bölüm sırasına göre okuyanlar şimdiye kadar hem LINQ hem de lambda ifadeleri hakkında biraz bilgiye sahip oldular. Artık LINQ sorgularında lambda ifadelerine yer verebiliriz.

Daha önce ifade edildiği gibi LINQ sorguları gerçekte Enumerable sınıfındaki genişletme metotlarına dayanmaktadır. Bir önceki bölümde LINQ sorgularını hazırlarken genişletme metotları yerine bu metotları temsil eden operatörleri kullanmıştık. Örneğin Select() metodu yerine bu metodu temsil eden select operatörünü kullanmıştık. Bu sayede LINQ sorgularımız SQL sorgularına benzemişti. Sorgulama yaparken Enumerable sınıfında tanımlı olan genişletme metotlarının nasıl kullanıldığını göstermek için bir önceki bölümde hazırladığımız basit LINQ sorgusunu buraya aldık.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] dizi = { 30, 36, 45, 50, 15, 55, 63, 70, 75 };
 var tekler = from eleman in dizi
    where eleman % 2 != 0
       select eleman;
 foreach (int i in tekler)
 {
  listBox1.Items.Add(i.ToString());
 }
}
```

Daha önce üzerinde uzunca konuştuğumuz bu LINQ sorgusunda Where ve Select operatörlerinden yararlandık. Derleme sırasında geri planda bu operatörlere karşılık gelen genişletme metotlarına göre işlem yapıldığını biliyorsunuz. Şimdi aynı sonucu almak üzere operatörlerin yerine genişletme metotlarından ve lambda ifadelerinden yararlanma yoluna gideceğiz. Bu amaçla yukarıda verdiğimiz koda aşağıdaki gibi ekleme yaptık.

```
private void button1_Click(object sender, EventArgs e)
{
 int[] dizi = { 30, 36, 45, 50, 15, 55, 63, 70, 75 };
 var tekler = from eleman in dizi
    where eleman % 2 != 0
        select eleman;
 var tekler1 = dizi.Where(eleman => eleman % 2 != 0);
 foreach (int i in tekler)
  {
   listBox1.Items.Add(i.ToString());
  }
  foreach (int i in tekler1)
  {
   listBox2.Items.Add(i.ToString());
  }
}
```

Bir sonraki bölümde generic yapıya ve değişik sayıda parametreye sahip **Action** delegate tipi hakkında bilgi verilecektir. LINQ sorgularını ve lambda ifadelerini bütün yönleri ile anlatmaya bu kitabın kapsamı elvermediği için burada kesiyoruz.

Kavram Dizini

A

Abstract Class'lar ... 245
Add New Item ... 61
Add Reference ... 87
Add Windows Form ... 52
AddDays Metodu ... 166
AddHours Metodu ... 167
AddYear Metodu ... 166
Alanlar ... 210
Anonymous Types ... 604
Application Sınıfı ... 38, 67
ArrayList Sınıfı ... 494
As Operatörü ... 181
Atama İşlemleri ... 179
Automatic Properties ... 226

B

Başlangıç Formu ... 55
Binary Sayılar ... 120
BinarySearch Metodu ... 155
Bool Değişkenler ... 115
Byte Değişkenler ... 116

C

Char Değişkenler ... 110
Checked ve Unchecked Deyimleri ... 489
Class Kavramı ... 5
Class'lar ... 205
Clear Metodu ... 152
CollectionBase Sınıfı ... 530
Console Application ... 63
Console Sınıfı ... 10
Constant Pattern'ler ... 592
Convert Sınıfı ... 16, 129
ConvertAll() Metodu ... 574
Copy Metodu ... 153
CreeateInstance Metodu ... 145
Çok Boyutlu Dizi Değişkenler ... 148

D

DateSerial Metodu ... 165
DateTime Değişkenler ... 123
DateTime Yapısı ... 161
Day Özelliği ... 165
DayOfWeek Özelliği ... 163
DayOfYear Özelliği ... 163
DaysInMonth Metodu ... 166
Decimal Değişkenler ... 124
Deferred Execution ... 644
Değişken Tanımlamak ... 91
Değişkenlere İlk Değer Aktarmak ... 93
Delegate Hazırlamak ... 357
Derleme İşlemi ... 4
Developer Command Prrompt ... 1
Dinamik Değişkenler ... 136
Dispose() Metodu ... 42
DivideByZeroException Sınıfı ... 471
Dizi Değişkenler ... 141
Double Değişkenler ... 121
Düzensiz Diziler ... 158

E

Enum Hazırlamak ... 454
Enum Sınıfı ... 457
Enum'larda Veya Operatörü ... 464
Escape Karakterler ... 114
Exception Sınıfı Hazırlamak ... 486
Exception Sınıfları ... 467
Exists() Metodu ... 570
ExpandoObject Sınıfı ... 293
Explicit Operatörünü Aşırı Yüklemek ... 406
Extension Metotlar ... 609

F

FileNotFoundException Sınıfı ... 480
Finally Bloğu ... 476
Float Değişkenler ... 121
For Döngüsü ... 200
Foreach Döngüsü ... 201
ForEach() Metodu ve Action Delegesi ... 677
ForEach() Metodu ... 575
Format Metodu ... 463
FormClosing Olayı ... 55
Formlar İçin Hazırlanan Kodlar ... 40

