

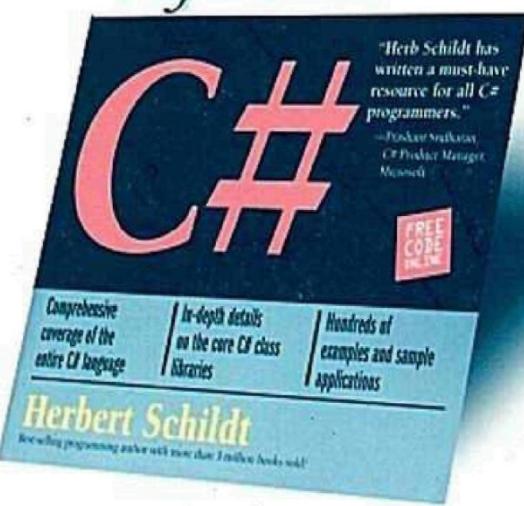
**"Herb Schildt, her programcının sahip olması gereken bir kaynak hazırlamış."**

**- Prashant Sridharan, C# Ürün Müdürü, Microsoft**

- C# dilinin tam ve anlaşılır incelemesi
- Çekirdek C# sınıf kütüphanelerinin ayrıntılı açıklaması
- Yüzlerce örnek program ve uygulama
- Veri tipleri ve operatörler
- Kontrol ifadeleri
- Sınıflar ve nesneler
- Yapılandırıcılar, yok ediciler ve metodlar
- Arayüzler, diziler, numaralandırmalar ve yapılar
- Metotların ve operatörlerin aşırı yüklenmesi
- Kalıtım ve sanal metodlar
- Yansıma ve çalışma zamanı tip tanımlaması
- Delegeler, özellikler, olaylar ve indeksleyiciler
- Kural dışı durum yönetimi
- Nitelikler
- İşaretçiler ve "emniyetsiz kod"
- Çok kanallılık
- Çekirdek C# kütüphaneleri ve
- System isim uzayı
- I/O sınıfları
- Ağ oluşturma
- Koleksiyon sınıfları
- Önişlemci

# C#

## The Complete Reference



**Herbert Schildt**

# Herkes İçin!

OSBORNE  
MC  
Graw  
Hill



---

**KISIM I C# DİLİ .....**

<b>BÖLÜM 1: C#'IN DOĞUŞU .....</b>	13
C# Aile Ağacı .....	14
C: Modern Programlama Çağının Başlangıcı .....	14
Nesne Yönelimli Programlamanın ve C++'ın Doğuşu .....	15
İnternet ve Java'nın Geliş'i .....	16
C#'in Doğuşu .....	17
C# ile .NET Framework Arasındaki Bağlantı .....	18
.NET Framework Nedir? .....	19
Common Language Runtime Nasıl Çalışır? .....	19
Kontrol Altına Alınan ve Alınmayan Kodlar .....	20
Common Language Specification .....	20
<b>BÖLÜM 2: C#'A GENEL BİR BAKIŞ .....</b>	21
Nesne Yönelimli Programlama .....	22
İlişkili Verilerin Paketlenmesi .....	23
Çok Böçümlülük .....	23
Kalıtım .....	24
İlk Basit Program .....	25
C# Komut Satırı Derleyicisini (csc.exe) Kullanmak .....	25
Visual Studio IDE'yi Kullanmak .....	26
Satır Satır İlk Örnek Program .....	30
Söz Dizimi Hatalarını Kontrol Altına Almak .....	33
Küçük Bir Değişiklik .....	34
İkinci Basit Program .....	35
Başka Bir Veri Tipi .....	37
İki Kontrol İfadesi .....	39
if İfadesi .....	39
for Döngüsü .....	40
Kod Bloklarını Kullanmak .....	42
Noktalı Virgüler ve Konumlandırma .....	43
Girintileme Alıştırmaları .....	44
C# Anahtar Kelimeleri .....	45
Tanımlayıcılar .....	45
C# Sınıf Kütüphaneleri .....	46
<b>BÖLÜM 3: VERİ TIPLERİ, LİTERALLER VE DEĞİŞKENLER .....</b>	47
Veri Tipleri Neden Önemlidir? .....	48
C#'in Değer Tipleri .....	48
Tamsayılar .....	49
decimal Tipi .....	53
Karakterler .....	54
bool Tipi .....	55
Bazı Çıktı Seçenekleri .....	56
Literaller .....	59
Onaltılık Literaller .....	60
Karakter Kaçış Sekansları .....	61
Karakter Katarı Literalleri .....	62

Değişkenlere Yakından Bir Bakış .....	63
Bir Değişkene İlk Değer Atamak .....	64
Dinamik İlk Değer Ataması .....	64
Değişkenlerin Kapsamı ve Yaşam Süreleri .....	65
Tip Dönüşümleri ve Tip Atamaları .....	68
Otomatik Dönüşümler .....	69
Uyumsuz Tiplerin Atamaları .....	70
Deyimlerde Tip Dönüşümleri .....	72
Deyimlerde Tip Ataması Kullanmak .....	74
<b>BÖLÜM 4: OPERATÖRLER .....</b>	77
Aritmetik Operatörler .....	78
Artırma ve Eksiltme .....	79
İlişkisel ve Mantıksal Operatörler .....	82
Kısa Devre Mantıksal Operatörler .....	85
Atama Operatörü .....	87
Bileşik Atamalar .....	87
Bit Tabanlı Operatörler .....	88
Bit Tabanlı VE, VEYA, XOR ve DEĞİL Operatörleri .....	88
Kaydırma Operatörleri .....	94
Bit Tabanlı Bileşik Atamalar .....	97
? Operatörü .....	97
Boşluklar ve Parantezler .....	99
Operatörlerin Öncelik Sırası .....	99
<b>BÖLÜM 5: PROGRAM KONTROL İFADELERİ .....</b>	101
if ifadesi .....	102
Kümelenmiş if'ler .....	103
if-else-if Merdiveni .....	104
switch İfadesi .....	105
Kümelenmiş switch İfadeleri .....	109
for Döngüsü .....	109
for Döngüsü Üzerinde Bazı Varyasyonlar .....	112
while Döngüsü .....	117
do-while Döngüsü .....	119
foreach Döngüsü .....	120
Döngüden Çıkmak İçin break Kullanmak .....	120
continue İfadesini Kullanmak .....	123
return .....	123
goto .....	124
<b>BÖLÜM 6: SINIFLARA, NESNELERE VE METOTLARA GİRİŞ .....</b>	127
Sınıfların Temel Özellikleri .....	128
Sınıfın Genel Biçimi .....	128
Sınıfı Tanımlamak .....	129
Nesneler Nasıl Oluşturulur? .....	134
Referans Değişkenleri ve Atama .....	135
Metotlar .....	135
Building Sınıfına Bir Metot Ekleme .....	136

Metottan Dönüş .....	139
Bir Değer Döndürmek .....	140
Parametre Kullanmak .....	142
Building'e Parametreli Bir Metot EklemeK .....	145
Erişilemeyen Kodları Önlemek .....	146
Yapilandırıcılar .....	147
Parametreli Yapilandırıcılar .....	148
Building Sınıfına Bir Yapılandırcı EklemeK .....	149
new Operatörüne Tekrar Bir Bakış .....	150
new'u Değer Tipleriyle Birlikte Kullanmak .....	151
Anlamsız Verilerin Toplanması ve Yok ediciler .....	152
Yok Ediciler .....	152
this Anahtar Kelimesi .....	154
<b>BÖLÜM 7: DİZİLER VE KARAKTER KATARLARI .....</b>	<b>157</b>
Diziler .....	158
Tek Boyutlu Diziler .....	158
Diziyi İlk Kullanıma Hazırlamak .....	161
Sınırlar Zorlanır .....	162
Çok Boyutlu Diziler .....	162
İki Boyutlu Diziler .....	162
Üç veya Daha Fazla Boyutlu Diziler .....	164
Çok Boyutlu Dizileri ilk Kullanıma Hazırlamak .....	165
Düzensiz Diziler .....	166
Dizi Referanslarını Atamak .....	168
Length Özelliğini Kullanmak .....	169
Düzensiz Dizilerde Length Kullanmak .....	172
foreach Döngüsü .....	173
Karakter Katarları .....	177
Karakter Katarlarını Oluşturmak .....	177
Karakter Katarları Üzerinde İşlem Yapmak .....	178
Karakter Katarı Dizileri .....	181
Karakter Katarları Değişmez .....	182
Karakter Katarları switch ifadelerinde Kullanılabilir .....	184
<b>BÖLÜM 8: METOT VE SINIFLARA DAHA YAKINDAN BİR BAKIŞ .....</b>	<b>185</b>
Sınıf Üyelerine Erişimi Kontrol Etmek .....	186
C# ' ta Erişim Belirleyicileri .....	186
Public ve Private Erişimi Uygulamak .....	188
Erişimi Kontrol Altına Almak: Bir Örnek Çalışma .....	189
Metotlara Nesne Aktarmak .....	194
Argümanlar Nasıl Aktarılır? .....	195
ref ve out Parametrelerini Kullanmak .....	198
ref Kullanmak .....	198
out Kullanmak .....	200
Referans Parametrelerinde ref ve out Kullanmak .....	202
Değişen Sayıda Argüman Kullanmak .....	204
Nesneleri Döndürmek .....	207
Bir Diziyi Döndürmek .....	209
Metotların Aşın Yüklenmesi .....	210
Yapilandırıcıları Aşırı Yüklemek .....	216

this Aracılığıyla Aşırı Yüklenmiş Bir Yapılandırıcıyı Çağırma	220
Main() Metodu	221
Main()'den Değer Döndürmek	221
Main()'e Argüman Aktarmak	222
Yinelenme	224
static'i Anlamak	227
static Yapılandırıcılar	232
<b>BÖLÜM 9: OPERATÖRLERİN AŞIRI YÜKLENMESİ</b>	234
Operatörlerin Aşırı Yüklenmesiyle İlgili Temel Kavramlar	235
İkili Operatörleri Aşırı Yüklemek	236
Tekli Operatörleri Aşırı Yüklemek	238
Standart C# Tipleri Üzerindeki İşlemleri Kontrol Altına Almak	242
İlişkisel Operatörleri Aşırı Yüklemek	246
true ve false'ı Aşırı Yüklemek	248
Mantıksal Operatörleri Aşırı Yüklemek	251
Mantıksal Operatörlerin Aşırı Yüklenmesine Basit Bir Yaklaşım	251
Kısa Devre Operatörleri Mümkün Kılmak	253
Dönüşüm Operatörleri	257
Operatörleri Aşırı Yüklemekle İlgili İpuçları ve Kısıtlamalar	262
Operatörlerin Aşırı Yüklenmesine Bir Başka Örnek	263
<b>BÖLÜM 10: İNDEKSLEYİCİLER VE ÖZELLİKLER</b>	268
İndeksleyiciler	269
Tek Boyutlu indeksleyici Oluşturmak	269
İndeksleyiciler Aşırı Yüklenebilir	273
İndeksleyiciler, Temel Niteliğinde Bir Dizi Gerektirmez	275
Çok Boyutlu İndeksleyiciler	276
Özellikler	278
Özelliklerle İlgili Kısıtlamalar	284
İndeksleyiciler ve Özelliklerin Kullanımı	284
<b>BÖLÜM 11: KALITIM</b>	290
Kalitimın Temel Unsurları	291
Üye Erişimi ve Kalitim	294
protected Erişim Kullanımı	297
Yapilandırıcılar ve Kalitim	298
Temel Sınıfın Yapılandırıcılarını Çağırma	300
Kalitim ve isim Gizleme	305
Gizli Bir isme Erişmek İçin base Kullanımı	306
Çok Katmanlı Hiyerarşi Oluşturmak	308
Yapilandırıcılar Ne Zaman Çağrılır?	311
Temel Sınıf Referansları ve Türetilmiş Nesneler	312
Sanal Metotlar ve Devre Dışı Bırakma (Overriding)	316
Devre Dışı Bırakılan Metotlara Neden Gerek Var?	320
Sanal Metotları Uygulamak	321
Özet Sınıfların Kullanımı	324
Kalitimı Önlemek İçin sealed Kullanmak	328
object Sınıfı	329

---

Kutulama ve Kutudan Çıkarma .....	331
object Genel Bir Veri Tipi midir? .....	333
<b>BÖLÜM 12: ARAYÜZLER, YAPILAR VE NUMARALANDIRMALAR .....</b>	<b>335</b>
Arayüzler .....	336
Arayüzleri Uygulamak .....	337
Arayüz Referanslarının Kullanımı .....	341
Arayüz Özellikleri .....	344
Arayüz İndeksleyiciler .....	345
Arayüzler Kalıtım Yoluyla Aktarılabilir .....	347
Arayüz Kalıtımı İle Birlikte İsim Gizleme .....	348
Açık Uygulamalar .....	348
Özel Bir Uygulama Geliştirmek .....	349
Belirsizliği Ortadan Kaldırmak İçin Açık Uygulamaların Kullanımı .....	350
Bir Arayüz ve Özet Sınıf Arasında Tercihte Bulunmak .....	351
.NET Standart Arayüzleri .....	351
Arayüzlerle İlgili Bir Örnek Çalışma .....	352
Yapılar .....	357
Yapılara Neden Gerek Vardır? .....	360
Numaralandırmalar .....	362
Bir Numaralandırmaya İlk Değer Atamak .....	364
Bir Numaralandırmanın Temel Tipini Belirtmek .....	365
Numaralandırmaların Kullanımı .....	365
<b>BÖLÜM 13: KURALDIŞI DURUM YÖNETİMİ .....</b>	<b>367</b>
System.Exception Sınıfı .....	368
Kural Dışı Durum Yönetimiyle İlgili Esaslar .....	369
try ve catch'in Kullanımı .....	369
Basit Bir Kural Dışı Durum Örneği .....	370
İkinci Bir Kural Dışı Durum Örneği .....	371
Yakalanmayan Kural Dışı Durumların Sonuçları .....	373
Kural Dışı Durumlar, Hataları Zarifçe Kontrol Altına Almanıza Olanak Tanır .....	374
Birden Fazla catch İfadelerinin Kullanımı .....	375
Kural Dışı Durumların Tümünü Yakalamak .....	376
try Bloklarını Kümelemek .....	377
Bir Kural Dışı Durum Fırlatmak .....	379
Kural Dışı Durumu Yeniden Fırlatmak .....	380
finally'nin Kullanımı .....	381
Kural Dışı Durumlara Yakından Bir Bakış .....	383
Sıkça Kullanılan Kural Dışı Durumlar .....	384
Kural Dışı Durum Sınıflarını Türetmek .....	386
Türetilmiş Sınıflarla İlgili Kural Dışı Durumları Yakalamak .....	390
checked ve unchecked Kullanımı .....	392
<b>BÖLÜM 14: I/O KULLANIMI .....</b>	<b>396</b>
C#'ın I/O Sistemi, Akışlar Üzerine Kurulmuştur .....	397
Byte Akışları ve Karakter Akışları .....	397
Önceki Tanımlı Akışlar .....	397
Akış Sınıfları .....	398
Stream Sınıfı .....	398

Byte Stream Sınıfları .....	399
Karakter Akışını Saran Sınıflar .....	400
İkili Akışlar .....	401
Konsol I/O .....	402
Konsol Girdisini Okumak .....	402
Konsol Çıktısı Yazmak .....	404
FileStream ve Byte Yönelimli Dosya I/O İşlemleri .....	405
Bir Dosyayı Açmak ve Kapatmak .....	405
Bir FileStream'den Byte'lar Okumak .....	407
Bir Dosyaya Yazmak .....	408
Dosya Kopyalamak İçin FileStream Kullanmak .....	410
Karakter Tabanlı Dosya I/O İşlemleri .....	411
StreamWriter Kullanımı .....	412
StreamReader Kullanımı .....	414
Standart Akışları Yönlendirmek .....	415
İkili Verileri Okumak ve Yazmak .....	417
BinaryWriter .....	417
BinaryReader .....	418
İkili I/O İşlemlerinin Gösterilmesi .....	419
Rasgele Erişimli Dosyalar .....	423
MemoryStream'in Kullanımı .....	425
StringReader ve StringWriter Kullanımı .....	427
Nümerik Karakter Katarlarını Dahili Gösterimlerine Dönüştürmek .....	429
<b>BÖLÜM 15: DELEGELER VE OLAYLAR .....</b>	432
Delegeler .....	433
Çoklu Çağrı .....	437
System. Delegate .....	439
Delegelere Neden Gerek Vardır? .....	440
Olaylar .....	440
Bir Çoklu Çağrı Olay Örneği .....	442
Olay Yöneticileri Olarak static Metotlara Karşı Örnek Metotlar .....	444
Olay Erişimcilerinin Kullanımı .....	446
Olayların Çeşitli Özellikleri .....	450
.NET'te Olaylarla İlgili Esaslar .....	451
EventHandler Kullanımı .....	453
Olayları Uygulamak: Bir Örnek Çalışma .....	454
<b>BÖLÜM 16: İSİM UZAYLARI, ÖNİŞLEMCI VE ASSEMBLY'LER .....</b>	456
İsim Uzayları .....	457
İsim Uzayını Deklare Etmek .....	457
using .....	461
using'in İkinci Kullanım Şekli .....	463
İsim Uzayları Eklenebilir .....	464
İsim Uzayları Kümelenebilir .....	466
Varsayılan İsim Uzayı .....	467
Önişlemci .....	467
#define .....	468
#if ve #endif .....	468
#else ve #elif .....	470

---

#undef .....	472
#error .....	472
#warning .....	472
#line .....	472
#region ve #endregion .....	473
Assembly'ler ve internal Erişim Niteleyicisi .....	473
internal Erişim Niteleyicisi .....	474

## BÖLÜM 17: ÇALIŞMA ZAMANI TİP TANIMLAMASI, YANSIMA VE NİTELİKLER ..... 475

Çalışma Zamanı Tip Tanımlaması .....	476
Bir Tipi is İle Test Etmek .....	476
as Kullanımı .....	477
typeof Kullanımı .....	479
Yansıma .....	480
Yansımanın Çekirdeği: System.Type .....	480
Yansımanın Kullanımı .....	481
Metotlar Hakkında Bilgi Edinmek .....	482
Metotları Yansıma Kullanarak Çağırmak .....	486
Type'in Yapılandırmacılarını Elde Etmek .....	489
Assembly'lerden Tip Elde Etmek .....	493
Tiplerin Ortaya Çıkarılmasını Tam Olarak Otomatize Etmek .....	499
Nitelikler .....	501
Niteliklerle ilgili Temel Bilgiler .....	502
Konumsal ve İsimlendirilmiş Parametreler .....	505
Standart Niteliklerin Kullanımı .....	509
AttributeUsage .....	509
Conditional Niteliği .....	510
Obsolete Niteliği .....	511

## BÖLÜM 18: EMNİYETSİZ KOD, İŞARETÇİLER VE ÇEŞİTLİ KONULAR ..... 513

Emniyetsiz Kod .....	514
İşaretçilerin Esasları .....	515
unsafe Kullanımı .....	517
fixed Kullanımı .....	517
Yapı Üyelerine Bir İşaretçi Aracılığıyla Erişmek .....	518
İşaretçi Aritmetiği .....	519
İşaretçilerin Karşılaştırılması .....	520
İşaretçiler ve Diziler .....	521
İşaretçiler ve Karakter Katarları .....	523
Çoklu Dolaylılık .....	524
İşaretçi Dizileri .....	526
Çeşitli Anahtar Kelimeler .....	526
sizeof .....	526
lock .....	526
readonly .....	527
stackalloc .....	527
using ifadesi .....	528
const ve volatile .....	529

<b>KISIM II: C# KÜTÜPHANESİSİ KEŞFETMEK .....</b>	530
<b>BÖLÜM 19: SYSTEM İSİM UZAYI .....</b>	531
System 'in Üyeleri .....	532
Math Sınıfı .....	533
Değer Tipindeki Yapılar .....	538
Tamsayı Tipinde Yapılar .....	539
Kayan Nokta Tipinde Yapılar .....	541
Decimal .....	544
Char .....	549
Boolean Yapısı .....	553
Array sınıfı .....	554
Dizileri Sıralamak ve Aramak .....	554
Diziyi Ters Çevirmek .....	557
Diziyi Kopyalamak .....	557
BitConverter .....	563
Random ile Rasgele Sayılar Üretmek .....	566
Bellek Yönetimi ve GC Sınıfı .....	567
Object .....	568
IComparable Arayüzü .....	569
IConvertible Arayüzü .....	569
ICloneable Arayüzü .....	570
IFormatProvider ve IFormattable .....	572
<b>BÖLÜM 20: KARAKTER KATARLARI VE BİÇİMLENDİRME .....</b>	573
C#'ta Karakter Katarları .....	574
String Sınıfı .....	574
String Yapılandırıcıları .....	575
String Alanı, İndeksleyici ve Özellik .....	576
String Operatörleri .....	576
String Metotları .....	577
Karakter Katarlarını Doldurmak ve Budamak .....	589
Ekleme, Çıkarma ve Değiştirme .....	590
Harf Kipini Değiştirmek .....	592
Substring() Metodunun Kullanımı .....	592
Biçimlendirme .....	593
Biçimlendirmeye Genel Bakış .....	593
Nümerik Biçim Belirleyicileri .....	594
Verileri Biçimlendirmek için String.Format() ve ToString() Kullanımı .....	596
Değerleri Biçimlendirmek İçin String.Format( ) Kullanımı .....	597
Verileri Biçimlendirmek İçin ToString() Kullanımı .....	599
Özel Bir Nümerik Biçim Oluşturmak .....	600
Özel Biçim Yer Tutucu Karakterleri .....	601
Tarih ve Saati Biçimlendirmek .....	604
Özel Tarih ve Saat Biçiminizi Oluşturmak .....	607
Numaralandırmaları Biçimlendirmek .....	609

<b>BÖLÜM 21: ÇOK KANALLI PROGRAMLAMA .....</b>	611
Çok Kanallılıkla İlgili Esaslar .....	612
Thread Sınıfinin Kullanımı .....	613
Bir Kanal Oluşturmak .....	613
Bazı Basit Yenilikler .....	616
Birden Fazla Kanal Oluşturmak .....	617
Çalışma Kanalının Ne Zaman Sona Ereceğini Belirlemek .....	619
IsBackground Özelliği .....	622
Kanal Öncelikleri .....	623
Senkronizasyon .....	625
Alternatif Yaklaşım .....	629
Bir Statik Metodu Kilitlemek .....	631
Monitör Sınıfı ve lock .....	631
Wait(), Pulse() ve PulseAll() Kullanarak Kanal İletişimi .....	632
Wait() ve Pulse() Kullanan Bir Örnek .....	633
Kilitlenme .....	636
MethodImplAttribute Kullanımı .....	637
Kanalları Askıya Almak, Sürdürmek ve Durdurmak .....	639
Alternatif Abort().....	641
Abort()'u İptal Etmek .....	642
Çalışma Kanalının Durumunu Belirlemek .....	644
Temel Kanalın Kullanımı .....	645
Çok Kanallılık Önerileri .....	646
Ayrı Bir Görevi Başlatmak .....	646
<b>BÖLÜM 22: KOLEKSİYONLARLA ÇALIŞMAK .....</b>	649
Koleksiyonlara Genel Bakış .....	650
Koleksiyon Arayüzleri .....	651
ICollection Arayüzü .....	651
IList Arayüzü .....	652
IDictionary Arayüzü .....	653
IEnumerable, IEnumerator ve IDictionaryEnumerator .....	655
IComparer .....	655
IHashCodeProvider .....	655
The DictionaryEntry Yapısı .....	656
Genel Amaçlı Koleksiyon Sınıfları .....	656
ArrayList .....	656
Hashtable .....	664
SortedList .....	666
Stack .....	670
Queue .....	672
BitArray İle Bitleri Saklamak .....	675
Özelleştirilmiş Koleksiyonlar .....	678
Bir Numaralandırıcı Üzerinden Bir Koleksiyona Erişmek .....	678
Numaralandırıcı Kullanmak .....	679
IDictionaryEnumerator Kullanımı .....	680
Koleksiyonlar içinde Kullanıcı Tarafından Tanımlanan Sınıfları Saklamak .....	682
IComparable'i Uygulamak .....	683
IComparer'i Belirtmek .....	685
Koleksiyonların Özeti .....	687

<b>BÖLÜM 23: INTERNET İLE AĞ UYGULAMALARI .....</b>	688
System.Net Üyeleri .....	689
Uniform Resource Identifier'lar .....	690
Internet Erişim Esasları .....	691
WebRequest .....	692
WebResponse .....	693
HttpWebRequest ve HttpWebResponse .....	694
İlk Basit Örnek .....	694
Ağ Hatalarını Kontrol Altına Almak .....	697
Create() Tarafından Üretilen Kural Dışı Durumlar .....	697
GetReponse() Tarafından Üretilen Kural Dışı Durumlar .....	698
GetResponseStream() Tarafından Üretilen Kural Dışı Durumlar .....	698
Kural Dışı Durum Yönetimini Kullanmak .....	698
URI Sınıfı .....	700
Ek HTTP Yanıt Bilgilerine Erişim .....	701
Başlığa Erişim .....	702
Çerezlere Erişim .....	703
LastModified Özelliğinin Kullanımı .....	705
MiniCrawler: Örnek Çalışma .....	706
WebClient Kullanımı .....	709
<b>KISIM III: C#'I UYGULAMAYA GEÇİRMEK .....</b>	713
<b>BÖLÜM 24: BİLEŞENLERİN OLUŞTURULMASI .....</b>	714
Bileşen Nedir? .....	715
Bileşen Modeli .....	715
C# Bileşeni Nedir? .....	716
Konteynerler ve Siteler .....	716
C# ite COM Bileşenlerinin Karşılaştırması .....	716
IComponent .....	717
Bileşen .....	717
Basit Bir Bileşen .....	719
CipherLib'i Derlemek .....	720
CipherComp Kullanan Bir İstemci .....	720
Dispose()'u Devre Dışı Bırakmak .....	721
Dispose(bool)'un Kullanımı .....	723
Dispose Edilmiş Bir Bileşenin Kullanımını Önlemek .....	728
using İfadesini Kullanmak .....	729
Konteynerler .....	730
Konteynerin Gösterilmesi .....	732
Bileşenler, Programlamanın Geleceği mi? .....	733
<b>BÖLÜM 25: FORM TABANLI WİNDOWS UYGULAMALARI GELİŞTİRMEK .....</b>	734
Windows Programlamanın Kısa Tarihçesi .....	735
Form Tabanlı Bir Windows Uygulaması Yazmanın İki Yöntemi .....	736
Windows'un Kullanıcı ile Etkileşimi .....	736
Windows Formları .....	737
Form Sınıfı .....	737
Form Tabanlı Bir Windows Program Taslağı .....	737

Windows Taslağını Derlemek .....	740
Düğme Eklemek .....	741
Button Sınıfının Esasları .....	741
Bir Forma Düğme Eklemek .....	742
Basit Bir Düğme Örneği .....	742
Mesajları Yönetmek .....	743
Alternatif Uygulama .....	746
Mesaj Kutusunun Kullanımı .....	746
Menü Eklemek .....	749
Şimdi Ne Yapabilirsiniz? .....	754
<b>BÖLÜM 26: YİNELENEREK İNEN BİR DEYİM AYRIŞTIRICISI .....</b>	<b>755</b>
Deyimler .....	756
Deyimleri Ayırtırmak: Problem .....	757
Bir Deyimi Ayırtırmak .....	758
Bir Deyimi Parçalara Ayırmak .....	759
Basit Bir Deyim Ayırtıcısı .....	762
Ayırtıcıyı Anlayalım .....	767
Ayırtıcıya Değişken Eklemek .....	768
Yinelenerek inen Ayırtırıcıda Söz Dizimi Kontrolü .....	776
Denenecek Birkaç Şey .....	777
<b>EK A: XML AÇIKLAMALARI REFERANSI .....</b>	<b>779</b>
XML Açıklama İmleri .....	780
XML Belgelemesinin Derlenmesi .....	781
XML Belgeleme Örneği .....	782
<b>EK B: C# VE ROBOTİK .....</b>	<b>784</b>

---

# C#'IN DOĞUŞU

C#, programlama dillerinin süregelen gelişiminde bir sonraki adımı temsil eder, C#'ın ortaya çıkışının kökleri derinlere, geçmiş birkaç yıl boyunca bilgisayar dillerinin gelişimini nitelemekte olan geliştirme ve adaptasyon süreçlerine kadar uzanır. Daha önce ortaya çıkan başarılı dillerin tümünde olduğu gibi, C# da bir yandan programlama sanatını daha ileriye taşıırken, öte yandan geçmişin üzerine inşa edilmiştir.

Microsoft tarafından .NET Framework’ün (.NET Çatısı) gelişimini desteklemek amacıyla geliştirilen C#, zamanın sınavından geçmiş özelliklerinin ve en son teknolojik yeniliklerin gücünden yararlanmaktadır. C#, modern kurumsal bilişim ortamları için programlar yazmaya yönelik oldukça kullanışlı ve verimli yöntemlerden biridir. Windows, internet, çeşitli bileşenler vs, bu modern kurumsal bilişim ortamları kapsamında yer almaktadır. Bu süreçte C#, programlama manzarasını yeniden tanımlamıştır.

Bu bölümün amacı C#'ı tarihsel açıdan yerine yerleştirmektir. Bunu gerçekleştirirken C#'ın ortaya çıkışına neden olan faktörler, C#'ın tasarım felsefesi ve diğer bilgisayar dillerinden nasıl etkilendiği de bu kapsamda ele alınacaktır. Bu bölümde ayrıca, C# ile .NET Framework arasında nasıl bir ilişki olduğu da açıklanmaktadır.

## C# Aile Ağacı

Bilgisayar dilleri boşlukta asılı değildir. Aslında, bilgisayar dilleri birbirleriyle bağlantılıdır; her yeni dil, kendisinden önceki dillerden bir şekilde etkilenmiştir. Farklı bitki türlerinin, birbirlerinin polenlerini kullanarak yeni bitki türleri oluşturması gibi bir durum söz konusudur. Benzer şekilde, bir dilin özellikleri bir diğerine uyarlanmış, varolan bir dile bir yenilik eklenmiş veya eski bir özellik varolandan çıkarılmıştır. Bu şekilde diller gelişmekte ve programlama sanatı ilerlemektedir. C# da bu bağlamda bir istisna değildir.

C# zengin bir programlama mirasına sahiptir. C# direkt olarak dünyanın en başarılı programlama dillerinin ikisinden türetilmiştir: C ve C++. Ayrıca bir başkasıyla da yakından ilişkilidir, Java. Bu ilişkilerin doğasını anlamak, C#'ı anlamak için kritik öneme sahiptir. Bu nedenle, C#'ı bu üç dilin tarihi çerçevesine oturtarak incelemeye başlıyoruz.

## C: Modern Programlama Çağının Başlangıcı

C'nin ortaya çıkışı programmanın modern çağının başlangıcına işaret eder. C, Dennis Ritchie tarafından 1970'te, UNIX işletim sistemi kullanan DEC PDP-11 üzerinde icat edildi. Daha önceki bazı diller, özellikle Pascal, kayda değer bir başarı elde etmiş olsa da, bugünkü programmanın gidişatını belirleyen modeli kuran C olmuştur,

C, 1960'ların *yapısal programlama* (*structured programming*) devrimi ile ortaya çıktı. Yapısal programlama öncesinde büyük programları yazmak zordu, çünkü program mantığı "spaghetti kod" olarak bilinen, takibi güç, bir takım arapsaçına dönümüş atlamlalar, çağrılar ve dönüşlerden ibaret olan bir program yapısına dönüşme eğilimindeydi. Yapısal dillerin bu probleme getirdiği çare; düzgün tanımlanmış kontrol ifadeleri, yerel değişkenli alt rutinler (*subroutines*) ve diğer gelişmiş özelliklerdir. Yapısal dilleri kullanarak bir dereceye kadar büyük programları yazmak mümkün hale geldi.

O zamanlar başka yapısal diller de mevcut olmasına rağmen gücü, estetiği ve ifade edilebilirliği başarıyla birleştiren ilk dil C oldu. C'nin kısa ve öz ama aynı zamanda kullanımı kolay söz dizimi, kontrolün (dilde değil de) programcada olmasına dayanan felsefesiyle birleşince çok kısa bir sürede birçok kişinin diğer dilleri bırakıp C'ye dönmesiyle sonuçlandı. Bugünün bakış açısıyla anlaşılması biraz güç olabilir ama C, programcılar uzun zamandır beklediği taze bir soluktu. Sonuç olarak; C, 1980'lerin en yaygın olarak kullanılan yapısal programlama dili halini aldı.

Her şeye rağmen, saygın C dilinin bile kısıtlayıcı yönleri vardı. En zahmetli yönlerinden biri büyük programları ele almadaki yetersizliğiydı. Proje belirli bir boyuta ulaştığı zaman C dili adeta bir engele çarpar ve bu noktadan sonra C programları anlaşılması ve sürdürülmesi zor bir hal alır. Bu limite tam olarak ne zaman ulaşılacağı programa, programcıya ve eldeki araçlara bağlı olmakla birlikte 5,000 satır gibi kısa bir kodda bile bu durumla karşılaşılabilmektedir.

## Nesne Yönelimli Programlamanın ve C++'ın Doğuşu

1970'lerin sonlarına doğru birçok projenin boyutu, yapısal programlama yöntembilimlerinin ve C dilinin başa çıkabileceğinin sınırlara ya yaklaşmıştı ya da ulaşmıştı. Bu problemi çözmek amacıyla nesne yönelimli programlama (OOP – Object Oriented Programming) denilen yeni bir programlama yöntemi ortaya çıkmaya başladı. Nesne yönelimli programlamayı kullanarak, bir programcı çok daha büyük programların üstesinden gelebilecekti. Sorun, o zamanlar en popüler dil olan C'nin nesne yönelimli programlamayı desteklememesiydi. C'nin nesne yönelimli bir versiyonu için duyulan istek en sonunda C++'ın ortayamasına neden oldu.

C++, Bjarne Stroustrup tarafından Murray Hill, New Jersey'deki Bell Laboratuarlarında 1979'da icat edilmeye başlandı. Stroustrup, bu yeni dili ilk önce "C with Classes" ("Sınıflı C") olarak adlandırdı. Ancak, 1983'te dilin ismi C++ olarak değiştirildi. C++ , C dilinin bütününe içermektedir. Bu nedenle C, C++'ın üzerine inşa edildiği bir temeldir. Stroustrup'un C üzerine yaptığı değişikliklerin pek çoğu, nesne yönelimli programlamayı desteklemek amacıyla tasarlanmıştır. Yeni dili C temeli üzerine inşa ederek Stroustrup, nesne yönelimli programlamaya yumuşak bir geçiş sağlamıştır. Bir C programcısının nesne yönelimli yöntembilimin avantajlarından yararlanmadan önce, bütünüyle yeni bir dil öğrenmek zorunda kalmak yerine, sadece bir-iki yeni özellik öğrenmesi gerekecekti.

C++ 1980'lerin büyük bölümünü, arka planda, kapsamlı bir gelişme kaydederek geçirdi. 1990'ların başından itibaren C++ genel kullanım için hazırıldı; üstelik, popüleritesi de adeta patlama yapmıştır. On yıllık sürecin sonuna gelindiğinde C++ en yaygın olarak kullanılan programlama dili halini almıştı. Bugün halen, yüksek performanslı, sistem seviyesinde kod geliştirmek için C++ üstün bir dildir.

C++'ın keşfinin bir programlama dili ortaya çıkarmaya yönelik bir girişim olmadığını anlamak kritik öneme sahiptir. Bilakis, C++ zaten çok başarılı bir dilin geliştirilmiş haliydi. Dillerin gelişimi ile ilgili bu yaklaşım - mevcut bir dil ile başlayıp, onu daha da ileriye taşımak - bugün bile devam etmekte olan bir trendin öncüsü oldu.

---

## İnternet ve Java'nın Gelişimi

Programlama dillerindeki bir sonraki başlıca ilerleme Java ile gerçekleştirilmiştir. Önceleri Oak olarak adlandırılan Java ile ilgili çalışmalar 1991'de Sun Microsystems'da başladı. Java'nın tasarıminının arkasında yer alan esas sürücü güç James Gosling'di. Patrick Naughton, Chris Warth, Ed Frank ve Mike Sheridan'ın da Java üzerinde ayrıca rolü oldu.

Java, söz dizimi ve felsefesi C++'tan alınan yapısal ve nesne yönelimli bir dildir. Java'nın yenilikçi özellikleri, programlama sanatındaki ilerlemelerden çok fazla etkilenmedi (gerci bu ilerlemelerin bir kısmının kuşkusuz etkisi olmuştur); bu özellikleri daha ziyade bilişim ortamında yaşanan değişiklikler biçimlendirdi. İnternet'in genel kullanıma açılması öncesinde, programların birçoğu belirli bir CPU ve belirli bir işletim sistemi için yazılıyor, derleniyor ve yönlendiriliyordu. Programcılar kodlarını yeniden kullanmayı sevdikleri her zaman için geçerli bir durum olsa da, bir programı kolaylıkla bir ortamdan diğerine aktarma becerisi halen erişilmesi gereken bir özelliği ve taşınabilirlik (portability), diğer acil problemlerin yanında, ancak arka koltukta yer bulabiliyordu. Ancak, birçok farklı tipte CPU'nun ve işletim sisteminin birleştirildiği İnternet'in çıkışıyla birlikte, eski taşınabilirlik problemi de yeniden gündeme geldi. Taşınabilirlik problemini çözmek için yeni bir dile ihtiyaç vardı. İşte bu yeni dil Java'ydı.

Java'nın tek önemli özelliği (ayrıca, hızla kabul görmesinin de nedeni), platformlar arasında taşınabilir kod geliştirme becerisi olmasına rağmen Java'yı teşvik eden asıl enerjinin İnternet olmadığını kaydetmek enteresandır. Asıl teşvik, gömülü denetleyiciler (embedded controllers) için yazılım geliştirmek amacıyla kullanılabilecek platformdan bağımsız bir dile (cross-platform language) olan ihtiyaçtır. 1993'te, gömülü denetleyiciler için kod geliştirirken ortaya çıkan platformlar arasında taşınabilirliği ile ilgili hususlarla İnternet için kod geliştirmeye çalışırken de karşılaşıldığı arlık netleşmişti. Hatırlayınız: İnternet, birçok farklı tipte bilgisayarın yaşadığı çok geniş, dağıtık bir bilişim evrenidir. Taşınabilirlik problemini küçük ölçekte çözene tekniklerin aynları, büyük ölçekte İnternet'e de uygulanabilir.

Java, programın kaynak kodunu *bytecode* denilen bir ara dile dönüştürerek taşınabilirliği sağlıyordu. Bu "bytecode" daha sonra Java Sanal Makinesi (Java Virtual Machine - JVM) tarafından çalıştırılıyordu. Bu nedenle, bir Java kodu JVM'in mevcut olduğu her ortamda çalıştırılabilecekti. Ayrıca, JVM'i uygulamak nispeten kolay olduğu için JVM çok sayıda ortamda kolayca kullanılabiliyordu.

Java'nın "bytecode" kullanımını hem C'ye hem de C++'a göre kökten farklılık gösteriyordu. C ve C++ derlendiğinde hemen hemen her zaman çalıştırılabilir makine kodu elde ediliyordu. Makine kodu ise belirli bir CPU ve işletim sistemine sıkı sıkıya bağlıdır. Yani, bir C/C++ programını farklı bir sistemde çalıştırmak istemişseniz, programın yeniden derlenmesi ve bu ortam için spesifik olarak makine kodunun üretilmesi gerekecekti. Bu nedenle, çok çeşitli ortamlarda çalışacak bir C/C++ programı geliştirmek için programın birkaç farklı çalıştırılabilir versiyonuna ihtiyaç vardı. Bu pratik olmadığı gibi aynı zamanda maliyetliydi de. Java'nın ara dil kullanımını zarif ve maliyet açısından hesaplı bir çözümüdü. Aynca bu, C#'ın kendi amaçları doğrultusunda adapte edeceği bir çözümüdü.

Önceden bahsedildiği gibi, Java; C ve C++'ın soyundan gelmektedir. Söz dizimi C'yi temel alır; nesne modeli ise C++'tan geliştirilmiştir. Java kodu ne ileri ne de geri yönde C veya C++ ile uyumlu olmasa da, Java'nın söz dizimi yeteri kadar benzer olduğu için, çok sayıda programcının oluşan mevcut C/C++ programcıları ekibi çok az bir çaba ile Java'ya kayabileceklerdi. Üstelik Java mevcut bir yöntemin üzerine inşa edilmiş ve geliştirilmiş olduğu için Gosling ve arkadaşları yeni ve yenilikçi özelliklere odaklanmakta serbesttiler. Tıpkı Stroustrup'un C++'ı geliştirirken "tekerleği yeniden icat etmesine" gerek olmadığı gibi, Gosling'in de Java'yı geliştirirken bütünüyle yeni bir dil icat etmesi gerekmiyordu. İlaveten, Java'nın ortaya çıkışıyla C ve C++ yeni bilgisayar dillerinin üzerine inşa edileceği onaylanmış bir katman olmuş oluyordu.

## C#'in Doğuşu

Java, İnternet ortamında taşınabilirliği çevreleyen birçok hususu başarıyla ortaya koymuşmasına rağmen, Java'nın da yetersiz kaldığı bazı özellikler hala mevcuttur. Bunlardan biri, "diller arasında uyum içinde çalışma" (cross-language interoperability) olarak tarif edilebilecek özellikle Buna ayrıca, karışık dillerde programlama (mixed-language programming) da denir. Bu, bir dilde üretilmiş bir kodun bir başka dilde üretilmiş kodla birlikte kolaylıkla çalışma becerisidir. Diller arasında uyum içinde çalışma özelliği büyük, dağıtık yazılım sistemleri geliştirirken gereklidir. Bu, ayrıca programlama yazılım bileşenleri için de istenilen bir özellikle. Çünkü en kıymetli bileşen; en çok sayıda işletim ortamında, en çeşitli bilgisayar dilleri yelpazesinde kullanılabilen bileşendir.

Java'nın yoksun olduğu bir diğer özellik ise Windows platformlarıyla tam entegrasyondur. Java programları (Java Sanal Makinesinin kurulmuş olduğunu varsayıarak) Windows ortamında çalıştırılabilir olmalarına rağmen, Java ve Windows yakından ilişkili değildir. Windows dünyada en yaygın olarak kullanılan işletim sistemi olduğundan, Windows için doğrudan destekten yoksun olması, Java için bir dezavantajdır.

Bu ve diğer ihtiyaçlara cevap vermek amacıyla Microsoft, C#'ı geliştirdi. C#, 1990'ların sonlarına doğru Microsoft'ta ortaya çıktı ve Microsoft'un .NET stratejisinin bütününen bir parçası oldu. C# ilk kez 2000'in ortalarında alfa versiyonu olarak piyasaya çıktı. C#'ın baş mimarı, çeşitli büyük başarılıara imza atmış bulunan, dünyanın onde gelen dil uzmanlarından Anders Hejlsberg'dir. Örneğin, Hejlsberg 1980'lerde çok başarılı ve güçlü bir dil olan Turbo Pascal'ın orijinal yazarıdır. Turbo Pascal'ın sadeleştirilmiş uygulaması, geleceğin tüm derleyicileri için standartları belirlemiştir.

C#; C, C++ ve Java ile doğrudan bağlantılıdır. Bu tesadüf değildir. Bunlar, dünyada en yaygın olarak kullanılan - ve en fazla sevilen - programlama dillerinden üçdür. Üstelik bugün profesyonel programcılar neredeyse tamamı C ve C++'ı, birçoğu da Java'yı bilmektedir. C#'ı sağlam ve iyi anlaşılır bir temel üzerine inşa ederek bu dillerden C#'a kolay bir geçiş de sağlanmış oldu. Hejlsberg için "tekerleği yeniden icat etmek" ne gerekli ne de istenilir bir durum olmadığı için, kendisi belirli gelişmelere ve yeniliklere odaklanmakta serbestti.

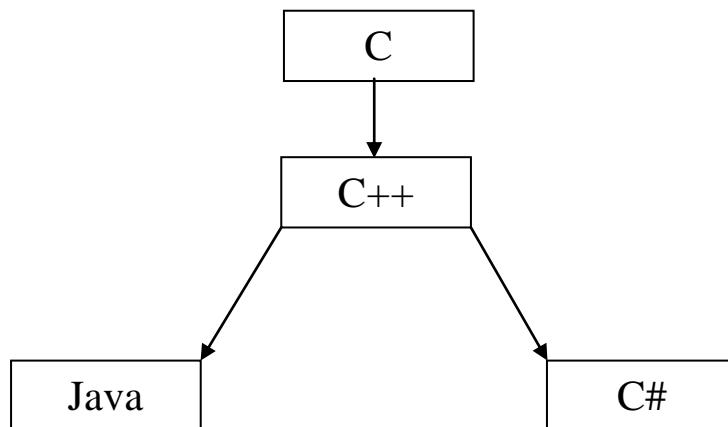
---

C#'ın aile ağacı Şekil 1.1'de gösterilmektedir. C#'ın dedesi C'dir. C'den C# söz dizimini, anahtar kelimelerinin birçoğunu ve operatörlerini almıştır. C#, C++ ile tanımlanan nesne modeli üzerine kurulmuştur ve bu nesne modelini geliştirmiştir. C veya C++'ı biliyorsanız C# ile adeta kendinizi evinizde hissedeceksiniz.

C# ve Java arasındaki ilişki biraz daha karmaşıktır. Daha önce açıklandığı gibi, Java da C ve C++ 'tan türemiştir. Java da C/C++ söz dizimini ve nesne modelini paylaşmaktadır. Tíkla Java gibi, C# da taşınabilir kod üretmek amacıyla tasarlanmıştır. Ancak C#, Java'dan türememiştir. Aksine, C# ve Java ortak bir soyu paylaşan, fakat birçok önemli açıdan farklılık gösteren birer kuzen gibidir. Her şeye rağmen, bunun iyi bir tarafı da vardır: Eğer Java'yı biliyorsanız birçok C# kavramı size tanıdık gelecektir. Aksi de doğrudur: Şayet gelecekte Java öğrenmeniz gereklı olursa, C# ile öğrendiğiniz birçok şey Java'da da geçerli olacaktır.

Bu kitabın konuları içinde uzun uzadıya inceleyeceğimiz C#'ın pek çok yeni özelliği mevcuttur. Ancak C#'ın en önemli özelliklerinden bir kısmı, standart olarak mevcut olan ve yazılım bileşenleri için sunulan destek ile bağlantılıdır. Aslında, C# bileşen yönelimli bir dil (component-oriented language) olarak nitelenmektedir; çünkü C#, yazılım bileşenlerini yazmak için bütünsel destek içermektedir.

Örneğin C#, bileşenleri oluşturan öğeleri, mesela özelliklerini, metotları ve olayları doğrudan destekleyen özellikler içerir. Ancak, C#'ın karışık dillerin hâkim olduğu bir ortamda çalışabilme becerisi belki de C#'ın bileşen yönelimli özelliklerinden en önemlididir.



**ŞEKİL 1.1:** C# aile ağacı

## C# ile .NET Framework Arasındaki Bağlantı

C# kendi başına incelenebilecek bir bilgisayar dili olmasına rağmen C#'ın çalışma ortamı (runtime environment) ile, yani .NET Framework ile özel bir ilişkisi vardır. Bunun iki sebebi vardır. Birincisi, C# başlangıçta Microsoft tarafından .NET Framework için kod geliştirmek amacıyla tasarlanmıştır. İkincisi, C# tarafından kullanılan kütüphaneler, .NET Framework tarafından tanımlanan kütüphanelerdir. Yani, C# dilini .NET ortamından ayırmak

mungkin olsa dahi her ikisi birbiriyle sıkı sıkıya bağlıdır. Bu nedenle, .NET Framework'ü genel olarak anlamak ve bunun C# için önemini kavramak gereklidir.

## .NET Framework Nedir?

.NET Framework, çok dağıtık (highly-distributed), bileşen yönelimli uygulamaların geliştirilmesini ve yürütülmesini destekleyen bir ortam tanımlamaktadır. .NET Framework, farklılık gösteren bilgisayar dillerinin birlikte çalışmasını mümkün kılar ve Windows platformu için güvenlik, taşınabilirlik (programlar açısından) ve ortak bir programlama modeli sağlar. .NET Framework'ün doğası gereği yalnızca Windows ile sınırlı olmadığını (gerci bugün için mevcut tek ortam bu olmasına rağmen) belirtmemiz gereklidir. Bunun anlamı şudur: .NET Framework için yazılmış programlar gelecekte Windows olmayan ortamlara da taşınabilecektir.

.NET Framework C# ile ilişkili olduğu için iki çok önemli unsuru tanımlamaktadır. Bunlardan ilki *Common Language Runtime*'dır (CLR - Ortak Dil Çalışma Zamanı) Bu, programınızın çalışmasını idare eden sistemdir. Diğer avantajlarının yanı sıra CLR, .NET Framework'ün programların taşınabilirliğini mümkün kıyan, karışık dilde programlamayı destekleyen ve güvenliği sağlayan bir parçasıdır.

İkinci unsur ise .NET sınıf kütüphanesidir. Bu kütüphane, programınızın çalışma ortamına erişmesine imkân verir. Örneğin, ekranda bir şey göstermek gibi bir I/O (Giriş/Cıkış) işlemi gerçekleştirmek istiyorsanız, bunun için .NET sınıf kütüphanesini kullanacaksınız. Şayet programlamayı yeni öğreniyorsanız, sınıf terimi size yeni gelebilir. Bu terim kitabın daha sonraki bölümlerinde ayrıntılı açıklanıyor olmakla birlikte, şimdilik kısa bir tanım yeterli olacaktır: Sınıf, programları organize etmeye yarayan nesne yönelimli bir özelliktir. Bir program .NET sınıf kütüphanesinde tanımlanan özelliklerle kısıtlı kalırsa, .NET çalışma zamanı sisteminin desteklendiği her yerde çalıştırılabilir. C# otomatik olarak .NET sınıf kütüphanesini kullandığı için, C# programları .NET ortamlarının tümüne otomatik olarak taşınabilir,

## Common Language Runtime Nasıl Çalışır?

CLR, .NET kodunun çalıştırılmasını (yürütülmesini) idare eder. CLR şu şekilde çalışır: Bir C# programını derlediğinizde, derleyiciden elde edilen çıktı çalıştırılabilir bir kod değildir. Bilakis bu, *Microsoft Intermediate Language* (MSIL) denilen özel bir tipte sözde kod (pseudocode) içeren bir dosyadır. MSIL, spesifik bir CPU'dan bağımsız olan birtakım taşınabilir komutlar tanımlar. Astında, MSIL taşınabilir bir Assembly dili tanımlar. Diğer bir husus da şudur: MSIL her ne kadar Java'nın bytecode'una benziyor ise de her ikisi aynı değildir.

Program çalıştığı zaman ara kodu çalıştırılabilir bir koda dönüştürmek CLR'in görevidir. Böylece, MSIL elde edilecek şekilde derlenen her program CLR'in gerçeklendiği her ortamda çalıştırılabilir. Bu, .NET Framework'ün taşınabilirimi temin etme biçiminin bir parçasıdır.

Microsoft Intermediate Language, bir *JIT* derleyici kullanılarak çalıştırılabilir bir koda çevrilir, "JIT", "Just-In-Time'a karşılık gelir (yani, "Tam Zamanında" demektir). Süreç söyle

---

gelişir: Bir .NET programı çalıştığı zaman CLR, JIT derleyiciyi etkin kılar. JIT derleyici programınızın parçalarının her birine duyulan ihtiyacı temel olarak MSIL'i yerel dile dönüştür. Böylece, C# programınız başlangıçta MSIL üretecek şekilde derlenmiş olsa bile aslında yerel kod gibi çalışacaktır. Bunun anlamı şudur: Programınız başlangıçta yerel kod üretecek şekilde derlenmiş olsaydı, elde etmiş olacağı çalışma hızına yakın bir hızda çalışacak, üstelik MSIL'in taşınabilirlik avantajlarına da kavuşacaktır.

Bir C# programını derlendiğinizde, MSIL'e ek olarak elde edilen bir diğer şey de *metadata* olarak adlandırılanichtetidir. Metadata, programınızın kullandığı verileri tarif eder ve kodunuzun diğer kodlarla etkileşmesini mümkün kılar. Metadata, MSIL ile aynı dosya içinde yer alır.

## Kontrol Altına Alınan ve Alınmayan Kodlar

Genel olarak, bir C# programı yazdığınızda *kontrol altına alınan kod* (*managed code*) denilen bir kod geliştirirsiniz. Kontrol altına alınan kod, Common Language Runtime'in kontrolünde az önce anlatılan şekilde çalıştırılır. Kontrol altına alınan kod CLR'in kontrolünde çalıştığı için, belirli kısıtlamalara tabidir - bunun yanında, bazı avantajlar da elde eder. Kısıtlamalar kolaylıkla tarif edilir ve karşılanır: Derleyici, CLR'a yönelik bir MSIL dosyası üretmelidir (C# bunu gerçekleştirir) ve .NET Framework kütüphanelerini kullanmalıdır (C# bunu da gerçekleştirir). Kontrol altına alınan kodun avantajları pek çoktur. Modern bellek yönetimi, dilleri karıştırma becerisi, daha sıkı güvenlik, sürüm kontrol desteği ve etkileşecek yazılım bileşenleri için net bir yöntem, bu avantajlar arasında yer alır.

Kontrol altına alınan kodun tam tersi ise *kontrol altına alınmayan koddur* (*unmanaged code*). Kontrol altına alınmayan kod, Common Language Runtime allında çalışmaz. Yani, .NET Framework'ün geliştirilmesi öncesinde yer alan Windows programlarının tümü kontrol altına alınmayan kod kullanırlar. Kontrol allına alınan ve alınmayan kodların bir arada çalışması mümkün değildir; böylece C#'ın kontrol altına alınabilir kod ürettiği gerçeği, C#'ın önceden mevcut programlarla birlikte çalışabilme becerisini kısıtlamaz.

## Common Language Specification

Kontrol altına alınan kodların tümü CLR ile sağlanan avantajlardan yararlanıyor olsa da, şayet kodunuz farklı dillerde yazılmış programlar tarafından kullanılacaksa, kullanılabilirliği en üst seviyeye çıkarmak için kodunuzun *Common Language Specification*'a (CLS - Ortak Dil Spesifikasiyonu) sıkı sıkıya yapışması gereklidir. CLS, farklı dillerde ortak olan özellikleri tarif eder. CLS uyumu, diğer diller tarafından kullanılacak yazılım bileşenleri geliştirirken özellikle önemlidir. CLS, *Common Type System*'in (CTS - Ortak Tip Sistemi) bir altkümesini içerir. CTS, veri tiplerini ilgilendiren kuralları tarif eder. Elbette, C# hem CLS'i hem de CTS'i desteklemektedir.

İKİNCİ BÖLÜM

2

---

# C#'A GENEL BİR BAKIŞ

Bir programlama dilini öğrenirken zorlanılan konuların açık farkla en başında yer alan husus, öğelerin birbirinin tek başına mevcut olmadığı gerçeğidir. Aksine, dilin bileşenleri birlikle çalışırlar. Bu şekilde birbiriyle bağıntılı olmak, C#'ın diğer özelliklerine dokunmadan tek özelliğini anlatmayı zorlaştırır. Genellikle, bir özelliği ele alırken bir başkası hakkında ön bilgi sahibi olmak gereklidir. Bu problemin üstesinden gelmeye yardımcı olmak amacıyla, bu bölümde bazı C# özelliklerinin kısa bir özeti sunulmaktadır. Bir C# programının genel yapısı, bazı temel kontrol ifadeleri ve operatörler, bu bölümde ele alınan özellikler arasında yer almaktadır. Bu bölümde çok fazla ayrıntıya girilmezken, daha çok herhangi bir C# programında rastlanabilen genel kavramlar üzerine konsantr olunuyor. Burada ele alınan konuların birçoğu, Kısım I'in kalan bölümlerinde daha ayrıntılı olarak incelenmektedir.

## Nesne Yönelimli Programlama

C#'ın merkezinde nesne yönelimli programlama (OOP - Object Oriented Programming) yer almaktadır Nesne yönelimli yöntembilim C#'tan ayrılmaz; tüm C# programları en azından bir ölçüye kadar nesne yönelimlidir. Nesne yönelimli programmanın C# için öneminden dolayı, basit bile olsa bir C# programı yazmaya başlamadan önce nesne yönelimli programmanın temel prensiplerini kavramak yararlıdır.

Nesne yönelimli programlama, programlama yaklaşımı için güçlü bir yöntemdir. Bilgisayarın icadıyla birlikte programlama yöntembilimleri, her şeyden önce programların artan karmaşıklığına uyum sağlamak amacıyla çok ciddi ölçüde değişti. Örneğin, bilgisayarların icat edildiği yıllarda programlama, bilgisayarın ön paneli kutlanılarak, ikili (binary) makine komutlarına karşılık gelen düğmelere basmak suretiyle gerçekleştiriliyordu. Programlar birkaç yüz satır uzunluğunda olduğu müddetçe bu yöntem ise yarıyordu. Programlar büyüğünce Assembly dili icat edildi; böylece programcılar, makine komutlarına karşılık gelen sembolik işaretleri kullanarak daha büyük ve artan karmaşıklıktaki programları ele alabileceklerdi. Programlar büyümeye devam ettikçe, programcılara karmaşıklıkla başa çıkılmak için daha fazla araç sunan FORTRAN ve COBOL gibi yüksek seviyeli diller ortaya çıktı. Bu ilk diller, artık çatlama noktasına geldiğinde yapısal programlama icat edildi.

Şunu bir düşünün: Programmanın her bir kilometre taşında, programcılar giderek daha da artan karmaşıklıkla başa çıkmalarına imkân veren teknikler ve araçlar geliştirilmiştir. Bu sürecin her adımında yeni yöntem, önceki yöntemlerin en iyi elemanlarını almış ve ilerlemiştir. Bu, nesne yönelimli programlama için de geçerlidir. Nesne yönelimli programlama öncesinde birçok proje, yapısal programmanın artık işe yaramadığı noktaya yaklaşmıştı (veya aşmıştı). Karmaşıklığı daha iyi ele alabilen bir yöntemlere ihtiyaç vardı ve çözüm, nesne yönelimli programlamayı.

Nesne yönelimli programlama, yapısal programlamaya özgü en iyi kavramları almış ve bunları bazı yeni kavramlarla birleştirmiştir. Sonuçta programlar farklı ve daha iyi şekilde organize edilebilecekti. En genel anlamda, bir program iki yöntemden biri kullanılarak organize edilebilir: kodu etrafında (meydana gelenler) veya verileri etrafında (etkilenenler). Sadece yapısal programlama teknikleri kullanılarak programlar genellikle kod etrafında organize edilir. Bu yöntem ayrıca “veriye etki eden kod” olarak da düşünülebilir.

Nesne yönelimli programlar diğer şekilde çalışırlar. Bu tür programlar “koda erişimi kontrol eden veri” temel prensibine dayanarak veri etrafında organize edilirler. Nesne yönelimli bir dilde, verileri ve bu verilere etki etmeye izin verilen kodu tanımlarsınız. Böylece, bir veri tipi bu veri üzerine uygulanabilecek işlemleri tam olarak tanımlar.

Nesne yönelimli programmanın prensiplerini desteklemek amacıyla, C# da dâhil olmak üzere tüm nesne yönelimli diller üç ortak özelliğe sahiptir: ilişkili verilerin paketlenmesi (encapsulation), çok biçimlilik (polymorphism) ve kalıtım. Şimdi bunları tek tek inceleyelim.

## İlişkili Verilerin Paketlenmesi

*Verilerin paketlenmesi* (encapsulation), kodun manipüle ettiği verilerle kodu birbirine bağlayan ve her ikisini dışarıdan gelebilecek istenmeyen etkilerden ve hatalı kullanımlardan koruyan bir programlama mekanizmasıdır. Nesne yönelimli bir dilde kod ve veri, kendi kendisini içeren bir kara kutu oluşturacak şekilde birbirlerine bağlanabilir. Gerekli veri ve kodun tümü kutunun içinde mevcuttur. Kod ve veri bu şekilde bağlanınca bir nesne (*object*) oluşturulmuş olur. Diğer bir deyişle nesne, verilerin paketlenmesini (encapsulation) destekleyen bir aygittır.

Nesne içindeki kod, veri veya her ikisi de bu nesneye *özel* (*private*) veya *açık* (*public*) olabilirler. Özel kod veya veri, sadece bu nesnenin parçaları tarafından bilinir ve erişilebilir. Yani özel kod veya veri, söz konusu nesnenin dışında kalan program parçaları tarafından erişilemez. Kod veya veri açık olduğunda ise, söz konusu kod ve veri bir nesnenin içinde tanımlanmış olsa bile programınızın diğer parçaları tarafından erişebilirler. Nesnelerin açık kısımları tipik olarak, özel öğelere kontrollü bir arayüz sağlamak amacıyla kullanılır.

C#'ın temel veri paketleme birimi *sınıftır* (*class*). Sınıf, bir nesnenin şeklini tanımlar. Sınıf, hem verileri hem de bu veriler üzerinde işlem yapacak kodu belirtir. C#'ta, *nesneleri* inşa etmek amacıyla sınıf spesifikasyonu kullanılmaktadır. Nesneler bir sınıfın örnekleridir. Yani, sınıf aslında bir nesnenin ne şekilde inşa edileceğini belirten birtakım planlardan ibarettir.

Sınıfı oluşturan kod ve veri söz konusu sınıfın *üyeleri* (*members*) olarak adlandırılırlar. Özellikle, sınıf tarafından tanımlanan verilerden *üye değişkenler* (*member variables*) veya *örnek değişkenler* (*instance variables*) olarak söz edilir. *Metot* (*method*), C#'ta alt rutin için kullanılan bir terimdir. Eğer C/C++'a aşınaysanız C# programcısının *metot* diye adlandırdığı şeyin C/C++ programcısı tarafından *fonksiyon* (*islev*) olarak adlandırıldığını bilmek yararlı olabilir. C# dili C++'tan doğrudan türediği için C# metotlarından söz edildiğinde *fonksiyon* (*islev*) terimi de yaygın olarak kullanılmaktadır.

## Çok Biçimlilik

Çok biçimlilik (polymorphism - Yunanca'da “birçok form” anlamına gelir), genel bir etkinlikler sınıfına erişmek için bir arayüze imkân veren bir özelliktir. Çok biçimliliğin en basit örneği, otomobil direksiyonudur. Gerçekte hangi tipte bir vites mekanizması kullanılırsa

kullanılsın direksiyon (arayüz) hep aynı şekildedir. Yani, arabanız ister manuel vitesli, ister otomatik vitesli olsun, direksiyon hep aynı şekilde çalışır, Böylece, direksiyonu sola çevirmek, hangi tipte vites kullanılırsa kutlanılsın arabanın sola dönmesine neden olur. Tek tip bir arayüzün avantajı, elbette, direksiyonun nasıl çalıştıracağınızı bildikten sonra herhangi bir tipte arabayı sürebilecek olmanızda yatar.

Aynı prensip programlamaya da uygulanabilir. Örneğin, bir *yığın* (ilk giren son çıkar esasına dayanan bir liste) düşünün. Üç farklı tipte yiğina gereksinim duyan bir programınız olabilir. Öyle ki, yiğinlardan biri tamsayılar için, biri kayan noktalı değerler için, diğeri de karakterler için kullanılmaktadır. Bu örnekte, yiğinlarda saklanan veriler farklı bile olsa, yiğinların her birini gerçekleyen algoritma aynıdır. Nesne yönelimli olmayan bir dilde, her biri farklı bir isim kullanan üç farklı yiğin rutini geliştirmeniz gerekecekti. Ancak, C#'ta çok biçimlilik özelliğinden ötürü üç spesifik durumun üçyle de çalışan tek bir genel yiğin rutini geliştirebilirsiniz. Bu sayede, bir yiğin kullanmayı öğrendiğiniz andan itibaren hepsini kullanabilirisiniz.

Daha genel bir yaklaşımla, çok biçimlilik kavramı sık sık “tek arayüz, çok sayıda metot” deyişile de ifade edilir. Bu, bir grup ilişkili etkinlik için genel bir arayüz tasarlamak mümkündür, anlamına gelmektedir. Çok biçimlilik, *genel bir etkinlik sınıfını* belirtmek amacıyla aynı arayüzün kullanılmasına imkân vererek karmaşıklığı azaltmaya yarar. Her bir durum için hangi *spesifik etkinliğin* (yani metodun) uygulanacağını tercih etmek derleyicinin görevidir. Siz programcılar bu tercihi elle yapmanız gereklidir. Sizin sadece genel arayüzü hatırlamamız ve değerlendirmeniz gereklidir.

## Kalıtım

Kalıtım (*inheritance*), bir nesnenin diğer bir nesnenin özelliklerini ele geçirebilmesini sağlayan bir yöntemdir. Bu önemlidir, çünkü kalıtım, hiyerarşik sınıflandırma kavramını destekler. Bir düşünürseniz, bilgilerin büyük bölümü hiyerarşik (yani, yukarıdan aşağıya doğru) sınıflandırma sayesinde yönetilebilir kılınır. Örneğin, kırmızı tatlı elma, *elma* sınıfımasının bir parçasıdır. Elma, *meyve* sınıfının bir parçasıdır. Meyve ise daha büyük bir sınıf olan *yiyecek* sınıfının altında yer alır. Yani, *yiyecek* sınıfı, aynı zamanda mantıksal olarak kendisinin bir alt sınıfı olan *meyve* sınıfına da uygulanabilecek belirli özelliklere (yenilebilir, besleyici vs.) sahiptir. Bu özelliklere ek olarak, *meyve* sınıfının da kendisini diğer sınıflardan ayırt eden, kendisine özgü özellikleri (sulu, tatlı vs.) vardır. *Elma* sınıfı bu tür özelliklerden elmaya özgü olanları (ağaçta yetişir, tropik meyve değildir vs.) tanımlar. Kırmızı Tatlı elma bu durumda, kendisinden önce gelen tüm sınıfların tüm özelliklerini kalıtım yoluyla devralacaktır ve bu özelliklerden sadece kendisini eşsiz kıılanları tanımlayacaktır.

Kalıtım kullanılmasydı, her nesne kendi özelliklerinin tümünü açıkça tanımlamak zorunda kalacaktı. Kalıtım sayesinde bir nesnenin kendi sınıfı içinde sadece kendisini eşsiz kıılan özellikleri tanımlaması yeterlidir. Nesne, genel özelliklerini atalarından kalıtım yoluyla devralabilir. Böylece, bir nesnenin daha genel bir durumun spesifik bir örneği olmasını mümkün kıılan, kalıtım mekanizmasıdır.

## İlk Basit Program

Artık gerçek bir C# programına göz atma zamanı geldi. Aşağıda gösterilen programı derleyerek ve çalıştırarak başlayacağınız:

```
/*
    Bu basit bir C# programıdır.
    Bu programa Example.cs adını verelim.
*/
using System;

class Example {

    // Bir C# programı Main()'e çağrıda bulunarak başlar.
    public static void Main() {
        Console.WriteLine("Basit bir C# programı.");
    }
}
```

Bu kitap baskıya hazırlanırken mevcut tek C# ortamı Visual Studio .NET idi. Visual Studio .NET'i kullanarak bir C# programını iki şekilde düzenleyip derleyebilir ve çalıştırabilirsiniz.

Birincisi, komut satırından çalıştırılan derleyiciyi (**csc.exe**) kullanabilirsiniz, İkincisi, Visual Studio Integrated Development Environment'ı (IDE - Bütünleşik Geliştirme Ortamı) kullanabilirsiniz. Burada her iki yöntem de ele alınıyor, (Şayet farklı bir derleyici kullanıyorsanız, derleyicinizin talimatlarını izleyin.)

### C# Komut Satırı Derleyicisini (**csc.exe**) Kullanmak

Ticari projelerinizde muhtemelen Visual Studio IDE'yi kullanacak olmanızı rağmen C# komut satırı derleyicisi, bu kitapta yer alan örnek programların birçoğunu derlemek ve çalıştmak için en kolay yöntemdir. Programları C# komut satırı derleyicisi kullanarak geliştirmek ve çalıştmak için aşağıda yer alan üç adımı izleyeceksiniz:

1. Bir metin editörü kullanarak programı kodlayın.
2. Programı derleyin.
3. Programı çalıştırın.

### Programı Kodlamak

Bu kitapta yer alan programlar Osborne'un Web sitesinde mevcuttur: [www.osborne.com](http://www.osborne.com). Ancak, programları elle kodlamak isterseniz, bu konuda özgürsünüz. Bu durumda, Notepad gibi bir metin editörü kullanarak programı bilgisayarınıza kodlamalısınız. Unutmayın, salt metin dosyaları oluşturmalısınız; biçimlendirilmiş kelime işlemci dosyaları oluşturmamaya dikkat edin, çünkü kelime işlemci dosyasındaki biçimlendirme bilgileri C# derleyicisini şaşırtacaktır. Programı kodlarken, dosyaya `Example.cs` adını verin.

## Programı Derlemek

Programı derlemek için, kaynak dosyanın ismini komut satırında aşağıda gösterildiği gibi belirterek, C# derleyicisini (**csc.exe**) çalıştırın:

```
C:\>csc Example.cs
```

**csc** derleyicisi, programın MSIL versiyonunu içeren **Example.exe** adında bir dosya oluşturur. MSIL çalıştırılabilir bir kod olmamasına rağmen, yine de bir **exe** uzantılı dosya içinde yer alır. **Example.exe** dosyasını çalıştırılmaya kalkıştığınızda Common Language Runtime otomatik olarak JIT derleyiciyi çağırır. Her şeye rağmen dikkatli olun; şayet .NET Framework’ün kurulu olmadığı bir bilgisayarda **Example.exe** dosyasını (veya MSIL içeren herhangi bir başka **exe** dosyasını) çalıştırmayı deniyorsanız, CLR mevcut olmadığı için program çalışmayaacaktır.

**NOT** **csc.exe** derleyicisini çalıştırılmaya başlamadan önce tipik olarak //Program Files/Microsoft Visual Studio .NET/Vc7/Bin dizini altında yer alan vcvars32.bat isimli toplu işlem (batch) dosyasını çalıştırmanız gerekebilir. Alternatif olarak, C# için halen ilk kullanımına hazırlanmış bir komut satırı oturumunu da başlatabilirsiniz. Bunun için, görev çubüğünün Start | Programs menüsündeki Microsoft Visual Studio .NET seçeneği altında gösterilen araç listesinden Visual Studio .NET Command Prompt seçeneğini işaretlemelisiniz.

## Programı Çalıştırmak

Programı gerçekten çalıştırmak için, aşağıda gösterildiği gibi, komut satırında sadece programın ismini yazmak yeterlidir:

```
C:\>Example
```

Program çalışlığında, aşağıdaki çıktı ekranda görünür:

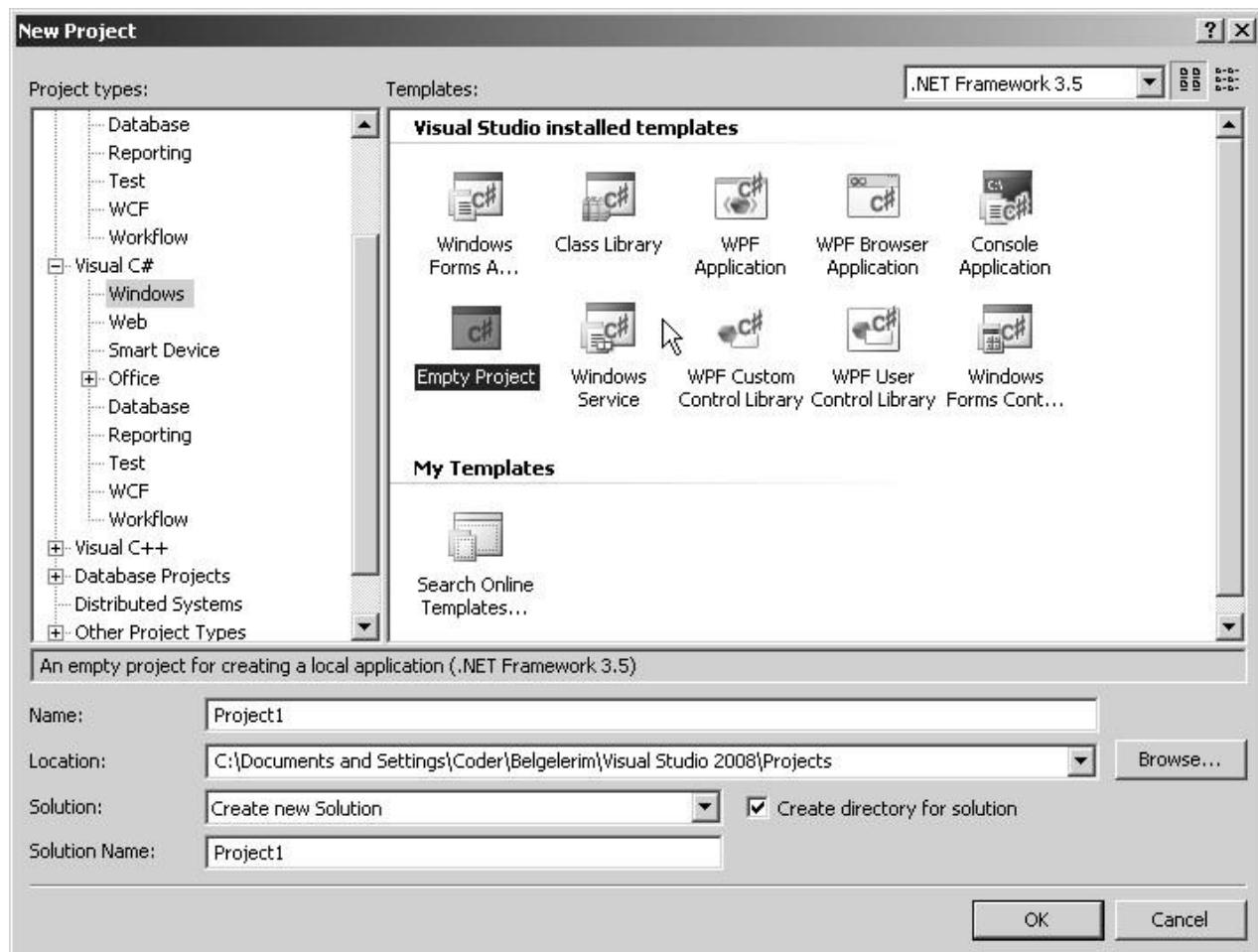
```
Basit bir C# programı.
```

## Visual Studio IDE'yi Kullanmak

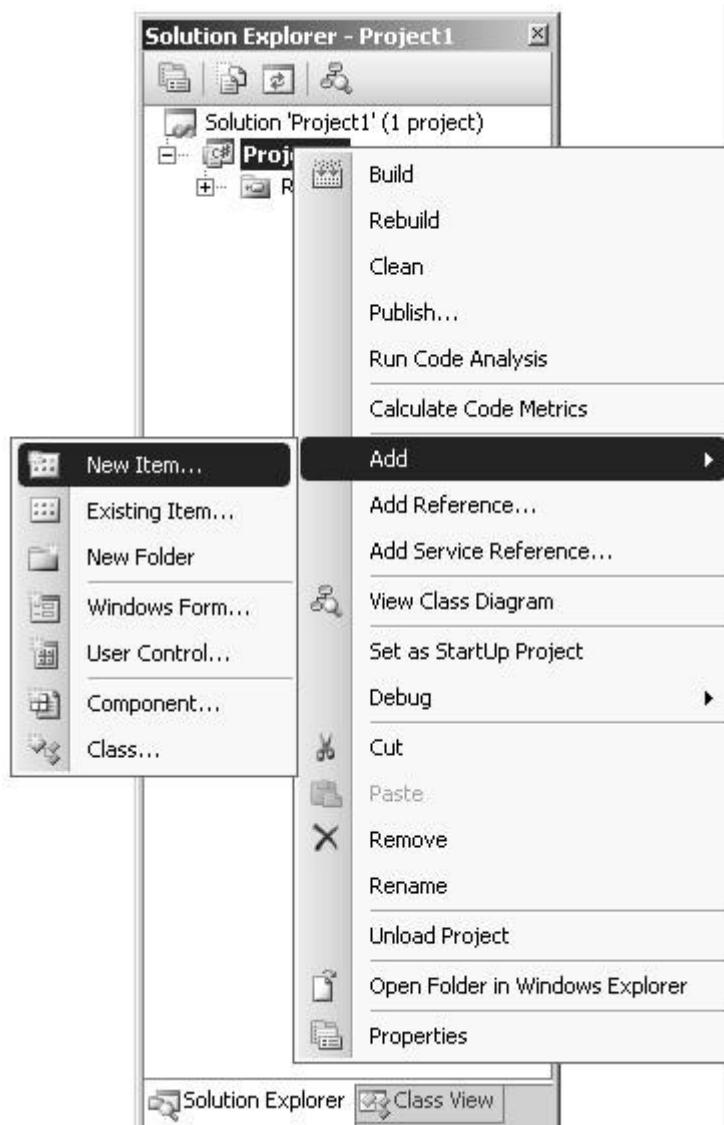
Visual Studio .NET sürüm 7'den itibaren Visual Studio IDE, C# programlarını derleyebilir. Visual Studio .NET sürüm 7'yi kullanarak bir C# programını düzenlemek, derlemek ve çalıştırmak için aşağıdaki aşamaları izleyeceksiniz. (Visual Studio'nun farklı bir sürümüne sahipseniz, farklı aşamaları izlemeniz gerekebilir.)

1. File | New | Project komutunu seçerek yeni, boş bir C# projesi oluşturun.

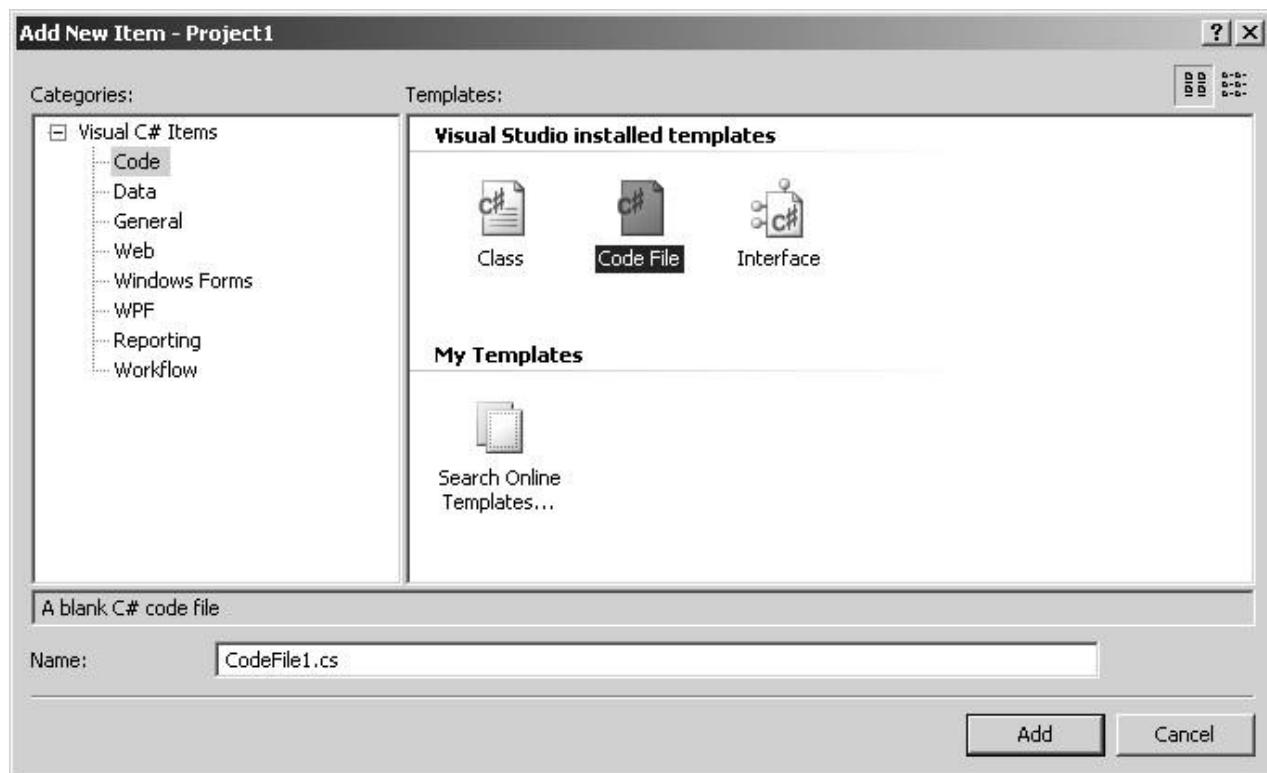
2. Visual C# | Windows seçeneğini işaretleyin, sonra aşağıda gösterildiği gibi Empty Project seçeneğini işaretleyin.



3. Proje oluşturulduktan sonra, Solution Explorer penceresinde görünen proje isminin üzerine sağ fare tuşıyla tıklayın. Daha sonra, çıkan bağlamsal menüyü kullanarak Add seçeneğini işaretleyin. Bundan sonra, Add New Item komutunu seçin. Ekrانınız şu şekilde görünecektir:



4. Add New Item iletişim kurusunu görünce Visual C# Items | Code seçeneğini ve son olarak da, Code File seçeneğini işaretleyin. Ekranınız şu şekilde görünecektir:



5. Programınızı kodlayın ve dosyaya `Example.cs` ismini vererek kaydedin. (Bu kitaptaki kodları [www.osborne.com](http://www.osborne.com) adresinden indirebileceğiniz hatırlınızdan çıkarmayın.) Kaydetme işlemi bittiğinde ekranınız Şekil 2.1'deki gibi görünecektir.
6. Build | Build Solution komutunu seçerek programı derleyin.
7. Debug | Start Without Debugging komutunu seçerek programı çalıştırın.

Programı çalıştırdığınızda Şekil 2.2'deki pencereyi göreceksiniz.

#### NOT

**Bu kitaptaki örnek programları derlemek ve çalıştırmak amacıyla her biri için yeni bir proje dosyası oluşturmanıza gerek yoktur. Bunun yerine, aynı C# projesini kullanabilirsiniz. Tek yapmanız gereken mevcut dosyayı silip yeni dosyayı eklemekten ibarettir. Bunun ardından yeniden derleyin ve çalıştırın.**

Önceden açıklandığı gibi, bu kitabın birinci kısmındaki kısa programlar için `csc` komut satırı derleyicisini kullanmak çok daha basit bir yaklaşımındır. Elbette tercih sizindir.

```

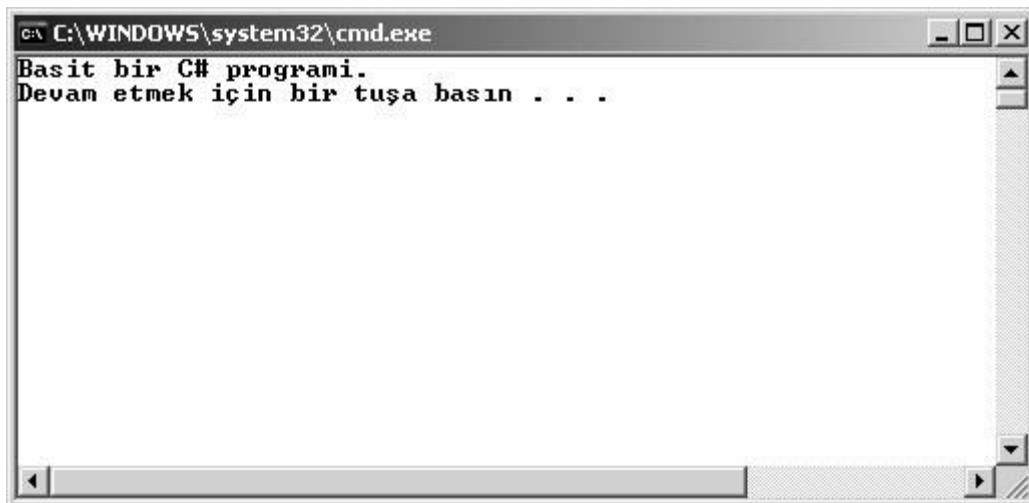
/*
    Bu basit bir C# programidir.
    Bu programa Example.cs adini verelim.
*/
using System;

class Example {

    // Bir C# programi Main()'e cagrida bulunarak baslar.
    public static void Main() {
        Console.WriteLine("Basit bir C# programi.");
    }
}

```

**ŞEKİL 2.1:** `Example.cs` proje ekranı



**ŞEKİL 2.2:** `Example` programının Visual Studio IDE altında çalışırken çıktısı

## Satır Satır İlk Örnek Program

`Example.cs` oldukça kısa olmasına rağmen, tüm C# programlarında ortak olan temel özelliklerden birkaçını içerir. Şimdi gelin, programın isminden başlayarak her parçasını yakından inceleyelim.

C# programının ismi isteğe bağlıdır. Program dosyasının isminin çok önem taşıdığı bazı bilgisayar dillerinden (en meşhurlarından biri Java) farklı olarak, C#'la böyle bir durum söz konusu değildir. Örnek programa `Example.cs` ismini vermeniz istendi, böylece programı derleme ve çalışma ile ilgili komutlar etkili olacaktır. Fakat, C# söz konusu olduğu müddetçe dosyaya başka bir isim de verebilirdiniz. Örneğin, önceki örnek program `Sample.cs`, `Test.cs`, hatta `x.cs` olarak adlandırılabilir.

Geleneksel olarak, C# programları .cs uzantısını kullanırlar. Siz de bu geleneğe uymalısınız. Ayrıca, birçok programcı dosyaya dosya içinde tanımlanmış ana sınıfın ismini verirler. **Example.cs** dosya isminin tercih edilmesinin nedeni de budur. C# programlarının isimleri keyfi olduğu için, bu kitaptaki örnek programların birçoğuunda isimler açıkça belirtilmiyor. Kendi tercih ettiğiniz isimleri kullanabilirsiniz.

Program aşağıdaki satırlarla başlıyor:

```
/*
    Bu basit bir C# programıdır.
    Bu programa Example.cs adını verelim.
*/
```

Bu bir *açıklamadır* (comment). Diğer pek çok programlama dilinde olduğu gibi, C# da programın kaynak dosyasına kendi görüşlerinizi eklemenize imkan verir. Açıklamanın içeriği, derleyici tarafından dikkate alınmaz. Açıklama, programın işleyişini programın kaynak kodunu okumakta olan herhangi birine tarif eder veya açıklar. Bu örnekteki açıklama, programı tarif ediyor ve kaynak dosyayı **Example.cs** olarak adlandıracığınızı hatırlatıyor. Elbette, gerçek uygulamalarda açıklamalar, genellikle programın bazı parçalarının işleyişini veya belirli bir özelliğin ne işe yaradığını açıklarlar.

C# üç farklı açıklama stilini destekler. Programın en üstünde yer alan ve yukarıda da gösterilen birden çok satırdan oluşan açıklamadır. (multiline comment). Bu tür bir açıklama /\* ile başlamalı ve \*/ ile sona ermelidir. Bu iki açıklama simgesi arasında yer alanların hiçbirini derleyici tarafından dikkate alınmaz. İsminden anlaşılacağı gibi, birden çok satırdan oluşan açıklama, birkaç satır uzunluğunda olabilir. Programdaki bir sonraki satır şudur:

```
using System;
```

Bu satır programın **System** isim uzayını kullandığını gösteriyor. C#'ta *isim uzayı* (namespace), deklaratif (tanımlamalara ayrılan) bir bölge tanımları. İsim uzaylarını daha sonra ayrıntılı olarak ele alacağız. Simdilik şu kadar bahsedelim: Bir isim uzayı, bir takım isimleri diğerlerinden ayırmaya yaranan yöntemlerden biridir. Aslında, bir isim uzayında deklare edilen isimler başka bir isim uzayında deklare edilen aynı isimlerle çelişmeyecektir. Örnek programda **System** isim uzayı kullanıyor **System**, C#'ın kullandığı bir kütüphane olan .NET Framework sınıf kütüphanesi ile ilişkili öğeler için ayrılmış bir isim uzayıdır, **using** anahtar kelimesi sadece, programın belirtilen isim uzayındaki isimleri kullandığını ifade eder.

Programda yer alan bir sonraki kod satırı şudur:

```
class Example {
```

Bu satırda yeni bir sınıfın tanımlandığını deklare etmek için **class** anahtar kelimesi kullanılıyor. Önceden bahsedildiği gibi sınıf, C#'ın temel veri paketleme birimidir. **Example** sınıfının ismidir. Sınıf tanımı açılış kümeye parantezi ( ) ile başlar ve kapanış kümeye parantezi ( ) ile sona erer. İki kümeye parantezi arasında yer alan elemanlar sınıfın üyeleriidir. Şu an için sınıfın ayrıntılarını çok fazla dert etmeyin. Yalnız, C#'ta tüm program faaliyetlerinin tek bir

sınıf içinde meydana geldiğini de gözden kaçırmayın. C# programlarının tümünün neden (en azından bir parça) nesne yönelimli olduğunun bir sebebi de işte budur.

Programdaki bir sonraki satır aşağıda da gösterilen *tek satırlık açıklamadır*:

```
// Bir C# programı Main()'e çağrıda bulunarak baslar.
```

Bu, C# tarafından desteklenen bir diğer açıklama tipidir. Tek satırlık açıklama // ile başlar ve satırın sonunda biter. Genel bir kural olarak, programcılar birkaç satırlık açıklamaları daha uzun görüşlerini açıklamak için, tek satırlık açıklamaları ise kısa, satır-satır açıklamalar için tercih ederler.

Bir sonraki kod satırı aşağıda gösterilmiştir:

```
public static void Main() {
```

Bu satır **Main()** metodu ile başlıyor. Önceden bahsedildiği gibi, C#'ta alt rutinler metot olarak adlandırılır. Bu satırın öncesinde yer alan açıklamadan da anlaşılacağı gibi bu satır, programın çalışmaya başlayacağı satırdır. C# uygulamalarının tümü **Main()**'i çağırarak çalışmaya başlarlar. (Bu, C/C++ programlarının çalışmaya **main()**'den başlamalarına benzer.) Bu satırın her parçasını tam olarak açıklamak için henüz erken, çünkü bu, bazı başka C# özelliklerinin ayrıntılı olarak anlaşılmamasını gerektiriyor. Ancak, bu kitaptaki örneklerin birçoğu bu kod satırını kullanacağı için şimdilik buna kısaca göz atalım.

**public** anahtar kelimesi bir erişim belirleyicisidir, Bir erişim belirleyicisi (*access specifier*), programın diğer bölümlerinin bir sınıfın üyesine nasıl erişebileceklerini belirler. Bir sınıf üyesinin öncesinde **public** yer alırsa bu üye, deklare edildiği sınıfın dışında kalan kod tarafından erişilebilir. (**public**'ın karşıtı **private**'dır. **private**, bir üyenin tanımlı olduğu sınıfın dışındaki kod tarafından kullanılmasını öner.) Bu örnekte **Main()**, **public** olarak deklare edilmektedir, çünkü **Main()**, program çalıştığında kendisinin ait olduğu sınıf dışındaki kod (yani, işletim sistemi) tarafından çağrılacaktır.

#### NOT

**Bu kitap baskıya hazırlanırken, C# aslında Main()'in public olarak deklare edilmesini şart koşmuyordu.** Ancak, Visual Studio .NET tarafından desteklenen bir çok örnekte bu şekilde deklare edilmektedir. Bu, ayrıca birçok C# programcısının da tercih ettiği bir yöntemdir. Bu nedenlerden ötürü, bu kitapta da **Main()**, **public** olarak deklare edilecektir. Yine de, biraz farklı bir şekilde deklare edildiğini görürseniz şaşırmayın.

**static** anahtar kelimesi **Main()**'in sınıfına ait bir nesne tanımlanmadan önce **Main()**'in çağrılmasına imkan veriyor. **Main()**, programın başlangıcında çağrıldığı için **static** anahtar kelimesinin kullanılması gereklidir. **void** anahtar kelimesi sadece **Main()**'in bir şey döndürmediğini derleyiciye bildiriyor. Daha sonra öğreneceğiniz gibi, metodlar da değer döndürebilir. **Main'**ın peşinden gelen boş parantezler, **Main'**e hiçbir bilgi aktarılmadığını gösteriyor. **Main()**'e veya herhangi bir başka metoda bilgi aktarmanın mümkün olduğunu daha sonra öğreneceksiniz. Bu satırda en son karakter ise { karakteridir.

Bu, **Main()**'in gövdesinin başladığını işaret ediyor. Bir metodu oluşturan kodun tümü, metodun açılış küme parantezi ile kapanış parantezi arasında yer alacaktır.

Kodun bir sonraki satırı aşağıda gösterilmiştir. Bu satırın **Main()**'in içinde yer aldığına dikkat edin.

```
Console.WriteLine("Basit bir C# programı.");
```

Bu satır ekranda "Basit bir C# programı." karakter katarını çıktı olarak gösterir. Bu karakter katarının peşinden yeni satır gelir. Program çıktısı aslında standart olarak yerleşik bir metot olan **WriteLine()** ile gerçekleştirilir. Bu örnekte **WriteLine()**, kendisine aktarılan karakter katarını ekranda gösterir. Bir metoda aktarılan bilgiye *argüman (argument)* denir. **WriteLine()**, karakter katarlarına ek olarak başka tür bilgilerin gösterilmesi için de kullanılabilir. Bu satır, konsol giriş/çıkışlarını (I/O) destekleyen bir önceden tanımlı sınıfın ismi olan **Console** ile başlıyor. **Console**'u **WriteLine()** ile birleştirerek **WriteLine()**'in **Console** sınıfının bir üyesi olduğu derleyiciye bildiriliyor. C#'ın konsol çıktısını tanımlamak için nesne kullanıyor olması gerçeği, C#'ın nesne yönelimli özelliğinin bir başka kanıtıdır.

Dikkat ederseniz, **WriteLine()** ifadesi noktalı virgül ile sona eriyor, keza programın başlarında yer alan **using System** ifadesi de. C#'ta tüm ifadeler noktalı virgül ile sona erer. Programdaki kimi diğer satırların noktalı virgül ile sona ermemesinin nedeni bunların teknik olarak birer ifade olmamasıdır.

Program içindeki ilk } simgesi **Main()**'i sonlandırıyor, sonuncu } simgesi de **Example** sınıf tanımını sona erdiriyor.

Son bir husus: C#'ta büyük - küçük harf (harf kipi) ayrimı vardır. Bunu unutmak ciddi problemlere neden olabilir. Örneğin, kazara **Main** yerine **main** ya da **WriteLine** yerine **writeline** yazarsanız, yukarıdaki program hatalı olacaktır. Üstelik, C# derleyicisi **Main()** metodunu içermeyen sınıfları *derleyecek* olmasına rağmen, bunları çalıştırmanın yolu mevcut değildir. Bu nedenle, **Main**'i yanlış yazmış olsanız bile derleyici programınızı derleyecektir. Ancak, **Example.exe**'nin başlangıç noktasının tanımlanmadığını bildiren bir de hata mesajı ile karşılaşacaksınız.

## Söz Dizimi Hatalarını Kontrol Altına Almak

Eğer şimdije kadar yapmamışsanız önceki programı kodlayın, derleyin ve çalıştırın. Önceden sahip olduğunuz programlama deneyiminizden de bilebileceğiniz gibi, bilgisayarınıza kodları girerken bir şeyi kazara hatalı yazmak oldukça kolaydır. Neyse ki, hatalı bir şey girdiğinizde ve derleyici bunu derlemeye çalıştığında, *söz dizimi hatası (syntax error)* mesajını verecektir. C# derleyicisi ne yazmış olursanız olun kaynak kodunuzdan anlamlı bir şeyler çıkarmaya çalışır. Bu nedenle, rapor edilen hata problemin asıl nedenini her zaman yansıtmayabilir. Önceki programda, örneğin, **Main()** metodundan sonraki açılış küme parantezinin kazara eksik olması, **csc** komut satırı derleyicisi kullanılarak derlendiğinde,

aşağıdaki bir dizi hata mesajının üretilmesine neden olur. (IDE kullanılarak derlendiğinde de benzer hatalar ortaya çıkar.)

```
Example.cs(12, 28) : error CS1002: ; expected
Example.cs(12, 22) : error CS1002: Invalid token '(' in class,
struct, or interface member declaration
Example.cs(15, 1) : error CS1022: Type or namespace
definition, or end-of-file expected
```

Açıkça görülmektedir ki, ilk hata mesajı tamamen yanlıştır, çünkü eksik olan noktalı virgül değil, kümeye parantezidir. Son iki mesaj ise aynı ölçüde karışiktır.

Konunun özeti şu: Programınız bir söz dizimi hatası içerdiginde, derleyicinin mesajlarını her zaman olduğu gibi değerlendirmeyin. Bu mesajlar sizi yanıtabilir. Problemi saptamak için bir hata mesajını "ikinci kere tahmin etmeniz" gerekebilir. Ayrıca, hata olduğu rapor edilen satırın hemen öncesinde yer alan son birkaç kod satırına da bakın. Kimi zaman bir hata gerçekten meydana geldiği noktadan birkaç satır sonrasında rapor edilebilir.

## Küçük Bir Değişiklik

Bu kitaptaki programların tümünde kullanılıyor olsa da ilk programın başında yer alan

```
using System;
```

ifadesi teknik olarak gerekli değildir. Ancak, bu ifade aynı zamanda değerli bir kolaylıktır. Bunun gerekli olmamasının nedeni, C#'ta her zaman bir ismi ait olduğu isim uzayı ile *tam olarak* niteleyebiliyor olmanızdır. Örneğin, şu satır

```
Console.WriteLine("Basit bir C# programı.");
```

su şekilde yeniden yazılabılır:

```
System.Console.WriteLine("Bait bir C# programı.");
```

Yani, ilk örnek aşağıda gösterildiği gibi yeniden kodlanabilir:

```
//Bu versiyon, using System ifadesini içermiyor
class Example{
    // Bir C# programı Main()'e çağrıda bulunarak baslar.
    public static void Main(){
        // Burada, Console.WriteLine tam olarak niteleniyor
        System.Console.WriteLine("Basit bir C# programı.");
    }
}
```

**System** isim uzayına ait bir üye kullanıldığında, her seferinde **System** isim uzayını belirtmek oklukça bezdirici olduğu için, C# programclarının birçoğu, bu kitaptaki programların tümünde olduğu gibi programlarının en başına **using System** ifadesini dâhil ederler. Yine de, bir ismi, gerekli olduğunda isim uzayı ile açıkça niteleyebileceğinizi kavramak önemlidir.

## İkinci Basit Program

Bir programlama dilinde belki de hiçbir özellik, bir değişkene değer atama işlemi kadar önemli değildir. *Değişken*, bir değer atanabilen ve ismi olan bellek alanıdır. Üstelik, bir değişkenin değeri programın çalışması esnasında değiştirilebilir. Yani, değişkenin içeriği sabit değildir, değiştirilebilir.

Aşağıdaki program, **x** ve **y** adında iki değişken tanımlıyor.

```
//Bu program degiskenleri tanitiyor.

using System;

class Example2 {
    public static void Main(){
        int x;                  //bu, bir degisken deklare ediyor
        int y;                  //bu, baska bir degisken deklare ediyor

        x = 100;    //bu, x'e 100 degerini atiyor

        Console.WriteLine("x contains " + x) ;

        y = x / 2;

        Console.WriteLine("y contains x / 2: ");
        Console.WriteLine(y);
    }
}
```

Bu programı çalıştırıldığınızda, aşağıdaki çıktıyı ekranda göreceksiniz:

```
x contains 100
y contains x / 2 : 50
```

Bu program birkaç yeni kavramı tanıtıyor. Öncelikle,

```
int x;          //bu, bir degisken deklare ediyor
```

ifadesi tamsayı tipinde **x** adında bir değişkeni deklare ediyor. C#'ta tüm değişkenler kullanılmadan önce deklare edilmelidir. Üstelik, değişkenin tutabileceği değerlerin cinsi de ayrıca belirtilmelidir. Buna değişkenin *tipi* denir. Bu örnekte **x**, tamsayı değerleri tutabilir. C#'ta bir değişkeni tamsayı tipinde deklare etmek için isminin önüne **int** anahtar kelimesini yerleştirin. Böylece, önceki ifadede **int** tipinde **x** adında bir değişken deklare ediliyor.

Bir sonraki satırda, **y** adında bir değişken deklare edilmektedir.

```
int y;          //bu, baska bir degisken deklare ediyor
```

Dikkat ederseniz bu, bir öncekiyle aynı biçimde kullanılıyor; yalnızca, değişken isimleri farklılık gösteriyor.

Genel olarak, bir değişken deklare etmek için şu tür bir ifade kullanacaksınız:

```
tip değişken-ismi;
```

Burada, **tip** deklare edilen değişkenin tipini, **değişken-ismi** ise değişkenin ismini belirtiyor. **int**'e ek olarak C# bazı başka veri tiplerini de destekler, Aşağıdaki kod satırı **x**'e 100 değerini atıyor:

```
x = 100; //bu, x'e 100 değerini atıyor
```

C#'ta değer atama operatörü tek eşittir işaretidir. Atama operatörü, sağ tarafındaki değeri sol tarafındaki değişkene kopyalar.

Bir sonraki kod satırı, önce "**x contains**" karakter katarını, bunun peşinden de **x**'in değerini ekranda gösterir.

```
console.WriteLine("x contains " + x);
```

Bu ifadedeki artı işaretin **x**'in değerinin, **x**'in önünde yer alan karakter katarından sonra ekranda gösterilmesine neden olur. Bu yaklaşım genelleştirilebilir. + operatörünü kullanarak tek **WriteLine()** ifadesi içinde istediğiniz kadar çok sayıda öğeyi ekranda peş peşe gösterebilirsiniz.

Bir sonraki kod satırında, **x**'in 2'ye bölünmüş değeri **y**'ye atanıyor:

```
y = x / 2;
```

Bu satır **x**'deki değeri 2'ye böler, sonucu **y**'de saklar. Böylece, bu satır çalıştırıldıkten sonra **y**, 50 değerini içerecektir. **x**'in değeri ise değişimeyecektir. Diğer bilgisayar dillerinin pek çokunda olduğu gibi C# da aşağıda gösterilenlerin yanı sıra aritmetik operatörlerinin tümünü desteklemektedir:

- + Toplama
- Çıkartma
- \* Çarpma
- / Bölme

Programın diğer iki satırı şu şekildedir:

```
Console.Write("y contains x / 2: ");
Console.WriteLine(y);
```

Burada iki yeni durum söz konusudur. Öncelikle, "**y contains x / 2:** " karakter katarını göstermek için standart **Write()** metodu kullanılıyor. Bu karakter katarının peşinden yeni bir satır gelmiyor. Bunun anlamı şudur: Bir sonraki çıktı üretildiğinde bu çıktı ekranda aynı satırda başlayacaktır. **Write()** metodu tipki **WriteLine()** gibidir; yalnızca, **Write()** metodu, her çağrıdan sonra yeni bir satırda başlamaz. İkincisi, **WriteLine()** çağrısında **y**'nin kendisinin kullanıldığına dikkat edin. C#'ta tanımlı herhangi bir standart tipteki değerlerin çıktısını almak için hem **Write()** hem de **WriteLine()** kullanılabilir.

Bir başka konuya geçmeden önce değişkenleri deklare etmekle ilgili bir hususa daha değinelim: Aynı deklarasyon ifadesini kullanarak bir veya daha fazla değişkeni deklare etmeniz mümkündür. Sadece değişken isimlerini virgül ile ayırmınız yeterlidir. Örneğin, **x** ve **y** şu şekilde de deklare edilebilirdi:

```
int x, y; //tek bir ifade ile ikisi de deklare ediliyor
```

## Başka Bir Veri Tipi

Önceki programda **int** tipinde bir değişken kullanılmıştı. Ancak, **int** tipinde bir değişken yalnızca tamsayıları tutabilir. Yani, ondalık kısmı gerekli olduğunda **int** kullanılamaz. Örneğin, bir **int** değişkeni **18** değerini tutabilir, fakat **18.3** değerini tutamaz. Neyse ki **int**, C#'ta tanımlı olan birkaç veri tipinden yalnızca biridir. Ondalık kısmı olan sayılarla da imkân vermek amacıyla C#'ta iki kayan noktalı tip tanımlanıyor: **float** ve **double**. Bunlar sırasıyla tek ve çift duyarlıklı sayıları simgelerler. Bu ikisinden **double**, genellikle daha yaygın olarak kullanılır,

**double** tipinde bir değişkeni deklare etmek için aşağıdakine benzer bir ifade kullanın:

```
double result;
```

Burada **result**, **double** tipindeki değişkenin ismidir, **result** değişkeni kayan noktalı bir tipte olduğu için **122.23**, **0.034** veya **-19.0** gibi değerleri tutabilir.

**int** ve **double** arasındaki farkı daha iyi anlamak için aşağıdaki programı deneyin:

```
/*
Bu program, int ve double arasindaki
farki ortaya koyar.
*/
using System;

class Example3 {
    public static void Main() {
        int ivar; // bu, bir int degisken deklare eder
        double dvar; // bu, kayan noktalı bir degisken deklare eder

        ivar = 100; // ivar'a 100 degerini atar
        dvar = 100,0; // dvar'a 100.0 degerini atar

        Console.WriteLine("Original value of ivar: " + ivar);
        Console.WriteLine("Original value of dvar: " + dvar);

        Console.WriteLine(); // bos bir satir basar

        // simdi, her ikisini de 3'e bol
        ivar = ivar / 3;
        dvar = dvar / 3.0;
    }
}
```

```

        Console.WriteLine("ivar after division: " + ivar);
        Console.WriteLine("dvar after division: " + dvar);
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Original value of ivar: 100
Original value of dvar: 100

ivar after division: 33
dvar after division: 33.3333333333333

```

Gördüğünüz gibi, **ivar** 3 ile bölündüğünde tamsayı bölme işlemi gerçekleştiriliyor ve sonuç **33** oluyor - ondalık kısmı anlıyor. Ancak, **dvar** 3 ile bölündüğünde ondalık kısmı korunuyor.

Programda gösterildiği gibi, bir programda kayan noktalı bir değer belirtmek istiyorsanız ondalık noktayı dâhil etmelisiniz. Eğer dâhil etmezseniz, söz konusu değer tamsayı olarak yorumlanacaktır. Örneğin, C#'ta **100** değeri bir tamsayıdır; fakat **100.0** ise kayan noktalı bir değerdir.

Programda dikkat edilecek bir başka yeni durum daha var. Boş bir satır yazmak için **WriteLine()**'ı argümansız olarak çağrımanız yeterlidir.

Kayan noktalı veri tipleri genellikle, ondalık bileşenlerin sık sık gereklili olduğu gerçek dünyaya ait niceliklerle çalışırken kullanılır. Örneğin, aşağıdaki program dairenin alanını hesaplıyor. Program **pi** için **3.1416** değerini kullanıyor.

```

// Bir dairenin alanını hesaplar.

using System;

class Circle {
    static void Main() {
        double radius;
        double area;

        radius = 10.0;
        area = radius * radius * 3,1416;

        Console.WriteLine("Area is " + area);
    }
}

```

Bu programın çıktısı şöyledir:

```
Area is 314.16
```

Şüphesiz, dairenin alanı kayan noktalı veriler kullanılmadan tam olarak hesaplanamazdı.

## İki Kontrol İfadesi

Bir metodun içinde kontrol, yukarıdan aşağıya doğru bir ifadeden diğerine geçerek ilerler. Ancak, bu akışı C# tarafından desteklenen çeşitli program kontrol ifadeleri kullanarak değiştirmek de mümkündür. Kontrol ifadelerine daha sonra yakından göz atacak olmamıza rağmen, iki tanesini burada kısaca tanıtacağız, çünkü örnek programları yazmak için bunları kullanacağımız.

### if İfadesi

C#'ın koşul ifadesini kullanarak programın bir parçasını seçerek çalıştırabilirsiniz. Bu koşul ifadesi **if**'dir. C#'taki **if** ifadesi diğer dillerdeki **if** ifadesine çok benzer şekilde çalışır. Örneğin, C#'taki **if** ifadesi C, C++ ve Java'daki **if** ifadeleriyle aynı söz dizimine sahiptir. Bu ifade en basit şekliyle şöyledir:

```
if (koşul) ifade;
```

Burada, **koşul** bir Boolean (yani, **true** (doğru) veya **false** (yanlış)) deyimdir. Eğer **koşul** doğruysa, ifade gerçekleşir, **koşul** yanlış ise ifade atlanır. İşte bir örnek:

```
if (10 < 11) Console.WriteLine("10 is less than 11");
```

Bu örnekte **10**, **11**'den küçük olduğu için koşul deyimi doğru olur ve **WriteLine()** gerçekleşir. Ancak, bir de aşağıdakine bakın:

```
if (10 < 9) Console.WriteLine("this won't be displayed");
```

Bu durumda **10**, **9**'dan küçük değildir. Yani, **WriteLine()** çağrısı gerçekleşmeyecektir. C#, koşul deyimlerinde kullanılan ilişkisel operatörlerin tam kadro tanımını içerir. Bunlar aşağıda gösterilmiştir:

#### Operatör Anlamı

<	Küçüktür
<=	Küçüktür veya eşittir
>	Büyüktür
>=	Büyüktür veya eşittir
==	Eşittir
!=	Eşit değildir

Aşağıda, **if** ifadesini gösteren bir program örneği bulunuyor:

```
// if'i tanitir.

using System;

class IfDemo {
    public static void Main() {
```

```

int a, b, c;

a = 2;
b = 3;

if(a < b) Console.WriteLine("a is less than b");

// bu, ekranda hiçbir şey göstermez
if(a == b) Console.WriteLine("you won't see this");
Console.WriteLine();

c = a - b; // c, -1 değerini icerir

Console.WriteLine("c contains -1");
if(c >= 0) Console.WriteLine("c is non-negative");
if(c < 0)   Console.WriteLine("c is negative");

Console.WriteLine();

c = b - a; // c şimdi 1 değerini icerir
Console.WriteLine("c contains 1");
if(c >= 0) Console.WriteLine("c is non-negative");
if(c < 0)   Console.WriteLine("c is negative");

}
}

```

Bu programın çıktısı aşağıdaki gibidir;

```

a is less than b

c contains -1
c is negative

c contains 1
c is non-negative

```

Bu programdaki bir başka noktaya dikkat edin. Aşağıdaki satır, virgül ile ayrılmış bir liste kullanarak **a**, **b** ve **c** isimli üç değişken deklare eder:

```
int a, b, c;
```

Önceden bahsedildiği gibi, aynı tipte iki veya daha fazla değişkene ihtiyacınız varsa bu değişkenler tek bir ifade ile deklare edilebilir. Tek yapmanız gereken, değişken isimlerini virgül ile ayırmaktır.

## for Döngüsü

Kod parçalarını bir *döngü* oluşturarak tekrar tekrar çalıştırabilirsiniz, C#, çok çeşitli döngü ifadelerinden oluşan güçlü döngü özelliklerine sahiptir. Burada **for** döngüsünü ele atacağız. C, C++ veya Java'ya aşınaysanız, C#'taki **for** döngüsünün bu dillerdekiyle aynı şekilde çalıştığını öğrenmek sizi mutlu edecektir, **for** döngüsünün en basit şekli aşağıda gösterilmiştir:

```
for (başlangıç; koşul; iterasyon) ifade;
```

En yaygın şekilde döngünün **başlangıç** bölümü, döngü kontrol değişkenine ilk değer atar. **koşul**, döngü kontrol değişkenini test eden bir Boolean deyimdir. Bu testin sonucu doğruysa **for** döngüsü iterasyona devam eder. Testin sonucu yanlışsa döngü sona erer. **iterasyon** deyimi, döngünün her iterasyonunda döngü kontrol değişkeninin ne şekilde değişeceğini belirler. İşte, **for** döngüsünü göz önünde canlandıran kısa bir program:

```
// for dongusunu tanitir.

using System;

class ForDemo {
    public static void Main() {
        int count;

        for (count = 0; count < 5; count = count + 1)
            Console.WriteLine("This is count: * + count");

        Console.WriteLine("Done!");
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
This is count: 0
This is count: 1
This is count: 2
This is count: 3
This is count: 4
Done!
```

Bu örnekte **count**, döngü kontrol değişkenidir, **count**, **for** döngüsünün ilk değer ataması bölümünde 0 değerini alır. Her iterasyonun başında (ilk iterasyon da dâhil olmak üzere) **count < 5** koşul testi gerçekleştirilir. Bu testin sonucu doğruysa, **WriteLine()** ifadesi gerçekleşir, sonra da döngünün iterasyon bölümünü çalıştırılır. Bu işlemler koşul testi yanlış olana kadar sürer. Koşul testi yanlış olduğunda kontrol, döngünün sonuna geçer.

İlginc olan şudur: Profesyonel olarak yazılmış C# programlarında, önceki örnekte gösterildiği gibi döngünün iterasyon bölümünü hemen hemen hiç rastlamazsınız. Yani, şu tür ifadelerle nadiren karşılaşırınız:

```
count = count + 1;
```

Bunun nedeni, C#'ta bu işlemi daha verimli biçimde gerçekleştiren özel bir artırma operatörünün mevcut olmasıdır. Artırma operatörü **++**'dır (yani, peş peşe artı işaretleri). Artırma operatörü, operandını bir artırır. Artırma operatörünü kullanarak önceki ifade şu şekilde yazılabilir:

```
count++;
```

Böylece, önceki programdaki **for** döngüsü genellikle şöyle yazılır:

```
for (count = 0; count < 5; count++)
```

Bunu denemek isleyebilirsiniz. Döngünün öncekiyle tamamen aynı şekilde çalıştığını göreceksiniz.

C#'ta ayrıca, - - olarak belirtilen eksiltme operatörü de bulunur. Bu operatör, operandını bir azaltır.

## Kod Bloklarını Kullanmak

C#'ın bir başka temel ögesi *kod blogudur*. Bir kod blogu iki veya daha fazla ifadenin gruplanmasıdır. Bu, ifadeleri açılış ve kapanış küme parantezleri içine alarak gerçekleştirilir. Kod blogunun hazırlanmasıyla, söz konusu kod blogu arlık tek bir ifadenin kullanılabileceği her yerde kullanılabilecek olan mantıksal bir birim halini alır. Örneğin, bir blok **if** ve **for** ifadeleri için hedef olabilir. Şu **if** ifadesine bir göz atın:

```
if (w < h) {
    v = w * h;
    w = 0;
}
```

Burada **w**, **h**'den küçükse blogun içindeki her iki ifade de gerçekleşecektir. Böylece, blogun içindeki iki ifade mantıksal bir birim oluştururlar; bir ifade ancak diğer de gerçekleşendiğinde gerçekleşebilir. Buradaki temel konu, iki veya daha fazla ifadeyi ne zaman birbirine bağlamanız gerekse, bunu bir blok oluşturarak yapabilecek olmanızdır. Kod blokları algoritmaların birçoğunun daha net ve verimli uygulanmasına imkân verir.

Şimdi, sıfıra bölmeyi önleyen bir kod blogu kullanan bir programa göz atalım:

```
// Bir kod blogunu gösterir.

using System;

class BlockDemo {
    public static void Main() {
        int i, j, d;

        i = 5;
        j = 10;

        // bu if'in hedefi bir kod blogudur.
        if (i != 0) {
            Console.WriteLine("i does not equal zero");
            d = j / i;
            Console.WriteLine("j / i is " + d);
        }
    }
}
```

Bu programın çıktısı şu şekildedir:

```
i does not equal to zero  
j / i is 2
```

Bu örnekte **if** ifadesinin hedefi tek bir ifade değil, bir kod bloğudur, **if**'i kontrol eden koşul doğruysa (bu örnekle olduğu gibi), bloğun içindeki üç ifade gerçekleşecektir. **i** değişkenine sıfır değerini vermeye çalışın ve sonucu gözleyin.

İşte bir başka örnek. Bu kez, 1'den 10'a kadar sayıların toplamlarını ve çarpımlarını hesaplamak için kod bloğu kullanılıyor.

```
// 1'den 10'a kadar sayilarin toplam ve carpimlarini hesaplar.  
  
using System;  
  
class ProdSum {  
    static void Main() {  
        int prod;  
        int sum;  
        int i;  
  
        sum = 0;  
        prod = 1;  
  
        for (i = 1; i <= 10; i++) {  
            sum = sum + i;  
            prod = prod * i;  
        }  
        Console.WriteLine("Sum is " + sum);  
        Console.WriteLine("Product is " + prod);  
    }  
}
```

Cıktı aşağıdadır:

```
Sum is 55  
Product is 3628800
```

Burada kod bloğu sayesinde, toplam ve çarpımın her ikisinin tek bir döngü içinde hesaplanması mümkün oluyor. Blok kullanılmamasayı iki ayrı **for** döngüsü gerekecekti.

Son bir husus: Kod blokları programın çalışmasıyla ilgili verimsizlige neden olmazlar. Başka bir deyişle, { ve } programın çalışması sırasında ekstra zaman harcamaz. Aslında, belirli algoritmaları sadeleştirme becerilerinden ötürü, kod bloklarının kullanım genellikle hızı ve verimliliği artırır.

## Noktalı Virgüler ve Konumlandırma

C#'ta noktalı virgül, ifadenin sonunu gösterir. Yani, tek başına ifadelerin her biri, noktalı virgül ile sona ermeliidir.

Bildiğiniz gibi bir blok, açılış ve kapanış küme parantezleri ile çevrelenmiş mantıksal olarak birbirine bağlı bir takım ifadelerden oluşur. Blok noktalı virgül ile sona *ermez*. Blok, her biri noktalı virgül ile sona eren bir grup ifadeden oluştuğu için bloğun noktalı virgül ile bitmemesi mantıklıdır. Noktalı virgül yerine kapanış küme parantezi bloğun sonuna işaret eder.

C# satırın sonunu ifadenin sonu olarak algılamaz - sadece noktalı virgül ifadeyi sonlandırır. Bu nedenle, ifadeyi satır üzerinde nereye yerleştirdiğiniz önemli değildir. Örneğin, C#'ta şu,

```
x = y;
y = y + 1;
Console.WriteLine(x + " " + y) ;
```

aşağıdaki ile aynıdır:

```
x = y;    y = y + 1;  Console.WriteLine(x + " " + y);
```

Üstelik, bir ifadeyi oluşturan elemanları da tek tek ayrı satırlara yerleştirebilirsiniz. Örneğin, aşağıdaki tamamen kabul edilebilir:

```
Console.WriteLine("This is a long line of output" +
                  x + y + z +
                  "more output");
```

Uzun satırları bu şekilde bölmek, genellikle programlan daha okunaklı kılar. Ayrıca aşırı uzun satırların bir sonraki satıra taşmasını da önlemeye yardımcı alabilir.

## Girintileme Alistirmaları

Önceki örneklerde, belirli ifadelerin girintili olduğuna dikkat etmişsinizdir. C# serbest formlu bir dildir. Yani, ifadeleri satır üzerinde birbirine kıyasla nereye yerleştirdiğiniz önemli değildir. Ancak, yıllar geçtikçe, çok okunaklı programlara imkân veren ortak ve onaylanmış bir girintileme stili gelişmiştir. Bu kitaptaki örnekler bu stili izliyorlar; sizin de bu şekilde yapmanız tavsiye edilir. Bu stili kullanarak, her açılış küme parantezinden sonra girintiyi bir seviye artırırsınız ve her kapanış küme parantezinden sonra da bir seviye azaltırsınız. Kimi ifadeler ilave bazı girintileri de teşvik eder; bunlar daha sonra ele alınacaktır.

**TABLO 2.1: C# Anahtar Kelimeleri**

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally

fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	unit	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

## C# Anahtar Kelimeleri

C# dilinde tanımlı halihazırda 77 anahtar kelime mevcuttur (Tablo 2.1 'e bakın). Bu anahtar kelimeler, operatör ve separatör sözdizimleriyle birleştirilerek C# dilinin tanımını oluştururlar. Bu anahtar kelimeler değişken, sınıf ve metot ismi olarak kullanılamazlar.

## Tanımlayıcılar

C#'ta bir tanımlayıcı (identifier), bir metoda, değişkene veya kullanıcı tarafından tanımlanan herhangi başka bir öğeye atanın isimdir. Tanımlayıcılar bir karakterden birkaç karaktere kadar değişen uzunluklarda olabilir. Değişken isimleri alfabetin herhangi bir harfi ile veya alt çizgi (\_) ile başlayabilir. Sonra harf, rakam veya alt çizgi ile devam edebilir. Alt çizgi, değişken isminin okunaklılığını artırmak için kullanılabilir; `line_count` örneğinde olduğu gibi. Büyük ve küçük harfler farklıdır; yani, C#'ta `myvar` ve `MyVar` ayrı isimlerdir. Kabul edilebilir tanımlayıcılar için şunlar verilebilir:

Test	x	y2	MaxLoad
up	_top	my_var	sample23

Bir tanımlayıcının rakam ile başlamayacağını hatırlızdan çıkarmayın. Yani, örneğin `12x` geçerli değildir. İsimlendirilen ögenin anlamını ya da kullanımını yansıtan tanımlayıcı isimleri kullanmak, iyi bir programlama alışkanlığıdır.