G

Generic Class'larda Miras 557
Generic Delegate 565
Generic Interface'ler 560
Generic Metotlara Sınırlama Koymak ... 555
Generic Queue Sınıfı 578
Generic Sınıflar 537
Generic Sınıflarda Metotlar 549
Generic Stack Sınıfı 576
GetEnumerator() Metodu 503
GetInterface Metodu 349
GetInvocationList() Metodu 393
GetType() Metodu 108, 436
GetValue Metodu 144
GetValues() Metodu 458

H

Hashtable Sınıfı 510
Hata Yakalama Filtreleri 481
Hata Yakalama Sınıfları 477
Hata Yakalamak 467
Hour Özelliği 167

I, İ, J

ICollection Arayüzü 528
IDictionary Arayüzü 533
IEnumerable ve IEnumerator Arayüzleri 520
If Deyimi 193
Implicit Operatörünü Aşırı Yüklemek ... 406
Indexer'ları Aşırı Yüklemek 423
IndexOf Metodu 156
InitializeComponent Metodu 33
Integer Değişkenler 116
Interface Referansları 353
Interface'ler ve Özellikler 344
Interface'ler 339
Interface'lerde Kalıtım 349
InvalidCastException Sınıfı 481
Is Operatörü 181
IsDefined() Metodu 465
IsLeapYear Metodu 166
İndeksleyici Destekli Generic Sınıf 547
İndeksleyiciler 415
İsimsiz Metotlar 622
İsimsiz Tipler 604
İsteğe Bağlı Parametreler 311
Join Metodu 157

K

Kalıtım Kavramı 240
Kaydırma Operatörleri 180
KeyDown Olayı 58
KeyPress Olayı 57
Koleksiyon Sınıfları 491
Konsol Uygulamaları 59

L

Lambda İfadeleri 655
LastIndexOf Metodu 157
Length Özelliği 150
LINQ İle Dosyaları Listelemek 653
LINQ Sorguları ve İsimsiz Tipler 650
LINQ Sorguları 633
LinkedList Sınıfı 582
List Sınıfı 567
Long Değişkenler 116

M

Main Metodu 75
Main() Metodu 6
MessageBox Sınıfı 34, 49
Metot Hazırlamak 8, 297
Metot Override 268
Metotlar İçin Parametre Tanımlamak ... 300
Metotları Aşırı Yüklemek 328
Minute Özelliği 167
Month Özelliği 165
Multicast Delegate'ler 369

N

Name Hidding 265
Namespace Hazırlamak 79
Namespace'e Takma Ad Vermek 87
Namespace'ler Hakkında Önbilgi 16
Nested Class'lar 236
New Anahtar Kelimesi 27
Now Özelliği 162
Nullable Tipler 133

O, Ö

Object Değişkenler 106
Object Inializers 231
Object Initializer 629
Object Sınıfı 429
Olay Hazırlamak 381

Olaylar ve Metot Hazırlamak 44
Olaylarla Metotları İlişkilendirmek..... 47
Open Project 31
OperatöriAaşırı Yüklemek..... 395
Operatörler..... 173
OrderedDictionary Sınıfı..... 517
Out Anahtar Kelimesi 313
Output Type..... 63
Override Anahtar Kelimesi 247
Özellikler 220

P

Parse() Metodu.....15, 130
Partial Metotlar 334
Partial Types..... 233
Pattern Matching..... 585
Polimorfizm Kavramı 272
Private Alanlar 217
Program.cs Dosyası 35
Public Değişkenler..... 96

Q, R

Queue Sınıfı 508
Rank Metodu..... 149
Read() Metodu..... 11
ReadKey Metodu..... 69
ReadOnly Değişkenler..... 104
Ref Anahtar Kelimesi 313
ReferanceEquals Metodu..... 434
RemoveAll Metodu 572
Reset Layout Window 32
Resize Metodu 147
Return Anahtar Kelimesi 305
Run Metodu 36

S

Sabit Tanımlamak 104
Second Özelliği..... 167
SetValue Metodu 144
Show Metodu 54
ShowDialog() Metodu..... 53
Sıfıra Bölme Hatasını Yakalamak 471
Solution Explorer..... 33
Sort Metodu 154
SortedDictionary Sınıfı..... 580
SortedList Sınıfı..... 515
Stack Sınıfı 505
Static Yapıcı Metotlar..... 289
Statik Değişkenler 99
Statik Olan-Olmayan Metotlar 23
Statik Sınıflar 253
String Değişkenler..... 111
Struct Hazırlamak 441
Struct'lar ve Interface'ler..... 451
Struct'lar ve Yapıcı Metotlar..... 445
Struct'larda Sabit Hazırlamak 448
Switch Case..... 195
Switch Pattern..... 596

T

Tarih ve Zaman İşlemleri 161
Ternary Operatörü 187
TextBox Kontrolü..... 56
This Anahtar Kelimesi 216
Throw Deyimi 483
Ticks Özelliği 169
TimeSpan Yapısı 170
Tip Dönüşümleri..... 125
ToDateTime Metodu..... 162
Today Özelliği 161
ToString Metodu 127
ToString Metodu 16
TryParse Metodu..... 130
Tuple Sınıfı 324
Type Pattern 585

U, V

Using Deyimi..... 82
Using Static Ayarlaması 262
ValueType Sınıfı 327
Var Anahtar Kelimesi 601
Var İle Değişken Tanımlamak 110
Var Pattern..... 593
View Form..... 33
Visual Studio..... 31

W

Where Operatörü..... 642
While Döngüsü 199
Windows Forms Application..... 32
Write Metodu 18
WriteLine Metodu 10

Y

Yapıcı Metotlar 277
Year Özelliği..... 165
Yerel Değişkenler 7, 94
Yerel Metotlar 336