C# anahtar kelimelerinden hiç birini tanımlayıcı ismi olarak kullanamıyor olmanızı rağmen C#, bir anahtar kelimenin önünde @ kullanmanızı izin vererek söz konusu anahtar kelimenin geçerli bir tanımlayıcı olmasını mümkün kılar. Örneğin, @`for` geçerli bir tanımlayıcıdır. Bu örnekte, tanımlayıcı aslında `for`'dur; @ dikkate alınmaz. İşte, @ tanımlayıcısının kullanımını göz önünde canlandıran bir program:

```
// @ tanımlayıcısını göstermektedir.
using System;
```

```
class IdTest {
    static void Main()  {
        int @if; // if' i tanımlayıcı olarak kullan

        for (@if = 0; @if < 10; @if++)
            Console.WriteLine("@if is " + @if);
    }
}
```

Burada gösterilen çıktı, `@if`'in gerçekten de bir tanımlayıcı olarak yorumlandığını kanıtlamaktadır:

```
@if is 0
@if is 1
@if is 2
@if is 3
@if is 4
@if is 5
@if is 6
@if is 7
@if is 8
@if is 9
```

Dürlüst olmak gereklirse, özel amaçlar hariç tanımlayıcılar İçin `@` ile nitelenen anahtar kelimeler kullanmak tavsiye edilmez. Ayrıca, `@` herhangi bir tanımlayıcının önünde de kullanılabilir, fakat bu kötü bir programlama alışkanlığı olarak dikkate alınır.

## C# Sınıf Kütüphaneleri

Bu bölümde gösterilen örnek programlar C#'ın standart metodlarının ikisinden yararlanıyor: `WriteLine()` ve `Write()`. Önceden bahsedildiği gibi, bu metodlar `System` isim uzayının bir parçası olan `Console` sınıfının üyeleridir. `System` isim uzayı ise .NET Framework'ün sınıf kütüphaneleri ile tanımlanır. Bu bölümde daha önceden açıklandığı gibi C# ortamı; I/O, karakter katarlarının ele alınması, ağ uygulamaları ve GUI gibi işlemler için destek sağlamak amacıyla, .NET Framework sınıf kütüphanelerine dayanmaktadır. Yani, C# bütün olarak C# dilinin kendisinin ve .NET standart sınıflarının birleşiminden oluşmaktadır. Daha sonra göreceğiniz üzere, herhangi bir C# programının parçası olan işlevselligin büyük bölümünü sınıf kütüphaneleri sağlar. Gerçekten, C# programcısı olmanın bir parçası da bu sınıf kütüphanelerini kullanmayı öğrenmektir. Kısım I boyunca .NET sınıf kütüphanelerinin ve metodlarının çeşitli elemanları ele alınıyor. Kısım II'de .NET kütüphanesi daha ayrıntılı inceleniyor.

---

**VERİ TİPLERİ,  
LİTERALLER VE  
DEĞİŞKENLER**

Bu bölümde C#'ın üç temel ögesi inceleniyor: Veri tipleri, literaller ve değişkenler. Genel olarak, bir dilin sunduğu veri tipleri, bu veri tiplerinin uygulanabileceği problem sınıflarını tanımlar, Tahmin edebileceğiniz gibi, C# zengin bir standart veri tipleri kümesi içerir. Bu, C#'ı çok çeşitli uygulamalar için uygun kılar. Bu tiplerin herhangi birinden değişkenler oluşturabilirsiniz. Her tipten sabit de belirtebilirsiniz. C# dilinde bu sabitler *literal* olarak adlandırılır.

## Veri Tipleri Neden Önemlidir?

Veri tipleri C#'ta özellikle önemlidir, çünkü C# sıkı sıkıya tipe dayalı bir dildir. Bu, tip uyumu için tüm işlemlerin derleyici tarafından tip kontrolünden geçirileceği anlamına gelir. Kural dışı İşlemler derlenmeyecektir. Böylece, sıkı tip kontrolü, hataları önlemeye yardımcı olur ve güvenilirliği artırır. Sıkı tip kontrolünü mümkün kılmak için tüm değişkenler, deyimler ve değerler bir tipe sahiptir. “Tipsız” bir değişken şeklinde bir kavram söz konusu değildir örneğin. Üstelik bir değerin tipi, bu değer üzerinde hangi işlemlere izin verildiğini belirler. Bir tip üzerinde izin verilen işlemlere bir başka tip üzerinde izin verilmeyebilir.

## C#'ın Değer Tipleri

C#'ın içeriği yerleşik veri tipleri iki genel kategoride toplanır: *değer tipleri* ve *referans tipleri*. C#'ın referans tipleri sınıflarla tanımlanır; sınıflarla ilgili tartışma ise daha sonra ele alınacaktır, Ancak, C#'ın özünde 13 değer tipi yer alır. Bunlar Tablo 3.1'de gösterilmiştir. Bu değer tipleri, C# dilinin anahtar kelimeleri tarafından tanımlanan yerleşik tiplerdir ve herhangi bir C# programı tarafından kullanılmak üzere hazırlıdır.

*Değer tipi* terimi, bu tipteki değişkenlerin, değerlerini doğrudan içerdiklerini belirtir. (Bu; değişkenin, gerçek değere bir referans içeriği referans tiplerinden farklıdır.) Böylece, değer tipleri diğer programlama dillerinde, mesela C++'ta, bulunan veri tiplerine çok benzer şekilde davranışırlar. Değer tipleri ayrıca basit tipler olarak da bilinir.

C#, her değer tipi için değişmez bir değer aralığı ve davranış biçimini belirler. Taşınabilirlik gereksinimlerinden dolayı C# bu konuda taviz vermez. Örneğin, `int` tüm çalışma ortamlarında aynıdır. Kodu, belirli bir platforma uydurmak için yeniden yazmaya gerek yoktur. Değer tiplerinin büyüğünü değişimde belirlemek, küçük bir performans kaybına neden olabilirken, taşınabilirliği mümkün kılmak için gereklidir.

**TABLO 3.1: C# Değer Tipleri**

Tip	Anlamı
<code>bool</code>	true/false değerlerini simgeler
<code>byte</code>	8 bit işaretsiz tamsayı
<code>char</code>	Karakter
<code>decimal</code>	Mali hesaplamalar için nümerik tip

<b>double</b>	Çift duyarlıklı kayan noktalı sayı
<b>float</b>	Tek duyarlıklı kayan noktalı sayı
<b>int</b>	Tamsayı
<b>long</b>	Uzun tamsayı
<b>sbyte</b>	8 bit işaretli tamsayı
<b>short</b>	Kısa tamsayı
<b>uint</b>	İşaretsiz tamsayı
<b>ulong</b>	İşaretsiz uzun tamsayı
<b>ushort</b>	İşaretsiz kısa tamsayı

## Tamsayılar

C#'ta dokuz tamsayı tipi tanımlıdır: **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long** ve **ulong**. Ancak, **char** tipi özellikle karakterleri simgelemek için kullanılır ve bu bölümde daha sonra ele alınacaktır. Geriye kalan sekiz tamsayı tipi nümerik hesaplamalar için kullanılır. Bunların bit genişliği ve değer aralıkları aşağıda gösterilmiştir:

Tip Genişlik	Bit Olarak Değer Aralığı
byte 8	0'dan 255'e
sbyte 8	-128'den 127'ye
short 16	-32,768'den 32,767'ye
ushort 16	0'dan 65,535'e
int 32	-2,147,483,648'den 2,147,483,647'ye
uint 32	0'dan 4,294,967,295'e
long 64	-9,223,372,036,854,775,808'den 9,223,372,036,854,775,807'ye
ulong 64	0'dan 18,446,744,073,709,551,615'e

Tablodan görüldüğü gibi, C#'ta çeşitli tamsayı tiplerinin hem işaretli hem de işaretsiz versiyonları tanımlıdır. İşaretli ve işaretsiz tamsayılar arasındaki fark, tamsayının üst bitinin (high-order) yorumlanma şékline bağlıdır. Bir işaretli tamsayı belirtildiğinde C# derleyicisi, tamsayının üst bitinin *işaret bayrağı* (*sign flag*) olarak kullanılacağını varsayan bir kod üretir. Eğer işaret bayrağı **0** ise sayı pozitiftir; eğer **1** ise, sayı negatiftir. Negatif sayılar hemen hemen her zaman *ikinin tümleyeni* (*two's complement*) yöntemi kullanılarak simgelenirler. Bu yöntemde, sayının tüm bitleri (işaret bayrağı hariç) ters çevrilir ve sayıya **1** eklenir. Son olarak, işaret bayrağı **1** olarak ayarlanır.

İşaretli tamsayılar pek çok sayıda algoritma için önemlidir ama, işaretsiz akrabalarının mutlak büyüklüğünün sadece yansına sahiptirler. Örneğin, **short** olarak **32.767** şu şekildedir:

01111111 11111111

İşaretli bir değer için eğer üst bit **1** olarak ayarlanırsa, sayı **-1** olarak yorumlanacaktır (“ikinin tümleyeni” yönteminin kullanıldığını varsayıyoruz). Ancak, eğer bu sayı **ushort** olarak deklare edilmemişse, üst bit’i **1** olarak ayarlandığında sayı **65.535** olur.

Muhtemelen en yaygın olarak kullanılan tamsayı tipi **int**’tir. **int** tipinde değişkenler genellikle döngülerini kontrol etmek için, dizi indekslerinde ve genel amaçlı tamsayı matematiğinde kullanılır. **int**’ten daha geniş değer aralığına sahip bir tamsayıya ihtiyacınız olduğunda, birçok seçenekiniz vardır. Eğer saklamak istediğiniz değer işaretetsiz ise, **uint**’i kullanabilirsiniz. Büyük, işaretli sayılar için **long**’u kullanın. Büyük, işaretetsiz sayılar için **ulong**’u kullanın. Örneğin işte size dünyadan güneşe olan uzaklıği inç cinsinden hesaplayan bir program. Bu değer çok büyük olduğu için program, bu değeri tutmak amacıyla **long** tipinde bir değişken kullanıyor.

```
// Dunyadan gunese inc cinsinden uzakligi hesaplar.

using System;

class Inches {
    public static void Main(){
        long inches;
        long miles;

        miles = 93000000; // Gunes 93,000,000 mil uzakliktadir

        // Bir mil 5,280 ayaktir, bir ayak 12 inctir,
        inches = miles * 5280 * 12;

        Console.WriteLine("Distance to the sun: " + inches +
                           " inches.");
    }
}
```

Programın çıktısı işte şöyledir;

```
Distance to the sun: 5892480000000 inches.
```

Açıkça görülmüyor ki, sonuç bir **int** veya **uint** değişkeninde saklanamazdı.

En küçük tamsayı tipleri **byte** ve **sbyte**’tir. **byte** tipi, 0 ile 255 arasında işaretetsiz bir değerdir. **byte** tipinde değişkenler özellikle ham ikili verilerle (raw binary data) çalışırken, örneğin bir aygit tarafından üretilen bytestream halindeki veri kullanılırken, yararlı olur. Küçük, işaretli tamsayılar için **sbyte** kulanın. Aşağıda örnek bir program yer alıyor. Bu örnek programda 100’e kadar olan tamsayıların toplamını hesaplayan bir **for** döngüsünü kontrol etmek için **byte** tipinde bir değişken kullanılıyor.

```
// byte kullan.

using System;

class Use_byte {
    public static void Main() {
```

```
byte x;
int sum;

sum = 0;
for (x = 1; x <= 100; x++)
    sum = sum + x;

Console.WriteLine("Summation of 100 is " + sum);
}
```

Programın çıktısı aşağıdaki gibidir:

```
// Math.Sin(), Math.Cos() ve Math.Tan()'i tanitir.

using System;

class Trigonometry {
    public static void Main() {
        Double theta; // aci radyan cinsinden.

        for (theta = 0.1; theta <= 1.0; theta = theta + 0.1) {
            Console.WriteLine("Sine of " + theta + " is " +
                Math.Sin(theta));
            Console.WriteLine("Cosine of " + theta + " is " +
                Math.Cos(theta));
            Console.WriteLine("Tangent of " + theta + " is " +
                Math.Tan (theta));
            Console.WriteLine ();
        }
    }
}
```

Program çıkışının bir kısmı şu şekildedir:

```
Sine of 0.1 is 0.0998334166468282
Cosine of 0.1 is 0.995004165278026
Tangent of 0.1 is 0.100334672085451
```

```
Sine of 0.2 is 0.198669330795061
Cosine of 0.2 is 0.980066577841242
Tangent of 0.2 is 0.202710035508673
```

```
Sine of 0.3 is 0.29552020666134
Cosine of 0.3 is 0.955336489125606
Tangent of 0.3 is 0.309336249609623
```

Sinüs, kosinüs ve tanjantı hesaplamak için standart kütüphane metotları olan **Math.Sin()**, **Math.Cos()** ve **Math.Tan()** kullanılmaktadır. Tıpkı **Math.Sqrt()** gibi, trigonometrik metotlar da bir **double** argüman ile çağrılır ve **double** bir sonuç döndürürler. Açılar radyan olarak belirtilmelidir.

## decimal Tipi

C# nümerik tiplerinin belki de en ilginci, parasal hesaplamalarda kullanılması planlanan **decimal** tipidir, **decimal** tipi, **1E-28** ile **7.9E+28** arasındaki değerleri simgelemek için **128** bit kullanır. Bildiğiniz gibi, kayan noktalı normal aritmetik, ondalık sayılar uygulandığında çeşitli yuvarlama hatalarına maruz kalır. **decimal** tipi bu hataları ortadan kaldırır ve sayıları **28** ondalık basamağa kadar (bazı durumlarda **29** basamağa kadar) hatasız simgeleyebilir. Ondalık sayıları yuvarlama hataları olmadan temsil etme becerisi, bu tipi, özellikle bellek gerektiren hesaplamalarda kullanışlı kılar.

İşte size, **decimal** tipini mali hesaplamada kullanan bir program. Program, orijinal fiyat ve indirim yüzdesi verildiğinde indirimli fiyatı hesaplıyor.

```
//Bir fiyat indirimini hesaplamak için decimal tipini kullanmak.

using System;

class UseDecimal {
    public static void Main() {
        decimal price;
        decimal discount;
        decimal discounted_price;

        // indirilmis fiyat hesapla
        price = 19.95m;
        discount = 0.15m;    // indirim orani: %15

        discounted_price = price - (price * discount);

        Console.WriteLine("Discounted price: $" + discounted_price);
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Discounted price: $16.9575
```

Programda, ondalık sabitlerin **m** soneki ile bitliğine dikkat edin. Bu gereklidir, çünkü **m** soneki olmadan bu değerler, **decimal** veri tipi ile uyumlu olmayan standart kayan noktalı sabitler olarak yorumlanır. Her şeye rağmen, bir **decimal** değişkene **m** sonekini kullanmadan, mesela **10** gibi bir tamsayı değer atayabilirsiniz. (Nümerik sabitlerin ne şekilde belirtildiğine bu bölüm içinde daha yakından göz atacağız.)

Aşağıda, **decimal** tipini kullanan bir başka örnek görülmektedir. Bu örnekle, bir yatırımın sabit faiz oranı üzerinden birkaç yıl sonundaki değeri hesaplanıyor.

```
/*
    Bir yatırının gelecekteki değerini hesaplamak için decimal
    tipini kullanmak.
*/
```

```

using System;

class FutVal {
    public static void Main() {
        decimal amount;
        decimal rate_of_return;
        int years, i;

        amount = 1000.0m;
        rate_of_return = 0.07m;
        years = 10;

        Console.WriteLine("Original investment: $" + amount);
        Console.WriteLine("Rate of return: " + rate_of_return);
        Console.WriteLine("Over " + years + " years");

        for (i = 0; i < years; i++)
            amount = amount + (amount * rate_of_return);
        Console.WriteLine("Future value is $" + amount);
    }
}

```

İşte programın çıktısı:

```

Original investment: $1000
Rate of return: 0.07
Over 10 years
Future value is $1967.15135728956532249

```

Sonucun birkaç ondalık basamağa - bu, muhtemelen arzu edeceğinizden daha da fazladır! - kadar hatasız olduğuna dikkat edin. Bu tür bir çıktıının daha hoş görünecek şekilde nasıl biçimlendirileceğini bu bölümde öğreneceksiniz.

## Karakterler

C#'ta karakterler, diğer bilgisayar dillerinde, örneğin C++'ta, olduğu gibi **8** bitlik nicelikler değildir. Bunun yerine C#'ta, *Unicode* denilen **16** bit karakter tipi kullanılır. Unicode, bütün insanı dillerde mevcut olan karakterlerin tümünü simgelemeye yetecek kadar büyük bir karakter seti tanımlar. İngilizce, Fransızca veya Almanca gibi birçok dil nispeten küçük alfabeler kullanıyor olsa da, bazı diller, örneğin Çince, sadece **8** bit kullanılarak simgelenemeyen çok büyük karakter setleri kullanır. Tüm dillerin karakter setlerine yer vermek için **16** bit değerler gereklidir. Böylece, C#'ta **char**, **0**'dan **65.535**'e uzanan bir menzile (değer aralığına) sahip, işaretsiz bir **16** bit tiptir. Standart **8** bit ASCII karakter seti, Unicode'un bir alt kümeleridir ve **0**'dan **127**'ye kadar uzanır. Yani, ASCII karakterleri halen geçerli C# karakterleridir.

Bir karakter değişkenine tek tırnak içine alınan bir karakter ile değer atanabilir. Örneğin, aşağıda **ch** değişkenine X değeri atanıyor:

```

char ch;
ch = 'X';

```

Bir **char** değerinin çıktısını bir **WriteLine()** ifadesini kullanarak elde edebilirsiniz. Örneğin aşağıdaki satırda, **ch**'in değeri çıktı olarak elde ediliyor:

```
Console.WriteLine("This is ch: " + ch);
```

**char**, C#'ta tamsayı tipi olarak tanımlanmış olmasına rağmen her hal ve şartta tamsayılarla serbestçe karıştırılarak kullanılamaz. Bu, tamsayılardan **char**'a otomatik tip dönüşümü olmamasından kaynaklanmaktadır. Örneğin, aşağıdaki program parçası geçerli değildir:

```
char ch;
ch = 10; //hata; bu, calismaz
```

Yukarıdaki kodun çalışmayacak olmasının nedeni, **10**'un bir tamsayı değer olmasıdır. **10**, **char**'a otomatik olarak dönüştürilmeyecektir. Bu kodu derleme girişiminde bulunursanız, bir hata mesajı ile karşılaşırısz. Bu bölüm içinde ileriki sayfalarda bu kısıtlamadan kurtulmanın yöntemini öğreneceksiniz.

## bool Tipi

**bool** tipi doğru/yanlış değerlerini simgeler. C#'ta doğru ve yanlış değerleri **true** ve **false** özel amaçlı kelimeleriyle tanımlanır. Yani, **bool** tipinde bir değişken veya deyim bu iki değerden birine eşit olabilir. Ayrıca, **bool** ve tamsayı değerler arasında tanımlı bir tip dönüşümü söz konusu değildir. Örneğin, **1 true'ya** dönüştürülmez; **0** da **false'a** dönüştürülmez. İşle, **bool** tipini gösteren bir program:

```
// bool degerleri gosterir.

using System;

class BoolDemo {
    public static void Main() {
        bool b;

        b = false;
        Console.WriteLine("b is " + b);
        b = true;
        Console.WriteLine("b is " + b);

        // bir bool degeri, if ifadesini kontrol edebilir
        if(b) Console.WriteLine("This is executed,");

        b = false;
        if(b) Console.WriteLine("This is not executed.");

        // ilişkisel operatorun sonucu bir bool degerdir
        Console.WriteLine("10 > 9 is " + (10 > 9));
    }
}
```

Bu programın ürettiği çıktı aşağıda gösterilmiştir:

```
b is False  
b is True  
This is executed.  
10 > 9 is True
```

Bu programda dikkat çeken üç ilginç nokta var. Öncelikle, gördüğünüz gibi, bir **bool** değerinin **WriteLine()** ile çıktısı alındığında ekranda "**True**" veya "**False**" görünüyor. İkincisi, **if** ifadesini kontrol etmek için **bool** değişkenin değeri kendi başına yeterlidir. Şu şekilde bir **if** ifadesi yazmaya gereklidir:

```
if (b == true) ...
```

Üçüncüsü, bir ilişkisel operatörün, örneğin **<** operatörünün çıktısı bir **bool** değeridir. **10 > 9** deyiminin ekranda "**True**" değerini göstermesi bu sebeptendir. Ayrıca, **10 > 9** deyiminin etrafındaki ekstra parantez çifti gereklidir, çünkü **+** operatörü, **>** operatöründen daha yüksek önceliğe sahiptir.

## Bazı Çıktı Seçenekleri

Şu ana kadar, **WriteLine()** ifadesi kullanılarak verilerin çıktısı alındığında çıktı, C# tarafından sağlanan varsayılan biçim kullanılarak görüntüleniyordu. Ancak, C# verinin nasıl gösterileceği konusunda size ayrıntılı kontrol olanakları sunan gelişkin bir biçimlendirme mekanizmasına sahiptir. Biçimlendirilmiş I/O (formatted I/O) bu kitapta daha sonra ele alınacak olmasına rağmen, şu aşamada bazı biçimlendirme seçeneklerini tanıtmak yararlı olacaktır. Bu seçenekleri kullanarak değerlerin **WriteLine()** ifadesi üzerinden çıktıları alındığında ne şekilde görüneceklerini belirleyebilirsiniz. Bu, çok daha hoş çıktı üretmenizi mümkün kılar. C# biçimlendirme mekanizmasının burada anlatılan özelliklerden çok daha fazlasını desteklediğini hatırlınızdan çıkarmayın.

Verinin listeler halinde çıktısını alırken, listenin her parçasını artı işaretti ile birbirinden ayırabilirsiniz:

```
Console.WriteLine("You ordered " + 2 + " items at $" + 3 + " each.");
```

Çok uygun olmakla birlikte, nümerik bilgilerin bu şekilde çıktısını almak bu bilgilerin görüntümleri üzerinde size hiçbir kontrol sağlamaz. Örneğin, kayan noktalı bir değer için görüntülenecek ondalık basamak sayısını kontrol edemezsiniz. Aşağıdaki ifadeyi ele alın:

```
Console.WriteLine("Here is 10/3: " + 10.0/3.0);
```

Bu, şu çıktıyı üretir:

```
Here is 10/3: 3.33333333333333
```

Bazı amaçlar için bu makul olsa da, bu kadar çok ondalık basamak göstermek diğer amaçlar için uygun olmayabilir. Örneğin, mali hesaplamalarda genellikle iki ondalık basamak göstermeyi tercih edeceksiniz.

Nümerik verilerin ne şekilde biçimlendirildiğini kontrol etmek için, aşağıda gösterildiği gibi, biçimlendirme bilgisini ifadenin içine gömmenize imkân veren ikinci bir **WriteLine()** stili kullanmanız gerekecektir:

```
WriteLine("bicimlendirme karakter katari", arg0, arg1, ..., argN);
```

Bu versiyonda **WriteLine()**'a aktarılan argümanlar + işaretti ile değil, virgül ile ayrılır. **bicimlendirme karakter katari**, olduğu gibi gösterilen sıradan baskı karakterleri ve biçimlendirme belirleyicileri olmak üzere iki öğe içerir. Biçimlendirme belirleyicileri aşağıdaki gibi bir genel görünüşe sahiptir:

```
{argno, genişlik:fmt}
```

Burada **argno**, gösterilecek argüman sayısını (0'dan başlayarak) belirtir. Minimum alan genişliği **genişlik** ile; biçim ise **fmt** ile belirtilir.

Programın çalışması sırasında, biçimlendirme karakter katarı içinde bir biçimlendirme belirleyicisine rastlanınca, **argno**'ya karşılık gelen argüman **argno**'nun yerine yerleştirilir ve ekranda gösterilir. Yani **argno**, eşlenecek verinin nerede gösterileceğini belirleyen, biçimlendirme karakter katarıının içindeki biçimlendirme spesifikasyonunun konumudur. **genişlik** ve **fmt**'nin her ikisi de isteğe bağlıdır. Bu nedenle, bir biçimlendirme belirleyicisi, en basit şekilde, sadece hangi argümanın gösterileceğini belirtir. Örneğin **{0}**, **arg0**'ı gösterir; **{1}**, **arg1**'i gösterir vs.

Gelin, basit bir örnek ile başlayalım. Şu ifade

```
Console.WriteLine("February has {0} or {1} days." 28, 29);
```

aşağıdaki çıktıyı üretir:

```
February has 28 or 29 days.
```

Gördüğünüz gibi, **{0}** için **28** ve **{1}** için de **29** değeri yerine yerleştiriliyor. Yani, biçimlendirme belirleyicileri, sonradan gelen argümanların, bu örnekte **28** ve **29**'un, karakter katarı içinde gösterileceği konumları belirtiyorlar. Ayrıca, ekstra değerler + işaretti ile değil, virgül ile ayrılıyor.

Şimdi de önceki ifadenin minimum alan genişliğini de belirten bir versiyonuna bakalım:

```
Console.WriteLine("February has {0, 10} or (1,5} days", 28, 29);
```

Bu ifade aşağıdaki çıktıyı üretir:

```
February has 28 or 29 days
```

Gördüğünüz gibi alanların kullanılmayan kısımlarını doldurmak için boşluk karakterleri ilave ediliyor. Minimum alan genişliğinin aslında, sadece *minimum* genişlikten ibaret olduğunu hatırlınızdan çıkarmayın. Gerektiğinde çıktı, bu genişliği aşabilir.

Bir biçimlendirme komutuyla ilgili argümanların elbette sabit olması gereklidir. Örneğin aşağıdaki program, kareler ve küplerden oluşan bir tabloyu ekranda gösteriyor. Program, değerlerin çıktısını alırken biçimlendirme komutlarını kullanıyor.

```
// Bicimlendirme komutlarini kullan.

using System;

class DisplayOptions {
    public static void Main() {
        int i;

        Console.WriteLine("Value\tSquared\tCubed");

        for (i = 1; i < 10; i++)
            Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
    }
}
```

Çıktı aşağıda gösterilmiştir:

Value	Squared	Cubed
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729

Önceki örneklerde değerlerin kendilerine herhangi bir biçimlendirme uygulanmamıştı. Biçimlendirme belirleyicileri kullanılmasının nedeni elbette, verilerin görünüşlerini kontrol etmektedir. En çok biçimlendirilen veri tipleri kayan noktalı ve ondalık değerlerdir. Bir biçim belirtmenin en kolay yollarından biri, **WriteLine()**'ın kullanacağı bir şablon tarif etmektir. Bunun için, rakam konumlarını işaret etmek amacıyla # işaretleri kullanarak istediğiniz biçimin bir örneğini gösterin. Ayrıca, ondalık noktayı ve virgülleri de belirtebilirsiniz. Örneğin, 10/3 değerini göstermenin daha iyi bir yolu şu şekildedir:

```
Console.WriteLine("Here is 10/3: {0:#.##}", 10.0/3.0);
```

Bu ifadenin çıktısı aşağıdaki gibidir:

```
Here is 10/3: 3.33
```

Bu örnkle şablon, **WriteLine()**'a iki ondalık basamak ayırmamasını söyleyen **#.##** karakterleridir. Ancak, **WriteLine()**'ın gerektiğiinde, değeri hatalı göstermemek için ondalık noktanın solunda birden fazla basamaklık yer bırakacağını kavramak önemlidir.

İşte bir başka örnek daha.

```
Console.WriteLine("{0:###,###.###}", 123456.56);
```

Bu ifade aşağıdaki çıktıyı üretir;

```
123,456.56
```

Şayet değerleri dolar ve sent biçimini kullanarak göstermek istiyorsanız, **C** biçimlendirme belirleyicisini kullanın. Örneğin,

```
decimal balance;
balance = 12323.09m;
Console.WriteLine("Current balance is {0:C}", balance);
```

Bu kod parçasının çıktısı işte şöyledir:

```
Current balance is $12,323.09
```

Daha önce verilen fiyat indirimi programında çıktıyı daha sık göstermek amacıyla **C** biçimini kullanabilir:

```
/*
    Dolar ve cent'lerin ciktisini almak için C bicimlendirme
    Belirleyicisini kullanmak.
*/
using System;

class UseDecimal {
    public static void Main() {
        decimal price;
        decimal discount;
        decimal discounted_price;

        // indirimli fiyat hesapla
        price = 19.95m;
        discount = 0.15m; // indirim orani: %15

        discounted_price = price - (price * discount);

        Console.WriteLine("Discounted price: {0:C}", discounted_price);
    }
}
```

Çıktı artık, işe şöyle görünüyor:

```
Discounted price: $16.96
```

## Literaller

C#'ta *literaller*, insanların okuyabileceği biçimde temsil edilen sabit değerlere karşılık gelir. Örneğin, **100** sayısı bir literaldir. Literaller ayrıca yaygın olarak *sabit* (constant) olarak da

adlandırılır. Coğunlukla literallerin kendileri ve kullanımları o kadar bellidir ki önceki örnek programların tümünde, şu veya bu şekilde literal kullanılmıştır. Artık bunları kurallı olarak açıklamanın zamanı geldi.

C# literalleri herhangi bir değer tipinde olabilir, Her literalin gösteriliş şekli söz konusu literalin tipine bağlıdır. Önceden açıklandığı gibi, karakter sabitleri tek tırnak içine alınır. Örneğin, '**a**' ve '%' karakter sabitleridir.

Tamsayı literaller ondalık kısmı olmayan sayılar şeklinde belirtilir. Örneğin **10** ve **-100** tamsayı sabitlerdir. Kayan noktalı sabitler, sayının ondalık kısmının peşinden geldiği ondalık nokta kullanımını gerektirir. Örneğin, **11.123** bir kayan noktalı sabittir. C# ayrıca, kayan noktalı sayılar için bilimsel (scientific) notasyon kullanımına da imkan verir.

C# tipe sıkı sıkıya bağlı bir dil olduğu için literallerin de tipi vardır. Bu, doğal olarak akla şu soruyu getirir: Nümerik bir literalin tipi nedir? Örneğin, **12,123987** veya **0.23**'ün tipi nedir? Neyse ki C#, bu sorulara yanıt niteliğinde bazı takibi kolay kuralları da belirtir.

Oncellikle, tamsayı literaller için literalin tipi, söz konusu literalin tutabileceği **int**'ten başlayarak en küçük tamsayı tipidir. Böylece, bir tamsayı literal, değerine bağlı olarak ya **int**, **uint**, **long** ya da **ulong** tipinde olabilir. İkincisi, kayan noktalı literaller **double** tipindedir.

Eğer C#'ın varsayılan tipi, sizin literal için düşündüğünüz tipten değilse, literalin tipini bir sonek ilave ederek açıkça belirtebilirsiniz. Bir **long** literali belirtmek için **1** veya **L** ekleyin. Örneğin **12**, **int** tipindedir, fakat **12L**, **long** tipindedir. İşaretsiz bir tamsayı değeri belirtmek için **u** veya **U** ekleyin. Yani **100**, **int** tipindedir, fakat **100U**, **uint** tipindedir. İşaretsiz uzun bir tamsayıyı belirtmek için **ul** veya **UL** kullanın. Örneğin, **984375UL** **ulong** tipindedir.

Bir **float** literalı belirtmek için sabite **F** veya **f** ekleyin. Örneğin, **10.19F** **float** tipindedir.

Bir **decimal** literalı belirtmek için değerinin peşine **m** veya **M** yerleştirin. Örneğin, **9.95M** bir **decimal** literalıdır.

Tamsayı literaller varsayılan durumda **int**, **uint**, **long** veya **ulong** tipinde bir değer oluşturuyor olsa da, atanın değer hedef tip tarafından simgelenebildiği müddetçe tamsayı literaller, her şeye rağmen **byte**, **sbyte**, **short** veya **ushort** tipindeki değişkenlere atanabilir. Bir tamsayı literal bir **long** değişkene her zaman atanabilir.

## Onaltılık Literaller

Muhtemelen biliyorsunuzdur, programlamada onluk taban yerine kimi zaman onaltı tabanına dayanan sayı sistemini kullanmak daha kolaydır. Onaltı tabanına dayanan sayı sistemine *onaltılık (hexadecimal)* denir ve bu sistem, **0** ile **9** arasındaki rakamlarla, **A** ile **F** arasındaki harfleri kullanır. **A**'dan **F**'ye kadar olan harfler **10**, **11**, **12**, **13**, **14** ve **15**'e karşılık gelir. Örneğin, onaltılık sistemdeki **10** sayısı, onluk sistemde **16**'dır. Onaltılık sayıların

kullanım sikliğinden dolayı C#, tamsayı sabitleri onaltılık biçimde belirtmenize imkân verir. Bir onaltılık literal **0x** (sıfır ve x) ile başlamalıdır. İşte birkaç örnek:

```
count = 0xFF;      //ondalık tabanda 255
incr = 0x1a;       //ondalık tabanda 26
```

## Karakter Kaçış Sekansları

Karakter sabitlerini tek tırnak içine almak, basılan karakterlerin pek çoğu için işe yarar; fakat, metin editörü kullanıldığında birkaç karakter, mesela carriage return (paragraf sonu) karakteri, özel bir problemin ortaya çıkmasına neden olur. Üstelik, bazı başka karakterlerin, mesela tek ve çift tırnağın C#'ta özel bir anlamı da vardır. Bu yüzden, bunları doğrudan kullanamazsınız. Bu nedenlerden ötürü C# birkaç tane kaçış sekansi sunmaktadır. Tablo 3.2'de gösterilen bu karakterlerden kimi zaman *ters bölü işaretli karakter sabitleri* olarak da söz edilir. Bu kaçış sekansları simgeledikleri karakterlerin yerine kullanılırlar.

Örneğin, aşağıdaki ifade **ch** değişkenine sekme karakterini atıyor:

```
ch = '\t';
```

Sıradaki örnek ise **ch** değişkenine tek tırnak karakterini atar:

```
ch = '\';
```

**TABLO 3.2: Kaçış Sekansları**

Kaçış Sekansi	Anlamı
\a	Uyarı (Zil)
\b	Backspace (Geri al)
\f	Form feed (Form besleme)
\n	Newline (Yeni satır)
\r	Carriage return (Paragraf sonu)
\t	Yatay sekme
\v	Düşey sekme
\0	Null
\'	Tek tırnak
\"	Çift tırnak
\\\	Ters bölü

## Karakter Katarı Literalleri

C# bir başka tipte literali daha destekler: Karakter katarı literalı. *Karakter katarı*, çift tırnak içine alınan bir karakter kümesidir, Örneğin, aşağıda bir karakter katarı görüyorsunuz.

```
"bu bir denemedir"
```

Önceki örnek programlardaki `WriteLine()` ifadelerinin birçoğunda karakter katarı örneklerini görmüştünüz.

Normal karakterlere ek olarak bir karakter katan literalı ayrıca, az önce bahsedilen kaçış sekanslarından bir veya daha fazla içerebilir. Örneğin, aşağıdaki programı ele alın. Bu program `\n`, `\t` ve `\\"` kaçış sekanslarını kullanıyor.

```
// Karakter katarları icindeki kacis sekanslarini gosterir.

using System;

class StrDemo {
    public static void Main() {
        Console.WriteLine("Line One\nLine Two\nLine Three");
        Console.WriteLine("One\tTwo\tThree");
        Console.WriteLine("Four\tFive\tSix");

        // tırnak işaretlerini gom
        Console.WriteLine("Why?\\"", he asked.");
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Line One
Line Two
Line Three
One    Two    Three
Four   Five   Six
"Why?", he asked.
```

`\n` kaçış sekansının yeni bir satır oluşturmak için nasıl kullanıldığına dikkat edin. Çok satırlı çıktı almak için birkaç tane `WriteLine()` ifadesi kullanmanız gereklidir. Uzunca bir karakter katarı içinde hangi noktalarda yeni bir satır oluşturulmasını istiyorsanız bu noktalara `\n` yerleştirmeniz yeterlidir. Ayrıca, bir karakter katarı içinde soru işaretinin nasıl oluşturulduğuna da dikkat edin.

Şimdi bahsedilen karakter katarı literal stiline ek olarak, ayrıca *harfi harfine karakter katarı literalı (verbatim string literal)* de belirtebilirsiniz. Harfi harfine karakter katarı literalı @ ile başlar ve bunu tırnak içine alınmış karakter katarı izler. Tırnak içine alınmış karakter katarının içeriği değişiklik yapılmaksızın kabul edilir ve iki veya daha fazla satır uzunluğunda olabilir. Yani, newline, sekme gibi karakterleri dâhil edebilirsiniz, fakat kaçış sekanslarını kullanmanız gereklidir. Tek bir istisna, çift tırnak ("") elde etmek için bir satırda iki adet

çift tırnak ("") kullanmanız gerektidir. İşte size, harfi harfine karakter katarı literallerini gösteren bir program:

```
// harfi harfine karakter katarı literallerini gösterir.

using System;

class Verbatim {
    public static void Main() {
        Console.WriteLine(@"This is a verbatim
string literal that
spans several lines.
");
        Console.WriteLine(@"Here is some tabbed output:
1      2      3      4
5      6      7      8
");
        Console.WriteLine(@"Programmers say, " "I like C#." " ");
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
This is a verbatim
string literal
that spans several lines.
Here is some tabbed output:
1      2      3      4
5      6      7      8
Programmers say, "I like C#."
```

Önceki programla ilgili dikkat çeken önemli bir konu, harfi harfine karakter katarı literallerinin tam olarak programa girildikleri şekliyle gösterilmeleridir.

Harfi harfine karakter katarı literallerinin avantajı, çıktıyı aynen ekranda görüleceği şekilde programda belirtebiliyor olmanızdır. Ancak, çok satırlı karakter katarları söz konusu olduğunda, karakter katarının satıldan taşması programınızın hızmasını (girintilememesini) anlaşılır hale getirebilir. Bu nedenle, bu kitaptaki programlarda harfi harfine karakter katarı literallerinden sadece kısıtlı oranda yararlanılacaktır. Buna rağmen, bir çok biçimlendirme durumunda harfi harfine karakter katarı literalleri halen müthiş bir avantaj sağlamaktadır.

Son bir husus: Karakter katarlarını karakterlerle karıştırmayın. Bir karakter literalı, mesela 'X', **char** tipinde tek bir harfi simgeler. Sadece tek bir harf içeren bir karakter katarı, mesela 'X', yine de bir karakter katarıdır.

## Değişkenlere Yakından Bir Bakış

Değişkenler Bölüm 2'de tanıtılmıştı. Daha önceden öğrendiğiniz gibi, değişkenler şu ifade stili kullanılarak deklare edilirler:

*tip değişken-ismi*

Bu ifadede **tip**, değişkenin veri tipi; **değişken-ismi** ise değişkenin ismidir. Kısa bir süre önce bahsedilen veri tipleri de dâhil olmak üzere, herhangi geçerli bir tipte değişken deklare edebilirsiniz. Bir değişken tanımladığınızda değişkenin tipinin bir örneğini de tanımlarsınız. Böylece, bir değişkenin becerileri değişkenin tipi ile belirlenir. Örneğin, **bool** tipinde bir değişken, kayan noktalı değerleri saklamak için kullanılamaz. Ayrıca, bir değişkenin tipi, değişkenin yaşam süresi içinde değiştirilemez. Örneğin, bir **int** değişken, **char** değişkene dönüştürülemez.

C#'taki tüm değişkenler kullanılmadan önce deklare edilmelidir. Bu gereklidir, çünkü derleyici, söz konusu değişkeni kullanan herhangi bir ifadeyi derlemeden önce değişkenin ne tür bir veri içerdigini tam olarak bilmelidir. Bu ayrıca C#'ın sıkı bir tip kontrolü yapmasını da mümkün kılar.

C#'ta birkaç farklı türde değişken tanımlanır. Bizim kullanmakta olduğumuz tür *yerel değişkenler* olarak adlandırılır, çünkü bunlar bir metot içinde deklare edilirler.

## Bir Değişkene İlk Değer Atamak

Bir değişkeni kullanmadan önce değişkene bir değer vermelisiniz. Bunu gerçekleştirmenin bir yolu, önceden gördüğünüz gibi atama ifadesi aracılığıyladır. Diğer bir yolu ise değişken deklare edildiği sırada değişkene bir ilk değer atamaktır. Bunu gerçekleştirmek için değişkenin isminin ardından eşittir işareti ve atanacak değeri yazın, İlk değer atamasının genel şekli aşağıdaki gibidir:

`tip değişken = değer;`

Burada *değer*, *değişken* oluşturulduğunda *değişken*'e verilen değerdir. *Değer*, belirtilen tip ile uyumlu olmalıdır.

İşte birkaç örnek:

```
int count = 10; // count'a baslangicta 10 değerini ver
char ch = 'X'; // ch'e X ilk değerini ata
float f = 1.2F; // f'e 1.2 ilk değeri atanır
```

Virgül ile ayrılmış bir liste kullanarak aynı tipte iki veya daha fazla değişkeni deklare ederken, bu değişkenlerden bir veya daha fazlasına ilk değer atayabilirsiniz. Örneğin:

```
int a, b = 8, c = 19, d; // b ve c baslangic degerine sahipler
```

Bu durumda yalnızca **b** ve **c**'ye ilk değer atanıyor.

## Dinamik İlk Değer Ataması

Önceki örneklerde ilk değer olarak yalnızca sabitler kullanılmış olmasına rağmen C#, değişkenin deklare edildiği anda, geçerli olan herhangi bir deyim kullanılarak, değişkenlere dinamik olarak ilk değer atamanıza imkân verir. Örneğin aşağıda, iki dik kenarı verilen bir dik üçgenin hipotenüsünü hesaplayan kısa bir program görülmektedir:

```
// Dinamik ilk değer ataması işlemini gösterir.

using System;

class DynInit {
    public static void Main() {
        double s1 = 4.0, s2 = 5.0; // kenarların uzunlukları

        // hypot'a dinamik olarak ilk değer ata
        double hypot = Math.sqrt( (s1 * s1) + (s2 * s2) );
        Console.WriteLine("Hipotenuse of triangle with sides " + s1 +
                           " by " + s2 + " is ");

        Console.WriteLine("{0:#.###} .", hypot);
    }
}
```

Cıktı ise şöyle olur:

```
Hipotenuse of triangle with sides 4 by 5 is 6.403.
```

Burada üç yerel değişken - **s1**, **s2** ve **hypot** - deklare edilmektedir. İlk ikisine, - **s1** ve **s2** - sabitlerle ilk değer atanmaktadır. Ancak, **hypot**'a hipotenüsün uzunluğu dinamik olarak atanmaktadır. Dikkat ederseniz, ilk değer atama işlemi **Math.Sqrt()** çağrısını da içermektedir. Daha önce açıklandığı gibi, deklarasyon sırasında geçerli olan herhangi bir deyimi kullanabilirsiniz. Bu noktada **Math.Sqrt()**'a (veya diğer herhangi bir kütüphane metoduna) yapılan çağrı geçerli olduğu için **hypot**'a ilk değer atamak üzere **Math.Sqrt()** kullanılabilir. Buradaki en önemli nokta, ilk değer atayan deyimin, ilk değer atanırken geçerli olan herhangi bir öğeyi; metot çağrıları, diğer değişkenler veya literaller de dahil olmak üzere kullanabilmesidir.

## Değişkenlerin Kapsamı ve Yaşam Süreleri

Şimdiye kadar kullanmakta olduğumuz değişkenlerin tümü, **Main()** metodunun başında deklare edilmekteydi. Ancak, C# bir yerel değişkenin herhangi bir blok içinde deklare edilmesine imkân vermektedir. Blok, açılış küme parantezi ile başlar ve kapanış küme parantezi ile sona erer. Blok, bir *deklarasyon uzayı* (*declaration space*) veya *kapsam* (*scope*) tanımlar. Böylece, her yeni blok başlattığınızda yeni bir kapsam tanımlarsınız. Kapsam, programınızın diğer bölümleri tarafından hangi nesnelerin erişilebileceğini belirler. Ayrıca, bu nesnelerin yaşam sürelerini de saptar.

C#'taki en önemli kapsamlar, bir sınıf ve bir metot tarafından tanımlanlardır. Sınıfların kapsamı (ve bu kapsam içinde tanımlı değişkenler) ile ilgili tartışma sınıflar ele alınana kadar ertelenmiştir. Simdilik bir metot tarafından veya bir metot içinde tanımlanan kapsamları inceleyeceğiz.

Bir metot tarafından tanımlanan kapsam, metodun açılış küme parantezi ile başlar. Ancak, söz konusu metodun parametreleri varsa onlar da metodun kapsamına dahil edilir.

Genel bir kural olarak, bir kapsam içinde tanımlanan değişkenler, bu kapsamın dışında tanımlanmış kod tarafından görülemezler (yani. erişilemezler). Böylece, bir kapsam içinde bir değişken deklare ettiğiniz zaman, söz konusu değişkeni yerelleştirmiş ve izinsiz erişimden ve/veya değişikliklerden korumuş olursunuz. Gerçekten de, kapsam kuralları, verilerin paketlenmesi (*encapsulation*) kavramının temelini oluşturmaktadır.

Kapsamlar kümelenebilir. Örneğin, bir kod bloğu oluşturduğunuz anda bir de yeni, kümelenmiş bir kapsam da tanımlamış olursunuz. Bu durum söz konusu olduğunda dıştaki kapsam içekini kuşatır. Bu, dıştaki kapsam içinde deklare edilen nesnelerin içteki kapsam içindeki kod tarafından görüleceği anlamına gelir. Ancak, bunun tersi doğru değildir. İçteki kapsam içinde deklare edilen nesneler, bu kapsam dışından görülemeyeceklerdir.

Kümelenmiş kapsamların etkisini anlamak amacıyla aşağıdaki programa bir göz atın:

```
// Blokların kapsamlarını gösterir.

using System;

class ScopeDemo {
    public static void Main() {
        int x; // Main() içindeki kodların tümü tarafından bilinir

        x = 10;
        if(x == 10) { // yeri bir kapsam baslat

            int y = 20; // sadece bu blok tarafından bilinir

            // x ve y'nin her ikisi de biliniyor
            Console.WriteLine("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Hata! y burada bilinmiyor

        // x hala biliniyor.
        Console.WriteLine("x is * + x");
    }
}
```

Açıklamalardan görüleceği gibi, **x** değişkeni **Main()**'in kapsamının en başında deklare edilmektedir ve **Main()** içindeki kalan kodun tümü tarafından erişilebilir. **y**, **if** bloğu içinde deklare edilmektedir, Blok, bir kapsam tanımladığı için **y** sadece kendi bloğu içindeki diğer kodlar tarafından görülebilir. **y**'nin bloğu dışında kalan **y = 100;** satırının açıklama yoluyla koddan çıkarılmasının nedeni budur. Satırın başında yer alan açıklama simgesini kaldırırsanız, derleme sırasında hata ortaya çıkar, çünkü **y**, bloğu dışında erişilemez. **x**, **if** bloğu içinde kullanılabilir, çünkü bir blok içindeki kod (yani, kümelenmiş kapsam), kendisini kuşatan kapsam tarafından deklare edilen değişkenlere erişim hakkına sahiptir

Blok içinde, değişkenler herhangi bir noktada deklare edilebilir, fakat ancak deklare edildikten sonra geçerlidirler. Yani, bir metodun en başında bir değişken tanımlıyorsanız, bu değişken söz konusu metot içindeki kodun bütününe kullanımına hazırlıdır. Aksi halde, bir blo-

gün sonunda bir değişken tanımlıyorsanız, bu其实e işe yaramaz, çünkü hiç bir kod buna erişemeyecektir.

Hatırınızda tutmanızın önemli olduğu bir başka konu da şudur: Değişkenler, kendi kapsamlarına girildiği zaman oluştururlar ve kapsamlarından çıkışın yok edilirler. Bu, bir değişkenin kapsamını terk ettiği andan itibaren değerini saklamayacağı anlamına gelir. Bu nedenle, bir metot içinde deklare edilen değişkenler, bu metoda yapılan çağrılar arasında değerlerini saklamayacaklardır. Ayrıca, bir blok içinde deklare edilen bir değişken, blok terk edildiğinde kendi değerini kaybedecektir. Yani, bir değişkenin yaşam süresi, değişkenin kapsamı ile sınırlıdır.

Şayet bir değişken deklarasyonu ilk değer ataması içeriyorsa, bu değişkenin deklare edildiği blok içine her girişte, değişkene yeniden ilk değer atanacaktır. Örneğin, şu programa bir göz atın:

```
// Bir degiskenin yasam suresini gosterir.

using System;

class VarlnitDemo {
    public static void Main() {
        int x;

        for[x = 0;  x < 3; x++) {
            int y = -1; // bloga her giriste y'ye ilk deger atanir
            Console.WriteLine("y is: " + y); // bu, daima -1 yazar
            y = 100;
            Console.WriteLine("y  is now: " + y);
        }
    }
}
```

Bu programın ürettiği çıktı aşağıda gösterilmiştir:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Gördüğünüz gibi, **for** döngüsüne her girişte **y**'ye **-1** değeri yeniden atanıyor, **y**'ye sonradan 100 değeri atanıyor olsa bile, bu değer yitiriliyor.

C#'ın kapsamla ilgili kurallarının sizi şaşırtabilecek garip bir tarafı da vardır: Bloklar kümelenemelerine rağmen, içteki kapsam içinde deklare edilen değişkenlerin hiçbirini, içteki kapsamı kuşatan kapsam tarafından deklare edilen değişkenlerle aynı isimde olamaz. Örneğin, aynı isimde iki ayrı değişken deklare etmeye çalışan aşağıdaki program derlenmeyecektir.

```
/*
Bu program, distaki kapsam icinde tanimli
```

```

bir degisen ile ayni isimde, icteki kapsam icinde bir
degisen deklare etmeye calisir.
*** Bu program derlenmez, ***
*/
using System;

class NestVar {
    public static void Main() {
        int count;

        for(count = 0; count < 10; count = count+1) {
            Console.WriteLine("This is count: " + count);

            int count; // kural disi!!!
            for(count = 0; count < 2; count++)
                Console.WriteLine("This program is in error!");
        }
    }
}

```

C/C++ biliyorsanız, bu dillerde, içteki kapsam içinde deklare edilen değişkenlere verilen isimler üzerinde hiçbir kısıtlama olmadığını bilirsiniz. Yani, C/C++’ta dıştaki **for** döngüsü bloğu içinde yer alan **count** deklarasyonu tamamen geçerlidir. Ancak, C/C++’ta bu tür bir deklarasyon, dıştaki değişkeni gizler. C# tasarımcıları isim gizleme (*name hiding*) adlı bu özelliğin kolaylıkla programlama hatalarına yol açabileceğini hissettiler ve buna izin vermediler.

## Tip Dönüşümleri ve Tip Atamaları

Programlama yaparken bir tipteki değişkeni bir başkasına atamak çok yaygındır. Örneğin, aşağıda gösterildiği gibi bir **int** değerini bir **float** değişkenine atmak isteyebilirsiniz:

```

int i;
float f;
i = 10;
f = i; // int'i float'a atiyor

```

Birbiriyile uyumlu tipler bir atama ifadesi içinde karışık biçimde kullanılırsa, sağ taraftaki değer otomatik olarak sol taraftakının tipine dönüştürülür. Yani, yukarıdaki program parçasında **i**'nin içindeki değer **float**'a dönüştürülür, sonra **f**'ye atanır. Ancak, C#'ın sıkı tip kontrolünden ötürü tiplerin tümü uyumlu değildir. Bu nedenle, tip dönüşümlerinin tümüne kapalı olarak izin verilmez. Örneğin, **bool** ve **int** uyumlu değildir. Neyse ki, bir *tip ataması* (*cast*) kullanarak uyumsuz tipler arasında dönüşüm elde etmek, her şeye rağmen mümkündür. Tip ataması, tip dönüşümünü açıkça gerçekleştirir. Hem otomatik tip dönüşümü, hem de tip atamaları aşağıda ele alınmaktadır.

## Otomatik Dönüşümler

Bir tipteki bir veri, bir başka tipteki değişkene atandığında, aşağıdaki şartlar sağlandığı takdirde otomatik tip dönüşümü söz konusu olur:

- Her iki tip uyumludur.
- Hedef tip, kaynak tipten daha büyuktur.

Bu iki koşul sağlandığında *genişletici dönüşüm* (*widening conversion*) söz konusu olur. Örneğin, **int** tipi her zaman geçerli **byte** değerlerinin tümünü tutacak kadar büyütür. Üstelik, **int** ve **byte** tamsayı tipidir; bu nedenle otomatik dönüşüm uygulanabilir.

Genişletici dönüşümler için nümerik tipler, tamsayı ve kayan noktalı tipler de dahil olmak üzere birbirile uyumludur. Örneğin, aşağıdaki program tamamen geçerlidir, çünkü **long**'dan **double**'a yapılan dönüşüm, otomatik olarak gerçekleştirilen bir genişletici dönüşümüdür.

```
// long'dan double'a otomatik dönüşüm gösterir.

using System;

class LtoD {
    public static void Main() {
        long L;
        double D;

        L = 100123285L;
        D = L;
```



```
Console.WriteLine("u after assigning 64000: " + u +
    " -- no data lost.");  
  
// long'u uint'e donustur, veri kaybi var
l = -12;
u = (uint) l;
Console.WriteLine("u after assigning -12: " + u + " - data
    lost.");
Console.WriteLine();  
  
// int'i char'a donustur
b = 88; // x'in ASCII kodu
ch = (char) b;
Console.WriteLine("ch after assigning 88: " + ch);
}  
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Integer outcome of x / y: 3

b after assigning 255: 255 -- no data lost.
b after assigning 257: 1 --data lost.

s after assigning 32000: 32000 -- no data lost.
s after assigning 64000: -1536 -- data lost.

u after assigning 64000: 64000 -- no data lost.
u after assigning -12: 4294967284 -- data lost.

ch after assigning 88: X
```

Şimdi gelin, atamaların her birine göz atalım. (**x/y**)'nin tip ataması yoluyla **int**'e dönüştürülmesi ondalık kısmın kesilmesiyle sonuçlanır, bilgi kaybı olmuştur.

**b**'ye **255** değeri atandığında bilgi kaybı olmaz, çünkü **byte** **255** değerini tutabilir. Ancak, **b**'ye **257** değerini atama girişiminde bulunulunca bilgi kaybı meydana gelir, çünkü **257 byte**'nın değer aralığı dışındadır. Her iki durumda da tip ataması gereklidir, çünkü **int**'ten **byte**'a otomatik dönüşüm yoktur.

**short** değişkeni **s**'e, **uint** tipinde **u** değişkeni üzerinden **32.000** değeri atandığında hiç veri kaybı olmaz, çünkü **short 32.000** değerini tutabilir. Ancak, bir sonraki atamada **u**, **short**'un değer aralığı dışında kalan **64.000** değerine sahiptir ve veri kaybı söz konusu olur. Her iki durumda da tip ataması gereklidir, çünkü **uint**'ten **short**'a otomatik dönüşüm yoktur.

Bir sonraki atamada, **u**'ya **long** tipinde **1** değişkeni üzerinden **64.000** değeri atanır. Bu durumda hiç veri kaybı olmaz, çünkü **64.000 uint** menzili içindedir. Ancak, **u**'ya **-12** değeri atandığında veri kaybı olur, çünkü **uint** negatif sayıları tutamaz. Her iki durumda da tip ataması gereklidir, çünkü **long**'dan **uint**'e otomatik dönüşüm yoktur.

Son olarak, en son atamada bilgi kaybı olmaz, fakat **byte** değerini **char**'a atarken tip ataması gereklidir.

## Deyimlerde Tip Dönüşümleri

Bir atama ifadesinde yer alan tip dönüşümlerine ek olarak ayrıca deyimler içinde de tip dönüşümlerine rastlanır. Bir deyim içinde iki veya daha fazla sayıda farklı veri tipini, tiplerin birbirleriyle uyumlu olmaları şartıyla, özgürce karıştırılabilirsiniz. Örneğin, **short** ve **long**'u bir deyim içinde karışık kullanabilirsiniz, çünkü bunların her ikisi de nümerik tiplerdir. Farklı veri tipleri bir deyim içinde karıştırıldığında, tek tek işlem bazında aynı tipe dönüştürülür.

Dönüşümler, C#'ın *tip terfi kuralları* (*type promotion rules*) kullanılarak gerçekleştirilir. Bu kuralların ikili (binary) işlemler için tanımladığı algoritma şu şekildedir:

EĞER bir operand **decimal** ise, BU DURUMDA diğer operand **decimal**'e yükseltilir (terfi ettirilir). (Diğer operand, **float** veya **double** tipinde olmadığı sürece bu doğrudur. Aksi halde hata ortaya çıkar.)

AKSİ HALDE, EĞER bir operand **double** ise, ikinci operand **double**'a yükseltilir.

AKSİ HALDE, EĞER bir operand **float** ise, ikincisi **float**'a yükseltilir,

AKSİ HALDE, EĞER bir operand **ulong** ise, ikincisi **ulong**'a yükseltilir. (İkinci operand, **sbyte**, **short**, **int** veya **long** tipinde olmadığı sürece bu doğrudur. Aksi halde, hata ortaya çıkar.)

AKSİ HALDE. EĞER bir operand **long** ise, ikincisi **long**'a yükseltilir.

AKSİ HALDE, EĞER bir operand **uint** ise ve ikincisi **sbyte**, **short** veya **int** tipinde ise, her ikisi de **long**'a yükseltilir.

AKSİ HALDE, EĞER bir operand **uint** ise, ikincisi **uint**'e yükseltilir.

AKSİ HALDE her iki operand da **int**'e yükseltilir.

Tip terfi kuralları ile ilgili dikkat edilmesi gereken birkaç önemli husus vardır. Öncelikle, bir deyim içinde tiplerin tümü karıştırılamaz. Özellikle, **float** veya **double**'dan **decimal**'e kapalı dönüşüm söz konusu değildir. Ayrıca, **ulong**'u da herhangi işaretsiz bir tamsayı tip ile karıştırmak mümkün değildir. Bu tipleri karıştırmak, açık bir tip ataması kullanımını gerektirir.

İkinci olarak, son kurala özellikle dikkat edin. Bu kural, önceki kuralların hiçbirini geçerli olmadığı takdirde diğer operandların tümünün **int**'e yükseltileceğini bildiriyor. Bu nedenle, bir deyim içinde tüm **char**, **sbyte**, **byte**, **ushort** ve **short** değerleri hesaplama amaçlı kullanımlar için yükseltilir. Buna *tamsayı terfisi* (*integer promotion*) denir. Bu, ayrıca tüm aritmetik işlemlerin sonuçlarının **int**'ten daha küçük olmayacağı anlamına gelir.

Tip terfilerinin sadece, bir deyimin değeri hesaplanırken üzerinde işlem yapılan değerler üzerine etki ettiğini kavramak önemlidir. Örneğin, eğer bir **byte** değişkeninin değeri bir deyim içinde **int**'e yükseltilmişse, deyimin dışında değişken hala **byte**'dır. Tip terfisi yalnızca deyimin değerinin hesaplanması etkiler.

Tip terfisi, her şeye rağmen, bazı beklenmedik sonuçlara da yol açabilir. Örneğin, bir aritmetik işlem iki **byte** değeri içerirse, peş peşe şu durumlar ortaya çıkar, İlk önce, **byte** operandları **int**'e yükseltilir. Sonra, işlem yapılır, Bu işlem **int** tipinde bir sonuç üretir. Böylece, iki **byte** değeri içeren işlemin sonucu bir **int** olacaktır. Bu, içten içe beklemiş olabileceğiniz bir durum değildir. Bir de, aşağıdaki programı ele alın:

```
// Terfi surprizi!

using System;

class PromDemo {
```

```

public static void Main() {
    byte b;

    b = 10;
    b = (byte) (b * b); // tip ataması gereklidir!

    Console.WriteLine("b: " + b);
}
}

```

Sezgilerimize biraz ters gelebilse de, **b\*b** değerini tekrar **b**'ye atarken **byte** atamasına gerek vardır. Bunun nedeni, **b\*b** deyiminin değeri hesaplanırken **b**'nin değerinin **int**'e yükseltilmesidir. Yani, **b\*b** işlemi **int** değeri ile sonuçlanır ki bu, tip ataması kullanılmadan bir **byte** değişkenine atanamaz. Tamamen uygun olması beklenen deyimlerde tip uyumsuzluğu ile ilgili beklenmedik hata mesajlarına rastlıyorsanız, bu bahsedilen durumu hatırlınızdan çıkarmayın.

Aynı durum, **char**'lar üzerinde gerçekleştirilen işlemlerde de ortaya çıkmaktadır. Örneğin, aşağıdaki kod parçasında, **ch1** ve **ch2**'nin deyim içinde **int**'e yükseltilmesinden ötürü tip ataması yardımıyla tekrar **char**'a dönüşüm gereklidir:

```

char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);

```

Tip ataması olmadan **ch1**'i **ch2**'ye ekleme işlemi **int** olarak sonuçlanır. Bu sonuç ise **char**'a atanamaz.

Tip terfileri ayrıca tekli işlemlerde de, mesela tekli - işleminde, yer alabilir. Tekli işlemlerde, **int**'ten küçük operandlar (**byte**, **sbyte**, **short** ve **ushort**) **int**'e yükseltilir. Ayrıca, **char** operandı da **int**'e dönüştürülür. Bundan başka, bir **uint**'in değeri negatif olmuşsa **long**'a yükseltilir.

## Deyimlerde Tip Ataması Kullanmak

Büyükçe bir deyimin belirli bir kısmına tip ataması uygulanabilir. Bu, bir deyimin değeri hesaplanırken ortaya çıkan tip dönüşümlerinin meydana geliş şekilleri üzerinde hassas bir kontrol sağlar. Örneğin, aşağıdaki programı ele alın. Bu program 1'den 10'a kadar olan sayıların karelerini gösteriyor. Program ayrıca her sonucun hem tamsayı kısmını, hem de ondalık kısmını ayrı ayrı gösteriyor. Bunu gerçekleştirmek için **Math.sqrt()**'un sonucunu **int**'e dönüştürmek amacıyla tip ataması kullanıyor.

```

// Deyim içinde tip atamaları kullanmak.

using System;

class CastExpr {
    public static void Main() {
        double n;

```

```
for(n = 1.0; n <= 10; n++) {
    Console.WriteLine("The square root of {0} is {1}", n,
                      Math.Sqrt(n));

    Console.WriteLine("Whole number part: {0}" , (int)
                      Math.Sqrt(n));

    Console.WriteLine("Fractional part: {0}", Math.Sqrt(n)
                      - (int) Math.Sqrt(n));
    Console.WriteLine();
}
}
```

Programdan alınan çıktı şöyle olur:

```
The square root of 1 is 1
Whole number part: 1
Fractional part: 0
```

```
The square root of 2 is 1.4142135623731
Whole number part: 1
Fractional part: 0.414213562373095
```

```
The square root of 3 is 1.73205080756888
Whole number part: 1
Fractional part: 0.732050807568877
```

```
The square root of 4 is 2
Whole number part: 2
Fractional part: 0
```

```
The square root of 5 is 2.23606797749979
Whole number part: 2
Fractional part: 0.23606797749979
```

```
The square root of 6 is 2.44948974278318
Whole number part: 2
Fractional part: 0.449489742783178
```

```
The square root of 7 is 2.64575131106459
Whole number part: 2
Fractional part: 0.645751311064591
```

```
The square root of 8 is 2.82842712474619
Whole number part: 2
Fractional part: 0.82842712474619
```

```
The square root of 9 is 3
Whole number part: 3
Fractional part: 0
```

```
The Square root of 10 is 3.16227766016838
Whole number part: 3
Fractional part: 0.16227766016838
```

Cıktıdan görüleceği gibi, **Math.Sqrt()**'un tip ataması yoluyla **int**'e dönüştürülmesi, sonuç değerinin tamsayı bileşenine karşılık geliyor. Şu deyimde,

```
Math.Sqrt(n) = (int) Math.Sqrt(n)
```

**int**'e dönüştüren tip ataması ile tamsayı bileşen elde ediliyor. Bu değer daha sonra, asıl değerden çıkarılarak ondalık bileşen elde ediliyor. Yani, deyimin sonucu **double**'dır. **int**'e dönüştürülen yalnızca **Math.Sqrt()**'a yapılan ikinci çağrıının değeridir.

**4**

**DÖRDÜNCÜ BÖLÜM**

---

# **OPERATÖRLER**

C#, programcılara, deyimlerin kurulması ve değerlerinin hesaplanması üzerinde ayrıntılı bir kontrol sağlayan kapsamlı bir operatör seti sunmaktadır, C# dört adet genel operatör sınıfı içerir: *aritmetik*, *bit tabanlı*, *ilişkisel* ve *mantıksal*. Bunlar bu bölümde incelenmektedir. Ayrıca, ele alınan diğer konular arasında atama operatörü ve ? operatörü de yer almaktadır. C#'ta belirli özel durumları kontrol altına alan diğer birkaç operatör daha tanımlanmıştır. Bu özel operatörler bu kitapta daha sonra, bu operatörlerin uygulandıkları özellikler ele alındığında incelenmektedir.

## Aritmetik Operatörler

C#'ta aşağıdaki aritmetik operatörler tanımlıdır:

<b>Operatör</b>	<b>Anlamı</b>
-----------------	---------------

+	Toplama
-	Çıkarma
*	Çarpma
/	Bölme
%	Modülüs (Kalan)
++	Artırma
--	Eksiltme

+, -, \* ve / operatörleri diğer bilgisayar dillerinde (veya cebirde) nasıl çalışıyorlarsa, C#'ta da aynı şekilde çalışırlar. Bunlar herhangi bir standart nümerik veri tipine uygulanabilirler.

Aritmetik operatörlerin faaliyetleri okuyucuların tümü tarafından çok iyi biliniyor olsa da, birkaç özel durumu açıklamak gereklidir. Öncelikle, / operatörü bir tamsayıya uygulandığı zaman, kalanın kesilip atılacağını hatırlınızdan çıkarmayın. Örneğin, **10/3** tamsayı bölme işleminin sonucu **3**'e eşit olacaktır. Bu bölmenin kalanını modülüs operatörünü (%) kullanarak elde edebilirsiniz, Modülüs operatörü C#'ta, diğer dillerde çalıştığı gibi, tamsayı bölmesinin kalanını sonuç olarak verir. Örneğin, **10 % 3** işleminin sonucu **1**'dir. C#'ta % operatörü hem tamsayılara hem de kayan noktalı sayılarla uygulanabilir. Yani, **10.0 % 3.0** işleminin de sonucu **1**'dir. (Bu, modülüs işlemlerine yalnızca tamsayı tipler üzerinde izin veren C/C++'tan farklıdır.) Aşağıdaki program modülüs operatörünü göstermektedir:

```
// % operatorunun kullanımını gösterir.

using System;

class ModDemo {
    public static void Main() {
        int irest, irem;
        double drest, drem;

        irest = 10 / 3;
        irem = 10 % 3;
```

```

dresult = 10.0 / 3.0;
drem = 10.0 % 3.0;

Console.WriteLine("Result and remainder of 10 / 3: " +
                   irest + " " + irem);

Console.WriteLine("Result and remainder of 10.0 / 3.0: " +
                   dresult + " " + drem);
}
}

```

Programdan elde edilen çıktı aşağıda gösterilmiştir:

```

Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.333333333333333 1

```

Gördüğünüz gibi, `%` operatörü hem tamsayılarla hem de kayan noktalı sayılarla yapılan işlemlerde 1 kalanını veriyor.

## Artırma ve Eksiltme

Bölüm 2'de tanıtılan `++` ve `--` operatörleri, artırma ve eksiltme operatörleridir. Bu operatörleri oldukça ilginç kılan bazı özel özelliklere sahip olduklarını öğreneceksiniz. Gelin şimdi, artırma ve eksiltme operatörlerinin tam olarak ne işe yaradıklarını tekrarlayarak başlayalım.

Artırma operatörü operandına 1 ekler; eksiltme operatörü de 1 çıkartır. Bu nedenle,

```
x = x + 1;
```

aşağıdaki ifade ile aynıdır:

```
x++;
```

Ayrıca,

```
x = x - 1;
```

şu İfade ile aynıdır:

```
x--;
```

Artırma ve eksiltme operatörlerinin her ikisi de operandın önünde (önek) veya arkasında (sonek) yer alabilir. Örneğin, şu ifade

```
x = x + 1;
```

yeniden aşağıdaki gibi yazılabılır:

```
++x; // önek formunda
```

Veya şu şekilde de yazılabilir:

```
x--; // sonek formunda
```

Yukarıdaki örnekte artırmanın önekli veya sonekli olarak uygulanması arasında hiçbir fark yoktur. Ancak, artırma veya eksiltme operatörleri daha büyük bir deyimin bir parçası olarak kullanılıncı önemli bir fark vardır. Artırma veya eksiltme operatörü, operandının *önünde* yer alıysa C#, deyimin geri kalan kısmında kullanılmak üzere operandın değerini hesap etmeden önce işlemi gerçekleştirecektir. Eğer operatör operandının peşinden geliyorsa C#, artırma veya eksiltme işleminden önce operandın değerini hesap edecektir. Aşağıdaki ifadelere bir bakın;

```
x = 10;
y = ++x;
```

Bu durumda, **y**'ye **11** değeri verilecektir. Ancak, eğer kod şu şekilde yazılsırsa

```
x = 10;
y = x++;
```

**y**'ye **10** değeri verilecektir. Her iki durumda da **x**'e **11** değeri verilir; aradaki fark, bu işlemin ne zaman gerçekleştiğidir.

Artırma veya eksiltme işleminin ne zaman ele alınacağını kontrol edebilmenin sağladığı önemli avantajlar vardır. Bir sayı serisi üreten aşağıdaki programa bir göz atın:

```
/*
  +'nin önek ve sonek formları
  arasındaki farkı gösterir.
*/
using System;

class PrePostDemo {
    public static void Main() {
        int x, y;
        int i;

        x = 1;

        Console.WriteLine("Series generated using y = x + x++;");
        for(i = 0; i < 10; i++) {

            y = x + x++; // sonek ++
            Console.WriteLine(y + " ");
        }
        Console.WriteLine();

        x = 1;
        Console.WriteLine("Series generated using y = x + ++x;");
        for(i = 0; i < 10; i++) {

            y = a + ++x; // önek ++
            Console.WriteLine(y + " ");
        }
    }
}
```

```
        Console.WriteLine();  
    }  
}
```

Çıktı aşağıda gösterilmiştir:

```
Series generated using y = x + x++;  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20
```

```
Series generated using y = x + ++x;  
3  
5  
7  
9  
11  
13  
15  
17  
19  
21
```

Çıktının gösterdiği gibi, şu ifade

`y = x + x++;`

`x`'in değerini `x`'e ekler ve sonucu `y`'ye atar. Sonra `x`'i 1 artırır. Ancak, şu ifade

`y = x + ++x;`

`x`'in değerini alır, `x`'i 1 artırır, sonra bu değeri `x`'in asıl değerine ekler. Sonuç `y`'ye atanır. Çıktıdan da görüleceği gibi, sadece `x++` deyimini, `++x` şeklinde çevirmek, sayı serisini çift sayı serisinden tek sayı serisine dönüştürür,

Önceki örnekle ilgili bir başka husus ise şudur: Şu tür deyimlerin

`x + ++x`

gözünüzü korkutmasına izin vermeyin. İki operatörün peş peşe gelmesi ilk bakışta biraz rahatsız edici olmasına rağmen derleyici bunu düzgün olarak gerçekleştirir. Bu deyimin, `x`'i `x`'in artırılmış değerine eklemekten ibaret olduğunu hatırlınızdan çıkarmayın.

## İlişkisel ve Mantıksal Operatörler

*İlişkisel operatörler (relational operators) ve mantıksal operatörler (logical operators)* terimlerinde geçen *ilişkisel* sözcüğü değerlerin birbiri arasında sahip olabileceği ilişkiyi; *mantıksal* sözcüğü ise true ve false değerlerin birbiriyle bağlanabilme yollarını ifade eder. İlişkisel operatörler true veya false sonuçlarını üretikleri için, genellikle mantıksal operatörlerle birlikte kullanılırlar. Bu nedenle, burada mantıksal operatörlerle birlikte ele alınacaklardır.

İlişkisel operatörler aşağıdaki gibidir:

Operatör	Anlamı
==	Eşittir
!=	Eşit değildir
>	Büyükür
<	Küçükür
>=	Büyükür veya eşittir
<=	Küçükür veya eşittir

Sırada mantıksal operatörler yer almaktadır:

Operatör	Anlamı
&	VE
	VEYA
^	XOR (özel VEYA)
	Kısa devre VEYA
&&	Kısa devre VE
!	DEĞİL

İlişkisel ve mantıksal operatörlerin sonucu bir **bool** değerdir.

C#'ta == ve != kullanılarak tüm nesneler eşitlik veya eşitsizlik açısından karşılaştırılabilir. Ancak, karşılaştırma operatörleri olan <, >, <= veya >=, sadece sıralama ilişkisini destekleyen tiplere uygulanabilir. Bu nedenle, ilişkisel operatörlerin tümü bütün nümerik tiplere uygulanabilir. Ancak, **true** ve **false** değerleri sıralı olmadığı için **bool** tipindeki değerler sadece eşitlik veya eşitsizlik için karşılaştırılabilirler. Örneğin, **true>false** deyiminin C#'ta bir anlamı yoktur.

Mantıksal operatörlerin operandları **bool** tipinde olmalıdır; mantıksal bir işlemin sonucu da **bool** tipinde olmalıdır. Mantıksal operatörler olan &, |, ^ ve !, temel mantıksal işlemleri - yani, VE, VEYA, XOR ve DEĞİL - aşağıdaki doğruluk tablosuna göre destekler:

<b>p</b>	<b>q</b>	<b>p&amp;q</b>	<b>p   q</b>	<b>p ^ q</b>	<b>!p</b>
Yanlış	Yanlış	Yanlış	Yanlış	Yanlış	Doğru
Doğru	Yanlış	Yanlış	Doğru	Doğru	Yanlış
Yanlış	Doğru	Yanlış	Doğru	Doğru	Doğru
Doğru	Doğru	Doğru	Doğru	Yanlış	Yanlış

Tablodan da görüldüğü gibi, XOR işleminin sonucu, sadece ve sadece tek bir operand doğru ise doğrudur.

Aşağıda, ilişkisel ve mantıksal operatörlerin birkaçını gösteren bir program görüyorsunuz:

```
// İlişkisel ve mantıksal operatorleri gösterir.

using System;

class RelLogOps {
    public static void Main() {
        int i, j;
        bool b1, b2;
        i = 10;
        j = 11;
        if(i < j) Console.WriteLine("i < j");
        if(i <= j) Console.WriteLine("i <= j");
        if(i != j) Console.WriteLine("i != j");
        if(i == j) Console.WriteLine("this won't execute");
        if(i >= j) Console.WriteLine("this won't execute");
        if(i > j) Console.WriteLine("this won't execute");

        b1 = true;
        b2 = false;
        if(b1 & b2) Console.WriteLine("this won't execute");
        if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) is true");
        if(b1 | b2) Console.WriteLine("b1 | b2 is true");
        if(b1 ^ b2) Console.WriteLine("b1 ^ b2 is true");
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

C#'ın sunduğu mantıksal operatörler, en yaygın olarak kullanılan mantıksal işlemleri gerçekleştirmektedir. Bununla birlikte, genel mantık kuralları tarafından tanımlanan birçok başka işlem de mevcuttur. Bu diğer mantıksal işlemler, C# tarafından desteklenen mantıksal operatörler kullanılarak kurulabilir. Yani, C# diğer mantıksal işlemlerden herhangi birini kurmak için yeterli sayıda mantıksal operatör sağlamaktadır. Örneğin, bir başka mantıksal

İşlem ise *çıkarımdır* (implication). Çıkanım; sonucun yalnızca, soldaki operand doğru ve sağdaki operand yanlış iken yanlış olduğu, ikili bir işlemidir. (Çıkarım işlemi, doğrunun yanlış anlamına gelemeyeceği fikrini yansıtmaktadır.) Bu sonuçla, çıkarım operatörünün doğruluk tablosu aşağıda gösterilmiştir:

<b>Operatör</b>	<b>Anlamı</b>	<b>p gerektirir q</b>
Doğru	Doğru	Doğru
Doğru	Yanlış	Yanlış
Yanlış	Yanlış	Doğru
Yanlış	Doğru	Doğru

Çıkarım operatörü, aşağıda gösterildiği gibi, ! ve | operatörlerinin birleştirilerek kullanılmasıyla kurulabilir:

`!p | q`

Bunun uygulaması aşağıdaki programda gösterilmektedir:

```
// C#'ta bir çıkarım operatoru olusturmak.

using System;

class Implication {
    public static void Main() {
        bool p = false, q=false;
        int i, j;

        for(i =0; i < 2; i++){
            for(j = 0; j < 2; j++) {
                if(i == 0) p = true;
                if(i == 1) p = false;
                if(j == 0) q = true;
                if(j == 1) q = false;

                Console.WriteLine("p is " + p + ", q is " + q);
                if(!p | q) Console.WriteLine(p + " implies " + q +
                    " is " + true);
                Console.WriteLine();
            }
        }
    }
}
```

Çıktı aşağıda gösterilmiştir:

p is True, q is True  
True implies True is True

p is True q is False

p is False, q is True

```
False implies True is True
```

```
p is False, q is False
False implies False is True
```

## Kısa Devre Mantıksal Operatörler

C#, daha verimli kod üretmek amacıyla kullanabilecek VE ve VEYA mantıksal operatörlerinin özel *kısa devre* versiyonlarını sağlar. Bunun neden böyle olduğunu anlamak için şuna bir bakın. Bir VE işleminde, eğer ilk operand yanlış ise ikinci operand hangi değeri içerirse içersin sonuç yanlıştır. Bir VEYA işleminde, eğer ilk operand doğru ise ikinci operandın değeri ne olursa olsun işlemin sonucu doğrudur. Yani, bu iki durumda ikinci operandın değerini dikkate almaya gerek yoktur. İkinci operandın değerini işleme almayarak zamanдан kazanılmış ve daha verimli bir kod üretilmiş olur.

Kısa devre VE operatörü **&&** operatördür; kısa devre VEYA operatörü ise **||** operatördür. Önceden bahsedildiği gibi, bunların normal karşılıkları **&** ve **|** operatörleridir. Normal ve kısa devre versiyonlar arasındaki fark, normal operatörlerin her zaman operandların her birinin değerini hesaplaması, fakat kısa devre versiyonlarının ikinci operandın değerini yalnızca gerektiğiinde hesaplamasıdır.

Aşağıda, kısa devre VE operatörünü gösteren bir program görünsünüz. Program, **d**'nin içindeki değerin **n**'nin bir çarpanı olup olmadığını belirler. Bunu, modülüs işlemi ile gerçekleştirir. Eğer **n/d** işleminin kalanı sıfır ise **d** bir çarpandır. Ancak, modülüs işlemi bölme içерdiği için sıfıra bölme hatasını önlemek amacıyla VE'nin kısa devre versiyonu kullanılmıştır.

```
// Kısa devre operatorleri gösterir.

using System;

class SCops {
    public static void Main() {
        int n, d;

        n = 10;
        d = 2;
        if(d != 0 && (n % d) == 0)
            Console.WriteLine(d + " is a factor of " + n);

        d = 0; // Şimdi, d'ye sıfır değerini ver

        // d, sıfır olduğu için ikinci operator hesaplanmaz.
        if(d != 0 && (n % d) == 0)
            Console.WriteLine(d + " is a factor of " + n);

        /*Simdi, aynı işlemi kısa devre operatörü olmadan deneyin.
         Bu, sıfıra bölme hatasına neden olur.*/
        if(d != 0 & (n % d) == 0)
            Console.WriteLine (d + " is a factor of " + n);
    }
}
```

}

Sıfır'a bölme hatasını önlemek amacıyla **if** ifadesi öncelikle **d**'nin sıfır'a eşit olup olmadığını kontrol ediyor. Eğer eşitse, kısa devre VE bu noktada durur ve modülüs bölme işlemini gerçekleştirmez. Böylece, ilk testte **d**, 2'dir ve modülüs işlemi gerçekleştirilir. İkinci test başarısız olur, çünkü **d**'ye sıfır değeri verilmiştir; modülüs işlemi sıfır'a bölme hatası önlenerek atlanır. Son olarak, normal VE operatörü denenmektedir. Bu, her iki operandın değerlerinin hesaplanması neden olur. Bu durum, sıfır'a bölme söz konusu olduğunda çalışma zamanı (runtime) hatasına yol açar.

Kısa devre operatörler bazı durumlarda normal karşılaşıklarından daha verimli oldukları için C#'ın niye hala normal VE ve VEYA operatörlerini sağladığını merak ediyor olabilirsiniz. Bunun yanıtı şudur: Bazı durumlarda, ortaya çıkan yan etkilerden dolayı bir VE ya da VEYA işleminin her iki operandının da değerini hesaplamayı tercih edeceksiniz. Şunu ele alın:

```
// Yan etkiler onemli olabilir.

using System;

class SideEffects {
    public static void Main() {
        int i;

        i = 0;

        /* Burada, if başarılı olmasa da i, yine de
           artırılır. */
        if(false & (++i < 100)
            Console.WriteLine("this won't be displayed");
        Console.WriteLine("if statement executed: " + i);
        // ekranda 1 gösterir

        /* Bu kez, i artırılmıyor, çünkü
           kısa devre operatör artırımı atlıyor */
        if(false && (++i < 100))
            Console.WriteLine("this won't be displayed");
        Console.WriteLine("if statemant executed: " + i);
        // hala 1 !!
    }
}
```

Açıklamalardan görüleceği gibi, ilk **if** ifadesinde **if** doğrulansın veya doğrulanmasın, **i** artırılır. Ancak kısa devre operatör kullanıldığında, ilk operand yanlış olduğunda **i** değişkeni artırılmaz. Bundan çıkarılacak ders şudur: Eğer kodunuz VE ya da VEYA işleminin sađaki operandının değerinin hesaplanması gerektiğiniysa, C#'ın bu işlemlerle ilgili kısa devre olmayan formlarını kullanmalısınız.

Bir başka husus da şudur: Kısa devre VE, *koşullu VE* (*conditional-AND*) olarak da bilinir. Ayrıca, kısa devre VEYA ise *koşullu VEYA* (*conditional-OR*) olarak da adlandırılır.

## Atama Operatörü

Atama operatörünü Bölüm 2'den beri kullanmaktasınız. Şimdi bu operatörü kurallı olarak ele almanın vakti geldi. *Atama operatörü*, bir adet eşittir işaretidir (=). Atama operatörü C#'ta diğer bilgisayar dillerindekine çok benzer şekilde çalışır. Genel olarak atama operatörü şu stildedir:

*değişken* = *deyim*;

Burada, *değişken*'in tipi *deyim*'in tipi ile uyumlu olmalıdır.

Atama operatörü, size tanıdık gelmeyecek ilginç bir niteliğe sahiptir: Atama operatörü; bir atama zinciri oluşturmaya imkan verir. Örneğin, aşağıdaki kod parçasını ele alın:

```
int x, y, z;
x = y = z = 100; // x, y ve z'ye 100 değeri veriliyor
```

Bu kod parçası tek bir ifade kullanarak **x**, **y** ve **z** değişkenlerine **100** değerini veriyor. Bu işe yarar, çünkü **=**, sağ taraftaki deyimin değerini hesaplayan bir operatördür. Böylece, **z=100** ifadesinin değeri **100**'dir ve bu değer, **y**'ye atanır, sonra sırası geldiğinde **x**'e atanır. Bir "atama zinciri" kullanmak, bir grup değişkene ortak bir değer vermenin koiay bir yoludur.

## Bileşik Atamalar

C#, belirli atama ifadelerinin kodlanması kolaylaştıran özel bileşik atama operatörleri sağlar. Gelin, bir örnek ile başlayalım. Burada gösterilen atama ifadesi:

```
x = x + 10;
```

bileşik atama kullanılarak yeniden yazılabilir:

```
x += 10;
```

**+=** operatör çifti derleyiciye, **x**'e **x** artı **10** değerini atamasını söyler.

İşte bir başka örnek daha. Şu ifade

```
x = x - 100;
```

aşağıdaki ifadeyle aynıdır:

```
x -= 100;
```

Her iki ifade de, **x**'e **x** eksi **100** değerini atar.

İkili operatörlerin (yani, iki operand gerektirenlerin) tümü için bileşik atama operatörleri mevcuttur. Kısayol ifadesinin genel şekli şöyledir:

*değişken op* = *deyim*;

Böylece, aritmetik ve mantıksal atama operatörleri şu şekli alır:

+=	-=	*=	/=
%=	&=	=	^=

Bileşik atama ifadeleri bileşik olmayan eşdeğerlerinden daha kısa oldukları için, bileşik atama operatörleri kimi zaman kısayol atama operatörleri olarak da adlandırılır.

Bileşik atama operatörleri iki avantaj sağlarlar. Birincisi, “uzun yoldan” ifade edilen eşdeğerlerinden daha kompakttırlar. İkincisi, daha verimli çalıştırılabilir kod üretimesini sağlayabilirler (çünkü operandın değeri sadece bir kez hesaplanır). Bu nedenlerden ötürü, profesyonelce yazılmış C# programlarında sık sık bileşik alama operatörlerinin kullanıldığını göreceksiniz.

## Bit Tabanlı Operatörler

C#, uygulanabileceği problem tiplerini genişleten birtakım *bit tabanlı (bitwise)* operatörler sunmaktadır. Bit tabanlı operatörler doğrudan operandlarının bitleri üzerinde işlem yaparlar. Bit tabanlı operatörler yalnızca tamsayı operandlar için tanımlıdır. **bool**, **float** ve **double** üzerinde kullanılamazlar.

Bu operatörlere *bit tabanlı* denir, çünkü bir tamsayı değeri oluşturan bitleri test etmek, ayarlamak veya kaydirmakta kullanılırlar. Bit tabanlı işlemler, sistem seviyesindeki çok çeşitli görevler açısından - örneğin bir aygıtın durum bilgilerinin sorgulanması veya kurulması gerektiğinde - önemlidir. Bit tabanlı operatörler Tablo 4.1’ de listelenmiştir.

### Bit Tabanlı VE, VEYA, XOR ve DEĞİL Operatörleri

Bit tabanlı VE, VEYA, XOR ve DEĞİL operatörleri sırasıyla şunlardır: **&**, **|**, **^** ve **~**. Bu operatörler, daha önce bahsedilen Boolean mantık eşdeğerleriyle aynı işlemleri gerçekleştirirler. İkisi arasındaki fark, bit tabanlı operatörlerin bitleri esas alarak tek tek bitler üzerinde işlem yapmalarıdır.

**TABLO 4.1: Bit Tabanlı Operatörler**

Operatör	Anlamı
<b>&amp;</b>	Bit tabanlı VE
<b> </b>	Bit tabanlı VEYA
<b>^</b>	Bit tabanlı XOR
<b>&gt;&gt;</b>	Sağa kaydırma
<b>&lt;&lt;</b>	Sola kaydırma
<b>~</b>	Bir'in tümleyeni (tekli DEĞİL)

Aşağıdaki tablo, **1**'ler ve **0**'lar kullanarak her işlemin sonucunu gösteriyor:

<b>p</b>	<b>q</b>	<b>p&amp;q</b>	<b>p q</b>	<b>p^q</b>	<b>~p</b>
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

En yaygın kullanım şekliyle bir tabanlı VE işlemini, bitleri sıfırlamanın bir yolu olarak düşünebilirsiniz. Yani, her iki operandın içindeki **0** değerine sahip herhangi bir bit, sonuç değerde karşılık gelen bitin **0** olarak ayarlanması neden olur. Örneğin,

$$\begin{array}{r}
 1101\ 0011 \\
 1010\ 1010 \\
 \& \text{-----} \\
 1000\ 0010
 \end{array}$$

Aşağıdaki program, tek sayıları çift sayılaraya çevirmek için **&** operatörünü kullanarak bu operatörü tanıtıyor. Bu işlemi, sıfırıncı biti sıfırlayarak gerçekleştiriyor. Örneğin **9**, ikilik düzende **0000 1001**'dir. Sıfırıncı bit sıfırlandığında, bu sayı **8** ya da ikili düzende **0000 1000** olur.

```

/* Bir sayiyi cift sayiya donusturmek icin bit tabanli VE
   kullanir. */

using System;

class MakeEven {
    public static void Main(){
        ushort num;
        ushort i;

        for(i = 1; i <= 10; i++) {
            num = i;

            Console.WriteLine("num: " + num);

            num = (ushort) (num & 0xFFE); // num & 1111 1110

            Console.WriteLine("num after turning off bit zero: " +
                num + "\n");
        }
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

num: 1
num after turning off bit zero: 0

num: 2
num after turning off bit zero: 2

```

```

num: 3
num after turning off bit eero: 2

num: 4
num after turning off bit zero: 4

num: 5
num after turning off bit zero: 4

num: 6
num after turning off bit zero: 6

num: 7
num after turning off bit zero: 6

num: 8
num after turning off bit zero: 8

num: 9
num after turning off bit zero: 8

num: 10
num after turning off bit zero; 10

```

VE ifadesinde kullanılan **0xFFFFE** değeri, **1111 1110** sayısının onaltılık düzendeki gösterimidir. Böylece VE işlemi, **ch**'in sıfırıncı bit hariç tüm bitlerini değiştirmeden bırakır. Sıfırıncı biti ise sıfırlar. Yani, çift sayılar değişmeden kalır, ama tek sayılar, sayı değerleri bir azaltılarak çift sayı haline getirilir.

VE operatörü ayrıca, bir bitin **1** veya **0** olup olmadığını belirlemek istediğinizde de kullanışlıdır. Örneğin, aşağıdaki program bir sayının tek sayı olup olmadığını belirliyor:

```

/* Bir sayinin tek sayı olup olmadigini belirlemek icin bit
tabanli VE kullanir. */

using System;

class IsOdd {
    public static void Main() {
        ushort num;

        num = 10;

        if((num & 1) == 1)
            Console.WriteLine("This won't display.");

        num = 11;

        if((num & 1) == 1)
            Console.WriteLine(num + " is odd.");
    }
}

```

Cıktı şu şekildedir:

11 is odd.

**if** ifadeleri içinde **num** değerine 1 ile VE uygulanır. Eğer **num**'ın sıfırıncı biti 1 ise, **num&1** işleminin sonucu 1'dir; aksi halde, sonuç sıfırdır. Bu nedenle, **if** ifadesi yalnızca sayı tek olduğunda doğrulanır.

Bit tabanlı & operatörünün bit testi becerisini, ikili biçimdeki bir **byte** değerinin bitlerini göstermek üzere bit tabanlı & operatörü kullanan bir program geliştirmek için kullanabilirsiniz, Aşağıdaki gibi bir yaklaşım uygulanabilir:

```
// Bir byte icindeki bitleri gösterir.

using System;

class ShowBits {
    public static void Main() {
        int t;
        byte val;

        val = 123;
        for(t = 128; t > 0; t = t/2) {
            if((val & t) != 0) Console.Write("1 ");
            if((val & t) == 0) Console.Write("0 ");
        }
    }
}
```

Cıktı aşağıda gösterilmiştir:

01111011

**for** döngüsü, bit tabanlı VE operatörünü kullanarak **val**'ın bitlerinin 1 veya 0 olup olmadığını peş peşe test eder. Eğer bit açık (1) ise 1 rakamı ekranda gösterilir; aksi halde, 0 gösterilir.

Bit tabanlı VEYA, VE'nin tersi olarak, bitleri 1'e çevirmek için kullanılabilir. Her iki operandda 1 değerine sahip herhangi bir bit, değişken içinde karşılık gelen bitin 1 olarak ayarlanmasıına neden olacaktır. Örneğin,

```
1101 0011
1010 1010
| -----
1111 1011
```

Az önce gösterilen sayıları çift sayıya çeviren programı, VEYA kullanarak sayıları tek sayıya çeviren bir program haline dönüştürebilirsiniz:

```
// Bir sayiyi tek sayiya donusturmek icin bit tabanli VEYA kullanir.
```

```
using System;
```

```

class MakeOdd {
    public static void Main() {
        ushort num;
        ushort i;

        for(i = 1; i <= 10; i++) {
            num = i;

            Console.WriteLine("num: " + num);

            num = (ushort) (num | 1); // num | 0000 0001

            Console.WriteLine("num after turning on bit zero: " +
                num + "\n");
        }
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

num: 1
num after turning on bit zero: 1
num: 2
num after turning on bit zero: 3
num: 3
num after turning on bit zero: 3
num: 4
num after turning on bit zero: 5
num: 5
num after turning on bit zero: 5
num: 6
num after turning on bit zero: 7
num: 7
num after turning on bit zero: 7
num: 8
num after turning on bit zero: 9
num: 9
num after turning on bit zero: 9
num: 10
num after turning on bit zero: 11

```

Program, her karaktere, ikili düzende **0000 0001** olarak simgelenen **1** ile VEYA uygulayarak çalışmaktadır. Böylece **1**, ikili düzende sadece sıfırıncı biti **1** olan bir değer üretir. Bu değere herhangi bir başka değer ile VEYA uygulandığında, alt (low-order) biti **1** olarak ayarlanan, diğer bitleri değişmeden kalan bir sonuç üretir. Böylece, çift sayı olan bir değer bir artırılacak ve tek sayı haline gelecektir.

Genellikle XOR olarak kısaltılan özel VEYA, sadece ve sadece karşılaştırılmakta olan bitler farklısa bir bite **1** değerini verir. Bu durum aşağıda gösterilmiştir:

```

0111 1111
1011 1001
^ -----
1100 0110

```

XOR operatörünün ilginç bir özelliği de bir mesajı şifrelemenin kolay bir yolu olmasıdır. Herhangi bir **x** değerine bir başka **y** değeri ile XOR uygulandığında ve sonra, sonuca tekrar **y** ile XOR uygulandığında **x** elde edilir. Yani, aşağıdaki ifadelerde:

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

**R2**, **x** ile aynı değere sahiptir. Yani, aynı değerleri kullanan peş peşe iki XOR ifadesinin sonucu orijinal değeri üretir. Bu prensibi, basit bir şifreleme programı geliştirmek için kullanabilirsiniz. Bu durumda bir tamsayı, mesajdaki karakterlere XOR uygulayarak hem mesajı şifrelemek, hem de mesajın şifresini çözmek için kullanılan bir anahtardır. Şifrelemek içjn XOR işlemi ilk kez uygulanır ve şifrelenmiş metin elde edilir. Şifreyi çözmek için XOR işlemi ikinci kez uygulanır ve orijinal metin elde edilir. İşte size, kısa bir mesajı şifrelemek ve mesajın şifresini çözmek için bu yöntemi kullanan basit bir örnek:

```
// Bir mesajı şifrelemek ve mesajın şifresini cozmek icin XOR kullanır.
```

```
using System;

class Encode {
    public static void Main() {
        char ch1 = 'H';
        char ch2 = 'i' ;
        char ch3 = '!';

        int key = 88;

        Console.WriteLine("Original message: " + ch1 + ch2 + ch3);

        // mesajı şifrele
        ch1 = (char) (ch1 ^ key);
        ch2 = (char) (ch2 ^ key);
        ch3 = (char) (ch3 ^ key);

        Console.WriteLine("Encoded message: " + ch1 + ch2 + ch3);

        // mesajın şifresini coz
        ch1 = (char) (ch1 ^ key);
        ch2 = (char) (ch2 ^ key);
        ch3 = (char) (ch3 ^ key);

        Console.WriteLine("Decoded message: " + ch1 + ch2 + ch3);
    }
}
```

Çıktı şöyle olur:

```
Original message: Hi!
Encoded message: ly
Decoded message: Hi!
```

Gördüğünüz gibi, aynı anahtarları kullanan iki XOR'un sonucunda şifrelenmiş mesaj elde edilir.

Tekli “bir’in tümleyeni” (DEĞİL) operatörü, operandın bitlerinin tümünün durumlarını tersine çevirir. Örneğin, **A** adındaki bir tamsayı **1001 0110** bitlerine sahipse, **~A** ile **0110 1001** bitleri sonuç olarak elde edilir

Aşağıdaki program, bir sayıyı ve ikili düzende bu sayının tümleyenini göstererek DEĞİL operatörünü tanıtmaktadır:

```
// Bit tabanlı DEĞİL operatorunu tanıtır.

using System;

class NotDemo {
    public static void Main() {
        sbyte b = -34;
        int t;
        for(t = 128; t > 0; t = t/2) {
            if((b & t) != 0) Console.Write("1 ");
            if((b & t) == 0) Console.Write("0 ");
        }
        Console.WriteLine();

        // Tüm bitleri tersine çevir
        b = (sbyte) -b;

        for(t = 128; t > 0; t = t/2) {
            if((b & t) != 0) Console.Write("1 ");
            if((b & t) == 0) Console.Write("0 ");
        }
    }
}
```

İşte programın çıktısı:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

## Kaydırma Operatörleri

C#'ta, bir değeri oluşturan bitleri sağa veya sola belirli bir miktar kaydirmak mümkündür. C#'ta, aşağıda gösterilen iki bit kaydırma operatörü tanımlıdır:

<<	Sola kaydırma
>>	Sağ'a kaydırma

Bu operatörlerin genel formları şu şekildedir:

*değer << bit-sayısı*  
*değer >> bit-sayısı*

Yukarıdaki ifadede **değer**'in bitleri, **bit-sayısı** ile belirtilen sayıda ve kaydırma operatörünün belirttiği yönde kaydırılmaktadır.

Sola kaydırma, belirtilen deyerin içindeki tüm bitlerin bir bit sola kaydırılmasına ve sağ taraftan bir sıfır bitinin eklenmesine neden olur. Sağa kaydırma, tüm bitlerin bir bit sağa kaydırılmasına neden olur. İşaretsiz bir değerin sağa kaydırılması söz konusu olduğunda, soldan bir sıfır biti eklenir. İşaretli bir değerin sağa kaydırılması durumunda ise, işaret biti korunur. Hatırlarsanız, negatif sayılar, tamsayı değerinin üst biti **1** yapılarak simgelenir. Yani, kaydırılmakta olan değer negatifse, her sağa kaydırma işlemi soldan bir **1** biti ekler. Eğer değer pozitifse, her sağa kaydırma işlemi soldan bir **0** ekler.

Sağ ve sola kaydırma işlemlerinin her ikisinde de, kaydırılıp çıkarılan bitler yitirilir. Böylece, kaydırırmak döndürmek (*rotate*) demek değildir; kaydırılıp çıkarılan biti geri kazanmanın hiç bir yolu yoktur.

Şimdi, sola ve sağa kaydırma işlemlerinin etkisini grafiksel olarak göz önünde canlandıran bir programı inceleyelim. Öncelikle, bir tamsayıya ilk değer olarak **1** verilmektedir; yani bu, sayının alt biti **1** olarak ayarlanıyor demektir. Sonra, tamsayı üzerinde sekiz adet kaydırma işlemi gerçekleştirilmektedir. Her kaydırımdan sonra, sayının alt sekiz biti gösterilmektedir. Daha sonra, sekizinci bit konumuna yerleştirilen bir **1** biti haricinde, bu işlemler tekrarlanır ve sağa kaydırırmalar gerçekleştirilir.

```
// << ve >> kaydırma operatorlerini tanıtır.

using System;

class ShiftDemo {
    public static void Main() {
        int val = 1;
        int t;
        int i;
        for(i = 0; i < 8; i++) {
            for(t = 128; t > 0; t = t/2) {
                if((val & t) != 0) Console.Write("1 ");
                if((val & t) == 0) Console.Write("0 ");
            }
            Console.WriteLine();
            val = val << 1; // sola kaydır
        }
        Console.WriteLine();

        val = 128;
        for(i = 0; i < 8; i++) {
            for(t = 128; t > 0; t = t/2) {
                if((val & t) != 0) Console.Write("1 ");
                if((val & t) == 0) Console.Write("0 ");
            }
            Console.WriteLine();
            val = val >> 1; // sağa kaydır
        }
    }
}
```

```

        }
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

```

İkili düzen, **2**'nin kuvvetlerine dayandığı için kaydırma operatörleri, bir tamsayıyı çok hızlı bir biçimde **2**'ye bölmek veya **2** ile çarpmak amacıyla kullanılabilecek bir yöntemdir. Sola kaydırma, sayıyı ikiye çarpar. Sağa kaydırma, sayıyı ikiye böler. Elbette bu işlem, bitleri bir uçtan veya diğerinden dışarı kaydırıp, yitirmediginiz müddetçe işe yarar. İşte bir örnek:

```
// 2 ile çarpmak ve 2'ye bolmek icin kaydırma operatorlerini kullanır.
```

```

using System;

class MultDiv {
    public static void Main() {
        int n;

        n = 10;

        Console.WriteLine("Value of n: " + n);

        // 2 ile carp
        n = n << 1;
        Console.WriteLine("Value of n after n = n * 2: " + n);

        // 4 ile carp
        n = n << 2;
        Console.WriteLine("Value of n after n = n * 4: " + n);

        // 2'ye bol
        n = n >> 1;
        Console.WriteLine("Value of n after n = n / 2: " + n);

        // 4'e bol
        n = n >> 2;
    }
}

```

```

Console.WriteLine("Value of n after n = n / 4: " + n);
Console.WriteLine();

// n'e yeni değer ata
n = 10;
Console.WriteLine("Value of n: " + n);

// 30 kez 2 ile çarp
n = n << 30; // data is lost
Console.WriteLine("Value of n after left-shifting 30
                  places: " + n);
}
}

```

Çıktı aşağıda gösterilmiştir;

```

Value of n: 10
Value of n after n = n * 2: 20
Value of n after n = n * 4: 80
Value of n after n = n / 2: 40
Value of n after n = n / 4: 10

Value of n: 10
Value of n after left-shifting 30 places: -2147483648

```

Çıktının son satırına dikkat edin. **10** sayısı **30** kez sola kaydırıldığında (yani, sayı  $2^{30}$  ile çarpıldığında), bilgi kaybı olur; çünkü, bitler **int**'in değer aralığından dışarıya kaydırılmıştır. Bu durumda elde edilen anlamsız (garbage) değer negatiftir, çünkü üst bite **1** kaydırılmıştır. Bu bit işaret biti olarak kullanıldığı için, sayının negatif olarak yorumlanması neden olur. Bu, kaydırma operatörlerini **2** ile çarpmaya veya **2**'ye bölmeye işlemlerde kullanırken neden dikkatli olmanız gerektiğini göz önüne seriyor. (İşaretli ve işaretsiz veri tipleriyle ilgili açıklamalar için Bölüm 3'e bakın.)

## Bit Tabanlı Bileşik Atamalar

Bit tabanlı ikili operatörlerin tümü bileşik atama ifadelerinde kullanılabilir. Örneğin, aşağıdaki iki ifadenin her biri **x**'e, kendisinin **127** ile XOR uygulanmasından elde edilen değeri atamaktadır:

```

x = x ^ 127;
x ^= 127;

```

## ? Operatörü

C#'ın en etkileyici operatörlerinden biri **?** operatöridür. **?** operatörü genellikle belirli tipteki **if-then-else** ifadelerinin yerine kullanılır. **?** operatörüne *üçlü* (ternary) *operator* denir, çünkü **?** operatörü üç adet operand gerektirir. **?** operatörünün genel formu şu şekildedir:

*Deyim1* **?** *Deyim2* : *Deyim3*;

Burada **Deyim1** bir **bool** deyimdir, **Deyim2** ve **Deyim3** de birer deyimdir. **Deyim2** ve **Deyim3**'ün tipleri aynı olmalıdır. İki nokta işaretinin kullanımına ve konumuna dikkat edin.

Bir **?** deyiminin değeri şu şekilde belirlenir: **Deyim1**'in değeri hesaplanır. Eğer bu değer doğru ise, **Deyim2** hesaplanır ve bütün **?** deyiminin değeri halini alır. Eğer **Deyim1** yanlış ise, **Deyim3** hesaplanır ve bunun değeri bütün deyimin değeri olur. Aşağıdaki örneği ele alın. Burada **absval**'e **val**'ın mutlak değeri atanmaktadır.

```
absval = val < 0 ? -val : val; // val'in mutlak değerini bul
```

Bu örnekte, eğer **val** sıfır veya sıfırdan büyük ise **val**'in değeri **absval**'e atanacaktır. Eğer **val** negatif ise, **absval**'e bu değerin negatif (ki, bu pozitif bir değer üretir) atanacaktır.

Aşağıda, **?** operatörüne bir başka örnek daha görülüyor. Bu program iki sayıyı böler, ama sıfıra bölmeye izin vermez.

```
// ? kullanarak sıfıra bolmeyi onler.

using System;

class NoZeroDiv {
    public static void Main() {
        int result;
        int i;
        for(i = -5; i < 6; i++) {
            result = i != 0 ? 100 / i : 0;
            if(i != 0)
                Console.WriteLine("100 /" + i + " is " + result);
        }
    }
}
```

Bu programın çıktısı aşağıdaki gibidir;

```
100 / -5 is -20
100 / -4 is -25
100 / -3 is -33
100 / -2 is -50
100 / -1 is -100
100 / 1 is 100
100 / 2 is 50
100 / 3 is 33
100 / 4 is 25
100 / 5 is 20
```

Programdaki şu satıra dikkat edin:

```
result = i != 0 ? 100 / i : 0;
```

Burada, **result**'a **100**'ün **i** ile bölünmesiyle elde edilen sonuç atanmaktadır. Ancak, bu bölüm işlemi yalnızca **i** sıfırdan farklı olduğunda gerçekleşmektedir. **i** sıfır olduğunda, **result**'a bir yer tutucu değer olarak sıfır değeri atanır.

Aslında, ? ile elde edilen değeri herhangi bir değişkene atamak zorunda değilsiniz. Örneğin, bu değeri, bir metoda yapılan çağrıya argüman olarak kullanabilirsiniz. Ya da, eğer deyimlerin tümü **bool** tipinde ise ?, bir döngü veya **if** ifadesinde koşul deyimi olarak kullanılabilir. Söz gelisi, gelin şimdi, önceki programın biraz daha verimli biçimde yeniden yazılmış versiyonuna göz atalım. Bu program önceki ile aynı çıktıyı üretmektedir.

```
// Prevent a division by zero using the ?.

using System;

class NoZeroDiv2 {
    public static void Main() {
        int i;

        for(i = -5; i < 6; i++)
            if(i != 0 ? true : false)
                Console.WriteLine("100 / " + i + " is " + 100 / i);
    }
}
```

**if** ifadesine dikkat edin, **i** sıfır ise, **if**'in sonucu yanlıştır, sıfıra bölme işlemi önlenir ve hiçbir sonuç ekranda gösterilmez. Aksi halde, bölme işlemi gerçekleştirilir.

## Boşluklar ve Parantezler

C#ta bir deyim, sekülerler ve boşluk karakterleri içерerek daha okunaklı hale getirilebilir, Örneğin, aşağıdaki iki deyim aynıdır, fakat ikincisini okumak daha kolaydır:

```
x=10/y*(127/x);
x = 10 / y * (127/*);
```

Parantezler, tipki cebirdeki gibi, içerdikleri işlemlerin öncelliğini artırırlar. Gereksiz ve üave parantez kullanımı hataya neden olmaz ya da söz konusu deyimin çalışmasını yavaşılatmaz. Hesaplananın doğru sırasını netleştirmek amacıyla hem kendiniz, hem de programınızı daha sonra anlamak zorunda kalabilecek diğer kişiler için, parantez kullanmanız tavsiye edilir. Örneğin, aşağıdaki iki deyimden hangisini okumak daha kolaydır?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) * 127;
```

## Operatörlerin Öncelik Sırası

Tablo 4.2, tüm C# operatörleri için en yüksekten en düşüğe doğru öncelik sırasını göstermektedir. Bu tablo, bu kitapta daha sonra ele alınacak olan birkaç operatörü de içermektedir.

## TABLO 4.2: Operatörlerin Öncelik Sırası

**En yüksek**

( )	[ ]	.	++ (sonek)	-- (sonek)	checked	new	sizeof	typeof	unchecked
!	~	(tip ataması)	+(tekli)	-(tekli)	++ (önek)	-- (önek)			
*	/	%							
+	-								
<<	>>								
<	>	<=	>=	is					
==	!=								
&									
^									
&&									
? :									
=	op=								

**En düşük**

B E Ş İ N C İ   B Ö L Ü M

---

# **PROGRAM KONTROL İFADELERİ**

Bu bölümde, C#'ın program kontrol ifadeleri ele alınıyor. Program kontrol ifadeleri üç kategoride toplanır: *seçme* ifadeleri, **if** ve **switch** ifadeleridir; *iterasyon* ifadeleri, **for**, **while**, **do-while** ve **foreach** döngülerinden oluşur; *atlama* ifadeleri, **break**, **continue**, **goto**, **return** ve **throw**'u içerir. C#'ın kural dışı durum yürütme mekanizmasının bir parçası olan ve Bölüm 13'te ele alınan **throw**'un haricindekiler burada incelenmektedir.

## if İfadesi

Bölüm 2'de **if** ifadesi tanıtılmıştı. Burada ayrıntılı olarak incelenmektedir. **if** ifadesinin tamamlanmış şekli şöyledir:

```
if (koşul) ifade;
else ifade;
```

Burada **if** ve **else**'in hedefleri tek bir ifadedir, **else** kısmı isteğe bağlıdır. **if** ve **else**'in her ikisinin de hedefleri ifade blokları olabilir. İfade bloklarının kullanıldığı **if** ifadesinin genel formu şu şekildedir:

```
if (koşul)
{
    ifade sekansi
}
else
{
    ifade sekansi
}
```

Koşul deyimi doğru ise, **if**'in hedefi gerçekleşir; aksi halde, **if**'den çıkışlrsa **else**'in hedefi gerçekleşir. Hiçbir zaman her ikisi aynı anda gerçekleşmez. **if**'i kontrol eden koşul deyimi bir **bool** sonuç üretmelidir.

İşte, bir sayının pozitif veya negatif olduğunu bildirmek amacıyla **if** ve **else** ifadesi kullanan basit bir örnek:

```
// Bir degerin pozitif veya negatif olup olmadigini belirler.

using System;

class PosNeg {
    public static void Main() {
        int i;

        for(i = -5; i <= 5; i++) {
            Console.Write("Testing " + i + ": ");

            if(i < 0) Console.WriteLine("negative");
            else Console.WriteLine("positive");
        }
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Testing -5: negative
Testing -4: negative
Testing -3: negative
Testing -2: negative
Testing -1: negative
Testing 0: positive
Testing 1: positive
Testing 2: positive
Testing 3: positive
Testing 4: positive
Testing 5: positive
```

Bu örnekte, **i** sıfırdan küçükse **if**'in hedefi gerçekleşir. Aksi halde, **else**'in hedefi gerçekleşir. Her ikisinin aynı anda gerçekleşmesi asla söz konusu olamaz.

## Kümelenmiş if'ler

*Kümelenmiş if* (nested **if**), bir başka **if** veya **else**'in hedefi olan bir **if** ifadesidir. Kümelenmiş **if**'ler, programlamada çok yaygın olarak kullanılır. C#'taki kümelenmiş **if**'lerle ilgili hatırlamanız gereken başlıca konu, bir **else** ifadesinin her zaman kendisiyle aynı blokta yer alan ve bir **else** ile ilişkilendirilmemiş en yakın **if** ifadesine ait olduğunu. İşte bir örnek:

```
if (i == 10) {
    if (j < 20) a = b;
    if (k > 100) c = d;
    else a = c; // bu else, if (k > 100)'a aittir
}
else a = d; // bu else, if (i == 10)'a aittir
```

Açıklamaların da gösterdiği gibi, en son **else**, **if(j<20)** ile ilişkili değildir (**else**'i olmayan en yakın **if** bu olmasına rağmen), çünkü aynı blokta değildir. Daha doğrusu, en son **else**, **if(i==10)** ile ilişkilidir. İçteki **else** ise **if(k>100)**'e aittir, çünkü aynı blok içindeki en yakın **if** budur.

Aşağıdaki programda bir iç içe **if** ifadesi gösteriliyor. Önceki pozitif/negatif programında sıfır, pozitif olarak belirtilmektedir. Ancak, bazı uygulamalarda sıfır, işaretsiz olarak alınır. Söz konusu programın aşağıda yer alan versiyonunda sıfır ne pozitif ne de negatif olarak ele alınmaktadır.

```
/* Bir degerin pozitif, negatif veya sifir olup olmadigini
belirler. */

using System;

class PosNegZero {
    public static void Main() {
        int i;

        for(i = -5; i <= 5; i++) {
```

```

        Console.WriteLine("Testing " + i + ": ");
        if(i < 0) Console.WriteLine("negative");
        else if(i == 0) Console.WriteLine("no sign");
        else Console.WriteLine("positive");
    }
}
}

```

İşte çıktı:

```

Testing -5: negative
Testing -4: negative
Testing -3: negative
Testing -2: negative
Testing -1: negative
Testing 0; no sign
Testing 1: positive
Testing 2; positive
Testing 3: positive
Testing 4; positive
Testing 5: positive

```

## if-else-if Merdiveni

Kümelenmiş **if**'e dayanan çok yaygın bir programlama özelliği **if-else-if** merdivenidir. Bu, şu şekilde görünür;

```

if (koşul)
    ifade;
else if (koşul)
    ifade;
else if (koşul)
    ifade;
.
.
.
else
    ifade;

```

Koşul deyimleri yukarıdan aşağıya doğru gerçekleşir. Doğru bir koşul ile karşılaşır karşılaşmaz, bu koşulla ilişkili ifade gerçekleşir ve merdivenin geri kalan kısmı dikkate alınmaz. Koşulların hiçbirini doğru değilse, bu durumda en son else ifadesi gerçekleşecektir. En son **else** genellikle, varsayılan koşul olarak davranışır. Yani, diğer koşul testlerinin tümü başarısız olursa, en son **else** ifadesi gerçekleşir. Eğer en son **else** ifadesi mevcut değilse ve diğer koşulların tümü yanlış ise, bu durumda hiçbir işlem gerçekleştirilmmez.

Aşağıdaki program **if-else-if** merdivenini gösteriyor. Program, belirli bir değer için (1'den farklı) en küçük tek basamaklı çarpanı buluyor.

```

// En küçük tek basamaklı carpanı bulur.

using System;

```

```

class Ladder {
    public static void Main() {
        int num;

        for(num = 2; num < 12; num++) {
            if((num % 2) == 0)
                Console.WriteLine("Smallest factor of " + num +
                    " is 2.");
            else if((num % 3) == 0)
                Console.WriteLine("Smallest factor of " + num +
                    " is 3.");
            else if((num % 5) == 0)
                Console.WriteLine("Smallest factor of " + num +
                    " is 5.");
            else if((num % 7) == 0)
                Console.WriteLine("Smallest factor of " + num +
                    " is 7.");
            else
                Console.WriteLine("num + " is not divisible by
                    2, 3, 5, or 7.");
        }
    }
}

```

Program, aşağıdaki çıktıyı üretmektedir:

```

Smallest factor of 2 is 2.
Smallest factor of 3 is 3.
Smallest factor of 4 is 2.
Smallest factor of 5 is 5.
Smallest factor of 6 is 2.
Smallest factor of 7 is 7.
Smallest factor of 8 is 2.
Smallest factor of 9 is 3.
Smallest factor of 10 is 2.
11 is not divisible by 2, 3, 5, or 7.

```

Gördüğünüz gibi, varsayılan **else**, yalnızca önceki **if** ifadelerinden hiçbirini başarılı olmadığı takdirde gerçekleşir.

## switch İfadesi

C#'ın seçme ifadelerinden ikincisi **switch**'dır. **switch**, çok yolu dallanma sağlar. Böylece **switch**, programın birkaç alternatif arasından seçim yapmasını mümkün kılar. Bir dizi kümelenmiş **if** ifadeleri de çok yolu testler gerçekleştirebilir olsa da, birçok durum için **switch** çok daha verimli bir yöntemdir. **switch** şu şekilde çalışır: Bir deyimin değeri, sabitlerden oluşan bir listede peş peşe test edilir. Deyimin değeri sabitlerden biriyle eşlenince, bu eşleşmeyle ilişkili ifade sekansı gerçekleşir. **switch** ifadesinin genel olarak formu şöyledir:

```

switch(deyim) {
    case sabit1:

```

```

        ifade sekansı
        break;
    case sabit2:
        ifade sekansı
        break;
    case sabit3:
        ifade sekansı
        break;
    .
    .
    .
    default:
        ifade sekansı
        break;
}

```

**switch** deyimi bir tamsayı tipinde, örneğin, **char**, **byte**, **short** veya **int**, ya da **string** tipinde (bu kitapta daha sonra ele alınmaktadır) olmalıdır. Böylece söz geliş, kayan noktalı deyimlere izin verilmemektedir. Sık sık **switch**'i kontrol eden deyim tek bir değişkenden ibarettir. **case** sabitleri deyim ile uyumlu tipte lileraller olmalıdır. Aynı **switch** içinde herhangi iki **case** sabiti aynı değerlere sahip olamaz.

**default** ifade sekansı, **case** sabitlerinden hiçbir deyim ile eşlenmediğinde çalıştırılır. **default** isteğe bağlıdır; **default** mevcut değilse, eşleşmelerin tümü başarısız olduğunda hiçbir işlem yapılmaz. Bir eşleşme bulunduğuanda, bu **case** ile ilişkili ifadeler **break**'e rastlayana kadar çalıştırılır.

Aşağıdaki program **switch**'i göstermektedir:

```

// switch'i tanitir.

using System;

class SwitchDemo {
    public static void Main() {
        int i;

        for(i=0; i<10; i++)
            switch(i) {
                case 0:
                    Console.WriteLine("i is zero");
                    break;
                case 1:
                    Console.WriteLine("i is one");
                    break;
                case 2:
                    Console.WriteLine("i is two");
                    break;
                case 3:
                    Console.WriteLine("i is three");
                    break;
                case 4:
                    Console.WriteLine("i is four");

```

```
        break;
    default:
        Console.WriteLine("i is five or more");
        break;
    }
}
```

Bu programın ürettiği çıktı aşağıdaki gibidir:

i is zero  
i is one  
i is two  
i is three  
i is four  
i is five or more  
i is five or more  
i is five or more  
i is five or more  
i is five or more

Gördüğünüz gibi, döngünün her tekrarında `i` ile eşleşen `case` sabiti ile ilişkili ifadeler gerçeklenir. Diğerlerinin hiçbirini dikkate alınmaz. `i`, beş veya daha büyük olursa `case` sabitlerinden hiçbirini eşlenmez; bu nedenle, `default` ifadesi gerçekleşir.

Önceki örnekte **switch**, bir **int** değişken ile kontrol ediliyordu. Önceden açıklandığı gibi, **switch**'i **char** da dahil olmak üzere herhangi bir tamsayı tipi ile kontrol edebilirsiniz. İşte, **char** deyīimi ve **char case** sabitleri kullanan bir örnek:

```
// switch'i kontrol etmek icin bir char kullanir.

using System;

class SwitchDemo2 {
    public static void Main() {
        char ch;

        for(ch = 'A'; ch <= 'E'; ch++)
            switch(ch) {
                case 'A':
                    Console.WriteLine("ch is A");
                    break;
                case 'B':
                    Console.WriteLine("ch is B");
                    break;
                case 'C':
                    Console.WriteLine("ch is C");
                    break;
                case 'D':
                    Console.WriteLine("ch is D");
                    break;
                case 'E':
                    Console.WriteLine("ch is E");
                    break;
            }
    }
}
```

```

        }
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

ch is A
ch is B
ch is C
ch is D
ch is E

```

Dikkat ederseniz, bu örnekte **default** ifadesi yer almıyor. **default**'un istege bağlı olduğunu hatırlınızdan çıkarmayın, Gerekmiyorsa, **default** koda dahil edilmeyebilir.

C#'ta bir **case** ile ilişkili ifade sekansının bir sonraki **case** içinde devam etmesi hatalıdır. Buna “bir sonraki seçeneğe kaymama” (no fall-through) kuralı denir. **case** ifadelerinin **break** ifadesi ile sona ermesinin nedeni budur. (Bir sonraki seçeneğe kayma durumunu diğer yöntemlerle de, örneğin bu bölümde daha sonra ele alınacak bir konu olan **goto**'larla da önleyebilirsiniz; ancak, **break** açık farkla en yaygın olarak kullanılan bir yöntemdir.) Bir **case**'in ifade sekansı içinde **break**'le karşılaşınca **break** ifadesi, program akışının **switch** ifadesinin bütünü terk etmesine ve **switch**'in dışında kalan bir sonraki ifadeden devam etmesine neden olur. **default** ifadesi de ayrıca bir sonraki seçeneğe kaymamalı ve genellikle **break** ile sona ermeliidir.

“Bir sonraki seçeneğe kaymama” kuralı C#'ı; C, C++ ve Java'dan ayıran hususlardan biridir. Bu dillerde bir **case**, peşinden gelen **case** üzerinden devam edebilir (yani, bir sonraki seçeneğe kayabilir – “fall-through”). C#'ta **case**'ler için “bir sonraki seçeneğe kaymama” kuralı geliştirilmesinin iki nedeni vardır. Öncelikle, bu sayede derleyicinin **case** ifadelerini, belki de optimizasyon amaçlı olarak, serbestçe sıralamasına imkan verilir. Eğer bir **case**, kendisinden sonra gelen **case** içinde devam ediyor olsaydı, bu tür bir yeniden düzenleme mümkün olmayacaktı. İkincisi, her **case**'in açıkça sona ermesini gerekli kılmak programcıların kazara bir **case**'in kendisinden sonra gelen **case** içinde devam etmesine imkan vermelerine de engel olur.

Bir **case** sekansının bir diğeri içinde devam etmesine imkan veremiyor olmanızı rağmen, aşağıdaki örnekte gösterildiği gibi iki veya daha fazla **case** ifadesinin aynı kod sekansını içermesini sağlayabiliyorsunuz:

```

// Bos case'ler bir sonraki case icinde devam edebilir.

using System;

class EmptyCasesCanFall {
    public static void Main() {
        int i;

        for(i = 1; i < 5; i++)
            switch(i) {

```

```

        case 1:
        case 2:
        case 3: Console.WriteLine("i is 1, 2 or 3");
            break;
        case 4: Console.WriteLine("i is 4");
            break;
    }
}
}

```

Cıktı aşağıda gösterilmiştir:

```

i is 1, 2 or 3
i is 1, 2 or 3
i is 1, 2 or 3
i is 4

```

Bu örnekte, eğer **i** 1, 2 veya 3 değerine sahipse, ilk **WriteLine()** ifadesi çalıştırılır. **i**, 4 ise ikinci **WriteLine()** ifadesi çalıştırılır, **case**'lerin üst üste yiğilması “bir sonraki seçenekle kaymama” kuralını çiğnemez, çünkü **case** ifadelerinin tümü aynı ifade sekansını kullanmaktadır.

Birkaç **case** ifadesinin ortak bir kodu paylaştığı durumda **case** ifadelerini üst üste yiğmek sıkça kullanılan bir tekniktir. Bu teknik, kod sekansının gereksiz yere tekrarlanmasılığını önler.

## Kümelenmiş switch İfadeleri

Bir **switch**'i dıştaki bir **switch**'in ifade sekansının bir parçası olarak kullanmak mümkündür. Buna *kümelenmiş switch* (nested **switch**) denir. İçteki ve dıştaki **switch**'lerin **case** sabitleri ortak değerler içerebilir; bu durumda hiçbir karışıklık ortaya çıkmayacaktır. Örneğin, aşağıdaki kod parçaları tamamen kabul edilebilir:

```

switch(ch1) {
    case 'A': Console.WriteLine("This A is part of outer switch");
        switch(ch2) {
            case 'A':
                Console.WriteLine("This A is part of inner switch");
                break;
            case 'B': // ...
        } // içteki switch'in sonu
        break;
    case 'B': // ...
}

```

## for Döngüsü

Bölüm 2'den beri **for** döngüsünün basit bir formunu kullanmaktanız. Bu bölümde **for** döngüsü ayrıntılı alarak incelenmektedir. **for** döngüsünün ne kadar güçlü ve esnek olduğunu şaşıracaksınız. Gelin şimdi, **for**'un en alışıldık formlarından başlayarak, temel kavramları tekrarlayarak başlayalım.

Tek bir ifadeyi tekrarlamak için kullanılan **for** döngüsünün genel formu su şekilde dir:

```
for(başlangıç; koşul; iterasyon) ifade;
```

Tekrarlanan bir blok için genel form şu şekildedir:

```
for(başlangıç; koşul; iterasyon)
{
    ifade sekansı
}
```

**başlangıç** genellikle, *döngü kontrol değişkenine* ilk değerini veren bir atama ifadesidir. Döngü kontrol değişkeni, döngüyü kontrol eden bir sayaç gibi davranışır. **koşul**, döngünün tekrarlanıp tekrarlanmayacağını belirleyen **bool** tipinde bir deyimdir. **iterasyon** deyimi, döngünün her tekrarlanışında döngü kontrol değişkeninin ne ölçüde değiştirileceğini tanımlar. Döngünün bu üç ana bölümünün noktalı virgül ile birbirinden ayrıldığına dikkat edin. **for** döngüsü, koşul testi başarılı olduğu sürece çalışmaya devam edecektir. Koşul yanlış olduğu andan itibaren döngüden çıkışacaktır ve program çalışmasına **for**'u takip eden ifadeden başlayarak devam edecektir,

**for** döngüsü pozitif veya negatif biçimde ilerleyebilir; döngü, döngü kontrol değişkenini herhangi bir ölçüde değiştirebilir. Örneğin, aşağıdaki program 100 ile -100 arasındaki sayıları 5'er 5'er azaltarak yazdırır:

```
// Negatif olarak tekrarlayan dongu.

using System;

class DecrFor {
    public static void Main() {
        int x;

        for(x = 100; x > -100; x -= 5)
            Console.WriteLine(x);
    }
}
```

**for** döngüleriyle ilgili önemli bir husus, koşul deyiminin her zaman döngünün en başında test edildiğiidir. Bu, döngüye başlarken deyimin değeri yanlış ise döngünün içindeki kod hiç çalıştırılmayabilir, demektir. İşte bir örnek:

```
for(count = 10; count < 5; count++)
    x += count; // bu ifade calistirilmayacaktir
```

Bu döngü asla çalışmazdı, çünkü döngüye başlarken döngü kontrol değişkeni olan **count**, 5'ten büyuktur. Bu, koşul deyiminin (**count < 5**) başlangıçta yanlış olmasına neden olur; böylece, döngünün bir kez bile iterasyonu söz konusu olmaz.

**for** döngüsü en çok, iterasyonların bilinen sayıda yapılacağı durumlarda kullanışlıdır. Örneğin, aşağıdaki program 2 ile 20 arasındaki asal sayıları bulmak için iki **for** döngüsü kullanmaktadır. Eğer sayı asal değilse, sayının en büyük çarpanı ekranda gösterilmektedir:

```
/*
    Bir sayinin asal olup olmadigini belirler.
    Sayi asal degilse, sayinin en buyuk carpanini gosterir.
*/
using System;

class FindPrimes {
    public static void Main() {
        int num;
        int i;
        int factor;
        bool isprime;

        for(num = 2; num < 20; num++) {
            isprime = true;
            factor = 0;

            // num tam olarak bolunebilir mi?
            for(i = 2; i <= num/2; i++) {
                if((num % i) == 0) {
                    // num tam olarak bolunebilir - asal degil
                    isprime = false;
                    factor = i;
                }
            }

            if(isprime)
                Console.WriteLine(num + " is prime.");
            else
                Console.WriteLine("Largest factor of " + num + " is "
                                + factor);
        }
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
2 is prime.
3 is prime.
Largest factor of 4 is 2
5 is prime.
Largest factor of 6 is 3
7 is prime.
Largest factor of 8 is 4
Largest factor of 9 is 3
Largest factor of 10 is 5
11 is prime.
Largest factor of 12 is 6
13 is prime.
Largest factor of 14 is 7
Largest factor of 15 is 5
Largest factor of 16 is 8
17 is prime,
Largest factor of 18 is 9
```

19 is prime.

## for Döngüsü Üzerinde Bazı Varyasyonlar

**for**, C# dilindeki en çok yönlü ifadelerden biridir, çünkü çok çeşitli varyasyonlara imkan verir. Bu varyasyonlar burada incelenmektedir.

### Birden Fazla Döngü Kontrol Değişkeni Kullanmak

**for** döngüsü, döngüyü kontrol etmek amacıyla iki veya daha fazla değişken kullanmanıza imkan verir. Birden fazla döngü kontrol değişkeni kullanırken her değişken için başlangıç ve artırma ifadeleri virgül ile birbirinden ayrılr. Örneğin:

```
// for ifadesi içinde virgul kullanır.

using System;

class Comma {
    public static void Main() {
        int i, j;

        for(i = 0; j = 10; i < j; i++, j--)
            Console.WriteLine("i and j: " + i + " " + j);
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
i and j: 0 10
i and j: 1 9
i and j: 2 8
i and j: 3 7
i and j: 4 6
```

Burada virgüler, iki başlangıç ifadesini ve iki iterasyon deyimini ayırmaktadır. Döngü başladığında **i** ve **j**'nin her ikisine de ilk değer atanır. Döngünün her tekrarlanışında **i** bir artırılır, **j** bir azaltılır. Birden fazla döngü kontrol değişkeninden yararlanmak, genellikle kullanışlı bir yöntemdir. Bu sayede belirli algoritmalar kolaylaştırılabilir. Herhangi bir sayıda başlangıç ve iterasyon ifadelerine sahip olabilirsiniz; fakat, pratikte ikiden fazlası **for** döngüsünü hantallaştırır.

İşte, bir **for** ifadesi içinde çok sayıda döngü kontrol değişkeninin pratik kullanımına bir örnek. Bu program, bir sayının (bu örnekte 100'ün) en büyük ve en küçük çarpanını bulmak amacıyla tek bir **for** döngüsü içinde iki döngü kontrol değişkeni kullanmaktadır. Döngüyü sonlandıran koşula özellikle dikkat edin. Bu, her iki döngü değişkenine de dayanmaktadır.

```
/*
Bir sayinin en buyuk ve en kucuk carpanini bulmak icin for
ifadesi icinde virgul kullanir.
*/
```

```

using System;

class Comma {
    public static void Main() {
        int i, j;
        int smallest, largest;
        int num;

        num = 100;

        smallest = largest = 1;

        for(i = 2, j = num/2; (i <= num/2) & (j >= 2); i++, j--) {

            if((smallest == 1) & ((num % i) == 0))
                smallest = i;

            if((largest == 1) & ((num % j) == 0))
                largest = j;

        }

        Console.WriteLine("Largest factor: " + largest);
        Console.WriteLine("Smallest factor: " + smallest);
    }
}

```

Programın çıktısı şöyledir:

```

Largest factor: 50
Smallest factor: 2

```

İki döngü kontrol değişkeninin kullanılması sayesinde tek bir **for** döngüsü, bir sayının hem en küçük hem de en büyük çarpanını bulabilir. En küçük değeri aramak için **i** kontrol değişkeni kullanılmaktadır. **i**'ye öncelikle **2** değeri atanır ve **i**'nin değeri, **num**'ın yarısını aşana kadar **i** artırılır. **j** kontrol değişkeni en büyük çarpanı bulmak için kullanılır. **j**'nin değeri başlangıçta **num**'ın yarısına eşit olacak şekilde ayarlanır ve **2**'den küçük olana kadar azaltılır. Döngü, **i** ve **j**'nin her ikisi de son değerine ulaşana kadar tekrarlamayı sürdürür. Döngü sona erdiğinde, her iki çarpan da bulunmuş olacaktır.

## Koşul Deyimi

Bir **for** döngüsünü kontrol eden koşul deyimi, bir **bool** sonuç üreten herhangi bir geçerli deyim olabilir. Koşul deyimi, döngü kontrol değişkenini içermek zorunda değildir. Örneğin, bir sonraki programda **for** döngüsü, **done** değeri ile kontrol edilmektedir.

```

// Dongu kosulu herhangi bir bool deyim olabilir.

using System;

class forDemo {
    public static void Main() {

```

```

int i, j;
bool done = false;

for(i = 0, j = 100; !done; i++, j--) {
    if(i * i >= j) done = true;
    Console.WriteLine("i, j: " + i + " " + j);
}
}

```

Çıktı aşağıda gösterilmiştir:

```

i, j: 0 100
i, j: 1 99
i, j: 2 98
i, j: 3 97
i, j: 4 96
i, j: 5 95
i, j: 6 94
i, j: 7 93
i, j: 8 92
i, j: 9 91
i, j: 10 90

```

Bu örnekte, **for** döngüsü **bool** değişkeni olan **done** doğru olana kadar tekrarlanır. Bu değişken, döngü içinde **i**'nin karesi **j**'den büyük veya **j**'ye eşit olduğunda doğru değerine ulaşır.

## Eksik Parçalar

Bazı enteresan **for** döngüsü varyasyonları, döngü tanımındaki bazı parçaları boş bırakarak elde edilebilir. C#'ta **for** döngüsünün başlangıç, koşul ve iterasyon parçalarından herhangi birinin veya bu parçaların tümünün boş bırakılması mümkündür. Örneğin, aşağıdaki programı ele alın:

```

// for'un bazi parcalari bos olabilir.

using System;

class Empty {
    public static void Main() {
        int i;

        for(i = 0; i < 10; ) {
            Console.WriteLine("Pass #" + i);
            i++; // dongu kontrol degiskenini artir
        }
    }
}

```

Bu örnekte, **for**'un iterasyon deyimi boş bırakılmıştır. Bunun yerine, döngü kontrol değişkeni olan **i**, döngü gövdesi içinde artırılmaktadır. Bu, döngünün her tekrarlanışında **i**'nin 10'a eşit olup olmadığını görmek için **i** test edilir, fakat bundan başka hiçbir işlem gerçekleştirilmemektedir. Elbette, **i** döngü gövdesi içinde artırıldığı için, döngü normal şekilde çalışmaktadır ve aşağıdaki çıktıyı ekranada göstermektedir:

```
Pass #0
Pass #1
Pass #2
Pass #3
Pass #4
Pass #5
Pass #6
Pass #7
Pass #8
Pass #9
```

Bir sonraki örnekte, başlangıç bölümünü de **for**'dan çıkarılmıştır:

```
// for dongusunden biraz daha parca cikaralim.

using System;

class Empty2 {
    public static void Main() {
        int i;

        i = 0; // baslangici donguden cikart
        for(; i < 10; ) {
            Console.WriteLine("Pass #" + i);
            i++; // dongu kontrol degiskenini artir
        }
    }
}
```

Bu versiyonda **i**'ye döngünün bir parçası olarak değil, döngü başlamadan ilk değer atanıyor. Normalde döngü kontrol değişkenine **for**'un içinde ilk değer vermemeyi tercih edeceksiniz. Başlangıcı döngünün dışına yerleştirmek, genellikle ilk değerin **for** ifadesi içine katılamayacak kadar karmaşık bir işlem neticesinde elde edildiği durumlarda kullanılır.

## Sonsuz Döngü

Koşul deyimi boş bırakılan bir **for** döngüsü kullanarak *sonsuz döngü* (asla sona ermeyen döngü) hazırlayabilirsiniz. Örneğin, aşağıdaki program parçası birçok C# programcısının sonsuz döngü oluşturmak için kullandığı bir yöntemi göstermektedir:

```
for(;;) // Kasitli olarak hazırlanan bir sonsuz dongu
{
    // ...
}
```

Bu döngü sonsuza kadar çalışacaktır. Sonsuz döngü gerektiren bazı programlama görevleri olmasına rağmen, söz geliş, işaretim sistemi komut işlemcileri gibi “sonsuz döngülerin” birçoğu, aslında özel sonlandırma şartına sahip döngülerden ibarettir. Bu bölümün sonlarına doğru bu tür bir döngüyü nasıl durduracağınızı öğreneceksiniz. (İpucu: Bu işlem, **break** ifadesi kullanılarak gerçekleştirilmektedir.)

## Gövdesiz Döngüler

C#'ta **for** döngüsüyle (veya bir başka döngü ile) ilişkili gövde boş olabilir. Bunun nedeni, *null ifadenin* (*null statement*) söz diziimsel açıdan geçerli olmasıdır. Gövdesiz döngüler sık sık işe yarar. Örneğin, aşağıdaki program **1** ile **5** arasındaki sayıların toplamını bulmak için gövdesiz bir döngü kullanmaktadır:

```
// Dongunun govdesi bos olabilir.

using System;

class Empty3 {
    public static void Main() {
        int i;
        int sum = 0;

        // 5'e kadar sayilari topla
        for(i = 1; i <= 5; sum += i++) ;

        Console.WriteLine("Sum is " + sum);
    }
}
```

Programın çıktısı aşağıdaki gibidir;

```
Sum is 15
```

Dikkat ederseniz, toplama işlemi bütünüyle **for** ifadesi içinde ele alınmaktadır ve döngü gövdesine hiç gerek yoktur. İterasyon deyimine özellikle dikkat edin:

```
sum += i++;
```

Bu tür ifadeler gözünüzü korkutmasın. Bunlar, profesyonelce yazılmış C# programlarında sıkça kullanılır ve ifadeyi parçalara ayıırısanız anlaşılması kolaydır. Başka bir deyişle, “**sum'a sum** artı **i**’nin değerini ekle, sonra **i**’yi artır,” demektedir. Yani bu, aşağıdaki ifade sekansı ile aynıdır:

```
sum = sum + i;
i++;
```

## Döngü Kontrol Değişkenlerini for Döngüsü İçinde Deklare etmek

Genellikle bir **for** döngüsünü kontrol eden değişkene yalnızca döngünün amaçları doğrultusunda ihtiyaç vardır, geri kalan bölümlerde bu değişken kullanılmaz. Böyle bir durum

söz konusu olduğunda değişkeni, **for**'un başlangıç bölümü içinde deklare etmek mümkündür. Örneğin, aşağıdaki program 1 ile 5 arasındaki sayıların hem toplamını hem de faktöriyelini hesaplamaktadır. Program, döngü kontrol değişkeni olan **i**'yi **for**'un içinde deklare etmektedir:

```
// Dongu kontrol degiskenini for icinde deklare et.

using System;

class ForVar {
    public static void Main() {
        int sum = 0;
        int fact = 1;

        // 5'e kadar sayilarin faktoriyelini hesapla
        for (int i = 1; i <= 5; i++) {
            sum += i; // i dongu boyunca taniniyor
            fact *= i;
        }

        // fakat i burada taninmiyor.

        Console.WriteLine("Sum is " + sum);
        Console.WriteLine("Factorial is " + fact);
    }
}
```

Bir değişkeni bir **for** döngüsü içinde deklare ettiğiniz zaman hatırlınız tutmanız gereken bir husus vardır: Bu değişkenin kapsamı **for** ifadesi sona erdiğinde sonlanır. (Yani, değişkenin kapsamı **for** döngüsü ile sınırlıdır.) **for** döngüsü dışında, değişkenin varlığı sona erecektir. Bu nedenle, önceki örnekte **i**, **for** döngüsü dışından erişilememektedir. Döngü kontrol değişkenini programın geri kalan bölümlerinde kullanmanız gerekiyorsa, değişkeni **for** döngüsü içinde deklare etmeniz mümkün olmayacaktır.

Diğer konulara geçmeden önce, **for** döngüsü üzerinde kendi örneklerinizle deneyim kazanmak isteyebilirsiniz. Bunun etkileyici bir ifade olduğunu siz de keşfedeceksiniz.

## while Döngüsü

C#'ta mevcut bir başka döngü ise **while**'dır. **while** döngüsü genel olarak şu şekildedir:

```
While (koşul) ifade;
```

Burada **ifade**, tek bir ifade de olabilir, bir ifade bloğu da olabilir; **koşul** ise döngüyü kontra eden koşulu tanımlar ve geçerli herhangi bir Boolean deyim olabilir. İfade, koşul doğru iken gerçekleşir. Koşul yanlış değerini alırsa, program kontrolü hemen döngüyü takip eden satıra geçer.

İşte, bir tamsayının basamak sayısını hesaplamak için **while** döngüsü kullanan basit bir örnek.

```
// Bir tamsayinin basamak uzunlugunu hesaplar.

using System;

class WhileDemo {
    public static void Main() {
        int num;
        int mag;

        num = 435679;
        mag = 0;

        Console.WriteLine("Number: " + num);

        while(num > 0) {
            mag++;
            num = num / 10;
        }

        Console.WriteLine("Magnitude: " + mag);
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Number: 435679
Magnitude: 6
```

**while** döngüsü şu şekilde çalışır: **num**'ın değeri test edilir. Eğer **num**, sıfırdan büyükse **mag** sayacı bir artırılır ve **num**, 10'a bölünür. **num**'ın değer 10'dan büyük olduğu sürece döngü tekrarlamaya devam eder. **num**, sıfır olduğunda döngü sona erer ve **mag**, orijinal değerin basamak sayısını içerir.

**for** döngüsünde olduğu gibi, **while**'da da koşul deyimi döngünün en başında kontrol edilir. Bu, döngü kodunun hiç çalıştırılmama ihtimali olduğu anlamına gelir. Bu sayede, döngüden önce ayrı bir test yapma gerekliliği de ortadan kalkmış olur. Aşağıdaki program, **while** döngüsünün bu özelliğini göz önünde canlandırıyor. Program, 2'nin 0'dan 9'a kadar olan tamsayı kuvvetlerini hesaplamaktadır:

```
// 2'nin tamsayı kuvvetlerini hesaplar.

using System;

class Power {
    public static void Main() {
        int e;
        int result;

        for(int i=0; i < 10; i++) {
            result = 1;
            e = i;

            while(e > 0) {
```

```
        result *= 2;
        e--;
    }
    Console.WriteLine("2 to the " + i + " power is " +
                      result);
}
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

Dikkat ederseniz, **while** döngüsü, yalnızca **e**, 0'dan büyük olduğunda çalışmaktadır. Böylece, **e**, sıfır iken, **for** döngüsünün ilk tekrarında olduğu gibi, **while** döngüsü atlanır.

## do-while Döngüsü

Üçüncü C# döngüsü ise **do-while**'dır. Koşulun döngünün en başında test edildiği **for** ve **while**, döngülerinden farklı olarak **do-while** döngüsü, koşulu döngünün sonunda kontrol eder. Bu, **do-while** döngüsünün her zaman en az bir kez çalıştırılacağı anlamına gelir. **do-while** döngüsünün genel formu şu şekildedir:

```
do {
    ifadeler;
} while(koşul);
```

Sadece tek bir ifade mevcut olduğunda küme parantezleri gerekli olmasa da, **do-while** döngüsünün okunabilirliğini artırmak amacıyla küme parantezleri yoğunlukla kullanılmaktadır; böylece, **while** ile doğabilecek karışıklık da önlenmiş olur. **do-while** döngüsü, koşul ifadesi doğru olduğu sürece tekrarlamayı sürdürür.

Aşağıdaki program, bir tamsayının basamaklarını ters sırada göstermek için **do-while** döngüsü kullanmaktadır:

```
// Bir tamsayının basamaklarını ters sırada gösterir.

using System;

class DoWhileDemo {
    public static void Main() {
        int num;
        int nextdigit;
```

```

        num = 198;

        Console.WriteLine("Number: " + num);

        Console.Write("Number in reverse order: ");

        do {
            nextdigit = num % 10;
            Console.Write(nextdigit);
            num = num / 10;
        } while(num > 0);

        Console.WriteLine();
    }
}

```

Cıktı aşağıda gösterilmiştir:

```

Number: 198
Number in reverse order: 891

```

Döngü şu şekilde çalışmaktadır. Döngünün her iterasyonunda en sağdaki basamak, tamsayının **10** ile bölünmesinden kalan bulunarak elde edilir. Bu basamak daha sonra ekranda gösterilir. Sonra, **num**'ın değeri **10** ile bölünür. Bu bir tamsayı bölmesi olduğu için bölme, en sağdaki basamağın sayıdan çıkarılmasıyla sonuçlanır. Bu işlem **num**, sıfır olana kadar tekrarlanır.

## foreach Döngüsü

**foreach** döngüsü bir *koleksiyonun* elemanları üzerinden tekrarlanır. Koleksiyon, bir grup nesneden oluşur. C#'ta birkaç tip koleksiyon tanımlıdır. Bunlardan biri da dizidir. **foreach** döngüsü, dizilerin ele alındığı Bölüm 7'de incelenmektedir.

## Döngüden Çıkmak İçin break Kullanmak

**break** ifadesini kullanarak bir döngüden anında çıkmaya zorlamak mümkündür. Bu durumda, döngünün gövdesinde geriye kalan her türlü kod ve döngünün koşul testi pas geçilmiş olur. Bir döngünün içinde bir **break** ifadesi ile karşılaşılınca döngü sona erer ve program kontrolü, döngüyü takip eden bir sonraki ifadeden devam eder. İşte basit bir örnek:

```

// Bir donguden cikmak icin break kullanir.

using System;

class BreakDemo {
    public static void Main() {

        // bu donguden cikmak icin break kullan
        for(int i = -10; i <= 10; i++) {
            if(i > 0) break; // i pozitif oldugunda donguye son ver
    }
}

```

```

        Console.Write(i + " ");
    }
    Console.WriteLine("Done");
}
}

```

Bu program aşağıdaki çıktıyi üretir;

```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 Done
```

Gördüğünüz gibi, **for** döngüsü **-10**'dan **10**'a kadar çalışacak şekilde tasarlanmış olsa da **break** ifadesi döngünün daha erken, yani **i** pozitif olduğunda sona ermesine neden olmaktadır.

**break** ifadesi, kasıtlı sonsuz döngüler de dahil olmak üzere C#'ın herhangi bir döngüsü ile kullanılabilir. Örneğin, önceki programın **do-while** döngüsü kullanılarak yeniden kodlanmış şekli şöyledir:

```

// do-while dongusunden cikmak icin break kullan.

using System;

class BreakDemo2 {
    public static void Main() {
        int i;

        i = -10;
        do {
            if(i > 0) break;
            Console.Write(i + " ");
            i++;
        }while(i <= 10);

        Console.WriteLine("Done");
    }
}

```

Aşağıda, **break** ile ilgili daha pratik bir örnek görüyorsunuz. Bu program, bir sayının en küçük çarpanını bulmaktadır:

```

// Bir degerin en kucuk carpanini bulur.

using System;

class FindSmallestFactor {
    public static void Main() {
        int factor = 1;
        int num = 1000;

        for(int i = 2; i < num/2; i++) {
            if((num % i)== 0) {
                factor = i;
                break; // carpani bulunca donguyu durdur
            }
        }
        Console.WriteLine("Smallest factor is " + factor);
    }
}

```

```

        }
    }
}

```

Çıktı aşağıda gösterilmiştir:

```
Smallest factor is 2
```

Çarpan bulunur bulunmaz **break**, **for** döngüsünü durdurmaktadır. Bu durumda **break** kullanmak, bir kez bir çarpan bulunduktan sonra döngünün bir başka değer bulmaya çalışmasını önlemektedir. Böylece, verimsizlik de önlenmiş olur.

**break**, bir takım kümelenmiş döngülerin içinde kullanıldığında, sadece en içteki döngüden çıkış sağlar. Örneğin,

```
// Kumelenmis dongulerde break kullanmak.

using System;

class BreakNested {
    public static void Main() {

        for(int i = 0; i < 3; i++) {
            Console.WriteLine("Outer loop count: " + i);
            Console.Write("      Inner loop count: ");

            int t = 0;

            while(t < 100) {
                if(t == 10) break; // t, 10 olunca donguyu durdur
                Console.Write(t + " ");
                t++;
            }
            Console.WriteLine();
        }
        Console.WriteLine("Loops complete.");
    }
}
```

Bu program aşağıdaki çıktıyı üretir:

```
Outer loop count: 0
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

Gördüğünüz gibi, içteki döngüdeki **break** ifadesi yalnızca o döngünün sona ermesine neden olur. Dıştaki döngü bu durumdan etkilenmez.

**break** hakkında hatırlınızda tutmanız gereken diğer iki husus ise şöyledir: Öncelikle, bir döngü içinde birden fazla **break** ifadesi yer alabilir, fakat dikkatli olmalısınız. Aşırı sayıda **break** ifadesi kodunuzun yapısını bozma eğilimindedir. İkincisi, bir **switch** ifadesini

sonlandıran **break** sadece **switch** ifadesini etkiler; **break**'i içeren döngüler, söz konusu **break**'den etkilenmez.

## continue İfadesini Kullanmak

Bir döngünün normal kontrol yapısını pas geçerek, döngüyü erken iterasyona zorlamak mümkündür. Bu, **continue** kullanılarak gerçekleştirilir. **continue** ifadesi, döngüyü bir sonraki tekrarını gerçekleştirmeye zorlar, arada yer alan kod dikkate alınmaz. Yani **continue**, esasen **break**'in tamamlayıcısıdır. Örneğin aşağıdaki program, 0 ile 100 arasındaki çift sayıları yazdırmayı kolaylaştırmak için **continue** ifadesini kullanmaktadır.



kuralı çiğnenmiş olmaz; **break** ifadesine de gerek yoktur. Önceden açıklandığı gibi, bir **switch**'in içine atlamak için **goto** kullanmak mümkün değildir. Aşağıdaki satırın başındaki açıklama simgelerini kaldırırsanız program derlenmeyecektir:

```
// goto casel; // Hata! switch'in icine atlayamazsin.
```

Açıkçası, **goto**'yu **switch** ile birlikte kullanmak bazı özel durumlarda kullanışlı olabilir, ama genel olarak önerilen bir stil değildir.

**goto**'nın düzgün kullanımlarından biri, derin biçimde kümelenmiş bir rutinden çıkmak içindir. İşte basit bir örnek:

```
// goto'yu tanitir.

using System;

class Use_goto {
    public static void Main() {
        int i = 0, j = 0, k = 0;

        for(i = 0; i < 10; i++) {
            for(j = 0; j < 10; j++) {
                for(k = 0; k < 10; k++) {
                    Console.WriteLine("i, j, k: " + i + " " + j + " "
                        + k);
                    if(k == 3) goto stop;
                }
            }
        }

stop:
    Console.WriteLine("Stopped! i, j, k: " + i + ", " + j + " "
        + k);
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
i, j, k: 0 0 0  
i, j, k: 0 0 1  
i, j, k: 0 0 2  
i, j, k: 0 0 3  
Stopped! i, j, k: 0, 0 3
```

**goto**'yu kaldırmak, üç adet **if** ve **break** ifadesi kullanmayı mecbur kılar. Bu örnekte **goto**, kodu basitleştirmektedir. Bu, **goto** kullanımını göstermek amacıyla kullanılmak üzere uydurulmuş bir örnek olmakla birlikte, **goto**'nun avantajlı olabileceği durumları muhtemelen siz kendiniz de hayal edebilirsiniz.

ALTINCI BÖLÜM

---

# **SINIFLARA, NESNELERE VE METOTLARA GİRİŞ**

Bu bölümde sınıflar tanıtılmaktadır. Sınıf, C#'ın özüdür, çünkü sınıf, nesnenin doğasını tanımlar. Sınıf, bütün bir C# dilinin ürerine inşa edildiği bir temeldir. Öyle ki, sınıf, C#'ta nesne yönelimli programlamanın esaslarını oluşturur. Sınıf içinde veriler ve bu veriler üzerinde işlem yapan kod tanımlanır. Kod, metodlar içinde yer alır. Sınıf, nesne ve metodlar C#'ın belli başlı özelliklerini olduğu için, bunlar bu bölümde tanıtılmaktadır. Bu özellikleri esaslı kavramış olmak daha sofistike programlar yazmanıza ve sonraki bölümlerde ele alınacak olan büyük öneme sahip belirli C# öğelerini de daha iyi anlamana imkan verecektir.

## Sınıfların Temel Özellikleri

Tüm C# faaliyetleri bir sınıf içinde meydana geldiği için, aslında bu kitabın başından beri sınıfları kullanmaktasınız. Elbette, şimdiden kadar yalnızca son derece basit sınıflar kullandınız; sınıfların sahip olduğu özelliklerinin büyük çoğunluğundan yararlanmış değilsiniz. Sınıfların şu ana dek tanıtılan kısıtlı özelliğe sahip olanlardan çok daha güçlü olduklarına daha sonra şahit olacaksınız.

Gelin, temel özellikleri tekrar ederek başlayalım. Sınıf, bir nesnenin şeklini tanımlayan bir şablondur. Sınıf, hem verileri hem de bu veriler üzerinde işlem yapacak kodu belirtir. C#'ta *nesneleri* tanımlamak için bir sınıf spesifikasyonu kullanılmaktadır. Nesneler bir sınıfın örnekleridir. Yani, sınıf esasen nesnelerin ne şekilde inşa edileceğini belirten birtakım planlardan ibarettir. Bu noktada bir hususa açıklık kazandırmamız gerekiyor: Sınıf, mantıksal bir soyutlamadır. Bir sınıfa ait bir nesne oluşturulana kadar söz konusu sınıf bellekte fiziksel olarak temsil edilmez.

Bir diğer husus: Bir sınıfı oluşturan metod ve değişkenlerin söz konusu sınıfın *üyeleri* oldukları hatırlınızdan çıkarmayın.

## Sınıfin Genel Biçimi

Bir sınıf tanımladığınızda bu sınıfın içeriği verileri ve bu verilerin üzerinde işlem yapacak kodu da deklare etmiş olursunuz. Çok basit sınıfların yalnızca veri ya da yalnızca kod içermeleri mümkün olsa da, gerçek dünyaya ait sınıfların pek çoğu her ikisini de içerir.

Genel olarak, veriler sınıf tarafından tanımlanmış örnek değişkenlerde, kod ise metodlarda saklanır. Ancak anlatıma başlarken, C#'ta birkaç spesifik veri ve kod üyesinin tanımlı olduğunu açıklamak önemlidir. Örnek değişkenler, statik değişkenler, sabitler, metodlar, yapılandırıcılar, yok ediciler, indeksleyiciler, olaylar, operatörler ve özellikler bu kapsamda yer alır. Simdilik, sınıf ile ilgili anlatımımızı sınıfın başlıca öğeleriyle sınırlı tutacağız: örnek değişkenler ve metodlar. Bu bölümün ilerideki sayfalarında yapılandırıcı ve yok ediciler ele alınacaktır. Diğer üye tipleri daha sonraki bölümlerde anlatılacaktır.

Sınıf **class** anahtar kelimesi kullanılarak oluşturulur. Tek bir örnek değişken ve metod içeren bir **class** tanımının genel biçimi aşağıdaki gibidir:

```

class sınıfismi {
    // örnek değişkenleri deklare et

    erişim tip değişken1;
    erişim tip değişken2;
    // ...
    erişim tip değişkenN;

    // metotları deklare et

    erişim dönüş-tipi metot1(parametreler) {
        // metodun gövdesi
    }
    erişim dönüş-tipi metot2(parametreler) {
        // metodun gövdesi
    }
    // ...
    erişim dönüş-tipi metotN(parametreler) {
        // metodun gövdesi
    }
}

```

Her değişken ve metodun öncesinde **erişim** sözcüğünün yer aldığına dikkat edin. Burada **erişim**, söz konusu üyenin ne şekilde erişileceğini belirten bir erişim belirleyicisidir; örneğin, **public** gibi. Bölüm 2'de bahsedildiği gibi, sınıf üyeleri sınıfa özel (**private**) ya da daha erişilebilir açık olabilir. Erişim belirleyicisi, ne tür bir erişime izin verildiğini belirler. Erişim belirleyicisi isteğe bağlıdır. Erişim belirleyicisi mevcut değilse, bu durumda ilgili üye sınıfa özel (**private**) olur. **private** erişimli üyeler yalnızca kendi sınıflarının diğer üyeleri tarafından kullanılabilirler. Bu bölümdeki örneklerde tüm üyeleri **public** olarak belirtilecektir. Yani, bu üyeleri tüm diğer kodlar tarafından - hatta, sınıfın dışında kalan kodlar tarafından dahil - kullanılabilirler. Erişim belirleyicilerine Bölüm 8'de tekrar döneceğiz.

Aslında bu durumu zorlayan bir söz dizimi kuralı mevcut olmamasına rağmen, iyi tasarlanmış bir sınıfın içinde sadece ve sadece tek bir mantıksal bütün (varlık) tanımlanmalıdır. Örneğin, isim ve telefon numaralarını saklayan bir sınıf, normal olarak, borsa bilgilerini, ortalama yağış yoğunluğunu, güneşin hareketlerini ya da diğer ilgisiz bilgileri de aynı anda saklamayacaktır. Buradaki husus şudur; İyi tasarlanmış bir sınıf, mantıksal olarak bağlantılı bilgileri bir araya getirip, gruplar. İlgisiz bilgileri aynı sınıfın içine yerleştirmek, kodunuzun yapısının çok çabuk bozulmasına neden olur.

Şu ana kadar kullanmakta olduğumuz sınıfların, yalnızca tek bir metodu vardı: **Main()**. Yakında, diğerlerini de nasıl oluşturacağınızı göreceksiniz. Ancak, sınıfın genel yapısında **Main()** metodunun belirtilmemiş olmasına dikkat edin, **Main()** metodu yalnızca, söz konusu sınıf, programınızın başlangıç noktası olduğu taktirde gerekli olur.

## Sınıfı Tanımlamak

Sınıfları göz önünde canlandırmak için binalar, örneğin evler, dükkanlar, ofisler vs, hakkında bilgileri bir araya getiren bir sınıf geliştireceğiz. Bu sınıfı **Building** ismini verelim.

Bu sınıf, bir bina hakkında üç tür bilgi içerecek; Kat sayısı, toplam alan ve oturanların (sakinlerin) sayısı.

**Building**'in ilk versiyonu aşağıda gösterilmiştir, Bunda üç örnek değişken tanımlanmaktadır: **floors**, **area** ve **occupants**. **Building**'in hiç bir metot içermediğine dikkat edin. Yani, **Building** şimdilik sadece veri içeren bir sınıfıtır. (Sonraki bölümlerde buna başka metotlar da eklenecektir.)

```
class Building {
    public int floors; // katların sayısı
    public int area; // toplam bina alanı, fit kare olarak
    public int occupants; // oturanların sayısı
}
```

**Building**'de tanımlanan örnek değişkenler, örnek değişkenlerin genel olarak nasıl deklare edildiğini gösteriyor. Bir örnek değişkeni deklare etmenin genel biçimini aşağıdaki gibidir:

*erişim tip değişken-ismi;*

Burada **erişim**, erişimi; **tip**, değişkenin tipini ve **değişken-ismi** de değişkenin ismini belirtir. Böylece, erişim belirleyicisi bir tarafa, bir örnek değişkeni tipki yerel değişkenleri deklare eder gibi deklare edersiniz. **Building**'de, değişkenlerden önce **public** erişim belirleyicisi gelmektedir. Daha önce açıklandığı gibi bu, değişkenlerin **Building**'in dışındaki kod tarafından erişilmelerine imkan verir.

Sınıf tanımı, yeni bir veri tipi oluşturur. Bu örnekte, yeni veri tipi **Building** olarak adlandırılmıştır. **Building** tipinde nesneler deklare ederken bu ismi kullanacaksınız. Sınıf deklarasyonunun yalnızca tipi tarif eden bir ifade olduğunu, gerçekten bir nesne oluşturmadığını hatırlınızdan çıkarmayın. Bu nedenle, yukarıdaki kod **Building** tipinde herhangi bir nesnenin hayat bulmasına neden olmaz.

Gerçekten bir **Building** nesnesi oluşturmak için aşağıdaki gibi bir ifade kullanacaksınız:

```
Building house = new Building(); // Building tipinde bir nesne oluşturur
```

Bu ifade gerçeklendikten sonra **house**, **Building**'in bir örneği olacaktır. Böylece, "fiziksel" gerçeklik kazanacaktır. Şu an için bu ifadenin ayrıntılarını dert etmeyin.

Bir sınıfa ait bir örneği her oluşturduğunuzda, söz konusu sınıf tarafından tanımlanan her örnek-değişkenin kendi kopyasını içeren bir nesne oluşturmuş olursunuz. Yani, her **Building** nesnesi **floors**, **area** ve **occupants** örnek değişkenlerinin kendi kopyalarını içerecektir. Bu değişkenlere erişmek için **nokta** (.) operatörünü kullanacaksınız. Nokta operatörü, bir nesnenin ismini bir üyenin ismi ile bağlar. Nokta operatörünün genel biçimini şu şekildedir:

*nesne. üye*

Yani nesne, sol tarafta belirtilir ve üye, sağ tarafa yerleştirilir. Örneğin, **house**'un **floors** değişkenine 2 değerini atamak için aşağıdaki ifadeyi kullanın:

```
house.floors = 2;
```

Genel olarak, hem örnek değişkenlere hem de metotlara erişmek için nokta operatörünü kullanabilirsiniz.

İşte, **Building** sınıfını kullanan tamamlanmış bir program:

```
// Building sınıfını kullanan bir program.

using System;

class Building {
    public int floors; // kat sayısı
    public int area; // toplam bina alanı, fit kare olarak
    public int occupants; // oturanların sayısı
}

// Bu, Building tipinde bir nesne deklare eder.
class BuildingDemo {
    public static void Main() {
        Building house = new Building(); // bir Building nesnesi oluştur
        int areaPP; // kişi başına düşen alan

        // house içindeki alanlara değer ata
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;

        // kişi başına düşen alanı hesapla
        areaPP = house.area / house.occupants;

        Console.WriteLine("house has:\n " + house.floors +
                           " floors\n. " + house.occupants\n
                           " occupants\n " + house.area +
                           " total area\n " + areaPP +
                           " area per person");
    }
}
```

Bu program iki sınıfından oluşmaktadır: **Building** ve **BuildingDemo**. **BuildingDemo** içindeki **Main()** metodu, **Building** sınıfının **house** adında bir örneğini oluşturur. Sonra **Main()** içindeki kod, **house** ile ilişkili örnek değişkenlere değerler atayarak ve bu değerleri kullanarak bu değişkenlere erişir. **Building** ve **BuildingDemo**'nın iki ayrı sınıf olduğunu anlamak önemlidir. Bu sınıfların arasındaki tek ilişki, bir sınıfın diğerinin örneğini oluşturmasıdır. Ayrı birer sınıf olmalarına rağmen **BuildingDemo**'nın içindeki kod **Building**'in üyelerine erişebilir, çünkü bu üyeler **public** olarak deklare edilmiştir. Eğer bu üyelerle **public** erişim belirleyicisi kullanılmamış olsaydı, söz konusu üyelerin erişimi **Building** sınıfı ile kısıtlı kalmış olacaktı ve **BuildingDemo** bunları kullanmayacağındır.

Diyelim ki, yukarıdaki **UseBuilding.cs** dosyasını çağrıyorsunuz. Bu programı derlemek **UseBuilding.exe** isminde bir dosya oluşturur. **Building** ve **BuildingDemo**

sınıflarının her ikisi de bu çalıştırılabilir dosyanın birer parçasıdır. Program aşağıdaki çıktıyı ekranda gösterir:

```
house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person
```

Aslında, **Building** ve **BuildingDemo** sınıflarının aynı kaynak dosyası içinde bulunmaları şart değildir. Her sınıfı, örneğin **Building.cs** ve **BuildingDemo.cs** olarak adlandırılan, kendi dosyasına yerleştirebilirsiniz. C# derleyicisine her iki dosyayı derleyip bağlaması gerektiğini söylemeniz yeterlidir. Söz gelisi, programı eğer şimdi anlatılan şekilde iki parçaya ayırmışsanız, programı derlemek için şu komut satırını kullanabilirsiniz.

```
csc Building.cs BuildingDemo.cs
```

Eğer Visual Studio IDE'yi kullanıyorsanız, her iki dosyayı da projenize eklemeniz ve yeniden kurmanız gerekecektir.

Daha fazla ilerlemeden, gelin, temel prensibi tekrarlayalım: Her nesne, kendi sınıfı tarafından tanımlanan örnek değişkenlerin kendi kopyalarına sahiptir. Yani, bir nesnenin içindeki değişkenlerin içeriği, bir başka nesne içindeki değişkenlerin içeriğinden farklı olabilir. Her iki nesne arasında, aynı tipte nesne olmaları haricinde hiç bir bağ yoktur. Örneğin, iki **Building** nesneniz varsa, her nesne **floors**, **area** ve **occupants**'ın kendi kopyalarını içerir ve bunların içerikleri her iki nesne arasında farklılık gösterebilir. Aşağıdaki program bu gerçeği ortaya koymaktadır:

```
//8u program iki Building nesnesi olusturur.
using System;

class Building {
    public int floors;      // kat sayisi
    public int area;        // toplam bina alani, fit kare olarak
    public int occupants;   // oturanların sayisi
}

// Bu sinif, Building tipinde iki nesne deklare eder.
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Building office = new Building();

        int areaPP; // Kisi basina dusen alan

        // house icindeki alanlara deger at
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;

        // office icindeki alanlara deger at
        office.occupants = 25;
```

```

        office.area = 4200;
        office.floors = 3;

        // ev icinde kisi basina dusen alani hesapla
        areaPP = house.area / house.occupants;

        Console.WriteLine("house has:\n " + house.floors +
                           " floors\n " + house.occupants +
                           " occupants\n " + house.area +
                           " total area\n " + areaPP +
                           " area per person");

        Console.WriteLine();

        // ofis icinde kisi basina dusen alani hesapla
        areaPP = office.area / office.occupants;

        Console.WriteLine("office has:\n " + office.floors +
                           " floors\n " + office.occupants +
                           " occupants\n " + office.area +
                           " total area\n " + areaPP +
                           " area per person");
    }
}

```

Bu programla elde edilen çıktı aşağıda gösterilmiştir:

```

house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person

office has:
  3 floors
  25 occupants
  4200 total area
  168 area per person

```

Gördüğünüz gibi, **house**'un verileri **office**'in içeriği verilerden tamamen ayrıdır. Şekil 6.1'de bu durum gösterilmektedir.

The diagram illustrates two separate objects, **house** and **office**, each represented by an arrow pointing to a table of properties and values. The **house** object has properties: floors (2), area (2500), and occupants (4). The **office** object has properties: floors (3), area (4200), and occupants (25).

house	floors	2
	area	2500
	occupants	4

office	floors	3
	area	4200
	occupants	25

**ŞEKİL 6.1:** Bir nesnenin örnek değişkenleri, diğer nesneninkilerden ayrıdır.

## Nesneler Nasıl Oluşturulur?

Önceki programlarda **Building** tipinde bir nesne deklare etmek için aşağıdaki satır kullanılmıştı;

```
Building house = new Building();
```

Bu deklarasyon iki işlevi gerçekleştirir; Öncelikle, **house** adında **Building** sınıfı tipinde bir değişken deklare eder. Bu değişken bir nesne tanımlamaz. Bilakis bu, bir nesneye *ilişkili* olabilecek bir değişkenden ibarettir. İkincisi, söz konusu deklarasyon, nesnenin asıl, fiziksel kopyasını oluşturur ve **house** değişkenine bu nesnenin bir referansını atar. Bu işlem **new** operatörü kullanılarak gerçekleştirilir. Böylece, bu satır gerçeklendikten sonra **house**, **Building** tipinde bir nesneye ilişkilendirilmiş olur.

**new** operatörü, bir nesne için bellekte dinamik olarak (yani, çalışma esnasında) yer ayırır ve buna bir referans döndürür. Bu referans, nesne için **new** ile ayrılan bellek alanının adresidir. Bu referans daha sonra, bir değişken içinde saklanır. Yani, C#'ta tüm sınıf nesneleri için bellek alanı dinamik olarak ayrılmalıdır.

Önceki ifadedeki birleştirilmiş iki adım, her adımı avn ayrı göstermek için şu şekilde yeni den yazılabilir:

```
Building house; // nesneyi gösteren bir referans deklare et
house = new Building(); /* bir Building nesnesi için
                           bellek tahsis et */
```

İlk satır, **Building** tipinde bir nesneye bir referans olarak **house** değişkenini deklare etmektedir. Böylece, **house** bir nesneye ilişkili bir değişkendir, fakat kendisi bir nesne değildir. Bu aşamada, **house**, **null** değerini içermektedir. **null**, değişken bir nesneye ilişkili değildir, anlamını taşır. Bir sonraki satır, yeni bir **Building** nesnesi oluşturur ve bunu gösteren bir referansı **house**'a atar. Artık, **house** bir nesne ile bağlanmıştır.

Sınıf nesnelerinin referans yoluyla erişilmeleri, sınıfların neden *referans tipleri* olarak adlandırıldıklarını da açıklamaktadır. Değer tipleri ve referans tipleri arasındaki en temel fark, her tipin değişkenlerinin sahip oldukları anamlardan kaynaklanmaktadır. Bir değer tipindeki bir değişken için, değişkenin kendisi ilgili değeri taşır. Örneğin,

```
int x;
x = 10;
```

**x**, **10** değerini taşır çünkü **x**, bir değer tipi olan **int** tipinde bir değişkendir. Ancak, şu durumda

```
Building house = new Building();
```

**house**'un kendisi nesneyi içermez. Bunun yerine, nesneyi gösteren bir referans içerir.

## Referans Değişkenleri ve Atama

Bir atama işleminde referans değişkenleri, bir değer tipinin, örneğin `int` tipinin, değişkenlerinden daha farklı davranışırlar. Bir değer tipi değişkenini bir diğerine atarken durum apaçık ortadadır. Sol taraftaki değişken, sağ taraftaki değişkenin *değerinin* bir *kopyasını* alır. Bir nesne referans değişkenini bir diğerine atarken ise, durum biraz daha karmaşıktır, çünkü referans değişkeninin ilişkili olduğu nesneyi değiştiriyorsunuz. Aradaki bu farkın etkisi beklenmeyen bazı sonuçlar doğurabilir. Örneğin, aşağıdaki kod parçasını ele alın:

```
Building house1 = new Building();
Building house2 = house1;
```

İlk bakışta, `house1` ve `house2`'nin farklı nesnelerle ilişkili olduğu kolaylıkla düşünülebilir; fakat bu söz konusu değildir, Bunun yerine, `house1` ve `house2`'nin her ikisi de aynı nesneye ilişkilendirilecektir. `house1`'i `house2`'ye atamak, `house1`'in ilişkilendirilmiş olduğu nesne ile `house2`'yi de ilişkilendirmekten ibarettir. Örneğin, aşağıdaki atama gerçeklendikten sonra

```
house1.area = 2600;
```

şu `WriteLine()` ifadelerinin her ikisi de

```
Console.WriteLine(house1.area);
Console.WriteLine(house2.area);
```

aynı değeri göstermektedirler: **2600**.

`house1` ve `house2`'nin her ikisi de aynı nesne ile ilişkili olsa da, başka hiçbir şekilde ilişkili degillerdir. Örneğin `house2`'ye sonradan bir başka değer atamak yalnızca `house2`'nin ilişkili olduğu nesneyi değiştirir. Örneğin,

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();

house2 = house3; /* artık, house2 ve house3 aynı nesneye
iliskiliidir. */
```

Bu ifade sekansı çalıştırıldıkten sonra `house2`, `house3`'ün ilişkili olduğu nesneye ilişkili hale gelir. `house1`'in ilişkili olduğu nesne değişiklige uğramaz.

## Metotlar

Önceden açıklandığı gibi, örnek değişkenler ve metotlar sınıfları oluşturan iki ana bileşendir. Şimdiye kadar, `Building` sınıfı veri içeriyor ama hiç metot içermiyordu. Sadece veri içeren sınıflar da tamamen geçerli olmasına rağmen, sınıfların birçoğu metot içerecektir. *Metotlar*, sınıf tarafından tanımlanan verileri manipüle eden ve bir çok durumda söz konusu

verilere erişim sağlayan alt rutinlerdir. Özellikle, programınızın diğer bölümleri sınıfı, sınıfın metotları yoluyla etkileşebileceklerdir.

Bir metot, bir veya daha fazla ifade içerir. İyi yazılmış bir C# kodunda her metot yalnızca tek bir görev görür. Her metodun bir ismi vardır ve metodu çağrılmak için bu isim kullanılır. Genel olarak, bir metoda istediğiniz ismi verebilirsiniz. Ancak, **Main()** isminin programınızı çalışmaya başlatan metoda ayrılmış olduğunu unutmayın. Ayrıca, C# anahtar kelimelerini metot isimleri olarak kullanmayın.

Metotları metin içinde belirtirken elinizdeki kitapta, C# hakkında yazarken ortak hale gelmiş bir konvansiyon kullanılmaktadır ve kullanılmaya devam edilecektir. Metot isminden sonra parantezler yer alacaktır. Örneğin, bir metodun ismi **getval** ise, bu metodun ismi bir cümle içinde kullanıldığında **getval()** olarak yazılacaktır. Bu notasyon, elinizdeki kitaptaki değişken isimlerini metot isimlerinden ayırt etmenize yardımcı olacaktır.

Metodun genel biçimi şu şekildedir:

```
erişim dönüş-tipi isim (parametre-listesi) {
    // metodun gövdesi
}
```

Burada **erişim**, programın diğer bölümlerinin bu metodu nasıl çağrıracagini belirleyen bir erişim niteleyicisidir. Önceden açıklandığı gibi, erişim niteleyicisi isteğe bağlıdır. Eğer erişim niteleyicisi mevcut değilse metot, tanımlı olduğu sınıfa özeldir (**private**). Simdilik, tüm metotları **public** olarak deklare edeceğiz, böylece bu metotlar programın içindeki diğer kodlar tarafından da çağrılabilecekler. **dönüş-tipi**, metodun döndürdüğü veri tipini belirtir. Bu, kendi oluşturduğunuz sınıf tipleri de dahil olmak üzere herhangi geçerli tipte olabilir. Eğer metot bir değer döndürmüyorsa, dönüş değeri **void** olmalıdır. Metodun ismi, **isim** ile belirtilir. Bu, mevcut kapsam içinde diğer öğeler tarafından zaten kullanılmakta olanlardan başka, geçerli herhangi bir tanımlayıcı olabilir. **parametre-listesi**, virgül ile ayrılmış tip ve tanımlayıcı çiftlerinden oluşan bir listedir. Parametreler, esas olarak, metod çağrılığında metoda aktarılan argümanların değerini alan değişkenlerdir. Metodun hiç parametresi yoksa parametre listesi de boş olacaktır.

## Building Sınıfına Bir Metot EklemeK

Az önce açıklandığı gibi, bir sınıfın metotları tipik olarak bu sınıfın verilerini manipüle eder ve bu verilere erişim sağlarlar. Bunu hatırlı tutarak, önceki örneklerde **Main()**'in toplam alanı oturanların sayısına bölerek kişi başına düşen alanı hesapladığı hatırlayın. Bu teknik olarak doğru olsa da, bu hesaplamayı gerçekleştirmek için en iyi yöntem değildir. Kişi başına düşen alan hesabı en iyi **Building** sınıfının kendisi tarafından gerçekleştirilebilir. Bu sonuca götüren neden kolaylıkla anlaşılabilir: Bir binada kişi başına düşen alan, **Building** sınıfı tarafından paketlenmiş **area** ve **occupants** alanlarının değerlerine bağlıdır. Böylece, **Building** sınıfının bu hesaplamayı kendi başına gerçekleştirmesi mümkündür. Üstelik, bu hesaplamayı **Building**'in içine ekleyerek, **Building**'i kullanan programların her birinin bu hesabı yapmak zorunda olmalarını da önlemiş olursunuz. Bu, kodun gereksiz yere

kopyalanmasına da engel olur. Son olarak, **Building**'in içine kişi başına düşen alanı hesaplayan bir metot eklemek suretiyle, bir bina ile doğrudan ilişkili nicelikleri **Building** içine zapt ederek **Building** sınıfının nesne yönelimli yapısını da geliştirmiş oluyorsunuz.

**Building**'e bir metot eklemek için söz konusu metodu **Building**'in deklarasyonu içinde belirtin. Örneğin, **Building**'in aşağıdaki versiyonu, bir binada kişi başına düşen alanı gösteren **areaPerPerson()** isimli bir metot içermektedir;

```
// Building'e bir metot ekle.

using System;

class Building {
    public int floors;      // kat sayisi
    public int area;        // toplam bina alani, fit kare olarak
    public int occupants;   // oturanların sayısı

    // Kişi başına düşen alanı göster.
    public void areaPerPerson() {
        Console.WriteLine(" " + area / occupants +
                           " area per person");
    }
}

// areaPerPerson() metodunu kullan.
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Building office = new Building();

        // house içindeki alanlara değer ata
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;

        // office içindeki alanlara değer ata
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;

        Console.WriteLine("house has:\n " +
                          house.floors + " floors\n " +
                          house.occupants + " occupants\n " +
                          house.area + " total area");
        house.areaPerPerson();

        Console.WriteLine();

        Console.WriteLine("office has:\n " +
                          office.floors + " floors\n " +
                          office.occupants + " occupants\n " +
                          office.area + " total area");
        office.areaPerPerson();
    }
}
```

```

        }
    }
}
```

Bu program, önceki ile aynı olan aşağıdaki çıktıyı üretir:

```

house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person

office has:
  3 floors
  25 occupants
  4200 total area
  168 area per person
```

Gelin, şimdi **areaPerPerson()** metodundan başlayarak bu programın temel öğelerine göz atalım. **areaPerPerson()**'un ilk satırı şudur:

```
public void areaPerPerson() {
```

Bu satır, **areaPerPerson()** adında, parametresiz bir metot deklare etmektedir. Bu metot **public** olarak belirtilmiştir; böylece, programın diğer bölümlerinin tümü tarafından kullanılabilir. Bu metodun dönüş tipi **void**'dır. Bu sayede, **areaPerPerson()** kendisini çağrıran program parçasına bir değer döndürmez. Satır, metodun açılış kümeye parantezi ile sona ermektedir.

**areaPerPerson()**'un gövdesi sadece şu ifadeden oluşmaktadır:

```
Console.WriteLine(" " + area / occupants + " area per person");
```

Bu ifade, alanı oturanların sayısına bölerek bir binada kişi başına düşen alanı göstermektedir. **Building** tipindeki her nesne **area** ve **occupants**'ın kendi kopyalarına sahip oldukları için, **areaPerPerson()** çağrıldığı zaman, hesaplarda metodu çağrıran nesneye ait değişken kopyaları kullanılmaktadır.

**areaPerPerson()** metodu, kapanış kümeye parantezine rastlayınca sona erer. Bu, program kontrolünün tekrar metodu çağrıran koda geri dönmesine neden olur.

Şimdi, **Main()**'in içindeki şu kod satırına yakından bir bakın:

```
house.areaPerPerson();
```

Bu ifade, **house** üzerindeki **areaPerPerson()** metodunu çağrıır. Yani, nesnenin isminin ardından nokta operatörünü kullanarak **areaPerPerson()**'u **house** nesnesine bağlı olarak çağrımaktadır. Bir metot çağrıldığı zaman, program kontrolü metoda aktarılır. Metot sona erdiğinde ise, kontrol tekrar metodu çağrıran koda geri döner ve program, çağrıyı takip eden kod satırı ile devam eder.

Bu örnekte, `house.areaPerPerson()` çağrısı, `house` tarafından tanımlanan binada kişi başına düşen alanı göstermektedir. Benzer şekilde, `office.areaPerPerson()` çağrısı ise `office` tarafından tanımlanan, binadaki kişi başına düşen alanı gösterir. `areaPerPerson()`, her çağrılmışında belirtilen nesne için kişi başına düşen alanı gösterir.

`areaPerPerson()` metodu içinde çok dikkat edilmesi gereklili olan bir şey vardır: `area` ve `occupants` örnek değişkenleri doğrudan kullanılırlar; isimleri önünde nesne ismi veya nokta operatörü kullanılmaz. Bir metot kendi sınıfı tarafından tanımlanmış bir örnek değişken kullandığında, bunu doğrudan gerçekleştirir; nesneye açıkça referansta bulunmaz veya nokta operatörünü kullanmaz. Bunun üzerinde biraz düşünürseniz mantıklı olduğunu görürsünüz. Bir metot her zaman kendi sınıfının belirli bir nesnesine bağlı olarak çağrılır. Bu tür bir çağrı söz konusu olduğu andan itibaren artık bu nesne bilinir. Böylece, bir metot içinde nesneyi ikinci kez belirtmeye gerek yoktur. Bu, `areaPerPerson()` içindeki `area` ve `occupants`'ın, `areaPerPerson()`'u çağıran nesne içindeki bu değişkenlerin kopyalarıyla dolaylı olarak ilişkili oldukları anlamına gelir.

## Metottan Dönüş

Genel olarak, bir metodun dönmesine neden olan iki koşul söz konusudur. İlk, önceki örnekteki `areaPerPerson()` metodundan da görüleceği gibi, metodun kapanış küme parantezi ile karşılaşınca söz konusu olur. İkinci ise `return` ifadesi çalıştırıldığında gerçekleşir, `return`'ün iki şekli vardır: biri `void` metodlarında (bir değer döndürmeyen metodlar) kullanılan şekli, biri de değer döndürmek için kullanılan şekli. Burada ilk şekil incelenmektedir. Bir sonraki bölümde nasıl değer döndürüldüğü açıklanmaktadır.

Bir `void` metotla, aşağıdaki gibi bir `return` kullanarak metodun yanında dönmesine neden olabilirsiniz:

```
return ;
```

Bu ifade çalıştırıldığında program kontrolü, metodun geri kalan kodunu atlayarak metodу çağırılan koda döner, Örneğin, şu metodu ele alın:

```
public void myMeth() {
    int i;

    for( i = 0; i < 10; i++) {
        if(i == 5) return; // 5'te dur
        Console.WriteLine();
    }
}
```

Burada, `for` döngüsü yalnızca 0'dan 5'e kadar donecektir, çünkü `i`, 5'e eşit olur olmaz metot döner.

Bir metot içinde, özellikle iki veya daha fazla dönüş yolu söz konusu olduğunda, birden fazla `return` ifadesinin bulunmasına izin verilebilir. Örneğin,

```
public void myMeth() {
    // ...
    if(done) return;
    // ...
    if(error) return;
}
```

Bu örnekte, metodun işi bittiğinde (*done*) veya bir hata (*error*) ortaya çıktığında metot döner. Ancak, siz yine de dikkatli olun. Bir metot içinde çok sayıda çıkış noktası olması kodunuzun yapısını bozabilir; bu nedenle, bunları sık sık kullanmaktan kaçının.

Tekrar söyleysek, bir **void** metot, iki yöntemden biri kullanılarak dönebilir- metodun kapanış küme parantezine ulaşıldığında veya bir **return** ifadesi gerçekleştiğinde.

## Bir Değer Döndürmek

**void** tipinde metodlar ender olmasa da, metodların çoğu bir değer döndürecektir. Aslında, bir değer döndürme becerisi bir metodun en yararlı özelliklerinden biridir. Bölüm 3’te kare kök elde etmek için **Math.Sqrt()** fonksiyonunu kullandığımızda, dönüş değerine bir örnek görmüştünüz.

Dönüş değerleri, programlamada çok çeşitli amaçlarda kullanılabilir. Bazı durumlarda, **Math.Sqrt()** örneğinde olduğu gibi, dönüş değeri bazı hesaplamalardan elde edilen sonucu içermektedir. Diğer durumlarda dönüş değeri, yalnızca başarı ve başarısızlığı göstermekten ibaret olabilir. Geri kalan durumlarda ise, dönüş değeri bir durum kodunu içerebilir. Amaç ne olursa olsun, metod dönüş değerlerini kullanmak C# programlamanın önemli bir parçasıdır.

Metotlar, kendilerini çağrıran rutine, aşağıdaki şekilde **return** kullanarak bir değer döndürürler:

```
return değer;
```

Burada **değer**, döndürülen değerdir.

**areaPerPerson()**’un uygulanışını geliştirmek için bir dönüş değeri kullanabilirsiniz. Kişi başına düşen alanı göstermek yerine, **areaPerPerson()**’un bu değeri döndürmesini sağlamak daha iyi bir yaklaşımdır. Bu değeri diğer hesaplamalarda da kullanılabilecek olmanız, bu yaklaşımın sağladığı avantajlar arasındadır. Aşağıdaki örnekte **areaPerPerson()** metodу, kişi başına düşen alanı göstermek yerine, bu değeri döndürecek şekilde değiştirilmektedir:

```
// areaPerPerson()'dan bir değer dondurmek.

using System;

class Building {
    public int floors; // kat sayısı
    public int area; // binanın toplam alanı, fit kare olarak
    public int occupants; // oturanların sayısı

    // Kişi başına düşen alanı dondur.
```

```
public int areaPerPerson() {
    return area / occupants;
}

// areaPerPerson()'dan donen değeri kullan.
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Building office = new Building();
        int areaPP; // kişi başına düşen alan

        // house içindeki alanlara değer ata
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;

        // office içindeki alanlara değer ata
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;

        // ev için kişi başına düşen alanı elde et
        areaPP = house.areaPerPerson();

        Console.WriteLine("house has:\n" +
                           house.floors + " floors\n" +
                           house.occupants + " occupants\n" +
                           house.area + " total area\n" +
                           areaPP + " area per person");

        Console.WriteLine();

        // ofis için kişi başına düşen alanı elde et
        areaPP = office.areaPerPerson();

        Console.WriteLine("office has:\n" +
                           office.floors + " floors\n" +
                           office.occupants + " occupants\n" +
                           office.area + " total area\n" +
                           areaPP + " area per person");
    }
}
```

Cıktı, önceden gösterilen çıktıyla aynıdır.

Programda dikkat ederseniz, **areaPerPerson()** çağrıldığı zaman, atama ifadesinin sağ tarafına yerleştirilmektedir. Solda, **areaPerPerson()**'un döndüreceği değeri alacak olan değişken yer alır. Böylece, aşağıdaki ifade çalıştırıldıktan sonra, **house** nesnesinin kişi başına düşen alanı **areaPP** içinde saklanır.

```
areaPP = house.areaPerPerson();
```

`areaPerPerson()`'un artık `int` tipinde bir değere sahip olduğuna dikkat edin. Bu, kendisini çağırın koda bir tamsayı değer döndürecegi anlamına gelir. Bir metodun dönüş değeri önemlidir, çünkü metot tarafından döndürülen değer, metot tarafından belirtilen değerle uyumlu olmalıdır. Yani, `double` tipinde bir değer döndüren bir metot istiyorsanız, bu durumda söz konusu metodun dönüş tipi `double` tipinde olmalıdır.

Önceki program doğru olmasına rağmen, olabildiğince verimli bir biçimde yazılmamıştır. Spesifik olarak, `areaPP` değişkenine hiç gerek yoktur. `areaPerPerson()`'a yapılan çağrı doğrudan `WriteLine()` ifadesinde kullanılabilir. Şu şekilde:

```
Console.WriteLine("house has:\n " +
    house.floors + " floors\n " +
    house.occupants + " occupants\n " +
    house.area + " total area\n " +
    house.areaPerPerson() + " area per person");
```

Bu örnekte, `WriteLine()` çalıştırıldığında `house.areaPerPerson()` otomatik olarak çağrılar ve bundan sonra dönen değer, `WriteLine()`'a aktarılır. Üstelik, bir `Building` nesnesi için kişi başına düşen alan ne zaman gerekli olarsa, her seferinde `areaPerPerson()`'a çağrıda bulunabilirsiniz. Örneğin, şu ifade iki binanın kişi başına düşen alanlarını karşılaştırmaktadır:

```
if(b1.areaPerPerson() > b2.areaPerPerson())
    Console.WriteLine("b1 has more space for each person");
```

## Parametre Kullanmak

Bir metot çağrıldığı zaman, metoda bir veya iki değer aktarmak mümkündür. Önceden açıklandığı gibi, bir metoda aktarılan değer *argüman* denir. Metot içinde, argümanı kabul eden değişkene *parametre* denir. Parametreler, metot isminin peşinden gelen parantezler içinde deklare edilirler. Parametreleri deklare etmek için kullanılan söz dizimi, değişkenler için kullanılan ile aynıdır. Bir parametre kendi metodunun kapsamı içindedir ve argüman kabul etmek gibi özel görevi bir tarafa, tipki diğer yerel değişkenler gibi davranışır.

İşte, parametre kullanan basit bir örnek. `ChkNum` sınıfı içinde `isPrime()` metodu, kendisine aktarılan değer asal ise `true` döndürmektedir. Aksi halde, `false` döndürür. Bu nedenle, `isPrime()`, `bool` dönüş tipine sahiptir.

```
// Parametre kullanan basit bir örnek.

using System;

class ChkNum {
    // x, asal ise true dondur.
    public bool isPrime(int x) {
        for(int i=2; i < x/2 + 1; i++)
            if((x % i) == 0) return false;

        return true;
    }
}
```

```

class ParmDemo {
    public static void Main() {
        ChkNum ob = new ChkNum();

        for(int i=1; i < 10; i++)
            if(ob.isPrime(i)) Console.WriteLine(i + " is prime.");
            else Console.WriteLine(i + " is not prime.");

    }
}

```

Programın çıktısı işte şöyledir:

```

1 is prime.
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.

```

Bu programda **isPrime()** dokuz kez çağrılmaktadır ve fonksiyona her seferinde farklı bir değer aktarılmaktadır. Gelin, şimdi bu işleme daha yakından göz atalım. Öncelikle, **isPrime()**'ın nasıl çağrıldığına dikkat edin. Argüman, parantez içinde belirtilmektedir. **isPrime()** ilk kez çağrıldığında, fonksiyona **1** değeri aktarılmaktadır. Yani, **isPrime()** gerçeklenmeye başladığında **x** parametresi **1** değerini alır. İkinci çağrıda argüman, **2**'dir ve **x**, **2** değerine sahip olur Üçüncü çağrıda argüman, **3**'tür ve bu, **x**'in kabul ettiği değerdir vs. Buradaki husus şudur: **isPrime()** çağrılığında argüman olarak aktarılan değer, fonksiyonun **x** parametresi tarafından kabul edilen değerdir.

Bir metodun birden fazla parametresi olabilir. Sadece, her parametreyi bir sonrakinden virgülle ayırarak deklare edin. Örneğin, burada **ChkNum** sınıfı, **lcd()** adında bir metot eklenerek genişletilmiştir. **lcd()**, kendisine aktarılan iki değerin en küçük ortak bölenini döndürür:

```

// iki arguman alan bir metot ekleyin.

using System;

class ChkNum {
    // x asal ise true dondur.
    public bool isPrime(int x) {
        for(int i=2; i < x/2 + 1; i++)
            if(x % i) == 0) return false;

        return true;
    }
    // En kucuk ortak boleni dondur.
    public int lcd(int a, int b) {

```

```

        int max;

        if(isPrime(a) | isPrime(b) return 1;

        max = a < b ? a : b;

        for(int i=2; i < max/2 + 1; i++)
            if(((a%i) == 0) & ((b%i) == 0)) return i;

        return 1;
    }
}

class ParmDemo {
    public static void Main() {
        ChkNum ob = new ChkNum();
        int a, b;

        for(int i=1; i < 10; i++)
            if(ob.isPrime(i)) Console.WriteLine(i + " is prime.");
            else Console.WriteLine(i + " is not prime.");

        a = 7;
        b = 8;
        Console.WriteLine("Least common denominator for " +
                           a + " and " + b + " is " +
                           ob.lcd(a, b));

        a = 100;
        b = 8;
        Console.WriteLine("Least common denominator for " +
                           a + " and " + b + " is " +
                           ob.lcd(a, b));

        a = 100;
        b = 75;
        Console.WriteLine("Least common denominator for " +
                           a + " and " + b + " is " +
                           ob.lcd(a, b));
    }
}

```

**lcd()** çağrılığında argümanların da virgül ile ayrıldığına dikkat edin. Programdan elde edilen çıktı aşağıda gösterilmiştir;

```

1 is prime.
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.
Least common denominator for 7 and 8 is 1

```

```
Least common denominator for 100 and 8 is 2
Least common denominator for 100 and 75 is 5
```

Birden fazla parametre kullanıldığında her parametre kendi tipini belirtir. Parametrelerin tipleri birbirinden farklı olabilir. Örneğin, şu ifade tamamen geçerlidir:

```
int myMeth(int a, double b, float c) {
    // ...
}
```

## Building'e Parametreli Bir Metot EklemeK

**Building** sınıfına yeni bir özellik katmak için parametreli bir metot kullanabilirsiniz: Bu özellik; bir bina sakinlerinin belirli bir minimal alan kapladıklarını varsayıarak, bina sakinlerinin maksimum sayısını hesaplama becerisi olabilir. Bu yeni metot **maxOccupant()** olarak adlandırılır ve aşağıda gösterildiği gibidir:

```
/* Bina sakinlerinin her biri en azından belirli bir minimum alan
   kaplamak zorundaysa, bina sakinlerinin maksimum sayısını
   dondur. */
public int maxOccupant(int minArea) {
    return area / minArea;
}
```

**maxOccupant()** çağrıldığında, **minArea** parametresi her bina sakini için gerekli olan minimum alanı alır. Metot, binanın toplam alanını bu değere böler ve sonucu döndürür.

**maxOccupant()**'ı da içeren bütün **Building** sınıfının tamamı aşağıda gösterilmiştir:

```
/*
   Bir binada oturan kişilerin maksimum sayısını hesaplayan
   parametreli bir metot ekleyin.
   Her bina sakininin belirli bir minimum alana gereksinim
   duyduğunu varsayı.
*/
using System;

class Building {
    public int floors;      // kat sayısı
    public int area;        // toplam bina alanı, fit kare olarak
    public int occupants;   // oturanların sayısı
    // Kişi başına düşen alanı dondur.
    public int areaPerPerson() {
        return area / occupants;
    }

    /* Bina sakinlerinin her biri en azından belirli bir minimum
       alan kaplamak zorundaysa, bina sakinlerinin maksimum
       sayısını dondur. */
    public int maxOccupant(int minArea) {
        return area / minArea;
    }
}
```

```

// maxOccupant() metodunu kullan.
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Building office = new Building();

        // house icindeki alanlara değer ata
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;

        // office icindeki alanlara değer ata
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;

        Console.WriteLine("Maximum occupants for house if each has " +
            300 + " square feet: " +
            house.maxOccupant(300));

        Console.WriteLine("Maximum occupants for office if each has " +
            + 300 + " square feet: " +
            office.maxOccupant (300));
    }
}

```

Programın çıktısı aşağıdaki gibidir;

```

Maximum occupants for house if each has 300 square feet: 8
Maximum occupants for office if each has 300 square feet: 14

```

## Erişilemeyen Kodları Önlemek

Metotları hazırlarken, kodun bazı parçalarının her ne koşulda olursa olsun çalıştırılamaz olmasına neden olabilecek durumları önlemelisiniz. Buna *erişilemeyen kod (unreachable code)* denir ve bu tür bir kod C#'ta hatalı olarak kabul edilir. Erişilemeyen kod içeren bir metot geliştirmişişeniz, derleyici bu durumda bir uyarı mesajı verecektir. Örneğin:

```

public void m() {
    char a, b;

    // ...

    if (a == b) {
        Console.WriteLine("equal");
        return;
    }
    else {
        Console.WriteLine("not equal");
        return;
    }
    Console.WriteLine("this is unreachable");
}

```

Burada, `m()` metodu daima son `WriteLine()` ifadesinden önce dönecektir. Bu metodu derlemeye çalışırsanız bir uyarı mesajı ile karşılaşırınsınız. Genel olarak, erişilmez kodlar sizin tarafınızdan yapılmış bir hata nedeniyle ortaya çıkar; bu nedenle, erişlemeyen kodlarla ilgili uyarıları ciddiye almanız tavsiye edilir.

## Yapilandırıcılar

Önceki örneklerde her `Building` nesnesinin örnek değişkenlerinin, aşağıdaki gibi bir ifade sekansı kullanılarak, ayarlanması gerekiyordu;

```
house.occupants = 4;  
house.area = 2500;  
house.floors = 2;
```

Bu tür bir yöntem, profesyonelce yazılmış C# programlarında asla kullanılmaz. Bu durumun hataya açık olması bir yana (alanlardan birine değer vermemeyi unutabilirsiniz), bunu çok daha iyi gerçekleştirebilecek bir yöntem de mevcuttur.

*Yapilandırıcı (constructor)*, bir nesne oluşturulduğu anda bu nesneyi ilk kullanıma hazırlar. Yapilandırıcı kendi sınıfı ile aynı isme sahiptir ve söz dizimsel olarak metoda benzer. Ancak, yapılandırıcıların açık bir dönüş tipi yoktur. Yapılandırıcıların genel yapısı aşağıda gösterilmiştir:

```
erisim sınıf-ismi () {  
    // yapılandırıcı kodu  
}
```

Genellikle, bir sınıf tarafından tanımlanan örnek değişkenlere ilk değer atarken veya bir nesnenin eksiksiz oluşturulması için gerekli diğer başlangıç prosedürlerini gerçekleştirirken yapılandırıcıları kullanacaksınız. Ayrıca, *erisim* genellikle `public`'tir, çünkü yapılandırıcılar normal olarak kendi sınıfları dışından çağrırlırlar.

Siz tanımlayın ya da tanımlamayın, tüm sınıfların yapılandırıcıları vardır, çünkü C# otomatik olarak, tüm üye değişkenlere sıfır (değer tipleri için) veya null (referans tipleri için) ilk değerini atayan bir varsayılan yapılandırıcı sağlar. Yine de, kendi yapılandırınızı tanımladığınız andan itibaren varsayılan yapılandırıcı bir daha kullanılmaz.

İşte size, bir yapılandırıcı kullanan basit bir örnek:

```
// Basit bir yapılandırıcı.  
  
using System;  
  
class MyClass {  
    public int x;  
  
    public MyClass() {  
        x = 10;  
    }  
}
```

```

class ConsDemo {
    public static void Main() {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();

        Console.WriteLine(t1.x + " " + t2.x);
    }
}

```

Bu örnekte **MyClass** için yapılandırıcı şudur:

```

public MyClass() {
    x = 10;
}

```

Dikkat ederseniz, yapılandırıcı **public** olarak belirtilmiştir. Bunun nedeni, yapılandırıcının kendi sınıfı dışında kalan kod tarafından çağrılmak olmalıdır. Bu yapılandırıcı **MyClass**'ın **x** örnek değişkenine **10** değerini atar. Yeni bir nesne oluşturulacağı zaman söz konusu yapılandırıcı **new** kullanılarak çağrılr. Örneğin, aşağıdaki satırda

```
MyClass t1 = new MyClass();
```

**MyClass()** yapıtaidricisi **t1** nesnesi üzerinde çağrılr ve **t1.x**'e **10** değeri atanmış oluyor. Aynı durum **t2** için de geçerlidir. Yapılandırıcı kullanımından sonra **t2.x**, **10** değerine sahip olur. Böylece, programın çıktısı şöyle olur:

```
10 10
```

## Parametreli Yapılandırıcılar

Önceki örnekte parametresiz yapılandırıcı kullanılmıştı. Bu, birçok durum için uygun olsa da, daha çok, bir veya daha fazla parametre kabul eden yapılandırıcılara ihtiyacınız olacak. Parametreler tipki metodlara eklendiği şekliyle yapılandırıcılara eklenir: Parametreleri, yapılandırıcının isminin ardından gelen parantezler içinde deklare etmeniz yeterlidir. Örneğin, burada **MyClass**'a parametreli bir yapılandırıcı verilmiştir:

```

// Parametreli yapılandirici.

using System;

class MyClass {
    public int x;

    public MyClass(int i) {
        x = i;
    }
}

class ParmConsDemo {
    public static void Main() {
        MyClass t1 = new MyClass(10);
    }
}

```

```

    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
}
}

```

Bu programın çıktısı aşağıdaki gibidir:

```
10 38
```

Programın bu versiyonunda **MyClass()** yapılandırıcısı, **i** adında bir parametre tanımlıyor. **i**, **x** örnek değişkenine ilk değer atamak için kullanılıyor. Böylece, aşağıdaki satır gerçekleşlendiğinde **10** değeri **i**'ye aktarılır ve bu, daha sonra **x**'e atanır;

```
MyClass t1 = new MyClass(10);
```

## Building Sınıfına Bir Yapılandırıcı EklemeK

Bir nesne oluşturulurken **floors**, **area** ve **occupants** alanlarına otomatik olarak ilk değer atayan bir yapılandırıcıyı **Building** sınıfına ekleyerek **Building** sınıfını geliştirebiliriz. **Building** nesnelerinin nasıl oluşturulduğuna özellikle dikkat edin.

```

// Building'e bir yapılandırıcı ekler.

using System;

class Building {
    public int floors;          // kat sayisi
    public int area;            // toplam bina alani, fit kare olarak
    public int occupants;       // oturanların sayisi

    public Building(int f, int a, int o) {
        floors = f;
        area = a;
        occupants = o;
    }

    // Kisi basina dusen alani dondur.
    public int areaPerPerson() {
        return area / occupants;
    }

    /* Bina sakinlerinin her biri en azindan belirli bir minimum
       alan kaplamak zorundaysa, oturanların maksimum sayisini
       dondur. */
    public int maxOccupant(int minArea) {
        return area / minArea;
    }
}

// Parametreli Building yapılandırıcısını kullan.
class BuildingDemo {
    public static void Main() {

```

```

Building house = new Building(2, 2500, 4);
Building office = new Building(3, 4200, 25);

Console.WriteLine("Maximum occupants for house if each has "
    + 300 + " square feet: " +
    house.maxOccupant (300));

Console.WriteLine("Maximum occupants for office if each has "
    + 300 + " square feet: " +
    office.maxOccupant(300));
}
}

```

Bu programın çıktısı önceki versiyonunkiyle aynıdır.

Hem **house**, hem de **office** oluşturuldukları anda **Building()** yapılandırıcısı ile ilk kullanıma hazırlanırlar. Nesnelerin her birine ilk değerleri, parametrelerde belirtildiği şekliyle kendi yapılandırıcısı tarafından atanır. Örneğin, aşağıdaki satırda

```
Building house = new Building(2, 2500, 4);
```

**new** nesneyi oluştururken **2, 2500** ve **4** değerleri **Building()** yapılandırıcısına aktarılır. Böylece, **house**'a ait **floors**, **area** ve **occupants** kopyaları sırasıyla **2, 2500** ve **4** değerlerini içerecektir.

## new Operatörüne Tekrar Bir Bakış

Sınıflar ve sınıfların yapılandırıcıları hakkında daha fazla bilgi sahibi olduğunuzu göre gelin, **new** operatörünü daha yakından inceleyelim. **new** operatörünün genel yapısı şu şekildedir:

```
sınıf-değişkeni = new sınıf-ismi();
```

Burada **sınıf-değişkeni**, oluşturulmakta olan sınıf tipindeki değişkendir, **sınıf-ismi**, örneklenmekte olan sınıfın ismidir. Ardından parantezlerin geldiği sınıf ismi, sınıfın ait yapılandırıcıyı belirtir. Eğer bir sınıf kendi yapılandırıcısını tanımlamıyorsa **new**, C# tarafından sağlanan varsayılan yapılandırıcıyı kullanacaktır. Böylece **new**, herhangi bir sınıf tipinde nesneleri oluşturmak için kullanılabilir.

Bellek sonlu olduğu için belleğin yetersiz olduğu durumlarda **new**'un bir nesne için bellekte yer ayıramaması mümkün değildir. Eğer böyle bir durum söz konusu ise, programın çalışması esnasında bir kural dışı durum ortaya çıkacaktır. (Bunun ve diğer kural dışı durumların nasıl kontrol altına alındığını Bölüm 13'te öğreneceksiniz.) Elinizdeki kitaptaki örnek programlarda belleğin tükenmesinden endişe etmenize gerek olmayacak, fakat sizin yazacağınız gerçek dünyaya ait programlarda bu ihtimali göz önünde bulundurmanız gerekecektir.

## new'u Değer Tipleriyle Birlikte Kullanmak

Bu aşamada, `new`'u `int` veya `float` gibi değer tiplerindeki değişkenlerle neden kullanmanız gerekmeyi merak ediyor olabilirsiniz. C#'ta bir değer tipindeki değişken kendi değerini içerir. Program derlendiğinde bu değeri tutacak bellek alanı, derleyici tarafından otomatik olarak ayrıılır. Böylece, `new` kullanarak bu bellek alanı açıkça ayırmaya gerek yoktur. Bunun karşıtı olarak bir referans değişkeni, bir nesneyi gösteren bir referans içerir. Bu nesneyi tutacak olan bellek alanı, programın çalışması sırasında dinamik olarak ayrılır.

Temel tiplerin - örneğin `int` veya `char` - referans tipi olmaması, programınızın performansını büyük ölçüde artırır. Bir referans tipi kullanıldığı zaman, bir dolaylılık katmanı söz konusu olur ki bu, nesne erişimlerinin her birine değer tiplerinde olmayan ekstra bir yük ekler.

İşin ilginci, aşağıda gösterildiği gibi, `new`'un değer tipleriyle kullanımına izin verilmiş olmasıdır:

```
int i = new int();
```

Bu ifade, `int` tipi için varsayılan yapılandırıcıyı çağırır ve `i`'ye sıfır ilk değerini atar. Örneğin:

```
// new'u bir değer tipiyle birlikte kullanmak.

using System;

class newValue {
    public static void Main() {
        int i = new int(); // i'ye sıfır ilk değerini ata

        Console.WriteLine("The value of i is: " + i);
    }
}
```

Bu programın çıktısı şöyledir:

```
The value of i is: 0
```

Cıktının da doğruladığı gibi, `i`'ye sıfır ilk değeri atanmaktadır. Sunu hatırlınızdan çıkarmayın: `new` kullanılmamış olsaydı `i`'ye ilk değer atanmamış olacaktı ve `i`'ye açıkça bir değer vermeden `i`'yi `WriteLine()` ifadesinde kullanmaya çalışmak hatalı olacaktır.

Genel olarak, `new`'u bir değer tipi için çağrılmak söz konusu tip için varsayılan yapılandırıcıyı çağrılmak demektir. Açıkçası, programcıların çoğu `new`'u değer tipleriyle birlikte kullanmazlar.

## Anlamsız Verilerin Toplanması ve Yok ediciler

Gördüğünüz gibi nesnelere **new** operatörü kullanılarak serbest bellek havuzundan dinamik olarak bellek alanı ayrıılır. Elbette, bellek sonsuz değildir ve serbest bellek tükenebilir. Yani, istenilen nesneyi oluşturmak için yeterli serbest bellek alanı olmaması durumunda, **new** işleminin başarısızlıkta sonuçlanması mümkündür. Bu nedenle, herhangi bir dinamik bellek alanı ayırma tekniğinin en temel bileşenlerinden biri, kullanılmayan nesnelerin işgal ettiği serbest bellek alanını geri kazanmak ve bu bellek alanını daha sonraki ayırma işlemleri için hazır tutmaktadır. Birçok programlama dilinde, önceden ayrılmış bellek alanının serbest bırakılması, elle kontrol edilir. Örneğin, C++'ta ayrılmış bellek alanını serbest bırakmak, için **delete** operatörünü kullanırsınız. Ancak, C# farklı ve daha problemsiz bir yöntem kullanmaktadır: Anlamsız verilerin toplanması (*garbage collection*).

C#'ın anlamsız verileri toplama sistemi, nesnelere ayrılan bellek alanını otomatik olarak geri alır ve bu işlemi programcıların müdahale etmesine gerek kalmadan; perde arkasında gerçekleştirir. Sistem şu şekilde çalışır: Bir nesneye hiçbir referans söz konusu değilse, bu nesneye artık ihtiyaç duyulmayacağı varsayılar ve bu nesnenin işgal ettiği bellek alanı serbest bırakılır. Bu geri kazanılan bellek alanı, daha sonraki ayırma işlemlerinde kullanılabilir.

Anlamsız verileri toplama işlemi, programınızın çalışması sırasında sadece ara sıra meydana gelir. Daha fazla kullanılmayacak bir veya daha fazla nesne her mevcut olduğunda gerçekleşmez. Anlamsız verileri toplama işlemi zaman aldığı için, C# çalışma zamanı (runtime) sistemi sadece ihtiyaç olduğunda veya diğer uygun zamanlarda bunu gerçekleştirir. Bu nedenle, anlamsız verileri toplama işleminin ne zaman gerçekleştirileceğini kesin olarak bilemezsiniz.

## Yok Ediciler

Bir nesnenin anlamsız veri toplayıcısı tarafından tamamen yok edilmesinden hemen önce çağrılmak üzere bir metot tanımlamak mümkündür. Bu metoda *yok edici (destructor)* denir ve bir nesnenin kesinlikle sonlandırıldığını garanti etmek için kullanılabilir. Örneğin, bu nesnenin sahip olduğu açık bir dosyanın kapatıldığından emin olmak için bir yok edici kullanabilirsiniz. Yok ediciler, genel olarak aşağıdaki bir yapıya sahiptir:

```
-sınıf-ismi() {  
    // yok edici kodu  
}
```

Burada ***sınıf-ismi***, sınıfın ismidir. Yani, bir yok edici tıpkı bir yapılandırıcı gibidir. Tek fark, en başta tilda karakteri (-) içermesidir, Dönüş tipinin olmadığına dikkat edin.

Bir sınıfa bir yok edici eklemek için, bunu bir üye olarak sınıfa dahil etmeniz yeterlidir. Yok edici, kendi sınıfının bir nesnesi geri dönüştürülmemek üzere olduğunda çağrılacaktır. Nesne yok edilmeden önce gerçekleştirilmesi gereken işlemleri yok edici içinde belirtirsiniz.

Yok edicinin anlamsız verileri toplama işleminden hemen önce çağrıdığını kavramak önemlidir. Örneğin, nesne kapsam dışına çıktıığında yok edici çağrılmaz. (Bu, nesnenin kapsam

dışına çıkması durumunda çağrılan C++'ın yok edicilerinden farklıdır.) Bu, yok edicilerin ne zaman çalıştırılacağını tam olarak bilemeyeceğiniz anlamına gelir. Yine de, program sona ermeden önce yok edicilerin tümü çağrılacaktır.

Aşağıdaki program bir yok ediciyi göstermektedir. Program, çok sayıda nesne oluşturarak ve yok ederek çalışmaktadır. Bu süreç zarfında, bir aşamada anlamsız veri toplayıcısı etkin kılınacaktır ve bu nesneler için yok ediciler çağrılacaktır.

```
// Bir yok ediciyi gösterir.

using System;

class Destruct {
    public int x;

    public Destruct(int i) {
        x = i;
    }

    // nesne geri donusturulurken cagrılır
    -Destruct() {
        Console.WriteLine("Destructing " + x);
    }

    // hemen yok edilmek üzere bir nesne üretir
    public void generator(int i) {
        Destruct o = new Destruct(i);
    }
}

class DestructDemo {
    public static void Main() {
        int count;

        Destruct ob = new Destruct(0);

        /* Şimdi, çok sayıda nesne üretin. Bir aşamada, anlamsız
           veri toplama işlemi meydana gelecektir.
           Dikkat: Anlamsız veri toplamayı zorlamak için, üretilen
           nesne sayısını artırmanız gerekebilir. */

        for(count = 1; count * 100000; count++)
            ob.generator(count);

        Console.WriteLine("Done");
    }
}
```

Programın çalışması şöyledir: Yapılandırıcı, **x** örnek değişkenine ilk değer olarak bilinen bir değer atar. Bu örnekte **x**, nesnenin kimlik bilgisi olarak kullanılmaktadır. Yok edici, nesne geri dönüştürülürken **x**'in değerini gösterir, **generator()** özellikle enteresandır. Bu metot, bir **Destruct** nesnesini oluşturur ve hemen ardından da yok eder. **DestructDemo** sınıfı, **ob**

adında ilk **Destruct** nesnesini oluşturur. Sonra **ob**'u kullanarak ve **generator()**'ı **ob** üzerinde çağırarak, **100.000** nesne üretir. Bu işlemin net etkisi **100.000** nesne üretip, yok etmektir. Bu sürecin ortalarında, çeşitli aşamalarda anlamsız verileri toplama işlemi yer alacaktır. Bu işlemin tam olarak ne zaman ve ne kadar sıklıkla gerçekleştirileceği, birkaç faktöre bağlıdır: Başlangıçtaki serbest bellek alanı miktarı, işletim sistemi vs. Ancak, bir aşamada yok edici tarafından üretilen uyarı mesajlarını görmeye başlayacaksınız. Program sona ermeden önce (yani, “**Done**” mesajından önce) bu mesajları görmemişseniz, **for** döngüsü içindeki **count**'u yükselterek üretilmekte oları nesne sayısını artırmayı deneyin.

Yok edicilerin belirsiz biçimde çağrılmalarından ötürü bu fonksiyonlar, programınızda belirli bir aşamada meydana gelmesi gereken işleri gerçekleştirmek için kullanılmamalıdır.

Diğer bir husus ise şudur: Anlamsız verilerin toplanması talebinde bulunmak mümkündür. Bu durum, Kısım II'de C#'ın sınıf kütüphanesi ele alınırken anlatılacaktır. Ancak, anlamsız verilerin toplanması işlemini elle başlatmak, birçok durum için tavsiye edilmez, çünkü bu, verimsizliklere yol açabilir Aynca, açıkça anlamsız verilerin toplanması talebinde bulunmuş olsanız bile, anlamsız veri toplayıcısının çalışma şeklinden dolayı, belirli bir nesnenin ne zaman geri dönüştürüleceğini tam olarak bilmenizin bir yolu yoktur.

## this Anahtar Kelimesi

Bu bölüme son vermeden önce **this**'i tanıtmak gereklidir. Bir metot çağrıldığı zaman, bu metoda, kendisini çağrıran nesnenin bir referansı olan dolaylı bir argüman otomatik olarak aktarılır. Bu referans **this** olarak adlandırılır. **this**'i anlamak için öncelikle bir program ele alalım. Bu program, bir dikdörtgenin enini ve boyunu içeren **Rect** adında bir sınıf oluşturur. **Rect** sınıfı ayrıca dikdörtgenin alanı döndüren **area()** adında bir de metot içermektedir.

```
using System;

class Rect {
    public int width;
    public int height;

    public Rect(int w, int h) {
        width = w;
        height = h;
    }

    public int area() {
        return width * height;
    }
}

class UseRect {
    public static void Main() {
        Rect r1 = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);

        Console.WriteLine("Area of r1: " + r1.area());
    }
}
```

```

        Console.WriteLine("Area of r2: " + r2.area());
    }
}

```

Bildiğiniz gibi, bir metot içinde sınıfın diğer üyeleri, herhangi bir nesne veya sınıf niteleyicisi olmaksızın doğrudan erişilebilirler. Böylece, `area()` içindeki şu ifade,

```
return width * height;
```

`width` ve `height`'ın `area()`'yı çağıran nesne ile ilişkili kopyalarının birbiriyle çarpılacağı ve sonucun döndürüleceği anlamına gelir. Ancak, aynı ifade şu şekilde de yeniden yazılabilir:

```
return this.width * this.height;
```

Burada `this`, `area()`'nın üzerinde çağrıldığı nesneyle ilişkilidir. Yani `this.width`, söz konusu nesnenin `width` kopyasına ve `this.height` ise nesnenin `height` kopyasına karşılık gelir. Örneğin, `area()`, `x` adında bir nesne üzerinde çağrılmış olsaydı, önceki ifadede geçen `this`, `x` ile ilişkili olmuş olacaktı. `this` kullanmadan ifadeyi yazmak ise gerçekten kısadır.

`this` referansı kullanılarak yazılmış bütün bir `Rect` sınıfı şu şekildedir:

```

using System;

class Rect {
    public int width;
    public int height;

    public Rect(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public int area() {
        return this.width * this.height;
    }
}

class UseRect {
    public static void Main() {
        Rect r1 = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);

        Console.WriteLine("Area of r1: " + r1.area());
        Console.WriteLine("Area of r2: " + r2.area());
    }
}

```

Aslında, hiçbir C# programcısı `this`'i az önce gösterildiği şekliyle kullanmaz, çünkü bu şekilde hiçbir şey kazanılmaz, üstelik standart form daha kolaydır. Yine de, bunun bazı önemli kullanımları da vardır. Örneğin, C# söz dizimi bir parametrenin ya da yerel değişkenin bir ör-

nek değişkenle aynı isimde olmasına izin vermektedir. Böyle bir durum söz konusu olduğunda yerel isim örnek değişkeni gizler. Gizlenmiş örnek değişkene **this** kullanarak erişim sağlayabilirsiniz. Örneğin, tavsiye edilmeyen bir stil olmasına rağmen aşağıdaki kod, **Rect()** yapılandırıcısını yazmak için söz dizimsel açıdan geçerli bir yöntemdir;

```
public Rect (int width, int height) {  
    this.width = width;  
    this.height = height;  
}
```

Bu versiyonda, parametrelerin isimleri örnek değişkenlerin isimleriyle aynıdır, yani onları gizlemektedir; **this**, örnek değişkenleri “ortaya çıkarmak” için kullanılmıştır.

# DİZİLER VE KARAKTER KATARLARI

Bu bölümde C#'in veri tipleri konusuna geri dönülüyor. Diziler ve **string** tipi bu bölümde ele alınmaktadır. **foreach** döngüsü de ayrıca incelenmektedir.

## Diziler

Dizi (*array*), ortak bir isim ile anılan aynı tipteki değişkenler topluluğudur. Tek boyutlu diziler çok daha yaygın olmasına rağmen C#'ta diziler bir veya daha fazla boyutlu olabilir. Diziler çok çeşitli amaçlarda kullanır, çünkü diziler, ilgili değişkenleri bir araya getirip, gruplamak için uygun bir yöntem sunmaktadır. Örneğin, bir ay için günlük en yüksek sıcaklık kayıtlarını, hisse senedi fiyatlarının listesini veya programlama kitaplarından oluşan koleksiyonunu tutmak için bir dizi kullanabilirsiniz.

Dizinin sağladığı başlıca avantaj, verileri, kolaylıkla manipüle edilecek şekilde organize etmesidir. Örneğin, seçilmiş bir grup hissenin kar payını tutan bir diziye sahipseniz dizi üzerinden ilerleyerek ortalama gelirinizi kolaylıkla hesaplarsınız. Ayrıca diziler, verileri kolaylıkla sıralanabilecek şekilde düzenlerler.

C#'taki diziler típkı diğer programlama dillerindeki diziler gibi kullanılabilir olsalar da özel bir niteliğe sahiptirler; C#'taki diziler, nesne olarak uygulanırlar. Dizilerin anlatımının nesneler tanıtılana kadar ertelenmesinin bir nedeni budur. Dizileri nesne olarak uygulayarak bazı önemli avantajlar kazanılır. Kullanılmayan dışlerin anlamsız veri kapsamında toplanması bu sayede elde edilen avantajların en önemlilerinden biridir,

### Tek Boyutlu Diziler

Tek boyutlu bir dizi, ilgili değişkenlerin bir listesidir. Programlamada bu tür listeler yaygındır. Örneğin, bir ağ üzerindeki aktif kullanıcıların hesap numaralarını saklamak için tek boyutlu bir dizi kullanabilirsiniz. Bir beyzbol takımının mevcut vuruş ortalamalarını ise bir başka dizide saklayabilirsiniz.

Tek boyutlu bir dizi deklare etmek için şu genel yapıyı kullanacaksınız:

```
tip[] dizi-ismi = new tip[büyüklük];
```

Burada **tip**, dizinin temel tipini deklare etmektedir. Temel tip, diziyi oluşturan her bir elemanın veri tipini belirler. **tip**'in peşinden köşeli parantez geldiğine dikkat edin. Köşeli parantezler, tek boyutlu bir dizinin deklare edilmekte olduğuna işaret ederler. Dizinin tutacağı eleman sayısı **büyüklük** ile belirtilir. Diziler, nesne olarak gerçeklendikleri için bir dizi oluşturmak iki aşamalı bir süreçtir. İlk aşamada dizi referans değişkenini deklare edersiniz. İkinci aşamada dizi değişkenine bir bellek referansını atayarak dizi için bellekten yer ayırsınız. Böylece, C#'ta dizilere **new** operatörü kullanılarak dinamik olarak bellek alanı ayrırlıır.

#### NOT

C veya C++ biliyorsanız, dizilerin deklare ediliş şékléne özellikle dikkat edin. Özellikle, köşeli parantezler dizi ismini değil, tip ismini takip etmektedir.

İşte bir örnek. Aşağıdaki ifade 10 elemanlı bir **int** dizi tanımlıyor ve bunu **sample** adında bir dizi referans değişkenine bağlıyor.

```
int[] sample = new int[10];
```

Bu deklarasyon tipki bir nesne deklarasyonu gibi çalışır. **sample** değişkeni, **new** ile tahsis edilen bellek alanına bir referans içerir. Bu bellek alam, **int** tipinde 10 elemanı tutabilecek büyüküktedir.

Nesnelerde olduğu gibi, yukarıdaki deklarasyonu da ikiye bölmek mümkündür. Örneğin:

```
int[] sample;
sample = new int[10];
```

Bu durumda, **sample** oluşturulduğunda herhangi bir fiziksel nesneye ilişkili değildir. Ancak ikinci ifade gerçeklendikten sonra **sample** bir diziye bağlanır.

Bir dizinin içindeki elemanlara tek tek dizi indeksi yardımıyla erişilir. *Dizi indeksi (array index)*, bir elemanın dizi içindeki konumunu ifade eder. C#'ta dizilerin tümü, ilk eleman indeksi olarak sıfır değerine sahiptir. **sample**'ın 10 elemanı olduğu için **sample** 0 ile 9 arasındaki indeks değerlerini içerir. Bir diziyi indekslemek için istediğiniz elemanın numarasını köşeli parantez içinde belirtin. Böylece, **sample**'ın ilk elemanı **sample[0]**, son elemanı ise **sample[9]** olur. Örneğin, aşağıdaki program **sample**'a 0 ile 9 arasındaki sayıları yükler:

```
// Tek boyutlu bir diziyi tanitir.

using System;

class ArrayDemo {
    public static void Main() {
        int[] sample = new int[10];
        int i;

        for(i = 0; i < 10; i = i+1)
            sample[i] = i;

        for(i = 0; i < 10; i = i+1)
            Console.WriteLine("sample[" + i + "]: " + sample[i]);
    }
}
```

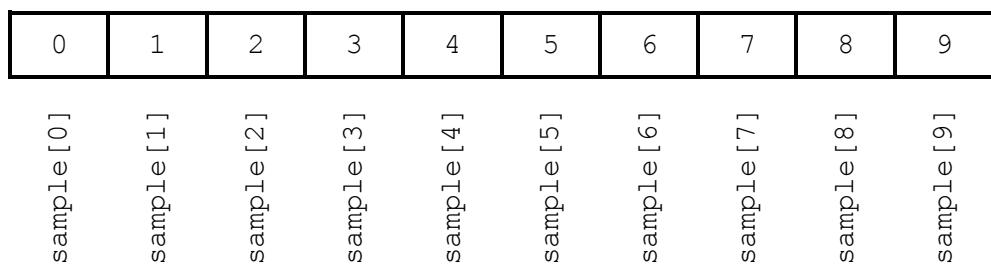
Programın çıktısı aşağıda gösterilmiştir:

```
sample[0]: 0
sample[1]: 1
sample[2]: 2
sample[3]: 3
sample[4]: 4
sample[5]: 5
sample[6]: 6
sample[7]: 7
sample[8]: 8
```

```
sample[9]: 9
```

Kavramsal olarak, **sample** dizisi aşağıdaki gibi görünür:

Diziler programlamada sıkça kullanılır, çünkü birbiriyle ilişkili çok sayıda değişkenle kolaylıkla ilgilenmenize imkan verirler. Örneğin, aşağıdaki program, **for** döngüsü yardımıyla dizi elemanları üzerinde ilerleyerek, **nums** dizisinde saklanan birtakım değerlerin ortalamasını bulur:



```
// Bir takim degerlerin ortalamasini hesaplar.

using System;

class Average {
    public static void Main() {
        int[] nums = new int[10];
        int avg = 0;

        nums[0] = 99;
        nums[1] = 10;
        nums[2] = 100;
        nums[3] = 18;
        nums[4] = 78;
        nums[5] = 23;
        nums[6] = 63;
        nums[7] = 9;
        nums[8] = 87;
        nums[9] = 49;

        for(int i=0; i < 10; i++)
            avg = avg + nums[i];

        avg = avg / 10;

        Console.WriteLine("Average: " + avg);
    }
}
```

Programdan elde edilen çıktı aşağıda gösterilmiştir:

```
Average: 53
```

## Diziyi İlk Kullanıma Hazırlamak

Önceki programda, on ayrı atama ifadesi kullanılarak `nums` dizisine elle değer atanmıştı. Bu tamamen geçerli olmasına rağmen bunu gerçekleştirmenin daha kolay bir yolu da vardır. Diziler tanımlanırken, ilk kullanıma hazırlanabilir. Tek boyutlu bir diziyi ilk kullanıma hazırlamanın genel yapısı şu şekildedir:

```
tip[ ] dizi-ismi = { değer1, değer2, değer3, ..., değerN };
```

Burada, ilk değerler `değer1` ile `değerN`'ye kadar yer alan değerlerle belirtilir. Bu değerler soldan sağa doğru indeks sırasına göre sırayla atanır. C#, belirttiğiniz ilk değerleri tutabilecek büyülükte bir diziyi, otomatik olarak bellek alanı ayırır. `new` operatörünü kullanmaya gerek yoktur. Örneğin, **Average** programını işte daha iyi yazmanın bir yolu şöyledir:

```
// Bir takım değerlerin ortalamasını hesaplar.
```

```
using System;

class Average {
    public static void Main() {
        int[] nums = { 99, 10, 100, 18, 78, 23,
                      63, 9, 87, 49 };
        int avg = 0;

        for(int i = 0; i < 10; i++)
            avg = avg + nums[i];

        avg = avg / 10;

        Console.WriteLine("Average: " + avg);
    }
}
```

İlginc bir konu, gerekli olmamasına rağmen bir diziyi ilk kullanıma hazırlarken `new` operatörünü kullanabilmenizdir. Örneğin, yukarıdaki programda `nums`'ı ilk kullanıma hazırlamak için bu, uygun fakat gereksiz bir yoldur:

```
int[] nums = new int[] { 99, 10, 100, 18, 78, 23,
                        63, 9, 87, 49 };
```

Burada gereksiz olmasına rağmen, diziyi `new` ile ilk kullanıma hazırlamak, önceden mevcut bir dizi referans değişkenine yeni bir dizi atayacağınız zaman işe yarar. Örneğin:

```
int[] nums;
nums = new int[] { 99, 10, 100, 18, 78, 23,
                  63, 9, 37, 49 };
```

Bu durumda, `nums` ilk ifadede deklare edilir ve ikincisinde ilk değerlerini alır.

Son bir husus: Bir diziyi ilk kullanıma hazırlarken dizi büyülüüğünü açıkça belirtmeye izin verilir, fakat büyülüük, atanacak değer sayısı ile uyuşmalıdır. Örneğin, `nums`'ı ilk kullanıma hazırlamanın bir başka yolu şöyledir;

```
int[] nums = new int[10] { 99, 10, 100, 18, 78, 23,
                         63, 9, 87, 49 };
```

Bu deklarasyonda, `nums`'ın büyülüüğü açıkça **10** olarak belirtilmiştir.

## Sınırlar Zorlanır

C#'ta dizi sınırlarına sıkı sıkıya uymak gereklidir. Dizinin üst sınırını aştığınızda yada alt sınırının altına düştüğünüzde çalışma zamanı hatası meydana gelir. Bunu kendi kendinize doğrulamak isterseniz, dizinin üst sınırını kasti olarak aşan aşağıdaki programı deneyin:

```
// Dizinin ust sinirinin asilmasini gosterir.

using System;

class ArrayErr {
    public static void Main() {
        int[] sample = new int[10];
        int i;

        // bir dizi asimi olustur

        for(i = 0; i < 100; i = i+1)
            sample[i] = i;
    }
}
```

`i`, **10**'a ulaşır ulaşmaz **IndexOutOfRangeException** (indeksin menzil dışına çıktığını belirten bir kural dışı durum) meydana gelir ve program sona erer.

## Çok Boyutlu Diziler

Programlamada en yaygın olarak tek boyutlu diziler kullanıyor olmasına rağmen, çok boyutlu diziler de kesinlikle ender olarak kullanılmaz. *Çok boyutlu bir dizi* (multidimensional array), iki veya daha fazla boyutu olan dizidir ve dizinin her elemanına iki veya daha fazla indeks kombinasyonu ile erişilir.

### İki Boyutlu Diziler

Çok boyutlu dizinin en basit şekli iki boyutlu dizidir. İki boyutlu dizide herhangi bir spesifik elemanın konumu iki indeks ile belirtilir. İki boyutlu bir diziyi bilgi içeren bir tablo olarak düşünürseniz, indekslerden biri satırı, diğeri de sütunu gösterir,

**table** adında ve **10x20** boyutunda iki boyutlu bir tamsayı dizisi deklare etmek için şu şekilde yazarsınız:

```
int[,] table = new int[10, 20];
```

Deklarasyona özellikle dikkat edin- İki boyutun birbirinden virgül ile ayrıldığına çok dikkat edin. Deklarasyonun ilk bölümünde, aşağıdaki gibi sözdizimi kullanılır:

```
[,]
```

iki boyutlu dizi referans değişkeninin oluşturulmakta olduğuna işaret eder. **new** kullanılarak dizi için gerçekten bellek alanı ayrıldığında, şu sözdizimi kullanılır:

```
int[10, 20]
```

Bu, **10x20** boyutunda bir dizi oluşturur ve boyutlar yine virgül ile ayrılır.

İki boyutlu dizinin bir elemanına erişmek için, her iki indeksi de virgül ile ayırarak belirtmelisiniz. Örneğin, **table** dizisinin **3, 5** konumuna **10** değerini atamak için şöyle yazarsınız:

```
table[3, 5] = 10;
```

Aşağıda tamamlanmış bir örnek görülmektedir. Bu örnekte, iki boyutlu bir diziye 1'den 12'ye kadar sayılar yüklenmekte ve dizinin içeriği ekranda gösterilmektedir.

```
// iki boyutlu bir diziyi gösterir.

using System;

class TwoD {
    public static void Main() {
        int t, i;
        int[,] table = new int[3, 4];

        for(t = 0; t < 3; ++t) {
            for(i = 0; i < 4; ++i) {
                table[t, i] = (t * 4) + i + 1;
                Console.Write(table[t,i] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Bu örnekte **table[0,0]**, 1 değerini; **table[0,1]**, 2 değerini; **table[0,3]**, 3 değerini alır vs. **table[2, 3]**'ün değeri 12 olacaktır. Kavramsal olarak, dizi Şekil 7 l'deki gibi görünecektir.

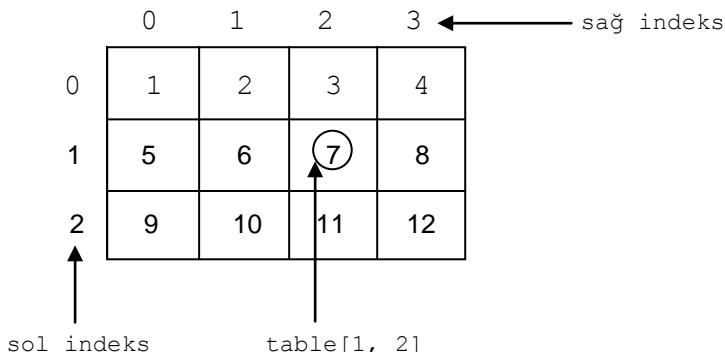
## NOT

C, C++ veya Java biliyorsanız, çok boyutlu dizileri deklare ederken veya bu dizi elemanlarına erişirken dikkatli olun. Bu dillerde, dizi boyutlarının ve indekslerinin her biri kendi köşeli parantezleri içinde belirtilir. C#'ta ise virgül kullanılarak boyutlar birbirinden ayrılır.

## Üç veya Daha Fazla Boyutlu Diziler

C#, ikiden fazla boyutlu dizilere imkan vermektedir. Çok boyutlu bir dizi deklarasyonunun genel yapısı şu şekildedir:

```
tip[, ..., ] isim = new tip[büyüklük1, büyüklük2, ..., büyüklükN];
```



**ŞEKİL 7.1:** *TwoD programıyla oluşturulan table dizisinin kavramsal görünümü*

Örneğin, aşağıdaki deklarasyon **4x10x3** büyülüüğünde üç boyutlu bir tamsayı dizisi oluşturur:

```
int[, ,] multidim = new int[4, 10, 3];
```

**multidim**'in **2, 4, 1** konumundaki elemanına **100** değerini atamak için şu ifadeyi kullanın:

```
multidim[2, 4, 1] = 100;
```

İşte size, **3x3x3** matris değerlerini tutmak için üç boyutlu bir dizi kullanan bir program. Program, daha sonra, küpün köşegenlerinden biri üzerindeki değerlerin toplamını bulmaktadır.

```
// 3x3x3'luk matrisin kösegeni üzerindeki sayıların toplamını bulur.

using System;

class ThreeDMatrix {
    public static void Main() {
        int[, ,] m = new int[3, 3, 3];
        int sum = 0;
        int n = 1;

        for(int x = 0; x < 3; x++)
            for(int y = 0; y < 3; y++)
                for(int z = 0; z < 3; z++)
                    m[x, y, z] = n++;

        sum = m[0,0,0] + m[1,1,1] + m[2,2,2];

        Console.WriteLine("Sum of first diagonal: " + sum);
    }
}
```

```

    }
}
```

Cıktı aşağıdaki gibidir:

```
Sum of first diagonal: 42
```

## Çok Boyutlu Dizileri İlk Kullanıma Hazırlamak

Çok boyutlu bir dizinin ilk kullanıma hazırlanması, her boyutun ilk değer listesini kendi küme parantezleri içine alarak gerçekleştirilir. Örneğin, iki boyutlu bir dizi için ilk kullanıma hazırlama işleminin genel yapısı aşağıda gösterilmiştir:

```

tip[,]  dizi_ismi = {
    { değer, değer, değer, ..., değer },
    { değer, değer, değer, ..., değer },
    .
    .
    .
    { değer, değer, değer, ..., değer },
};
```

Burada **değer**, atanacak ilk değeri gösterir. İç blokların her biri bir satır karşılık gelir. Her satır içinde, ilk değer dizinin ilk konumunda, ikinci değer ikinci konumda saklanacaktır vs. İlk değer bloklarının virgül ile ayrıldığına ve kapanış küme parantezinden sonra noktalı virgül geldiğine dikkat edin.

Örneğin, aşağıdaki program **sqrs** adında bir diziye **1** ile **10** arasındaki sayıları ve bunların karelerini ilk değer olarak atamaktadır:

```

// İki boyutlu bir diziyi ilk kullanıma hazırla.

using System;

class Squares {
    public static void Main() {
        int[,] sqrs = {
            { 1, 1 },
            { 2, 4 },
            { 3, 9 },
            { 4, 16 },
            { 5, 25 },
            { 6, 36 },
            { 7, 49 },
            { 8, 64 },
            { 9, 81 },
            { 10, 100 }
        };
        int i, j;

        for(i = 0; i < 10; i++) {
            for(j = 0; j < 2; j++)
                Console.Write(sqrs[i,j] + " ");
        }
    }
}
```

```

        Console.WriteLine();
    }
}
}

```

Programın çıktısı şöyle olur:

```

1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100

```

## Düzensiz Diziler

Önceki örneklerde, iki boyutlu bir dizi oluşturduğunuzda C#'ta *dikdörtgensel dizi* olarak adlandırılan bir dizi oluşturmuş oluyordunuz. İki boyutlu bir diziyi bir tablo olarak düşünürsek, dikdörtgensel bir dizi, satırların her birinin bütün bir dizi için aynı uzunlukta olduğu iki boyutlu bir dizidir. Ancak, C#'ta ayrıca *düzensiz dizi* (*jagged array*) denilen özel tipte iki boyutlu dizi oluşturmanıza da imkan verilir. Bir düzensiz dizi, her biri farklı uzunlukta dizilerin oluşturduğu bir dizidir. Böylece, satır uzunlukları aynı olmayan tabloları oluşturmak için bir düzensiz dizi kullanılabilir.

Düzensiz diziler, boyutların her biri bir çift köşeli parantez kullanılarak gösterilecek şekilde deklare edilir. Örneğin, iki boyutlu düzensiz bir dizi deklare etmek için, şu genel yapıyı kullanacaksınız:

```
tip[][] dizi-ismi = new tip[büyüklük] [ ];
```

Burada ***büyüklük***, dizideki satır sayısını gösterir. Satırların kendileri için yer ayrılmamıştır. Bunun yerine, satırlar için tek tek yer ayrılır. Bu, her satırın uzunluğunun farklımasına imkan verir. Örneğin aşağıdaki kod, **jagged**'in ilk boyutu için deklare edilirken bellekte yer ayırrı. Sonra, ikinci boyut için ayrıca yer ayırrı.

```

int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];

```

Bu ifade sekansı gerçeklendikten sonra **jagged** şu şekilde görünür:

jagged[0][0]	jagged[0][1]	jagged[0][2]	jagged[0][3]	
jagged[1][0]	jagged[1][1]	jagged[1][2]		
jagged[2][0]	jagged[2][1]	jagged[2][2]	jagged[2][3]	jagged[2][4]

Düzensiz dizilerin isimlerinin nereden geldiği açıkça görülüyor!

Bir düzensiz dizi oluşturulduğu andan itibaren dizinin elemanlarına, her indeks kendi köşeli parantezi içinde belirtilerek erişilebilir Örneğin, **jagged**'in **2, 1** konumundaki elemanına **10** değerini atamak için şu ifadeyi kullanırsınız:

```
jagged[2][1] = 10;
```

Bunun bir dikdörtgensel dizinin elemanına erişmek için kullanılan söz diziminden farklı olduğuna dikkat edin.

Aşağıdaki program iki boyutlu düzensiz bir dizi oluşturmayı gösteriyor:

```
// Duzensiz dizileri gosterir.

using System;

class Jagged {
    public static void Main() {
        int[][] jagged = new int[3][];
        jagged[0] = new int[4];
        jagged[1] = new int[3];
        jagged[2] = new int[5];

        int i;

        // birinci diziye deger yukle
        for(i = 0; i < 4; i++)
            jagged[0][i] = i;

        // ikinci diziye deger yukle
        for(i = 0; i < 3; i++)
            jagged[1][i] = i;

        // ucuncu diziye deger yukle
        for(i = 0; i < 5; i++)
            jagged[2][i] = i;

        // birinci dizideki degerleri goster
        for(i = 0; i < 4; i++)
            Console.Write(jagged[0][i] + " ");

        Console.WriteLine();
    }
}
```

```

// ikinci dizideki degerleri goster
for(i = 0; i < 3; i++)
    Console.Write(jagged[1][i] + " ");

Console.WriteLine();

// ucuncu dizideki degerleri goster
for(i = 0; i < 5; i++)
    Console.Write(jagged[2][i] + " ");

Console.WriteLine();
}
}

```

Çıktı aşağıdaki gibidir:

```

0 1 2 3
0 1 2
0 1 2 3 4

```

Düzensiz diziler, uygulamaların tümünde kullanılmaz, ama bazı durumlarda etkili olabilirler. Örneğin içi çok fazla dolu olmayan (yani elemanların tümü kullanılmayacak olan) çok büyük bir iki boyutlu diziye ihtiyaç duyuyorsanız, düzensiz dizi mükemmel bir çözüm olabilir.

Son bir husus: Düzensiz diziler, dizilerden oluşan diziler olduğu için her dizinin aynı cinsteki olması gereklidir. Söz gelisi, aşağıdaki ifade iki boyutlu dizilerden oluşan bir dizi tanımlar:

```
int[][] jagged = new int[3][,];
```

Bir sonraki ifade, **jagged[0]**'a **4x2**'lik bir diziyi gösteren bir referans atar:

```
jagged[0] = new int[4, 2];
```

Aşağıdaki ifade, **jagged[0][1,0]**'a bir değer atar:

```
jagged[0][1,0] = i;
```

## Dizi Referanslarını Atamak

Tıpkı diğer nesneler gibi, bir dizi referans değişkenini bir diğerine atadığınızda sadece değişkenin ilişkili olduğu nesneyi değiştiriyorsunuz. Ne dizinin bir kopyasının çıkarılmasına, ne de dizinin içeriğinin bir başka diziye kopyalanmasına neden olmuyorsunuz. Örneğin, aşağıdaki programı ele alın:

```

// dizi referans degiskenlerini atamak.

using System;

class AssignARef {
    public static void Main() {
        int i;

```

```

int[] nums1 = new int[10];
int[] nums2 = new int[10];

for(i = 0; i < 10; i++) nums1[i] = i;

for(i = 0; i < 10; i++) nums2[i] = -i;

Console.WriteLine("Here is nums1: ");
for(i = 0; i < 10; i++)
    Console.Write(nums1[i] + " ");
Console.WriteLine();

Console.WriteLine("Here is nums2: ");
for(i = 0; i < 10; i++)
    Console.Write(nums2[i] + " ");
Console.WriteLine();

nums2 = nums1; // artık nums2, nums1 ile ilişkili

Console.WriteLine("Here is nums2 after assignment: ");
for(i = 0; i < 10; i++)
    Console.Write(nums2[i] + " ");
Console.WriteLine();

// simdi nums2 üzerinden, nums1 dizisi üzerinde işlem yap
nums2[3] = 99;

Console.WriteLine("Here is nums1 after change through nums2: ");
for(i = 0; i < 10; i++)
    Console.Write(nums1[i] + " ");
Console.WriteLine();
}
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9

```

Çıktıdan görüldüğü gibi, **nums2**'ye **nums1**'in atanmasından sonra her iki dizi referans değişkenleri de aynı nesneyi göstermekteler.

## Length Özelliğini Kullanmak

C#'ta diziler nesne olarak gerçeklendikleri için bu durum birtakım avantajlar doğurur. Bu avantajlardan biri **Length** özelliğinden kaynaklanmaktadır. **Length**, dizinin tutabileceği eleman sayısını içerir. Böylece, her dizi dizinin uzunluğunu içeren bir alanı da beraberinde taşır. İşte, bu özelliği gösteren bir program:

```
// Length dizi ozelligini kullan.
```

```

using System;

class LengthDemo {
    public static void Main() {
        int[] nums = new int[10];

        Console.WriteLine("Length of nums is " + nums.Length);

        // nums'a ilk deger atamak icin Length'i kullan
        for (int i = 0; i < nums.Length; i++)
            nums[i] = i * i;

        // simdi, nums'i gostermek icin Length'i kullan
        Console.Write("Here is nums: ");
        for(int i = 0; i < nums.Length; i++)
            Console.Write(nums[i] + " ");

        Console.WriteLine();
    }
}

```

Bu program ekranda aşağıdaki çıktıyı gösterir:

```

Length of nums is 10
Here is nums: 0 1 4 9 16 25 36 49 64 81

```

**LengthDemo**'nun içinde **nums.Length**'in, **for** döngüsünde söz konusu tekrar sayısını kontrol etmek için döngü tarafından ne şekilde kullanıldığına dikkat edin. Her dizi kendi uzunluğunu birlikte taşıdığı için, dizinin büyüklüğünü elle takip etmek yerine bu bilgiyi kullanabilirsiniz. **Length**'in değerinin, gerçekte kullanımda olan eleman sayısı ile ilgili olmadığını hatırlınızda tutun. **Length**, dizinin tutabilecegi eleman sayısını içerir.

Çok boyutlu dizilerin uzunluğu elde edilirken dizinin tutabileceği toplam eleman sayısı döndürülür. Örneğin;

```

// Uc boyutlu bir dizi uzerinde Length dizi ozelligini kullan.

using System;

class LengthDemo3D {
    public static void Main() {
        int[, ,] nums = new int[10, 5, 6];

        Console.WriteLine("Length of nums is " + nums.Length);
    }
}

```

Çıktı aşağıdaki gibidir:

```

Length of nums is 300

```

Cıktının da doğruladığı gibi, **Length**, **nums**'in tutabileceği eleman sayısını elde eder. Bu örnekte eleman sayısı **300**'dür (**10x5x6**). Belirli bir boyutun uzunluğunu elde etmek mümkün değildir.

**Length** özelliğinin dahil edilmesi belirli tipteki dizi işlemlerinin daha kolay - ve daha güvenli - gerçekleştirilmesini sağladığı için bu sayede birçok algoritma sadeleştirilmektedir. Söz gelisi, aşağıdaki program, **Length** özelliğini kullanarak bir dizinin elemanlarını arkadan öne doğru bir başka diziye kopyalayıp dizinin içeriğini ters çevirmektedir.

```
// Diziyi ters çevir.

using System;

class RevCopy {
    public static void Main() {
        int i, j;
        int[] nums1 = new int[10];
        int[] nums2 = new int[10];

        for(i = 0; i < nums1.Length; i++) nums1[i] = i;

        Console.WriteLine("Original contents: ");
        for(i = 0; i < nums2.Length; i++)
            Console.Write(nums1[i] + " ");

        Console.WriteLine();

        // nums1'den nums'ye tersten kopyalama
        if(nums2.Length >= nums1.Length) /* nums2'nin yeterli
                                         uzunlukta olduğundan
                                         emin ol */
            for(i=0, j=nums1.Length-1; i < nums1.Length; i++, j--)
                nums2[j] = nums1[i];

        Console.WriteLine("Reversed contents: ");
        for(i = 0; i < nums2.Length; i++)
            Console.Write(nums2[i] + " ");

        Console.WriteLine();
    }
}
```

Cıktı şöyled olur:

```
Original contents: 0 1 2 3 4 5 6 7 8 9
Reversed contents: 9 8 7 6 5 4 3 2 1 0
```

Bu örnekte **Length**, iki önemli işlevi gerçekleştiriyor. Birincisi; **Length**, hedef dizinin, kaynak dizinin içeriğini tutabilecek büyülükte olduğunu doğrulamak için kullanılıyor. İkincisi; **Length**, ters kopyalama işlemini gerçekleştiren **for** döngüsü için sonlanma koşulu sağlıyor. Bu basit örnekte dizilerin büyülüğu elbette kolaylıkla biliniyor, ama aynı yöntem daha zorlayıcı, çok çeşitli durumlara uygulanabilir.

## Düzensiz Dizilerde Length Kullanmak

`Length`'in düzensiz dizilerle birlikte kullanılması özel bir durumdur. Bu durumda, her dizinin tek tek uzunluğunu elde etmek mümkün olur. Örneğin, dört düğümlü bir ağ üzerinde CPU etkinliğini simüle eden aşağıdaki programı ele alın:

```
// Duzensiz dizilerle Length kullanimini gosterir.

using System;

class Jagged {
    public static void Main() {
        int[][] network_nodes = new int[4][];
        network_nodes[0] = new int[3];
        network_nodes[1] = new int[7];
        network_nodes[2] = new int[2];
        network_nodes[3] = new int[5];

        int i, j;

        // Bazı sahte CPU kullanım verileri uyduralim
        for(i = 0; i < network_nodes.Length; i++)
            for(j = 0; j < network_nodes[i].Length; j++)
                network_nodes[i][j] = i * j + 70;

        Console.WriteLine("Total number of network nodes: " +
                           network_nodes.Length + "\n");

        for(i = 0; i < network_nodes.Length; i++) {
            for(j = 0; j < network_nodes[i].Length; j++) {
                Console.Write("CPU usage at node. " + i + " CPU " +
                             j + "; ");
                Console.Write(network_nodes[i][j] + "% ");
                Console.WriteLine();
            }
            Console.WriteLine();
        }
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Total number of network nodes: 4
CPU usage at node 0 CPU 0: 70%
CPU usage at node 0 CPU 1: 70%
CPU usage at node 0 CPU 2: 70%

CPU usage at node 1 CPU 0: 70%
CPU usage at node 1 CPU 1: 71%
CPU usage at node 1 CPU 2: 72%
CPU usage at node 1 CPU 3: 73%
CPU usage at node 1 CPU 4: 74%
CPU usage at node 1 CPU 5: 75%
CPU usage at node 1 CPU 6: 76%
```

```
CPU usage at node 2 CPU 0: 70%
CPU usage at node 2 CPU 1: 72%
```

```
CPU usage at node 3 CPU 0: 70%
CPU usage at node 3 CPU 1: 73%
CPU usage at node 3 CPU 2: 76%
CPU usage at node 3 CPU 3: 79%
CPU usage at node 3 CPU 4: 82%
```

**network\_nodes** düzensiz dizisi üzerinde **Length**'in ne şekilde kullanıldığına özellikle dikkat edin. Hatırlarsanız, iki boyutlu düzensiz dizi, dizilerden oluşan bir dizidir. Böylece, aşağıdaki ifade kullanıldığında, **network\_nodes** içinde saklanan dizilerin sayısı elde edilir;

```
network_nodes.Length
```

Bu örnekte bu sayı **4**'tür. Düzensiz bir dizi içinde herhangi bir dizinin uzunluğunu elde etmek için aşağıdaki gibi bir deyim kullanırsınız:

```
network_nodes[0].Length
```

Yukarıdaki deyim, ilk dizinin uzunluğunu elde eder.

## foreach Döngüsü

C#'ta **foreach** olarak adlandırılan bir döngünün tanımlı olduğundan, Bölüm 5'te bahsedilmişti, fakat bu ifadeyle ilgili açıklamalar ileriki konulara bırakılmıştı. İşte bu açıklamaların zamanı artık geldi.

**foreach** döngüsü, bir *koleksiyonun* elemanları üzerinde ilerleyerek tekrarlamak için kullanılır. Koleksiyon, bir grup nesneden ibarettir. C#'ta birkaç tip koleksiyon tanımlıdır. Bunlardan biri de dizidir. **foreach**'in genel yapısı aşağıdaki gibidir:

```
foreach(tip değişken-ismi in koleksiyon) ifade;
```

Burada **tip değişken-ismi**, **foreach** iterasyon yaparken koleksiyonun elemanlarından değer alacak olan *iterasyon değişkeninin* tipini ve ismini belirtir. Üzerinde ilerlenecek olan koleksiyon, **koleksiyon** ile belirtilir. Bu örnekte koleksiyon, bir dizidir. Bu nedenle **tip**, dizinin temel tipi ile aynı (veya uyumlu) olmalıdır. Hatırlanması gereken önemli bir nokta, diziler söz konusu olduğu sürece iterasyon değişkeninin salt okunur olduğunu. Yani, iterasyon değişkenine yeni bir değer atayarak dizinin içeriğini değiştiremezsiniz.

İşte, **foreach** kullanılan basit bir örnek. Bu örnekte, bir tamsayı dizisi oluşturuluyor ve bu dizi ilk kullanıma hazırlanıyor. Daha sonra, bu değerler bir yandan toplanırken diğer yandan ekranda gösteriliyor.

```
// foreach dongusunu kullanir.

using System;
```

```

class ForeachDemo {
    public static void Main() {
        int sum = 0;
        int[] nums = new int[10];

        // nums'a bazi degerler ata
        for(int i = 0; i < 10; i++)
            nums[i] = i;

        // degerleri gostermek ve toplamak icin foreach kullan
        foreach(int x in nums) {
            Console.WriteLine("Value is: " + x);
            sum += x;
        }
        Console.WriteLine("Summation: " + sum);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Value is: 0
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Summation: 45

```

Bu çıktıdan görüldüğü gibi; **foreach**, bir dizi içinde en düşük indeksten en yüksek indekse doğru tekrarlar.

**foreach** döngüsü, bir dizi içindeki tüm elemanlar incelenene kadar iterasyonu sürdürmesine rağmen, **foreach** döngüsünü **break** kullanarak daha erken sona erdirmek mümkündür. Örneğin, bu program **nums**'ın sadece ilk beş elemanını toplar:

```

// foreach ile birlikte break kullan.

using System;

class ForeachDemo {
    public static void Main() {
        int sum = 0;
        int[] nums = new int[10];

        // nums'a bazi degerler ver
        for(int i = 0; i < 10; i++)
            nums[i] = i;

        // degerleri toplamak ve gostermek icin foreach kullan
        foreach(int x in nums) {

```

```

        Console.WriteLine("Value is: " + x);
        sum += x;
        if(x == 4) break; // 4 elde edilince donguyu durdur
    }
    Console.WriteLine("Summation of first 5 elements: " + sum);
}
}

```

Elde edilen çıktı şöyle olur:

```

Value is: 0
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Summation of first 5 elements: 10

```

Gördüğü gibi, **foreach** döngüsü beşinci eleman elde edilince durur.

**foreach**, çok boyutlu diziler üzerinde de çalışır. Satır sırasına göre ilk satırdan sonuncu satıra kadar ilgili elemanları döndürür.

```

// foreach'i iki boyutlu dizi üzerinde kullan.

using System;

class ForeachDemo2 {
    public static void Main() {
        int sum = 0;
        int[,] nums = new int[3,5];

        // nums'a bazi degerler ver
        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                nums[i,j] = (i+1)*(j+1);

        // degerleri toplamak ve göstermek icin foreach kullan
        foreach(int x in nums) {
            Console.WriteLine("Value is: " + x);
            sum += x;
        }
        Console.WriteLine("Summation: " + sum);
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6

```

```

Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90

```

**foreach**, bir dizi üzerinde, yalnızca, dizinin başından sonuna, sırayla ilerleyebildiği için, bu döngünün kullanımının kısıtlı olduğunu düşünebilirsiniz. Oysa, bu doğru değildir. Çok sayıda algoritmada tam olarak bu mekanizmaya gerek duyulur. Bu algoritmaların en yaygın olanı da arama algoritmalarıdır. Örneğin, aşağıdaki program bir dizi içinde bir değer aramak için **foreach** döngüsü kullanmaktadır. Değer bulunursa döngü sona erer.

```

// foreach kullanarak bir diziyi aramak.

using System;

class Search {
    public static void Main() {
        int[] nums = new int[10];
        int val;
        bool found = false;

        // nums'a bazi değerler ver
        for(int i = 0; i < 10; i++)
            nums[i] = i;

        val = 5;

        // nums'da bir anahtar değer aramak için foreach kullan
        foreach(int x in nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            Console.WriteLine("Value found!");
    }
}

```

Diğer **foreach** uygulamaları arasında ortalama hesabı, bir kümenin en düşük ve en yüksek değerlerini bulmak, çift değerleri aramak gibi uygulamalar yer almaktır. Elinizdeki kitapta daha sonra göreceğiniz gibi **foreach**, diğer tipteki koleksiyonlar üzerinde işlem yaparken de özellikle kullanışlıdır.

## Karakter Katarları

Programlamaya ilgili görüşler günden güne değişse de, C#'ın en önemli veri tiplerinden biri **string**'dir. **string**, karakter katarlarını tanımlar ve destekler. Diğer birçok programlama dilinde **string**, bir karakter dizisidir. Oysa, C#'ta böyle bir durum söz konusu değildir. C#'ta **string**, bir nesnedir. Yani, **string** bir referans tipidir. C#'ta **string** standart bir veri tipi olmasına rağmen, **string** ile ilgili açıklamaların sınıf ve nesneler tanıtılanca kadar beklemesi gerekiydi.

Aslında siz, **string** sınıfını Bölüm 2'den beri kullanmactasınız, fakat bunu bilmiyordunuz. Bir karakter katarı literalı oluşturduğunuz zaman aslında bir **string** nesnesi oluşturmuş oluyorsunuz. Örneğin, şu ifadede

```
Console.WriteLine("In C#, strings are objects.");
```

“**In C#, strings are objects.**” karakter katarı, C# tarafından otomatik olarak bir **string** nesnesi haline getirilir. Yani, önceki programlarda **string** sınıfının kullanımı “arka plandaydı”. Bu bölümde bunların açıkça nasıl ele alındığını öğreneceksiniz.

## Karakter Katarlarını Oluşturmak

Bir **string** oluşturmanın en kolay yolu, karakter katarı literalı kullanmaktır. Örneğin, burada **str**, bir **string** referans değişkenidir. **str**'a, bir karakter katarı literalini gösteren bir referans atanmaktadır:

```
string str = "C# strings are powerful.;"
```

Bu örnekte, **str**'a ilk değer olarak “**C# strings are powerful**” karakter sekansı atanmaktadır.

Aynca, bir **char** dizisinden de bir **string** oluşturabilirsiniz. Örneğin:

```
char[] charray = {'t', 'e', 's', 't'};  
string str = new string(charray);
```

Bir kez bir **string** nesnesi oluşturuktan sonra, tırnak içinde karakter katarlarına izin verilen her yerde bunu kullanabilirsiniz. Örneğin, aşağıdaki örnekte gösterildiği gibi, bir **string** nesnesini **WriteLine()**'a argüman olarak kullanabilirsiniz:

```
// Karakter katarlarını tanitir.  
  
using System;  
  
class StringDemo {  
  
    public static void Main() {  
  
        char[] charray = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.'};  
        string str1 = new string(charray);  
    }  
}
```

```

        string str2 = "Another string.";

        Console.WriteLine(str1);
        Console.WriteLine(str2);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

A string.
Another string.

```

## Karakter Katarları Üzerinde İşlem Yapmak

**string** sınıfı, karakter katarları üzerinde işlem yapan birkaç metot içerir. Tablo 7.1'de bunların birkaçı gösterilmektedir.

**string** tipi, karakter katarının uzunluğunu içeren **Length** özelliğine de sahiptir.

Bir karakter katarının içinde yer alan karakterlerin değerlerini tek tek elde etmek için sadece indeks kullanmanız yeterlidir. Örneğin,

```

string str = "test";
Console.WriteLine(str[0]);

```

Bu örnekte, "**test**"in ilk karakteri olan '**t**' ekranda gösterilir. Tıpkı diziler gibi, karakter katarı indeksleri de sıfırdan başlar. Ancak önemli bir husus, indeks kullanarak bir karakter katarı içindeki bir karaktere yeni bir değer atayamıyor olmanızdır. İndeks yalnızca karakterleri elde etmek için kullanılabilir.

İki karakter katarının eşit olup olmadığını test etmek için **==** operatörünü kullanabilirsiniz. Normal olarak, **==** operatörü, nesne referanslarına uygulandığında her iki referansın da aynı nesneye bağlı olduğunu belirler. Ancak bu durum, **string** tipindeki nesneler için farklıdır. **==** operatörü iki **string** referansına uygulandığında, **string** içeriklerinin kendileri eşitlik açısından karşılaştırılır. Aynı durum **!=** operatörü için de geçerlidir; **string** nesneleri karşılaştırıldığında **string**'lerin içerikleri karşılaştırılır. Ancak, diğer ilişkisel operatörler, söz gelişi **<**, **>** ya da **=**, tıpkı diğer nesne tiplerinde olduğu gibi referansları karşılaştırırlar.

**TABLO 7.1: En Yaygın Karakter Katarı Düzenleyici Metotları**

Metot	Anlamı
<b>static string Copy(string str)</b>	<b>str</b> 'ın bir kopyasını döndürür.
<b>int CompareTo(string str)</b>	Metodu çağrıran karakter katarı <b>str</b> 'dan küçükse sıfırdan küçük bir değer döndürür; <b>str</b> 'dan büyükse, sıfırdan büyük bir değer döndürür; karakter katarları eşitse, sıfır döndürür.

<b>int IndexOf(string str)</b>	Metodu çağrıran karakter katarı içinde <b>str</b> ile belirtilen alt karakter katarını arar. İlk eşleşmenin indeksini döndürür. Eşleşme olmaması durumunda <b>-1</b> döndürür.
<b>int LastIndexOf(string str)</b>	Metodu çağrıran karakter katarı içinde <b>str</b> ile belirtilen alt karakter katarını arar. Son eşleşmenin indeksini döndürür. Eşleşme olmaması durumunda <b>-1</b> döndürür.
<b>string ToLower()</b>	Metodu çağrıran karakter katarının küçük harfli versiyonunu döndürür.
<b>string ToUpper()</b>	Metodu çağrıran karakter katarının büyük harfli versiyonunu döndürür.

Aşağıda, birkaç **string** işlemini gösteren bir program görüyorsunuz:

```
// Bazi string islemleri.

using System;

class StrOps {
    public static void Main() {
        string str1 =
            "When it comes to .Net programming, C# is #1.";
        string str2 = string.Copy(str1);
        string str3 = "C# strings are powerful.";
        string strUp, strLow;
        int result, idx;

        Console.WriteLine("str1: " + str1);

        Console.WriteLine("Length of str1: " + str1.Length);

        // str1'in buyuk ve kucuk harfli versiyonlarini olustur
        strLow = str1.ToLower();
        strUp = str1.ToUpper();
        Console.WriteLine("Lowercase version of str1:\n" +
            strLow);
        Console.WriteLine("Uppercase version of str1:\n" +
            strUp);

        Console.WriteLine();

        //str1'i her seferinde bir karakter goruntulenecek sekilde goster
        Console.WriteLine("Display str1, one char at a time.");
        for(int i=0; i < str1.Length; i++)
            Console.Write(str1[i]);
        Console.WriteLine("\n");

        // karakter katarlarini karsilastir
        if(str1 == str2)
            Console.WriteLine("str1 == str2");
        else
            Console.WriteLine("str1 != str2");
    }
}
```

```

        if(str1 == str3)
            Console.WriteLine("str1 == str3");
        else
            Console.WriteLine("str1 != str3");

        result = str1.CompareTo(str3);
        if(result == 0)
            Console.WriteLine("str1 and str3 are equal");
        else if(result < 0)
            Console.WriteLine("str1 is less than str3");
        else
            Console.WriteLine("str1 is greater than str3");

        Console.WriteLine();

        // yeni karakter katarini str2'ye ata
        str2 = "One Two Three One";

        // karakter katarini ara
        idx = str2.IndexOf("One");
        Console.WriteLine("Index of first occurrence of One: " + idx);
        idx = str2.LastIndexOf("One");
        Console.WriteLine("Index of last occurrence of One: " + idx);
    }
}

```

Bu program aşağıdaki çıktıyı üretir:

```

str1: When it comes to .NET programming, C# is #1.
Length of str1: 44
Lowercase version of str1:
    when it comes to .net programming, c# is #1.
Uppercase version of str1:
    WHEN IT COMES TO .NET PROGRAMMING, C# IS #1.

Display str1, one char at a time.
When it comes to .NET programming, C# is #1.

```

```

str1 == str2
str1 != str3
str1 is greater than str3

```

```

Index of first occurrence of One: 0
Index of last occurrence of One: 14

```

+ operatörünü kullanarak iki karakter katarını bitiştirebilirsiniz (peşpeşe ekleyebilirsiniz).

Örneğin, şu ifade **str4**'e "OneTwoThree" karakter katarını atar.

```

string str1 = "One";
string str2 = "Two";
string str3 = "Three";
string str4 = str1 + str2 + str3;

```

Bir diğer husus: **string** anahtar kelimesi, .NET Framework sınıf kütüphanesi tarafından tanımlanan **System.String** için kullanılan bir *takma isimdir* (*alias*) (yani, doğrudan **System.String** ile eşlenir). Böylece, **string** tarafından tanımlanan alanlar ve metodlar, burada gösterilen örneklerden çok daha fazlasını içeren **System.String** sınıfına aittir. **System.String**, Kısım II'de ayrıntılarıyla incelenmektedir.

## Karakter Katarı Dizileri

Tıpkı diğer veri tipleri gibi, karakter katarları da diziler içine toplanabilir. Örneğin:

```
// Karakter katarı dizilerini gösterir.

using System;

class StringArrays {
    public static void Main() {
        string[] str = { "This", "is", "a", "test." };

        Console.WriteLine("Original array: ");
        for(int i=0; i < str.Length; i++)
            Console.Write(str[i] + " ");
        Console.WriteLine("\n");

        // karakter katarını değiştire
        str[1] = "was";
        str[3] = "test, too!";

        Console.WriteLine("Modified array: ");
        for(int i=0; i < str.Length; i++)
            Console.Write(str[i] + " ");
    }
}
```

Bu programın çıktısı şöyle olur:

```
Original array:
This is a test.

Modified array:
This was a test, too!
```

İşte çok daha ilginç bir örnek. Aşağıdaki program, kelimeleri kullanarak bir tamsayıyı ekranda gösterir. Örneğin, 19 değeri - İngilizce – “one nine” olarak gösterilir.

```
// Bir tamsayının basamaklarını kelimelerle gösterir.

using System;

class ConvertDigitsToWords {
    public static void Main() {
        int num;
        int nextdigit;
        int numdigits;
```

```

int[] n = new int[20];

string[] digits = { "zero", "one", "two", "three", "four",
                    "five", "six", "seven", "eight",
                    "nine" };

num = 1908;

Console.WriteLine("Number: " + num);

Console.Write("Number in words: ");

nextdigit = 0;
numdigits = 0;

/*n'nin içinde saklanan basamakları al. Bu basamaklar ters
sırada saklanmıştır. */
do {
    nextdigit = num % 10;
    n[numdigits] = nextdigit;
    numdigits++;
    num = num / 10;
} while(num > 0);
numdigits--;

// kelimeleri göster
for( ; numdigits >= 0; numdigits--)
    Console.Write(digits[n[numdigits]] + " ");

Console.WriteLine();
}
}

```

Çıktı aşağıdaki gibidir:

```

Number: 1908
Number in words: one nine zero eight

```

Programda **digits** adındaki **string** dizisi, basamakların sıfırdan dokuza kadar kelime olarak karşılıklarını sırasıyla tutar. Program, bir tamsayıyı şu şekilde kelimelere dönüştürür: En sağdaki basamaktan başlayarak tamsayının basamaklarının her birini elde eder ve bunları **n** adındaki bir **int** dizisinde ters sırada saklar. Sonra program, bu dizi üzerinde arkadan öne doğru ilerler. Bu işlem sırasında **n**'deki her tamsayı değer **digits** için indeks olarak kullanılır ve karşılık gelen karakter katarı ekranda gösterilir.

## Karakter Katarları Değişmez

İşte sizi şaşırtabilecek bir konu: Bir **string** nesnesinin içeriği kesindir, değiştirilemez. Yani, bir kez oluşturulduktan sonra karakter katarını meydana getiren karakter sekansını değiştiremezsiniz. Bu kısıtlama, karakter katarlarının C#'ta daha verimli gerçeklenmesine imkan verir. Bu ciddi bir dezavantaj gibi görünse bile, aslında değildir. Önceden mevcut bir karakter katarının değiştirilmiş bir versiyonuna ihtiyacınız varsa, istenilen karakter katarını

İçeren yeni bir karakter katarı oluşturmanız yeterlidir. Kullanılmayan karakter katarı nesneleri, anlamsız veri kapsamında otomatik olarak toplanacağı için, kullanılmayan karakter katarlarına ne olacağı konusunda endişe etmenize gerek yoktur.

Ancak, bir konuya açıklık getirmek gerekiyor: **string** referans değişkenleri, bağlantılı oldukları nesneleri elbette değiştirebilirler. Oluşturulduktan sonra değiştirilemeyecek olan, belirli bir **string** nesnesinin içeriğidir.

Değiştirilemez karakter katarlarının neden bir engel olmadığını tam olarak anlamak için bir başka **string** metodu kullanacağımız: **Substring()**.**Substring()** metodu, kendisini çağırılan karakter katarının belirli bir parçasını içeren yeni bir karakter katarı döndürür. Alt karakter katarını içeren yeni bir **string** üretildiği için orijinal karakter katarı değişiklikle uğramaz ve değişmezlik kuralı halen geçerli kalır. Kullanmakta olacağımız **Substring()**'in yapısı aşağıda gösterilmiştir:

```
string Substring(int başlangıç, int uzunluk)
```

Burada **başlangıç**, alt karakter katarının başlangıç indeksini; **uzunluk** ise uzunluğunu belirtir.

**Substring()**'i ve karakter katarlarının değişmezliği prensibini gösteren bir program:

```
// Substring()'i kullanır.

using System;

class SubStr {
    public static void Main() {
        string orgstr = "C# makes strings easy.";

        // bir alt karakter katarı oluştur
        string substr = orgstr.Substring(5, 12);

        Console.WriteLine("orgstr: " + orgstr);
        Console.WriteLine("substr: " + substr);
    }
}
```

Programın çıktısı şu şekildedir:

```
orgstr: C# makes strings easy.
substr: kes strings
```

Gördüğünüz gibi, orijinal karakter katarı **orgstr** değişmeden kalır ve **substr**, alt karakter katarını içerir.

Bir husus daha: **string** nesnelerinin değiştirilemez oluşu, genellikle bir kısıtlama ya da engel olmamasına karşın, bir karakter katarını değiştirebilmenin avantajlı olduğu durumlar da söz konusu olabilir. Buna imkan vermek için C#, **System.Text** isim uzayında yer alan

**StringBuilder** adında bir sınıf sunmaktadır. Bu sınıf ile değiştirilebilen **string** nesneleri oluştururlar. Ancak, çoğu kez **StringBuilder** yerine **string**'ı kullanmayı tercih edeceksiniz.

## Karakter Katarları switch İfadelerinde Kullanılabilir

Bir **switch** ifadesini kontrol etmek için bir **string** kullanılabilir. **string**, **switch**'te kullanılabilen ve **int** tipinde olmayan tek tiptir. Karakter katarlarının **switch** ifadelerinde kullanılabilmesi gerçeği, bazı durumların oldukça kolay ele alınmasını mümkün kılmaktadır. Örneğin, aşağıdaki program “**one**”, “**two**” ve “**three**” kelimelerinin rakam karşılıklarını ekranda gösterir:

```
// Bir switch ifadesini bir string kontrol edebilir.

using System;

class StringSwitch {
    public static void Main() {
        string[] strs = { "one", "two", "three", "two", "one" };

        foreach(string s in strs) {
            switch(s) {
                case "one":
                    Console.Write(1);
                    break;
                case "two":
                    Console.Write(2);
                    break;
                case "three":
                    Console.Write(3);
                    break;
            }
        }
        Console.WriteLine();
    }
}
```

Cıktı aşağıdaki gibidir:

12321

# METOTLAR VE SINİFLARA DAHA YAKINDAN BİR BAKİŞ

Bu bölümde sınıf ve metotları incelenmeye devam ediyoruz. Bir sınıfın üyelerine erişimin nasıl kontrol edilebileceği incelenerek bölüme başlanıyor. Sonra nesnelerin aktarılması ve döndürülmesi, metotların aşırı yüklenmesi (overloading), **Main()**'in çeşitli şekilleri, yinelenme (recursion) ve **static** anahtar kelimesinin kullanımı ele alınıyor.

## Sınıf Üyelerine Erişimi Kontrol Etmek

Verilerin paketlenmesi (encapsulation) özelliğini destekleme kapsamında sınıflar başlıca iki avantaj sağlamaktadır. Öncelikle; sınıflar, verileri kodlarla bağlar. Sınıfların bu özelliğinden Bölüm 6'dan beri yararlanmaktadır. İkincisi; sınıflar, üyelere erişimi kontrol altında tutmak için çeşitli yöntemler sağlarlar. Burada incelenen işte bu ikinci özelliktir.

C#'ın yaklaşımı bir parça daha sofistike olsa da, aslında sınıf üyeleri iki temel tipte toplanır: **public** (açık) ve **private** (özel). **public** üye, kendi sınıfının dışında tanımlanmış kodlar tarafından serbestçe erişilebilir. Şu aşamaya kadar kullanmakta olduğumuz sınıf üyeleri bu tiptendir. **private** üye ise, yalnızca kendi sınıfı içinde tanımlanmış metotlar tarafından erişilebilir. Erişim, **private** üyelerin kullanımı sayesinde kontrol edilir.

Bir sınıfın üyelerine erişimi kısıtlamak nesne yönelimli programmanın en temel parçasıdır, çünkü bu, bir nesnenin yanlış kullanımını önlemeye yardımcı olur. İyi tanımlanmış birtakım metotlar yoluyla yalnızca **private** verilere erişim imkanı vererek söz konusu verilere uygunsuz değerlerin atanmasını önleyebilirsiniz -değer aralığı kontrol yaparak, örneğin. Sınıfın dışında kalan kodun **private** bir verinin değerini doğrudan ayarlaması mümkün değildir. Ayrıca, bir nesnenin içindeki verilerin nasıl ve ne zaman kullanıldığını da tam olarak kontrol edebilirsiniz. Böylece, doğru olarak uygulandığında bir sınıf, rahatlıkla kullanılabilen, fakat dahili çalışması kurcalamaya açık olmayan bir "kara kutu" oluşturur.

## C#'ta Erişim Belirleyicileri

Üye erişim kontrolü, dört adet erişim belirleyicisinin kullanımı sayesinde elde edilir: **public**, **private**, **protected** ve **internal**. Bu bölümde **public** ve **private** ile ilgileneceğiz. **protected** niteliyicisi yalnızca kalitim söz konusu olduğunda uygulanır ve Bölüm 9'da anlatılacaktır. **internal** niteliyicisi, daha çok *assembly* kullanımına uygulanır. Assembly de C#'ta genel bir tanımla bir program anlamına gelir. **internal** niteliyicisi Bölüm 16'da incelenmektedir.

Bir sınıfın bir üyesi **public** belirleyicisi ile nitelendiğinde bu üye, programınızdaki herhangi bir kod tarafından erişilebilir. Diğer sınıfların içinde tanımlanan metotlar da bu kapsamda yer alır.

Bir sınıfın bir üyesi **private** olarak belirtilmişse bu üye, yalnızca kendi sınıfının diğer üyeleri tarafından erişilebilir. Böylece, diğer sınıflardaki metotlar bir başka sınıfın **private** verilerine erişemezler. Bölüm 6'da açıklandığı gibi, eğer hiç erişim belirleyicisi kullanılmamışsa sınıf üyesi kendi sınıfına özeldir. (Yani; **private** varsayılan değerdir.)

Böylece, **private** sınıf üyeleri oluştururken **private** belirleyicisini kullanmak istegee bağlıdır.

Erişim belirleyicisi, üyenin geri kalan tip spesifikasyonundan önce gelir. Yani, üyenin deklarasyon ifadesi, erişim belirleyicisi ile başlamalıdır. İşte birkaç örnek:

```
public string errMsg;
private double bal;
private bool isError(byte status) { // ...
```

**public** ve **private** arasındaki farkı anlamak için aşağıdaki programı ele alın:

```
// public ve private erişim.

using System;

class MyClass {
    private int alpha;      // private erişim açıkça belirtiliyor
    int beta;                // varsayılan erişim sekli: private
    public int gamma;        // public erişim

    /* alpha ve beta'ya erişmek için gerekli metodlar. Bir sınıfın
       bir üyesinin, aynı sınıfın private üyesine erişmesinde
       problem yoktur.
    */
    public void setAlpha(int a) {
        alpna = a;
    }

    public int getAlpha() {
        return alpha;
    }

    public void setBeta(int a) {
        beta = a;
    }
    public int getBeta() {
        return beta;
    }
}

class AccessDemo {
    public static void Main() {
        MyClass ob = new MyClass();

        /* alpha ve beta'ya erişim yalnızca metodlar yardımıyla
           mümkün. */
        ob.setAlpha(-99);
        ob.setBeta(19);
        Console.WriteLine("ob.alpha is " + ob.getAlpha());
        Console.WriteLine("ob.beta is " + ob.getBeta());

        // alpha veya beta'ya bu şekilde erişemezsiniz:
        // ob.alpha = 10; // Yanlış! alpha private'tır!
    }
}
```

```

    // ob.beta = 9;      // Yanlis! beta private'tir!

    // gamma'ya doğrudan erişim olabilir cunku gamma public'tir
    ob.gamma = 99;
}
}

```

Gördüğünüz gibi, **MyClass** sınıfı içinde **alpha**, **private** olarak belirtilmektedir; **beta**'nın varsayılan değeri **private**'tir ve **gamma** ise **public** olarak belirtilmiştir. **alpha** ve **beta private** oldukları için kendi sınıfları dışındaki kod tarafından erişilemezler, Bundan dolayı, **AccessDemo** sınıfı içinde bu iki değişkenin ikisi de doğrudan kullanılamaz. Her ikisi de, **public** olan metodlar - söz gelisi **setAlpha()** ya da **getAlpha()** - yardımıyla kullanılabilir. Örneğin, aşağıdaki satırın başındaki açıklama simgesini kaldırınmak isteseydiniz, erişim kuralının çiğnenmesinden ötürü bu programı derleyemeyecektiniz.

```
// ob.alpha = 10; // Yanlis! alpha private'tir!
```

**MyClass** dışında yer alan kodun **alpha**'ya erişimine izin verilmemesine rağmen, **setAlpha()** ve **getAlpha()** metodlarından da görüldüğü gibi **MyClass** içindeki metodlar **alpha**'ya serbetçe erişebilirler. Bu durumun aynısı **beta** için de geçerlidir.

Temel husus şudur: Bir **private** üye kendi sınıfının diğer üyeleri tarafından serbestçe kullanılabilir, fakat kendi sınıfının dışında yer alan kod tarafından erişilemez.

## Public ve Private Erişimi Uygulamak

**public** ve **private** erişimi düzgün olarak kullanmak, başarılı nesne yönelimli programmanın temel bileşenidir. Katı kurallar olmasa da, kılavuz alınabilecek bazı genel prensipler şunlardır;

1. Bir sınıfın yalnızca bu sınıf içinde kullanılan üyelerini **private** olmalıdır.
2. Belirli bir değer aralığında olması gereken örnek veriler **private** olmalıdır; bu verilere erişim, değer aralığı kontrolü yapabilen **public** metodlar yardımıyla sağlanmalıdır.
3. Eğer bir üye üzerinde yapılan değişikliğin etkisi üyenin kendisini aşıyorsa (yani, nesnenin diğer taraflarını da etkiliyorsa), söz konusu üye **private** olmalı ve bu üyeye erişim kontrol altında tutulmalıdır.
4. Yanlış kullanıldıklarında bir nesneye zarar verebilen üyeler **private** olmalıdır. Bu üyelere erişim, yanlış kullanımı önleyen **public** metodlar yardımıyla gerçekleştirilmelidir.
5. **private** verilerin değerlerini alan ve ayarlayan (değiştiren) metodlar **public** olmalıdır.
6. **public** örnek değişkenlere, bu değişkenlerin **private** olmalarını gerektiren bir neden yoksa, izin verilir.

Yukarıdaki kuralların ele almadığı nüanslar elbette mevcuttur; üstelik, özel durumlar bir veya daha fazla kuralın çiğnenmesine neden olabilir. Fakat, genel olarak, bu kurallara uyarsanız, kolaylıkla yanlış kullanılmayacak esnek nesneler oluşturacaksınız demektir.

## Erişimi Kontrol Altına Almak: Bir Örnek Çalışma

Erişim kontrolünün “nasıl ve niçin” gerekliliğini daha iyi anlamak için bir örnek incelemek yararlıdır. Nesne yönelimli programmanın mükemmel örneklerinden biri, bir yiğini gerçekleyen bir sınıfıtır. Muhtemelen biliyorsunuzdur; yiğin, “ilk giren son çıkar” mantığına dayanan bir listeyi gerçekleyen veri yapısıdır. Yiğinin ismi, bir masa üzerindeki tabak yiğinına benzerliğinden gelmektedir. Masaya konan ilk tabak, en son kullanılacak olandır.

Yiğin, nesne yönelimli programmanın klasik örneklerindendir, çünkü bilgi için gerekli depolamayı, bilgiye erişmek için gerekli metotlarla birleştirir. Bu nedenle yiğin; ilk giren, son çıkar kullanımını mecbur kıلان bir *veri motorudur* (*data engine*). Bu tür bir birleşme, yiğin için depolama sağlayan üyeleri **private** olan ve **public** metotlarla erişim sağlanan bir sınıf için mükemmel bir seçenekdir. Bir sınıfın içinde yer alan verileri dış dünyadan gizleyerek yiğini kullanan kodun elemanlara kural tanımadan (metotları kullanmadan) erişmesi engellenir.

Yiğin ile ilgili iki temel işlem tanımlıdır: *ekleme* (*push*) ve *çıkarma* (*pop*). Ekleme, bir değeri yiğinin en üstüne yerleştirir. Çıkarma, yiğinin en üstünden bir değer alır. Yani çıkışma, tüketicidir; bir kez bir değer yiğinden alındıktan sonra, artık yiğinden çıkarılmıştır, bu değere tekrar erişilemez.

Aşağıda gösterilen örnek, bir yiğini gerçekleyen **Stack** adında bir sınıf tanımlamaktadır. Yiğinin temelini oluşturan depolama, özel bir dizi tarafından sağlanmaktadır. Ekleme ve çıkışma işlemleri **Stack** sınıfının **public** metotları sayesinde mümkündür. Yani, **public** metotlar ilk giren son çıkar mekanizmasını zorunlu kılar. Burada gösterildiği gibi, **Stack** sınıfı karakterleri saklar, ama aynı mekanizma, herhangi bir veri tipini saklamak amacıyla da kullanılabilir:

```
// Karakterlerden oluşan bir yigin sınıfı.

using System;

class Stack {
    // bu üyeler private'tır
    char[] stck; // yigini tutar
    int tos;    // yiginin tepesinin indeksi

    // buyuklugu bilinen bos bir Stack olustur
    public Stack(int size) {
        stck = new char[size]; // yigin için bellek alanı ayir
        tos = 0;
    }

    // Karakterleri yigina ekle.
```

```

public void push(char ch) {
    if(tos == stck.Length) {
        Console.WriteLine(" -- Stack is full.");
        return;
    }

    stck[tos] = ch;
    tos++;
}

// Yigindan bir karakter cikar.
public char pop() {
    if(tos == 0) {
        Console.WriteLine(" -- Stack is empty.");
        return (char) 0;
    }

    tos--;
    return stck[tos];
}

// Yigin doluya true dondur.
public bool full() {
    return tos == stck.Length;
}

// Yigin bossa true dondur.
public bool empty() {
    return tos == 0;
}

// Yiginin toplam kapasitesini dondur.
public int capacity() {
    return stck.Length;
}

// Su an yiginda olan nesnelerin sayisini dondur.
public int getNum() {
    return tos;
}
}

```

Gelin, şimdi bu sınıfı daha yakından inceleyelim. **stck** sınıfı, şu iki örnek değişkeni deklare ederek başlamaktadır:

```

// bu uyeler private'tir
char[] stok; // yigini tutar
int tos; // yiginin tepesinin indeksi

```

**stck** dizisi, yiğinin temelini oluşturan depolamayı sağlar. Bu örnekte yiğin, karakterleri tutar. Dizi için bellek alanı ayrılmadığına dikkat edin. Asıl dizi için bellek alanı ayırma işlemi, **Stack** yapılandırıcısı tarafından ele alınır. **tos** isimli üye, yiğinin en üstünün (tepesinin) indeksini tutar.

**tos** ve **stck** üyelerinin her ikisi de **private**'tir. Bu; ilk giren son çıkar mekanizmasını zorunlu kılmaktadır. **stck**'e **public** erişime izin verilseydi, yiğindaki elemanlar gelişigüzel bir sırada erişilebilirlerdi. Ayrıca; **tos**, yiğinin en üstteki elemanın indeksini tuttuğu için, yiğinin bozulmasını önlemek amacıyla, **Stack** sınıfı dışındaki kodun **tos** üzerinde değişiklik yapması da önlenmelidir. **Stack**'i kullananlar için, kısaca bahsedilen çeşitli **public** metodalar yardımıyla, **stck** ve **tos**'a erişim dolaylı olarak mümkündür.

Programda daha sonra yiğin yapılandırıcısı gösterilmektedir:

```
// buyuklugu bilinen bos bir Stack olustur
public Stack(int size) {
    stck = new char[size]; // yigin icin bellek alani ayir
    tos = 0;
}
```

İstenilen yiğin büyüklüğü yapılandırıcıya aktarılır. Yapılandırıcı, temel teşkil edecek dizi için bellek alanı ayırır ve **tos**'a sıfır değerini verir. Böylece, **tos**'un içindeki sıfır değeri, yiğinin boş olduğunu işaret eder.

Açık **push()** metodu, yiğine bir eleman yerleştirir. Bu metot aşağıda gösterilmiştir;

```
// Karakterleri yigina ekle.
public void push(char ch) {
    if(tos == stck.Length) {
        Console.WriteLine(" -- Stack is full.");
        return;
    }

    stck[tos] = ch;
    tos++;
}
```

Yiğine yerleştirilecek olan eleman **ch** üzerinden aktarılır. Söz konusu eleman yiğine eklenmeden önce yiğini oluşturan dizide hala yer olup olmadığı kontrol edilir. Bunun için **tos**'un **stck**'in uzunluğunu aşmadığından emin olmak gereklidir. Hala yer varsa söz konusu eleman, **stck** içinde **tos**'un belirttiği indekste saklanır ve **tos** bir artırılır. Böylece, **tos** her zaman **stck** içindeki bir sonraki boş elemanın indeksini içerir.

Yiğindan bir eleman çıkarmak için **pop()** açık metodunu çağırın. **pop()** aşağıda gösterilmiştir:

```
// Yigindan bir karakter cikar.
public char pop () {
    if(tos == 0) {
        Console.WriteLine(" -- Stack is empty.");
        return (char) 0;
    }

    tos--;
    return stck[tos];
}
```

Burada **tos**'un değeri kontrol edilmektedir. **tos**'un değeri sıfır ise yiğin boştur. Aksi halde, **tos** bir azaltılır ve o indekste bulunan eleman döndürülür.

Bir yiğini uygulamak için gerekli olan metodlar yalnızca **push()** ve **pop()**'tan ibaret olسا da, diğer bazı metodlar da oldukça kullanışlıdır. **Stack** sınıfında dört metot daha tanımlıdır. Bunlar **full()**, **empty()**, **capacity()** ve **getNum()**'dır. Bu metodlar yiğinin durumuyla ilgili bilgi sunarlar. Aşağıda gösterilmişlerdir:

```
// Yigin doluya true dondur.
public bool full() {
    return tos == stck.Length;
}

// Yigin bossa true dondur.
public bool empty() {
    return tos == 0;
}

// Yiginin toplam kapasitesini dondur.
public int capacity() {
    return stck.Length;
}

// Su an yiğinda olan nesnelerin sayisini dondur.
public int getNum() {
    return tos;
}
```

**full()** metodu, yiğin dolu iken **true**, aksi halde **false** döndürür. **empty()** metodu, yiğin boş olduğunda **true**, aksi halde **false** döndürür. Yiğinin toplam kapasitesini (yani, yiğinin tutabileceği toplam eleman sayısını) elde etmek için **capacity()**'yi çağırın. Yiğında şu an için mevcut olan elemanların sayısını elde etmek için **getNum()**'ı çağırın. Bu metodlar kullanışlıdır, çünkü bu metodların sağladığı bilgi **tos**'a erişim gerektirir; **tos** ise **private**'tir. Bunlar ayrıca, **public** metodların **private** üyelere nasıl güvenli erişim sağlayabileceğinin birer örneğidir.

Aşağıdaki program, yiğini göstermektedir:

```
// Stack sınıfını gösterir.

using System;

class StackDemo {
    public static void Main() {
        Stack stk1 = new Stack(10);
        Stack stk2 = new Stack(10);
        Stack stk3 = new Stack(10);
        char ch;
        int i;

        // stk1 icine birkac karakter yerlestir.
        Console.WriteLine("Push A through Z onto stk1.");
    }
}
```

```

        for(i = 0; !stk1.full(); i++)
            stk1.push((char) ('A' + i));

        if(stk1.full()) Console.WriteLine("stk1 is full.");

        // stk1'in icerigini goster.
        Console.Write("Contents of stk1: ");
        while( !stk1.empty() ) {
            ch = stk1.pop();
            Console.Write(ch);
        }

        Console.WriteLine();

        if(stk1.empty()) Console.WriteLine("stk1 is empty.\n");

        // stk1'e biraz daha karakter yerlestir
        Console.WriteLine("Again push A through Z onto stk1.");
        for(i = 0; !stk1.full(); i++)
            stk1.push((char) ('A' + i));

        /* Simdi, stk1'den al ve alınan elemani stk2'ye koy.
           Bu islem, stk2'nin elemanlari
           tek sirada tutmasına neden olur. */
        Console.WriteLine("Now, pop chars from stk1 and push " +
                           "them onto stk2.");
        while( !stk1.empty() ) {
            ch = stk1.pop();
            stk2.push(ch);
        }

        Console.Write("Contents of stk2: ");
        while( !stk2.empty() ) {
            ch = stk2.pop();
            Console.Write(ch);
        }

        Console.WriteLine("\n");

        // yigina 5 karakter yerlestir
        Console.WriteLine("Put 5 characters on stk3.");
        for(i = 0; i < 5; i++)
            stk3.push((char) ('A' + 1));

        Console.WriteLine("Capacity of stk3: " + stk3.capacity());
        Console.WriteLine("Number of objects in stk3: " +
                           stk3.getNum());
    }
}

```

Programın çıktısı aşağıda gösterilmiştir.

Push A through Z onto stk1.  
Stk1 is full.

Contents of stk1: JIHGFEDCBA  
Stk1 is empty.

Again push A through Z onto stk1.  
Now, pop chars from stk1 and push them onto stk2.  
Contents of stk2: ABCDEFGHIJ

Put 5 characters on stk3.  
Capacity of stk3: 10  
Number of objects in stk3: 5

## Metotlara Nesne Aktarmak

Bu noktaya kadar, elinizdeki kitaptaki örneklerde değer tipleri, örneğin **int** veya **double**, metotlara parametre olarak kullanılmaktaydı. Ancak, metotlara nesne aktarmak da hem doğru hem de yaygındır. Örneğin, aşağıdaki programı ele alın:

```
// Nesneler, metotlara aktarılabilir.

using System;

class MyClass {
    int alpha, beta;

    public MyClass(int i, int j) {
        alpha = i;
        beta = j;
    }

    /* Eğer ob, metodu çağrıran nesne ile aynı değerleri
       içeriyorsa, true dondur. */
    public bool sameAs(MyClass ob) {
        if((ob.alpha == alpha) & (ob.beta == beta))
            return true;
        else return false;
    }

    // ob'un kopyasını oluştur.
    public void copy(MyClass ob) {
        alpha = ob.alpha;
        beta = ob.beta;
    }

    public void show() {
        Console.WriteLine("alpha: {0}, beta: {1}", alpha, beta);
    }
}

class PassOb {
    public static void Main() {
        MyClass ob1 = new MyClass(4, 5);
        MyClass ob2 = new MyClass(6, 7);

        Console.Write("ob1: ");
    }
}
```

```
ob1.show();

Console.WriteLine("ob2: ");
ob2.show();

if(ob1.sameAs(ob2))
    Console.WriteLine("ob1 and ob2 have the same values.");
else
    Console.WriteLine("ob1 and ob2 have different values.");

Console.WriteLine();

// simdi ob1, ob2'nin kopyası olsun
ob1.copy(ob2);

Console.WriteLine("ob1 after copy: ");
Ob1.show();

if(ob1.sameAs(ob2))
    Console.WriteLine("ob1 and ob2 have the same values.");
else
    Console.WriteLine("ob1 and ob2 have different values.");
}
}
```

Bu program aşağıdaki çıktıyı üretir:

```
ob1: alpha: 4, beta: 5
ob2: alpha: 6, beta: 7
ob1 and ob2 have different values.

ob1 after copy: alpha: 6, beta: 7
ob1 and ob2 have the same values.
```

**sameAs()** ve **copy()** metodlarının her biri argüman olarak bir nesne alırlar. **sameAs()** metodu, metodu çağrıran nesnenin içindeki **alpha** ve **beta** değerlerini, argüman olarak **ob**'un içinde aktarılan nesnenin **alpha** ve **beta** değerleriyle karşılaştırır. Metot ancak bu örnek değişkenler için her iki nesne de aynı değerleri içeriyorsa **true** döndürür. **copy()** metodu, **ob** içinde aktarılan nesnenin **alpha** ve **beta** değerlerini, metodu çağrıran nesnenin **alpha** ve **beta** değerlerine atar. Her iki durumda da, **ob** parametresinin tip olarak **MyClass** sınıfını belirttiğine dikkat edin. Bu örnekte de görüldüğü gibi; nesne tipleri, söz dizimsel olarak, değer tipleriyle aynı şekilde metodlara aktarılmaktadır.

## Argümanlar Nasıl Aktarılır?

Önceki örnekte de gösterildiği üzere, bir metoda bir nesne aktarmak problemsiz bir iştır. Yine de, o örnekte gösterilmeyen bazı nüanslar da söz konusudur. Belirli durumlarda nesne aktarmanın etkileri, nesne olmayan argümanların aktarımında yaşanan deneyimlerden farklı olacaktır. Neden böyle olduğunu anlamak için gelin, bir argümanın bir alt rutine aktarılabilmesi için kullanılabilecek iki yöntemi yeniden gözden geçirelim.

İlk yöntem, *değer olarak çağrıdır*. Bu yöntem, argümanın *değerini* alt rutinin tanımlı parametresine kopyalar. Bu sayede, alt rutinin parametresi üzerinde yapılan değişikliklerin çağrıda kullanılan argüman üzerinde hiçbir etkisi yoktur. Argüman aktarmak için ikinci yöntem ise *referans olarak çağrıdır*. Bu yöntemde argümani gösteren bir referans (argümanın değeri değil) parametreye aktarılır. Alt rutinin içinde bu referans, çağrıda belirtilen asıl argümana erişmek için kullanılır. Bunun anlamı şudur: Parametre üzerinde yapılacak değişiklikler, alt rutini çağırmak için kullanılan argümani etkileyecektir. C#'ta her iki yöntemin de kullanılabileceğini daha sonra göreceksiniz.

**int** veya **double** gibi bir değer tipini bir metoda aktarırken bu değer tipi, değer olarak aktarılır. Böylece, argümani kabul eden parametre üzerinde meydana gelen değişikliklerin metod dışında hiçbir etkisi yoktur. Örneğin, aşağıdaki programı ele alın:

```
// Basit tipler değer olarak aktarılır.

using System;

class Test {
    /* Bu metod, çağrıda kullanılan argumanlar üzerinde değişiklik yapmaz. */
    public void noChange(int i, int j) {
        i = i + j;
        i = -j;
    }
}

class CallByValue {
    public static void Main() {
        Test ob = new Test();

        int a = 15, b = 20;

        Console.WriteLine("a and b before call: " + a + " " + b);

        ob.noChange(a, b);

        Console.WriteLine("a and b after call: " + a + " " + b);
    }
}
```

Bu programın çıktısı aşağıdaki gibidir:

```
a and b before call: 15 20
a and b after call: 15 20
```

Gördüğünüz gibi, **noChange()** içinde meydana gelen işlemlerin, çağrıda kullanılan **a** ve **b** değerleri üzerinde hiç etkisi yoktur.

Bir nesne referansını bir metoda aktardığınızda bu durum biraz daha karmaşıktır. Teknik olarak, nesne referansının kendisi değer olarak aktarılır. Böylece, referansın bir kopyası oluşturulur; parametre üzerindeki değişiklikler argümani etkilemeyecektir. (Örneğin, parametreyi yeni bir nesneyle ilişkilendirmek argümanın ilgili olduğu nesneyi

değiştirmeyecektir.) Ancak - ki bu, büyük bir anaktır - parametrenin ilişkili olduğu *nesne üzerinde* yapılan değişiklikler argümanın ilgili olduğu nesneyi *etkileyecektir*. Şimdi, neden böyle olduğunu görelim.

Hatırlarsanız, sınıf tipinde bir değişken oluştururken aslında bir nesneye bir referans oluşturmuş oluyorsunuz. Böylece, bu referansı bir metoda aktardığınızda, bunu alan parametre, argümanın ilişkili olduğu nesnelerle ilişkilendirilecektir. Bu gerçekten de nesnelerin metotlara “referans olarak çağrı” kullanılarak aktarıldığı anlamını taşır. Metodun içindeki nesneye yapılan değişiklikler argüman olarak kullanılan nesneyi *kesinlikle etkiler*. Örneğin, aşağıdaki programı ele alın:

```
// Nesneler referans olarak aktarılır.

using System;

class Test {
    public int a, b;

    public Test(int i, int j) {
        a = i;
        b = j;
    }

    /* Bir nesne aktar. Simdi, çağrıda kullanılan nesnenin
       içindeki ob.a ve ob.b degisecek. */
    public void change(Test ob) {
        ob.a = ob.a + ob.b;
        ob.b = -ob.b;
    }
}

class CallByRef {
    public static void Main() {
        Test ob = new Test(15, 20);

        Console.WriteLine("ob.a and ob.b before call: " +
                          ob.a + " " + ob.b);

        ob.change(ob);

        Console.WriteLine("ob.a and ob.b after call: " +
                          ob.a + " " + ob.b);
    }
}
```

Bu program aşağıdaki çıktıyi üretir:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 35 -20
```

Gördüğünüz gibi, bu örnekte **change()**’in içindeki faaliyetler argüman olarak kullanılan nesneyi etkilemiştir.

Tekrar edecek olursak: Bir nesne referansı bir metoda aktarıldığında, referansın kendisi “değer olarak çağrı” kullanılarak aktarılır. Böylece, söz konusu referansın bir kopyası oluşturulur. Ancak, aktarılmakta olan değer bir nesneye ilişkili olduğu için söz konusu değerin kopyası hala karşılık gelen argümanın bağlantılı olduğu nesneye ilişkili olacaktır.

## ref ve out Parametrelerini Kullanmak

Az önce açıklandığı gibi, **int** veya **char** gibi değer tipleri, metodlara değer olarak aktarılır. Bunun anlamı şudur: Bir değer tipini alan parametre üzerine yapılan değişiklikler çağrıda kullanılan asıl argümanı etkilemeyecektir. Ancak, bu davranışını değiştirebilirsiniz. **ref** ve **out** anahtar kelimelerinin kullanımını sayesinde değer tiplerini referans olarak aktarmak mümkündür. Bu sayede, bir metodun, çağrıda kullanılan argümanı değiştirmesine imkan verilir.

**ref** ve **out**’un nasıl kullanıldığına geçmeden önce, bir değer tipini neden referans olarak aktarmak isteyebileceğinizi anlamanız yararlıdır. Genel olarak, bunun iki nedeni vardır: Metodun kendi argümanının içeriğini değiştirebilmesine imkan vermek ya da metodun birden fazla değer döndürmesini mümkün kılmak. Gelin şimdi, bu nedenlerin her birini ayrıntılarıyla inceleyelim.

Bir metodun, kendisine aktarılan asıl argümanlar üzerinde işlem yapabilmesine sık sık ihtiyaç duyacaksınız. Bunun mükemmel bir örneği, iki argümanı değişim-tokus yapan **swap()** metodudur. Değer tipleri değer olarak aktarıldıkları için C#’in varsayılan “değer olarak çağrı” parametre aktarma mekanizması kullanılarak, örneğin iki **int** değerini takas eden bir metot yazmanız mümkün değildir. **ref** niteleyicisi problemi çözer.

Bildiğiniz gibi, **return** ifadesi, bir metodun kendisini çağrıran bir değer döndürmesini mümkün kılar. Ancak, bir metot her çağrılarında *yalnızca* bir değer döndürebilir. Peki, iki veya daha fazla bilgi parçası döndürmeniz gerekirse ne olacak? Örneğin, bir kayan noktalı sayıyı tamsayı ve ondalık kısımlarına ayıran bir metot geliştirmek istediğinizde ne yapacaksınız? Bu nın için iki bilgi parçasının da döndürülmesi gereklidir: Tamsayı kısmı ve ondalık bileşen. Bu metot tek bir dönüş değeri kullanılarak yazılamaz. **out** niteleyicisi bu problemi çözer.

## ref Kullanmak

**ref** parametre niteleyicisi C#’in “değer olarak çağrı” yerine, “referans olarak çağrı” oluşturmasını sağlar. **ref** niteleyicisi, metod deklare edildiğinde ve çağrıldığında kullanılır. Gelin, bir örnekle başlayalım. Aşağıdaki program **sqr()** adında bir metod kullanmaktadır. **sqr()**, bulunduğu yere argümanının karesini döndürür. **ref**’in kullanımına ve konumuna dikkat edin.

```
// Bir değer tipini referans olarak aktarmak için ref kullanır.

using System;

class RefTest {
    /* Bu metod, kendi argumanını değiştirir.
```

```

    ref'in kullanımına dikkat edin. */
public void sqr(ref int i) {
    i = i * i;
}
}

class RefDemo {
    public static void Main() {
        RefTest ob = new RefTest();

        int a = 10;

        Console.WriteLine("a before call: " + a);

        ob.sqr(ref a); // ref'in kullanımına dikkat edin

        Console.WriteLine("a after call: " + a);
    }
}

```

**ref**'in metot içindeki bütün parametre deklarasyonundan önce geldiğine ve metot çağrıldığında argüman isminin önünde yer aldığına dikkat edin. Aşağıda gösterilen program çıktısı da, argüman **a**'nın değerinin **sqr()** tarafından gerçekten değiştirildiğini doğrulamaktadır:

```

a before call: 10
a after call: 100

```

**ref** kullanarak, değer tipindeki iki argümanının değerlerini değiştirebilen bir metot yazmak artık mümkündür. Örneğin, işte size bir program. Bu program, kendisini çağıran iki tamsayı argümanının değerlerini değiştirebilen **swap()** adında bir metot içermektedir:

```

// iki değeri takas et.

using System;

class Swap {
    // Bu metot simdi argumanlarını değiştirir.
    public void swap(ref int a, ref int b) {
        int t;

        t = a;
        a = b;
        b = t;
    }
}

class SwapDemo {
    public static void Main() {
        Swap ob = new Swap();

        int x = 10, y = 20;

        Console.WriteLine("x and y before call: " + x + " " + y);

```

```

        ob.swap(ref x, ref y);

        Console.WriteLine("x and y after call: " + x + " " + y);
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

x and y before call: 10 20
x and y after call: 20 10

```

**ref** hakkında anlaşılması önemli bir konu da şudur: **ref** olarak aktarılan bir argümana çağrıdan önce bir değer atanmalıdır. Bunun nedeni şudur: Bu tür bir argüman alan bir metot, parametrenin geçerli bir veriyle ilişkili olduğunu varsayar. Yani, **ref** kullanarak, bir argümana ilk değer atamak için bir metot kullanmanız mümkün değildir.

## out Kullanmak

Kimi zaman bir metoda değer aktarmak yerine metottan bir değer almak için bir referans parametresi kullanmak isteyeceksiniz. Örneğin, referans parametresinde başarı/başarısızlık kodunu döndüren sözluğu bir ağ soketi açmak gibi bazı işlevleri gerçekleştiren bir metoda sahip olabilirsiniz. Bu durumda, metoda aktarılacak bilgi yoktur; metottan geri alınması gereken bilgi vardır. Bu senaryodaki problem, **ref** parametresine çağrıdan önce bir değer atanmış olması gereğidir. Sonuçta, bir **ref** parametresi kullanmak sırıbu koşulu sağlamak için argümana sahte bir değer verilmesini gerekli kılacaktır. Neyse ki, C# daha iyi bir alternatif sunmaktadır: **out** parametresi.

**out** parametresi, tek bir farkla **ref** parametresiyle aynıdır: **out** sadece bir metottan dışarıya bir değer aktarmak için kullanılabilir. **out** parametresi olarak kullanılan değişkene metodu çağrımadan önce ilk değer vermek gerekli (veya kullanışlı) değildir. Metot, söz konusu değişkene bir değer verecektir. Ayrıca, metodun içinde, bir **out** parametresi daima *atama yapılmamış* olarak kabul edilir; yani **out** parametresinin ilk değere sahip olmadığı varsayıılır. Bunun yerine metot, sona ermeden önce söz konusu parametreye bir değer *atamalıdır*. Böylece, metoda yapılan çağrıdan sonra **out** parametresi bir değer içerecektir.

İşte, **out** parametresi kullanılan bir örnek. **Decompose()** sınıfı içinde **parts()** metodu, kayan noktalı bir sayıyı tamsayı ve ondalık kısımlarına ayırr. Metodu çağrıran koda bileşenlerin her birinin nasıl döndürüldüğüne dikkat edin.

```

// out kullan.

using System;

class Decompose {
    /* Kayan noktalı bir degeri
       tamsayı ve ondalik kisimlarina ayir. */
    public int parts(double n, out double frac) {

```

```

        int whole;

        whole = (int) n;
        frac = n - whole; //ondalik kismi frac uzerinden geri aktar
        return whole; // tamsayi kismi dondur
    }
}

class UseOut {
    public static void Main() {
        Decompose ob = new Decompose();
        int i;
        double f;

        i = ob.parts(10.125, out f);

        Console.WriteLine("Integer portion is " + i);
        Console.WriteLine("Fractional part is " + f);
    }
}

```

Programın çıktısı aşağıdaki gibidir:

```

Integer portion is 10
Fractional part is 0.125

```

**parts()** metodu iki parça halinde bilgi döndürür: Öncelikle, **n**'nin tamsayı kısmı **parts()**'in dönüş değeri olarak döndürülür. İkincisi, **n**'nin ondalık kısmı metodu çağrıran koda, **frac** isimli **out** parametresi üzerinden geri aktarılır. Bu örnekte de görüldüğü gibi, **out** kullanılarak bir metodun iki değer döndürmesi mümkündür.

Elbette, yalnızca bir adet **out** parametresi ile kısıtlanmış değilsiniz. Bir metot, **out** parametreleri yardımıyla gerekli olduğu kadar çok sayıda bilgi döndürebilir. İşte, iki adet **out** parametresi kullanan bir örnek. Bu örnekte, **isComDenom()** metodu iki işlev görmektedir. Öncelikle, iki tamsayının ortak bir paydası olup olmadığını belirler. Eğer varsa, **true** döndürür; aksi halde, **false** döndürür. İkinci işlevi ise şudur: Eğer ortak bir payda varsa **isComDenom()**, ortak paydaların en küçük ve en büyüklerini **out** parametrelerinde döndürür.

```

// iki out parametresi kullan.

using System;

class Num {
    /* x ve y'nin ortak bir paydasi olup olmadigini belirler.
       Eger varsa, en kucuk ve en buyuk ortak paydalari
       out parametrelerinde dondurur. */
    public bool isComDenom(int x, int y,
                           out int least, out int greatest) {
        int i;
        int max = x < y ? x : y;
        bool first = true;

```

```

least = 1;
greatest = 1;

// en kucuk ve en buyuk ortak paydalari bulur
for(i = 2; i <= max/2 + 1; i++) {
    if( ((y%i)==0) & ((x%i)==0) ) {
        if(first) {
            least = 1;
            f0irst = false;
        }
        greatest = i;
    }
}

if(least != 1) return true;
else return false;
}
}

class DemoOut {
    public static void Main() {
        Num ob = new Num();
        int lcd, gcd;

        if(ob.isComDenom(231, 105, out lcd, out gcd)) {
            Console.WriteLine("Lcd of 231 and 105 is " + lcd);
            Console.WriteLine("Gcd of 231 and 105 is " + gcd);
        }
        else
            Console.WriteLine("No common denominator for 35 and 49.");
        if (ob.isComDenom(35, 51, out lcd, out gcd)) {
            Console.WriteLine("Lcd of 35 and 51 " + lcd);
            Console.WriteLine("Gcd of 35 and 51 is " + gcd);
        }
        else
            Console.WriteLine("No common denominator for 35 and 51.");
    }
}

```

**Main()** içinde, **isComDenom()**'a çağrı yapılmadan önce **lcd** ve **gcd**'ye değer atanmadığına dikkat edin. Parametreler **out** yerine **ref** olsaydı hatalı olurdu. Metot, her iki tamsayıının ortak bir paydası olup olmamasına bağlı olarak **true** veya **false** döndürür. Eğer ortak bir payda varsa, ortak paydaların en küçüğü ve en büyüğü **out** parametrelerinde döndürülür. Bu programın çıktısı aşağıdaki gibidir:

```

Lcd of 231 and 105 is 3
Gcd of 231 and 105 is 21
No common denominator for 35 and 51.

```

## Referans Parametrelerinde ref ve out Kullanmak

**ref** ve **out** kullanımı yalnızca değer parametreleriyle sınırlı değildir. Bunlar referans parametrelerinde de, örneğin bir nesne referansı aktarıldığı zaman da ayrıca kullanılabilir. **ref**

veya **out** bir referans parametresi üzerinde değişiklik yapınca, bu durum, referansın kendisinin referans olarak aktarılmasına neden olur. Böylece metodun, referansın ilişkili olduğu nesneyi değiştirmesine imkan verilir. Aşağıdaki programı ele alın. Bu programda, iki referansın ilişkili olduğu nesneleri değiştirmek için **ref** parametreleri kullanılmaktadır:

```
// iki referansı takas etmek.

using System;

class RefSwap {
    int a, b;

    public RefSwap(int i, int j) {
        a = i;
        b = j;
    }

    public void show() {
        Console.WriteLine("a: {0}, b: {1}", a, b);
    }

    // Bu metot argumanlarını değiştirir.
    public void swap(ref RefSwap ob1, ref RefSwap ob2) {
        RefSwap t;

        t = ob1;
        ob1 = ob2;
        ob2 = t;
    }
}

class RefSwapDemo {
    public static void Main() {
        RefSwap x = new RefSwap(1, 2);
        RefSwap y = new RefSwap(3, 4);

        Console.Write("x before call: ");
        x.show();

        Console.Write("y before call: ");
        y.show();

        Console.WriteLine();

        // x ve y'nin ilişkili olduğu nesneleri değiştirmek yap
        x.swap(ref x, ref y);

        Console.Write("x after call: ");
        x.show();

        Console.Write("y after call: ");
        y.show();
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
x before call: a: 1, b: 2
y before call: a: 3, b: 4

x after call: a: 3, b: 4
y after call: a: 1, b: 2
```

Bu örnekte **swap()** metodu, **swap()**'ın her iki argümanının ilişkili olduğu nesneleri değiş-tokuş eder. **swap()**'ı çağrımadan önce **x**, **1** ve **2** değerlerini içeren bir nesneyle; **y** ise **3** ve **4** değerlerini içeren bir nesneyle ilişkilidir. **swap()**'a yapılan çağrıdan sonra **x**, **3** ve **4** değerlerini içeren nesneyle; **y** ise **1** ve **2** değerlerini içeren nesneyle ilişkilendirilir. **ref** parametreleri kullanılmamış olsaydı. **swap()** içindeki değişim **swap()** dışında hiç bir etkisi olmamış olacaktı. Bunu kanıtlamak isterseniz, **ref**'i **swap()**'tan çıkarabilirsiniz.

## Değişen Sayıda Argüman Kullanmak

Bir metot geliştirdiğinizde, genellikle bu metoda aktaracağınız argümanların sayısını önceden bilirsiniz, fakat bu durum her zaman geçerli değildir. Bazen rasgele sayıda argüman aktarılabilen bir metot geliştirmek isteyebilirsiniz. Örneğin, bir değerler kümesinin en küçüğünü bulan bir metodu ele alın. Böyle bir metoda iki gibi az sayıda değer ya da üç, dört vs sayıda değer aktarılabilir. Her türlü durumda, metodun en küçük değeri döndürmesini istersiniz. Bu tür bir metot, normal parametreler kullanılarak oluşturulamaz. Bunun yerine, rasgele sayıda parametreye karşılık gelen özel tipte bir parametre kullanmalısınız. İşte bu, **params** parametresi kullanılarak gerçekleştirilir.

**params** nitelik, sıfır sayıda veya daha fazla argüman alabilen bir dizi parametre deklare etmek için kullanılır. Dizideki eleman sayısı, metoda aktarılan argüman sayısına eşit olacaktır. Programınız daha sonra argümanları elde etmek için dizisi erişir.

İste size bir örnek. Bu örnekte, bir değerler kümesinin en küçük değerini döndüren **minVal()** adında bir metot oluşturmak için **params** kullanılmaktadır:

```
// params'i gösterir.

using System;

class Min {
    public int minVal(params int[] nums) {
        int m;

        if(nums.Length == 0) {
            Console.WriteLine("Error: no arguments.");
            return 0;
        }

        m = nums[0];
        for(int i = 1; i < nums.Length; i++)
            if(nums[i] < m) m = nums[i];
```

```

        return m;
    }
}

class ParamsDemo {
    public static void Main() {
        Min ob = new Min();
        int min;
        int a = 10, b = 20;

        // 2 deger ile cagir
        min = ob.minVal(a, b);
        Console.WriteLine("Minimum is " + min);

        // 3 deger ile cagir
        min = ob.minVal(a, b, -1);
        Console.WriteLine("Minimum is " + min);

        // 5 deger ile cagir
        min = ob.minVal(18, 23, 3, 14, 25);
        Console.WriteLine("Minimum is " + min);

        // bir int dizisiyle bile cagrılabilir
        int[] args = { 45, 67, 34, 9, 112, 8 };
        min = ob.minVal(args);
        Console.WriteLine("Minimum is " + min);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Minimum is 10
Minimum is -1
Minimum is 3
Minimum is 8

```

`minVal()`'in her çağrılarında argümanlar, `nums` dizisi üzerinden metoda aktarılır. Dizinin uzunluğu eleman sayısına eşittir. Böylece, `minVal()`'i herhangi bir sayıda değerin en küçüğünü bulmak için kullanabilirsiniz.

`params` parametresine herhangi bir sayıda argüman aktarabilmenize rağmen, bunların tümü parametre olarak belirtilen dizi tipi ile uyumlu olmalıdır. Örneğin, `minVal`'ı şu şekilde çağırmak;

```
min = ob.minVal(1, 2.2);
```

kurallara uymaz, çünkü `double`'dan (2.2) `int`'e otomatik dönüşüm yoktur. Oysa `int`, `minVal()`'in içinde yer alan `nums`'ın tipidir.

`params`'ı kullanırken sınır koşullarıyla ilgili dikkatli olmanız gereklidir, çünkü `params` parametresi herhangi bir sayıda argüman kabul edebilir - *hatta, sıfır sayıda bile!* Örneğin, `minVal()`'i aşağıda gösterildiği gibi çağrılmak söz dizimsel açıdan geçerlidir:

```
min = ob.minVal(); // argumansız
min = ob.minVal(3) // 1 argumanlı
```

**nums** dizisinin elemanına erişmeye girişmeden önce dizide en az bir eleman olduğunu doğrulamak için **minVal()** içinde bir kontrolün yer alınmasının nedeni budur. Eğer burada kontrol olmasaydı, **minVal()**'in argümansız çağrılmaması durumunda, çalışma sırasında bir kural dışı durum ortaya çıkardı. (Bu kitabin ileriki bölümlerinde kural dışı durumlar ele alındığında, bu tip hataları kontrol altına almak için daha iyi bir yöntem öğreneceksiniz.) Üstelik, **minVal()**'in kodu, **minVal()**'i tek bir parametreyle çağırılmaya izin verecek şekilde yazılmıştır, Böyle bir durumda, tek arguman döndürülür.

Bir metot, normal parametrelere ve değişken sayıda bir parametreye sahip olabilir. Örneğin, aşağıdaki programda **showArgs()** metodu önce bir **string** parametresi, sonra da bir **params** tamsayı dizisi alır:

```
// Normal parametreyi params parametresiyle birlikte kullanmak.

using System;

class MyClass {
    public void showArgs(string msg, params int[] nums) {
        Console.WriteLine(msg + ": ");

        foreach(int i in nums)
            Console.Write(i + " ");

        Console.WriteLine();
    }
}

class ParamsDemo2 {
    public static void Main() {
        MyClass ob = new MyClass();

        ob.showArgs("Here are some integers", 1, 2, 3, 4, 5);
        ob.showArgs("Here are two more", 17, 20);
    }
}
```

Bu program aşağıdaki çıktıyı ekranda gösterir:

```
Here are some integers: 1 2 3 4 5
Here are two more: 17 20
```

Bir metodun hem normal parametreye hem de **params** parametresine sahip olması durumlarda **params** parametresi, parametre listesindeki en son eleman olmalıdır. Üstelik, her koşulda parametre listesinde yalnızca bir tane **params** parametresi olmalıdır.

## Nesneleri Döndürmek

Bir metot sınıf tipleri de dahil olmak üzere herhangi tipten verileri döndürebilir. Örneğin, `Rect` sınıfının aşağıdaki versiyonu `enlarge()` adında bir metot içermektedir. `enlarge()`, kendisini çağıran dikdörtgenle aynı orantıya sahip, fakat belirli bir faktör oranında daha büyük bir dikdörtgen oluşturmaktadır:

```
// Bir nesne dondur.

using System;

class Rect {
    int width;
    int height;

    public Rect(int w, int h) {
        width = w;
        height = h;
    }

    public int area() {
        return width * height;
    }

    public void show() {
        Console.WriteLine(width + " " + height);
    }

    /* Metodu çağırın dikdörtgenden belirtilen oranda
       daha büyük bir dikdörtgen dondur. */
    public Rect enlarge(int factor) {
        return new Rect(width * factor, height * factor);
    }
}

class RetObj {
    public static void Main() {
        Rect r1 = new Rect(4, 5);

        Console.Write("Dimensions of r1: ");
        r1.show();

        Console.WriteLine("Area of r1: " + r1.area());

        Console.WriteLine();

        // r1'in iki katı büyüklükte bir dikdörtgen oluştur
        Rect r2 = r1.enlarge(2);

        Console.Write("Dimensions of r2: ");
        r2.show();
        Console.WriteLine("Area of r2 " + r2.area());
    }
}
```

```
}
```

Çıktı aşağıda gösterilmiştir:

```
Dimensions of r1: 4 5
Area of r1: 20
```

```
Dimensions of r2: 8 10
Area of r2 80
```

Bir metot tarafından döndürülen bir nesne, artık kendisine herhangi bir referans kalmayana kadar varlığını sürdürür. Kendisine herhangi bir referans kalmadığında ise o nesne anlamsız veri toplama ve yok etme sürecine (garbage collection) tabi tutulur. Bir başka ifade ile, bir nesne, sırıf kendisini oluşturan metot sonlandı diye hemen yok edilmez.

Nesne dönüş tiplerinin bir uygulaması, *sınıf fabrikasıdır* (*class factory*). Bir sınıf fabrikası, kendi sınıfından nesneler kurmaya yarayan bir metottur. Güvenlik nedenleriyle ya da nesne kuruluşu bazı dış unsurlara bağlı olduğu için bazı durumlarda bir sınıfın kullanıcılarına o sınıfın yapılandırıcısına erişme olanağı vermek istemeyebilirsiniz. Böyle durumlarda nesneleri kurmak için bir sınıf fabrikası kullanılabilir. Aşağıda bunun basit bir örneği yer almaktadır.

```
// Bir sınıf fabrikası kullanır.

using System;

class MyClass {
    int a, b; // private

    // MyClass için bir sınıf fabrikası oluşturur.
    public MyClass factory (int i, int j) {
        MyClass t = new MyClass();

        t.a = i;
        t.b = j;

        return t; // bir nesne dondurur
    }

    public void show() {
        Console.WriteLine("a and b: " + a + " " + b);
    }
}

class MakeObjects {
    public static void Main() {
        MyClass ob = new MyClass();
        int i, j;

        // Fabrikayı kullanarak nesneler üretir.
        for(i=0, j=10; i < 10; i++, j--) {
            MyClass anotherOb = ob.factory(i, j); // Nesne oluştur
            anotherOb.show();
        }
    }
}
```

```

        Console.WriteLine();
    }
}

```

Programın çıktısı şöyledir:

```

a and b: 0 10
a and b: 1 9
a and b: 2 8
a and b: 3 7
a and b: 4 6
a and b: 5 5
a and b: 6 4
a and b: 7 3
a and b: 8 2
a and b: 9 1

```

Gelin, şimdi bu örneği yakından inceleyelim. **MyClass** bir yapılandıracı tanımlamadığı için sadece varsayılan yapılandıracı mevcuttur. Dolayısıyla **a** ve **b**'nin değerlerini bir yapılandıracı kullanarak atamak olanağlı değildir. Ancak sınıf fabrikası **factory()**, **a** ve **b**'nin değer atandığı nesneyle oluşturabilir. Üstelik, **a** ve **b** **private** olduğu için, onlara değer atamanın tek yolu **factory()** 'yi kullanmaktır.

**Main()** 'de bir **MyClass** nesnesi örneklenmektedir ve ilave **10** nesne oluşturmak için bu nesnenin fabrika metodu **for** döngüsü içinde kullanılmaktadır. Bu işlemin yapıldığı kod satırı şudur:

```
MyClass anotherOb = ob.factory(i, j); // Bir nesne al
```

Döngünün her adımında, **anotherOb** adlı bir nesne referansı oluşturulmakta ve buna, fabrika tarafından kurulan nesneye bir referans atanmaktadır. Her adının sonunda **anotherOb** kapsam dışına çıkmakta ve onun referansta bulunduğu nesne geri dönüşüm işlemine tabi tutulmaktadır.

## Bir Diziyi Döndürmek

C#'ta diziler, nesneler olarak uygulandığından bir metot bir dizi de döndürebilir. (Bu, dizilerin geçerli dönüş tipi olmadığı C++'tan farklı bir durumdur.) Örneğin, aşağıdaki programda **findfactors()** metodu kendisine aktarılan argümanların çarpanlarını tutan bir dizi döndürmektedir:

```

// Bir diziyi geri dondurmek.

using System;

class Factor {
    /* num'un carpanlarini iceren bir diziyi geri dondurur.
       Geri donustre numfactors bulunan carpanlarinin
       sayisini icerir. */

```

```

public int[] findfactors(int num, out int numfactors) {
    int[] facts = new int[80]; //80'lik büyüklik keyfi seçilmiştir
    int i, j;

    // carpanları bul ve bunları facts dizisine koy
    for(i=2, j=0; i < num/2 + 1; i++)
        if( (num%i) == 0 ) {
            facts[j] = i;
            j++;
        }

    numfactors = j;
    return facts;
}

class FindFactors {
    public static void Main() {
        Factor f = new Factor();
        int numfactors;
        int[] factors;

        factors = f.findfactors(1000, out numfactors);

        Console.WriteLine("Factors for 1000 are: ");
        for(int i = 0; i < numfactors; i++)
            Console.Write(factors[i] + " ");

        Console.WriteLine();
    }
}

```

Programın çıktısı aşağıda gösterilmiştir,

```

Factors for 1000 are:
2 4 5 8 10 20 25 40 50 100 125 200 250 500

```

**Factor**'de **findfactors()** şu şekilde deklare edilmektedir:

```

public int[] findfactors(int num, out int numfactors) {

    int dizisinin dönüş tipinin nasıl belirlendiğine dikkat edin. Bu söz dizimi
    genelleştirilebilir. Bir metot bir dizi geri döndürüğünde tip ve boyutları gerektiği şekilde
    ayarlayarak dönüş tipini yukarıdakine benzer şekilde belirleyebilirsiniz. Örneğin, aşağıdaki
    ifade iki boyutlu bir double dizisi geri döndüren someMeth() adlı bir metot tanımlamaktadır.

    public double[,] someMeth() { // ...

```

## Metotların Aşırı Yüklenmesi

Bu bölümde, C#'ın en heyecan verici özelliklerinden biri olan metotların aşırı yüklenmesi konusunu öğreneceksiniz. C#'ta aynı sınıfın içindeki iki ya da daha fazla metot, parametre deklarasyonları farklı olduğu sürece, aynı ismi paylaşabilirler. Böyle metotlara *aşırı yüklenmiş*

metot denir. Bu işleme de *metodun aşırı yüklenmesi* adı verilir. Metodun aşırı yüklenmesi, C#'ın çok biçimliliği uygulama yollarından biridir.

Genelde bir metodu aşırı yüklemek için onun farklı versiyonlarını deklare etmek yeterlidir. Gerisini derleyici halleter. Yalnız, önemli bir kısıtlamaya dikkat etmelisiniz: Aşırı yüklenmiş her bir metodun parametre tipleri ve/veya sayıları farklı olmalıdır. İki metodun sadece dönüş tiplerinin farklı olması yeterli değildir. Parametrelerinin tip ya da sayıları farklı olmalıdır. (Dönüş tipleri, C#'ın olası her durumda hangi metodу kullanacağına karar verebilmesi için yeterli bilgi vermez.) Kuşkusuz, aşırı yüklenmiş metodların geri dönüş tiplerinde de farklılıklar olabilir. Aşırı yüklenmiş bir metot çağrıldığında, çağrıdaki argümanlarla aynı parametre sayısına ve parametre tiplerine sahip olan versiyon gerçekleştirir.

İşte, metot aşırı yüklenmesini gösteren basit bir örnek:

```
// Metot aşırı yüklenmesini gösterir.

using System;

class Overload {
    public void ovlDemo() {
        Console.WriteLine("No parameters");
    }

    // ovlDemo'yu tek tamsayı parametre için aşırı yükle.
    public void ovlDemo(int a) {
        Console.WriteLine("One parameter: " + a);
    }

    // ovlDemo'yu iki tamsayı parametre için aşırı yükle.
    public int ovlDemo(int a, int b) {
        Console.WriteLine("Two parameters: " + a + " " + b);
        return a + b;
    }

    // ovlDemo'yu iki double parametre için aşırı yükle.
    public double ovlDemo(double a, double b) {
        Console.WriteLine("Two double parameters: " + a + " "+ b);
        return a + b;
    }
}

class OverloadDemo {
    public static void Main() {
        Overload ob = new Overload();
        int resI;
        double resD;

        // ovlDemo'nun tüm surumlerini çağır
        ob.ovlDemo();
        Console.WriteLine();

        ob.ovlDemo(2);
        Console.WriteLine();
```

```

        resI = ob.ovlDemo(4, 6);
        Console.WriteLine("Result of ob.ovlDemo(4, 6): " + resI);
        Console.WriteLine();

        resD = ob.ovlDemo(1.1, 2.32);
        Console.WriteLine("Result of ob.ovlDemo(1.1, 2.32): " +
                          resD;
    }
}

```

Programın çıktısı şu şekildedir:

No parameters

One parameter: 2

Two parameters: 4 6
Result of ob.ovlDemo(4, 6): 10

Two double parameters: 1.1 2.32
Result of ob.ovlDemo(1.1, 2.32): 3.42

Gördüğünüz gibi, **ovlDemo()** dört kez aşırı yüklenmiştir. İlk versiyonun parametresi yoktur. İkincisinde tek bir **int** parametre vardır. Üçüncüsünde iki **int** parametre, dördüncüsünde de iki **double** parametre yer almaktadır. **ovlDemo**'nın ilk iki versiyonunun **void**, kalan iki versiyonun ise bir değer döndürdüğüne dikkat edin. Bu tamamen geçerli bir durumdur, ancak daha önce de açıkladığımız gibi, aşırı yükleme bir metodun geri dönüş tipinden şu ya da bu şekilde etkilenmez. Dolayısıyla, **ovlDemo**'nın şu iki versiyonunu kullanmaya çalışırsak hata ile karşılaşırız:

```

// Tek ovlDemo(int)'de bir problem yok.

public void ovlDemo(int a) {
    Console.WriteLine("One parameter: " + a);
}

/* Hata! Geri donus tipleri farkli olsa bile
   iki ovlDemo(int) gecerli degil. */
public int ovlDemo(int a) {
    Console.WriteLine("One parameter: " + a);
    return a * a;
}

```

Kod içindeki açıklamaların da işaret ettiği gibi, iki metodun geri dönüş tiplerinin farklı olması aşırı yükleme işlemi için yeterli değildir.

Bölüm 3'ten hatırlayacağınız gibi, C# bazı otomatik tip dönüşümleri sağlar. Bu dönüşümler, aşırı yüklenmiş metodların tiplerine de uygulanır. Aşağıdaki örneğe bakalım:

```

/* Otomatik tip donusumleri asiri yuklenmis metodlardan
   hangisinin kullanilacaginin belirlenmesinde etkili olabilir. */

using System;

```

```

class Overload2 {
    public void f(int x) {
        Console.WriteLine("Inside f(int): " + x);
    }

    public void f(double x) {
        Console.WriteLine("Inside f(double): " + x);
    }
}

class TypeConv {
    public static void Main() {
        Overload2 ob = new Overload2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.f(i); // ob.f(int)'i cagirir
        ob.f(d); // ob.f(double)'i cagirir

        ob.f(b); // ob.f(int)'i cagirir -- tip donusumu
        ob.f(s); // ob.f(int)'i cagirir -- tip donusumu
        ob.f(f); // ob.f(double)'i cagirir -- tip donusumu
    }
}

```

Program, şu çıktıyı üretir:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(int): 99
Inside f(int): 10
Inside f(double): 11.5

```

Bu örnekte, **f()**'nin sadece iki versiyonu tanımlanmıştır: **int** parametresi olan bir versiyon ve **double** parametresi olan ikinci versiyon. Ancak **f()**'ye bir **byte**, **short** ya da **float** değeri aktarmak mümkündür. **byte** ve **short** kullanıldığında C# bunları otomatik olarak **int**'e dönüştürür. Dolayısıyla **f(int)** çağrıılır. **float** kullanıldığında ise bu parametre **double**'a dönüştürülür ve **f(double)** kullanılır.

Otomatik dönüşümlerin ancak parametrelerle argümanlar arasında doğrudan bir eşleşme olmadığı durumlarda kullanıldığını anlamak önemlidir. Örneğin, yukarıdaki programda **f()**'nin **byte** parametrelili bir versiyonu olduğunda farklı bir durum ortaya çıkacaktır:

```

// f(byte)'i ekle.

using System;

```

```

class Overload2 {
    public void f(byte x) {
        Console.WriteLine("Inside f(byte): " + x);
    }

    public void f(int x) {
        Console.WriteLine("Inside f(int): " + x);
    }

    public void f(double x) {
        Console.WriteLine("Inside f(double): " + x);
    }
}

class TypeConv {
    public static void Main() {
        Overload2 ob = new Overload2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.f(i); // ob.f(int)'i cagirir
        ob.f(d); // ob.f(double)'i cagirir

        ob.f(b); /* ob.f(byte)'i cagirir -
                    bu durumda tip donusumu olmaz */

        ob.f(s); // ob.f(int)'i cagirir -- tip donusumu
        ob.f(f); // ob.f(double)'i cagirir -- tip donusumu
    }
}

```

Şimdi bu programı çalıştırıldığımızda aşağıdaki çıktıyi elde ederiz:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(byte): 99
Inside f(int): 10
Inside f(double): 11.5

```

Artık **f()**'nin **byte** argümanı alan bir versiyonu mevcut olduğundan, **f()** bir **byte** argümanla çağrıldığında parametre otomatik olarak **int**'e dönüştürülmez ve **f(byte)** çağrıılır.

Aşırı yükleme versiyonlarından hangisinin kullanılacağını belirlemeye hem **ref** hem de **out** rol alır. Örneğin, aşağıdaki programda iki farklı metot tanımlanmaktadır:

```

public void f(int x) {
    Console.WriteLine("Inside f(int): " + x);
}

public void f(ref int x) {

```

```
        Console.WriteLine("Inside f(ref int): " + x);
    }
```

Bu nedenle,

```
ob.f(i)
f(int x)'i,
ob.f(ref i)
```

ise **f(ref int x)**'i çağrırmaktadır.

Metotların aşırı yüklenmesi, çok biçimliliği destekler, çünkü bu, C#'ın “tek arayüz, çok metot” modelini uygulamaya geçirme yöntemlerinden biridir. Şöyle ki: Metotların aşırı yüklenmesini desteklemeyen dillerde her metodun ismi farklı olmalıdır. Ancak sık sık, aslında tamamen aynı işlemi yapan, sadece kullandıkları veri tiplerinde ayrılan metotları uygulamak zorunda kalırsınız. Örnek olarak “mutlak değer” fonksiyonunu ele alalım. Aşırı yüklemeyi desteklemeyen dillerde bu fonksiyonun genelde üç ya da daha fazla versiyonu vardır. Her birinin adı biraz farklıdır. Örneğin C'de **abs()** fonksiyonu bir tamsayının mutlak değerini döndürken, **labs()** fonksiyonu **long int** için, **fabs()** da kayan noktalı değerler için kullanılır. C, aşırı yüklemeyi desteklemediği için, bu fonksiyonların yaptığı iş esasen aynımasına rağmen, üç farklı ismi hatırlamak gerekmektedir. Bu durum C#'ta geçerli değildir, çünkü her bir mutlak değer metodu aynı ismi kullanabilir. Hatta C#'in standart sınıf kütüphanesinde **Abs()** adlı bir mutlak değer fonksiyonu vardır. Bu metot C#'in **System.Math** sınıfı tarafından nümerik sınıflar üzerinde işlem yapabilecek şekilde aşırı yüklemeye tabi tutulmaktadır. C#, **Abs()**'ın hangi versiyonunun kullanılacağına, argümanın tipine bakarak kendisi karar vermektedir.

Aşırı yüklemenin başlıca yararlarından biri, birbirleriyle ilgili metotlara ortak bir isimle erişim sağlanmasıdır. Yani, **Abs** ismi, gerçekleştirilmekte olan *genel faaliyeti* simgelemektedir. Belli bir durumda metodun hangi *özel* versiyonunun kullanılacağına karar vermek derleyiciye kalmaktadır. Siz programcıların sadece yapılan genel işlemi hatırlaması yeterli olmaktadır. Çok biçimliliğin uygulanması ile, birden fazla isim tek bir isme indirgenmiştir. Belki bu örnek oldukça basittir ama kavramı genişlettiniz takdirde, aşırı yüklemenin çok daha büyük boyutlu karmaşık durumların yönetilmesini nasıl kolaylaştırdığını anlayabilirsiniz.

Bir metodu aşırı yüklediğiniz zaman, o metodun her bir versiyonu istediğiniz herhangi bir işi yapabilir. Aşırı yüklenmiş metodların birbirleriyle ilintili olması gerektigine dair bir kural yoktur. Ancak, stil açısından, metodların aşırı yüklenmesi böyle bir ilişkinin var olduğunu ima etmektedir. Dolayısıyla, birbirleriyle ilgisi olmayan metodları aşırı yüklemek için aynı ismi kullanmanız mümkün olsa bile bunu yapmamalısınız. Örneğin, **sqr** adı altında, parametresi **int** olunca sayının karesini, parametresi **float** olunca sayının karekökünü döndüren metodlar

oluşturmamalısınız. Metotların aşırı yüklenmesini bu şekilde uygulamak amaca ters düşer. Pratikte, sadece birbirleriyle çok yakından ilgili işlemler üzerinde aşırı yükleme yapmalısınız.

C#'ta *imza* (*signature*) terimi tanımlanmaktadır. Bu terim, bir metot ile onun parametre listesini içerir. Aşırı yükleme açısından değerlendirildiğinde, aynı sınıf içinde iki metot aynı imzaya sahip olamaz. İmza tanımının metodun dönüş tipini içermediğine dikkat edin. Bunun nedeni, C#'ın aşırı yüklemeyi çözümlemek için geri dönüş tipini kullanmamasıdır. Ayrıca, imza, eğer varsa **params** parametresini de içermemektedir, **params** da aşırı yükleme çözümlenmesinde kullanılmamaktadır.

## Yapilandırıcıları Aşırı Yüklemek

Tıpkı metotlar gibi, yapılandırıcılar da aşırı yüklenebilir. Bu şekilde, nesneleri çok çeşitli yollardan yapılandırmanız mümkün olur. Örneğin, aşağıdaki programı ele alın:

```
// Asırı yüklenmiş bir yapılandırıcıyı gösterir.

using System;

class MyClass {
    public int x;

    public MyClass() {
        Console.WriteLine("Inside MyClass()");
        x = 0;
    }

    public MyClass(int i) {
        Console.WriteLine("Inside MyClass(int)");
        x = i;
    }

    public MyClass(double d) {
        Console.WriteLine("Inside MyClass(double)");
        x = (int) d;
    }

    public MyClass(int i, int j) {
        Console.WriteLine("Inside MyClass(int, int)");
        x = i * j;
    }
}

class OverloadConsDemo {
    public static void Main() {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass(88);
        MyClass t3 = new MyClass(17.23);
        MyClass t4 = new MyClass(2, 4);

        Console.WriteLine("t1.x: " + t1.x);
        Console.WriteLine("t2.x: " + t2.x);
```

```

        Console.WriteLine("t3.x: " + t3.x);
        Console.WriteLine("t4.x: " + t4.x);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.X: 8

```

**MyClass()**, her biri ile farklı bir nesne kurulan dört şekilde aşırı yüklenebilir. **new** gerçeklendiğinde, normal yapılandırıcı belirtilen parametreler baz alınarak çağrılır. Bir sınıfın yapılandırıcısını aşırı yükleyerek sınıfınızın kullanıcısına nesnelerin yapılandırılma şekliyle ilgili esneklik verirsiniz.

Yapılandırıcıların aşırı yüklenmesinin en yaygın nedenlerinden biri, bir nesnenin diğerine ilk değer atamasına imkan vermektedir. Örneğin, işte bir yiğindan bir başka yiğin kurmayı mümkün kıyan ve daha önce geliştirilen **Stack** sınıfının ileri versiyonu:

```

// Karakterler için bir yiğin sınıfı.

using System;

class Stack {
    // bu üyeleri private'tır
    char[] stck; // yiğini tutar
    int tos;      // yiğinin tepesinin indeksi

    // buyuklugu bilinen bir boş Stack yapılandırır
    public Stack(int size) {
        stck = new char[size]; // yiğin için bellek alanı ayır
        tos = 0;
    }

    // Yiğindan bir Stack yapılandırır.
    public Stack(Stack ob) {
        // yiğin için bellek alanı ayır
        stck = new char[ob.stck.Length];

        // elemanları yeni yiğine kopyala
        for(int i=0; i < ob.tos; i++)
            stck[i] = ob.stck[i];

        // yeni yiğin için tos'u ayarla
        tos = ob.tos;
    }

    // Karakterleri yiğine ekle.
}

```

```

public void push(char ch) {
    if(tos == stck.Length) {
        Console.WriteLine(" -- Stack is full.");
        return;
    }

    stck[tos] = ch;
    tos++;
}

// Yigindan bir karakter cikar.
public char pop() {
    if(tos == 0) {
        Console.WriteLine(" -- Stack is empty.");
        return (char) 0;
    }

    tos--;
    return stck[tos];
}

// Yigin doluya true dondur.
public bool full() {
    return tos==stck.Length;
}

// Yigin bossa true dondur.
public bool empty() {
    return tos==0;
}

// Yiginin toplam kapasitesini dondur.
public int capacity() {
    return stck.Length;
}

// Su an yiginda olan nesnelerin sayisini dondur.
public int getNum() {
    return tos;
}
}

// Stack sinifini goster.

class StackDemo {
    public static void Main() {
        Stack stk1 = new Stack(10);
        char ch;
        int i;

        // stk1'e birkac karakter yerlestir
        Console.WriteLine("Push A through Z onto stk1.");
        for(i = 0; !stk1.full(); i++)
            stk1.push((char) ('A' + i));
    }
}

```

```

// stck1'in bir kopyasini olustur
Stack stk2 = new Stack(stk1);

// stk1' in icerigini goster.
Console.WriteLine("Contents of stk1: ");
while( !stk1.empty() ) {
    ch = stk1.pop();
    Console.Write(ch);
}

Console.WriteLine();

Console.WriteLine("Contents of stk2: ");
while ( !stk2.empty() ) {
    ch = stk2.pop();
    Console.Write(ch);
}

Console.WriteLine("\n");
}
}

```

Çıktı aşağıdaki gibidir:

```

Push A through Z onto stk1.
Contents of stk1: JIHGFEDCBA
Contents of stk2: JIHGFEDCBA

```

**StackDemo**'da ilk yığın olan **stk1**, yapılandırılır ve karakterlerle doldurulur. Bu yığın daha sonra ikinci yığınının, yani **stk2**'nin yapılandırılması için kullanılır. Bu işlem aşağıdaki **stack** yapılandırıcısının çalıştırılmasına neden olur:

```

// Yigindan bir Stack yapilandir

public Stack(Stack ob) {
    // yigin icin bellek alani ayir
    stck = new char[ob.stck.Length];

    // elemanlari yeni yigina kopyala
    for(int i = 0; i < ob.tos; i++)
        stck[i] = ob.stck[i];

    // yeni yigin icin tos'u ayarla
    tos = ob.tos;
}

```

Bu yapılandırıcı içinde, **ob** parametresi üzerinden aktarılan yığının elemanlarını tutabilecek uzunlukta bir dizi için bellek alanı ayrırlır. Sonra, **ob** dizisinin içeriği yeni diziye kopyalanır ve **tos**'a uygun bir değer atanır. Yapılandırıcısının işlemi bittikten sonra yeni ve orijinal yığınlar ayrı ama aynıdır.

## this Aracılığıyla Aşırı Yüklenmiş Bir Yapılandırıcıyı Çağırma

Aşırı yüklenmiş yapılandırıcılarla çalışırken bir yapılandırıcının diğerini çağırması kimi zaman kullanışlı olur. C#'ta bu, **this** anahtar kelimesinin bir başka şekli kullanılarak gerçekleştirilebilir. Genel yapı aşağıda gösterildiği gibidir:

```
yapilandirici-ismi(parametre-listesi1) : this(parametre-listesi2) {
    // ... yapılandırıcı gövdesi; gövde boş olabilir
}
```

Yapılandırıcı çalıştırıldığında, **parametre-listesi2** ile belirtilen parametre listesi ile eşleşen aşırı yüklenmiş yapılandırıcı ilk olarak çalıştırılır. Sonra, orijinal yapılandırıcı içinde mevcut ifade varsa bu ifadeler çalıştırılır. İşte bir örnek:

```
// this aracılığıyla bir yapılandırıcı çağrımayı gösterir.

using System;

class XYCoord {
    public int x, y;

    public XYCoord() : this(0, 0) {
        Console.WriteLine("Inside XYCoord()");
    }

    public XYCoord(XYCoord obj) : this(obj.x, obj.y) {
        Console.WriteLine("Inside XYCoord(obj)");
    }

    public XYCoord(int i, int j) {
        Console.WriteLine("Inside XYCoord(int, int)");
        x = i;
        y = j;
    }
}

class OverloadConsDemo {
    public static void Main() {
        XYCoord t1 = new XYCoord();
        XYCoord t2 = new XYCoord(8, 9);
        XYCoord t3 = new XYCoord(t2);

        Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
        Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
        Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```

Inside XYCoord(int, int)
Inside XYCoord()
Inside XYCoord(int, int)
Inside XYCoord(int, int)
Inside XYCoord(obj)
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9

```

Programın işleyişi şu şekildedir: **XYCoord** sınıfında, **x** ve **y** alanlarına gerçekten ilk değer atayan tek yapılandırıcı **XYCoord(int, int)** metodudur. Diğer iki yapılandırıcı sadece **this** aracılığıyla **XYCoord{int, int}** metodunu çağrırlar. Örneğin, **t1** nesnesi oluşturulduğunda bu nesnenin yapılandırıcısı, **XYCoord()** çağrılr. Bu, **this(0, 0)**'ın çalıştırılmasına neden olur ki bu, bu örnekte, **XYCoord(0, 0)**'a yapılan bir çağrıya dönüşmektedir. **t2**'nin oluşturulması da aynı şekilde gerçekleştirilir.

Aşırı yüklenmiş yapılandırıcıları **this** aracılığıyla çağrımanın kullanışlı olabilmesinin nedenlerinden biri, bu sayede kodun gereksiz yere tekrarının önlenmesidir. Yukarıdaki örnekte, yapılandırıcıların üçünün de aynı ilk değer atama sekansını tekrarlamaları için bir neden yok. Bu tekrarlar **this** kullanılarak önlenmektedir. Bir başka avantajı da şudur: Bu sayede, argümanlar açıkça belirtilmemişinde kullanılan dolaylı “varsayılan argümanlara” sahip yapılandırıcılar oluşturabilirsiniz. Örneğin, aşağıda gösterilen şekilde, bir başka **XYCoord** yapılandırıcısı oluşturabilirsiniz:

```
public XYCoord(int x) : this(x, x) { }
```

Bu yapılandırıcı otomatik olarak **y** koordinatına, **x** koordinatı ile aynı değeri varsayılan değer olarak atar. Bu “varsayılan argümanları” dikkatlice kullanmak elbette akıllıca olur; çünkü bu argümanların yanlış kullanımı, sınıflarınızı kullanan kullanıcıları kolaylıkla karışıklığa düşürebilir.

## Main () Metodu

Şu ana dek **Main()** metodunun tek şeklini kullanmaktayınız. Ancak, **Main()**'in aşırı yüklenmiş çeşitli şekilleri vardır. Kimisi bir değer döndürmek için kullanılabilir, kimisi argüman alabilir. Bunların her biri burada incelenmektedir.

### Main () 'den Değer Döndürmek

Program sona erdiğinde, **Main()**'den bir değer döndürerek **Main()**'i çağrıran prosese (genellikle bu işletim sistemidir) bir değer döndürebilirisiniz. Bunun için **Main()**'in şu şeklini kullanabilirsiniz:

```
public static int Main()
```

Dikkat ederseniz, **void** olarak deklare edilmek yerine **Main()**'in bu versiyonu **int** dönüş tipine sahiptir.

Genellikle **Main()**'in dönüş değeri, programın normal olarak mı, yoksa bazı anormal koşullara bağlı olarak mı sona erdiğini gösterir. Alışıldık olarak, **0** dönüş değeri genellikle normal bir sona işaret eder. Diğer değerlerin tümü bir tür bir hatanın meydana geldiğini gösterir.

## Main ()'e Argüman Aktarmak

Programların pek çoğu, *komut satırı argümanı* denilen argümanları alırlar. Komut satırı argümanı; program çalıştırıldığında, komut satırı üzerinde program isminin doğrudan peşinden gelen bilgidir. C# programlarında bu argümanlar **Main()** metoduna aktarılır. Söz konusu argümanları almak için **Main()**'in aşağıdaki şekillerinden birini kullanmalısınız:

```
public static void Main(string[] argümanlar)
public static int Main(string[] argümanlar)
```

İlk şekil **void** döndürür; ikincisi, önceki bölümde anlatıldığı gibi, bir tamsayı değer döndürmek için kullanılabilir. Her ikisi için de komut satırı argümanları **Main()**'e aktarılan bir **string** dizisinde birer karakter katarı olarak saklanırlar.

Örneğin, aşağıdaki program, kendisi ile birlikte çağrılan komut satırı argümanlarının tümünü ekranda gösterir:

```
// Tüm komut satırı bilgisini gösterir.

using System;

class CLDemo {
    public static void Main(string[] args) {
        Console.WriteLine("There are " + args.Length +
                           " command-line arguments.");

        Console.WriteLine("They are:");
        for(int i = 0; i < args.Length; i++)
            Console.WriteLine(args[i]);
    }
}
```

Eğer **CLDemo** şu şekilde çalıştırılırsa:

```
CLDemo one two three
```

Aşağıdaki çıktıyı ekranda göreceksiniz:

```
There are 3 command-line arguments.
They are:
one
two
three
```

Komut sahn argümanlarının ne şekilde kullanılabileceğinin tadına varmak için bir sonraki programı ele alın. Bu program mesajları şifreler ya da mesaj şifrelerini çözer. Şifrelenecek veya

şifresi çözülecek mesaj komut satırında belirtilir. Şifreleme yöntemi çok basittir: Bir kelimeyi şifrelemek için her harf bir artırılır. Böylece; A, B olur ve bu şekilde sürer. Şifre çözmek içinse her harf bir azaltılır.

```
// Mesaji sifreler veya mesajin sifresini cozer.

using System;

class Cipher {
    public static int Main(string[] args) {

        // argumanların mevcut olup olmadığını kontrol et
        if(args.Length < 2) {
            Console.WriteLine("Usage: encode/decode
                            word1 [word2...wordN]");
            return 1; // başarısızlık kodunu dondur
        }

        /* argumanlar mevcutsa, ilk arguman "encode" veya "decode"
           olmalıdır */
        if(args[0] != "encode" & args[0] != "decode") {
            Console.WriteLine("First arg must be encode or decode.");
            return 1; // başarısızlık kodunu dondur
        }

        // mesajı şifrele ya da şifresini coz
        for(int n = 1; n < args.Length; n++) {
            for(int i = 0; i < args[n].Length; i++) {
                if(args[0] == "encode")
                    Console.Write((char) [args[n][i] + 1] );
                else
                    Console.Write((char) (args[n][i] - 1) );
            }
            Console.WriteLine(" ");
        }
        Console.WriteLine();

        return 0;
    }
}
```

Programı kullanmak için şifrelenmesini ya da şifresinin çözülmesini istediğiniz kelime öbeğinin peşinden “**encode**” veya “**decode**” komutunu belirtin. Programın **Cipher** olarak adlandırıldığını varsayırsak iki örnek çalışma şekli şöyle olabilir:

```
C:Cipher encode one two
pof uxp
```

```
C:Cipher decode pof uxp
one two
```

Bu programda iki ilginç özellik mevcuttur. Birincisi; programın, çalışmasına devam etmeden önce, komut satırı argümanının mevcut olup olmadığını nasıl kontrol ettiğine dikkat edin.

Bu çok önemlidir ve genelleştirilebilir. Program, bir veya daha fazla komut satırı argümanının mevcut olmasına bel bağlarsa, doğru argümanların tedarik edilmiş olduğunu daima doğrulamalıdır. Bunu başaramaması, programın çökmesine yol açabilir. Ayrıca, komut satırı argümanlarının ilki “**encode**” ya da “**decode**” olması gereği için, program ilerlemeye başlamadan bunu da kontrol eder.

İkincisi, programın bitiş kodunu nasıl döndürdüğünə dikkat edin. Eğer gerekli olan komut satırı mevcut değilse, anormal sonlanmaya işaret eden **1** döndürülür. Aksi halde, program sona erdiğinde **0** döndürülür.

## Yinelenme

C#’ta bir metot kendi kendisini çağırabilir. Bu işleme *yinelenme (recursion)* denir ve kendi kendisini çağrıran bir metoda da *yinelenen (recursive)* metot denilir. Yinelenme genel olarak bir şeyi kendisiyle tanımlama anlamına gelir ve kendi kendisine referansta bulunarak yapılan tanıma (circular definition) biraz benzer. Yinelenen bir metodun en önemli bileşeni, kendi kendisini çağrıran bir ifadeye sahip olmasıdır. Yinelenme, güçlü bir kontrol mekanizmasıdır.

Yinelenmenin klasik örneği, bir sayının faktöriyelinin hesaplanmasıdır. **n** sayısının faktöriyeli, **1** ile **n** arasındaki tüm tam sayıların çarpımından oluşur. Örneğin **3** faktöriyel **1x2x3** yani **6**’dır. Aşağıdaki program bir sayının faktöriyelinin yinelenme ile nasıl hesaplandığını göstermektedir. Karşılaştırma yapabilmek için aynı programın yinelenme içermeyen bir versiyonu da verilmiştir.

```
// Basit bir yinelenme örneği.

using System;

class Factorial {
    // Bu yinelenmeli bir fonksiyondur.
    public int factR(int n) {
        int result;

        if(n == 1) return 1;
        result = factR(n - 1) * n;
        return result;
    }

    // Bu ise aynı fonksiyonun iterasyonlu esdeğeriidir.
    public int factI(int n) {
        int t, result;

        result = 1;
        for(t = 1; t = n; t++) result *= t;
        return result;
    }
}

class Recursion {
    public static void Main() {
```

```
Factorial f = new Factorial();

Console.WriteLine("Factorials using recursive method.");
Console.WriteLine("Factorial of 3 is " + f.factR(3));
Console.WriteLine("Factorial of 4 is " + f.factR(4));
Console.WriteLine("Factorial of 5 is " + f.factR(5));
Console.WriteLine();

Console.WriteLine("Factorials using iterative method.");
Console.WriteLine("Factorial of 3 is " + f.factI(3));
Console.WriteLine("Factorial of 4 is " + f.factI(4));
Console.WriteLine("Factorial of 5 is " + f.factI(5));
}
}
```

Programın çıktısı şöyledir:

```
Factorials using recursive method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Factorials using iterative method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Yinelemeli **factI()** metodunun nasıl çalıştığı herhalde açıkları. Bu metot, **1**'den başlayarak her adımda o ana kadar getirilen çarpım sonucunu o adımdaki sayı ile çarparak ilerlemektedir.

Yinelenmeli **factR()** metodunun işleyişi ise biraz daha karmaşıktr. **factR()**, **1** argümanı ile çağrıldığında metot, **1** döndürmektedir. Başka bir argümanla çağrıldığında ise döndürdüğü değer **factR(n-1)\*n** ifadesinin sonucudur. Bu ifadenin değerini bulmak için **factR()** bu kez de **n-1** argümanıyla çağrılmaktadır. Kullanılan argüman, yani **n**, **1** sayısına eşit olana kadar bu süreç devam etmektedir. Örneğin, **2**'nin faktöriyeli hesaplanırken, **factR()**'a yapılan ilk çağrı sonucunda **1** argümanı ile ikinci bir çağrı oluşturulmaktadır. İkinci çağrı **1** argümanı ile yapıldığı için **1** döndürmeye, bu rakam daha sonra **2** ile (yani **factR()**'a yapılan ilk çağrıının argümanı ile) çarpılmakta ve böylece ilk çağrıının döndürüceği sayı bulunmaktadır. Eğer denemek isterseniz, **factR()**'ın içine **WriteLine()** ifadeleri koyarak metodun her seviyede nasıl çağrıdığını ve ara çarpımların ne olduğunu görmek ilginç olabilir.

Bir metot kendi kendisini çağrılığında sistem yiğinında (system stack) yeni yerel değişkenlere ve parametrelerle yer ayrılır. Metodun kodu, baştan itibaren bu yeni değişkenler kullanılarak çalıştırılır. Yinelenen bir çağrı metodun yeni bir kopyasını oluşturmaz. Sadece argümanlar yenidir. Yinelenen çağrılar geri döndüğünde eski yerel değişkenler ve parametreler yiğinden çıkartılırlar ve programın işleyişi, metodun içinde çağrıının yapıldığı noktadan devam eder. Yinelenen metodların adeta bir "teleskop" gibi uzayıp geri kısaldığını düşünebiliriz.

İşte bir başka yinelenme örneği daha. Aşağıda yer verdigimiz **displayRev()** metodu, kendisine aktarılan karakter katarı argümanını tersinden yazmak için yinelenme kullanmaktadır.

```
// Bir karakter katarini yinelenme kullanarak tersinden yazar.

using System;

class RevStr {

    // Karakter katarini tersinden yaz.
    public void displayRev(string str) {
        if(str.Length > 0)
            displayRev(str.Substring(1, str.Length - 1));
        else
            return;

        Console.Write(str[0]);
    }
}

class RevStrDemo {
    public static void Main() {
        string s = "this is a test";
        RevStr rsOb = new RevStr();

        Console.WriteLine("Original string: " + s);

        Console.WriteLine("Reversed string: ");
        rsOb.displayRev(s);

        Console.WriteLine();
    }
}
```

Programın çıktısı şöyledir:

```
Original string: this is a test
Reversed string: tset a si siht
```

Programda **displayRev()** her çağrıldığında, önce **str**'nin uzunluğunun sıfırdan büyük olup olmadığını kontrol eder. Eğer uzunluk sıfırdan büyükse, **displayRev()** **str**'den ilk karakterin çıkarılmasıyla oluşan yeni bir karakter katarı ile yinelenmeli olarak çağrılır. Bu süreç, uzunluğu sıfır olan bir karakter katarı parametre olarak aktarılana kadar devam eder. Sıfır uzunluktaki karakter katarına ulaşıldığında, yani orijinal karakter katarının sonu geldiğinde, bu sefer de yinelenme işlemleri çözülerek geri dönülür ve her adımda o adımdaki karakter katarının baş harfi yazılır. Bu da orijinal karakter katarının tersten yazılmasını sağlar.

Pek çok program parçasının yinelenmeli versiyonları, iterasyonlu versiyonlardan biraz daha yavaş çalışır. Bunun nedeni, yapılan ek metod çağrılarının kaynak kullanımı ve yönetim bakımından ek iş çıkarmasıdır. Bir metodun aşırı sayıda yinelenmeli olarak çağrılması sistem yığınının taşmasına (stack overflow) neden olabilir. Parametreler ve yerel değişkenler sistem

yığınında saklandığı için, her yinelenmeli çağrı bu parametrelerin yeni bir kopyasını oluşturur. Bu da bazı durumlarda yığın kapasitesinin dolmasına halta taşmasına neden olabilir. Bu da C#'ın kural dışı durum oluşturulmasıyla sonuçlanır. Ancak, yinelenmeli bir çağrı kontrolden çıkmadıkça, büyük olasılıkla, bu konuda endişe etmenize gerek olmayacağından.

Yinelenmenin ana avantajı, bazı algoritmaların iterasyonlu biçimlerine göre yinelenmeli olarak daha basit ve net uygulanabilmesidir. Örneğin, **Quicksort** algoritmasını iterasyonlu olarak uygulamak oldukça güçtür. Aynca, bazı problemler, özellikle de yapay zekayla ilgili problemler, doğaları itibarıyle yinelenmeli çözümlere daha yatkın görülmektedir.

Yinelenmeli metotları yazarken mutlaka **if** gibi bir koşullu ifade kullanmalısınız. Bu ifade, yeni bir yinelenmeli çağrı yapmadan geri dönmelidir. Aksi halde metodu çağrıdığınızda asla geri dönmeyecektir. Yinelenme ile çalışırken bu hata çok yaygın olarak yapılar. Yinelenmeye uğraşırken bol bol **writeline()** ifadeleri kullanarak işleyişi izleyebilir ve bir hata yaptıysanız programın çalışmasını durdurabilirsiniz.

## static'i Anlamak

Bazen bir sınıfın herhangi bir nesnesinden bağımsız olarak kullanacağınız bir sınıf üyesi tanımlamanız gerekebilir. Normal olarak bir sınıf üyesine o sınıfın bir nesnesi üzerinden erişilmesi gereklidir. Ancak, belli bir örneğe referansta bulunmadan kendi başına kullanılabilecek bir üye oluşturmak da mümkündür. Böyle bir üye oluşturmak için, o üyenin deklarasyonundan önce başına **static** anahtar kelimesini yazın. Bir üye **static** olarak tanımlandığı zaman, o üye, kendi sınıfından herhangi bir nesne oluşturulmadan önce ve herhangi bir nesneye referansta bulunmadan kullanılabilir. Hem metotları hem de değişkenleri **static** olarak deklare edebilirsiniz. **static** bir üyenin en yaygın örneği **Main()**'dır. **Main()**, programınız başladığında işletim sistemi tarafından çağrılmak zorunda olduğu için **static** olarak deklare edilmek zorundadır.

**static** bir üyesi sınıfın dışında kullanmak için üyenin sınıfını yazmalı, peşinden de nokta operatörünü kullanmalısınız. Bu durumda herhangi bir nesne oluşturmanız gerekmeyez. Hatta, **static** bir üye bir nesne örneği üzerinden erişmek mümkün değildir. **static** bir üye ancak kendi sınıfının adı üzerinden erişilebilir. Örneğin, **Timer** adlı bir sınıfa ait **count** adlı bir **static** değişkene **10** değerini atamak istiyorsanız su satırı kullanın:

```
Timer.count = 10;
```

Bu biçim, normal örnek değişkenlerine bir nesne üzerinden erişme yöntemine benzemektedir; tek farkı, sınıf isminin kullanılıyor olmasıdır. **static** bir metot da aynı şekilde, yani sınıf isminden sonra nokta operatörü kullanılarak çağrılabılır.

**static** olarak deklare edilen değişkenler esas itibarıyle global değişkenlerdir. Bir **static** değişkenin sınıfına ait nesneler deklare edildiğinde **static** değişkenin herhangi bir kopyası oluşturulmaz. Sınıfın tüm örnekleri aynı **static** değişkeni paylaşır. **static** bir değişkene ilk değer ataması, o değişkenin sınıfı yüklenirken yapılır. İlk değer ataması açık bir

şekilde yapılmıyorsa, değişkene, nümerik değerler için 0, nesne referansları için **null**, **bool** tipi değişkenler için ise **false** değeri atanır. Dolayısıyla, **static** bir değişkenin her zaman bir değeri vardır.

**static** bir metotla normal bir metot arasındaki fark, **static** metodun herhangi bir nesne oluşturmaya gerek kalmadan sınıfının ismi kullanılarak çağrılabilmesidir. Bunun bir örneğini daha önce görmüştük: C#'ın **System.Math** sınıfı içinde **static** bir metot olan **Sqrt()** böyle bir metottu.

İşte, **static** bir değişken ve **static** bir metot oluşturan bir örnek:

```
// Static kullanır.

using System;

class StaticDemo {
    // Static bir degisken
    public static int val = 100;

    // Static bir metod
    public static int valDiv2() {
        return val / 2;
    }
}

class SDemo {
    public static void Main() {

        Console.WriteLine("Initial value of StaticDemo.val is "
            + StaticDemo.val);

        StaticDemo.val = 8;
        Console.WriteLine("StaticDemo.val is " + StaticDemo.val);
        Console.WriteLine("StaticDemo.valDiv2(): " +
            StaticDemo.valDiv2());
    }
}
```

Program şu çıktıyı üretir:

```
Initial value of StaticDemo.val is 100
StaticDemo.val is 8
StaticDemo.valDiv2(): 4
```

Çıktıdan anlaşıldığı gibi, **static** bir değişkene ilk değer program başladığında, değişkenin sınıfından herhangi bir nesne oluşturulmadan önce atanır.

**static** metodlarla ilgili çeşitli kısıtlamalar vardır:

1. **static** bir metodun bir **this** referansı olamaz.
2. **static** bir metot diğer **static** metodları doğrudan çağrıabilir. Ancak kendi sınıfından bir örnek metodu doğrudan çağrıramaz. Bunun nedeni, örnek metodların

bir sınıfa özgü örnekler üzerinde çalışması, **static** bir metodun ise bu özelliğe sahip olmamasıdır.

3. **static** bir metot sadece **static** olan verilere doğrudan erişmek zorundadır. **static** bir metot bir örnek değişkenine doğrudan erişemez, çünkü sınıfının bir örneği üzerinde işlem yapmak durumunda değildir.

Örneğin, aşağıdaki sınıfta, **valDivDenom()** adlı **static** metot geçersizdir:

```
class StaticError {
    int denom = 3; // normal bir örnek degisken
    static int val = 1024; // static bir degisken

    // Hata! static olmayan bir degiskene
    // static bir metodun icinden doğrudan erisilemez */
    static int valDivDenom() {
        return val/denom; // hata oldugundan derleme gereklesmez!
    }
}
```

Burada **denom**, **static** bir metodun içinden erişilmesi mümkün olmayan normal bir örnek değişkenidir. Ancak **val**'ın kullanımında problem yoktur, çünkü **val**, **static** bir değişkendir.

Aynı problem, **static** olmayan bir metot, aynı sınıfın **static** bir metodunun içinden çağrılmaya çalışıldığında da ortaya çıkar. Örneğin:

```
using System;

class AnotherStaticError {
    // static olmayan metod.
    void nonStaticMeth() {
        Console.WriteLine("Inside nonStaticMeth().");
    }

    /* Hata! static bir metodun icinden static olmayan
     * bir metot doğrudan çağrılamaz. */
    static void staticMeth() {
        nonStaticMeth(); /* hatalı olduğu için derleme
                           gereklesmeyecektir. */
    }
}
```

Bu durumda **static** olmayan (yani örnek olan) bir metodу **static** bir metodun içinden çağrırmaya çalışmak sonucunda derleyici hatası ortaya çıkmıştır.

Şu nokta önemlidir: **static** bir metot, kendi sınıfının örnek metodlarını çağırabilir ve örnek değişkenlerine erişebilir. Ancak bunu o sınıfın bir nesnesi üzerinden yapmak zorunluluğundadır. Bunlarla ilgili tek kısıtlama, örnek metodları ya da değişkenleri nesne belirtmeden kullanamamasıdır. Örneğin aşağıdaki program parçası gayet geçerlidir:

```
class MyClass {
    // static olmayan metod.
```

```

void nonStaticMeth() {
    Console.WriteLine("Inside nonStaticMeth().");
}

/* static olmayan bir metot static bir metodun icinden nesne
referansi kullanarak cagrilabilir. */
public static void staticMeth(MyClass ob) {
    ob.nonStaticMeth(); // Bu ifade gecerlidir.
}
}

```

**static** alanlar herhangi spesifik bir nesneden bağımsız olduğu için bu alanlar tüm bir sınıfı uygulanması söz konusu bilgileri tutmak için yararlıdır. Böyle bir durumun örneği aşağıda verilmiştir. Bu örnekte, eldeki nesne sayısını tutmak için **static** bir alan kullanılmaktadır.

```

// Ornekleri saymak icin static bir alan kullanir.

using System;

class CountInst {
    static int count = 0;

    // nesne olusturulduğunda count'u artir
    public CountInst() {
        count++;
    }

    // nesne yok edildiginde count'u eksilt
    -CountInst() {
        Count--;
    }

    public static int getCount() {
        return count;
    }
}

class CountDemo {
    public static void Main() {
        CountInst ob;

        for(int i = 0; i < 10; i++) {
            ob = new CountInst();
            Console.WriteLine("Current count: " +
                CountInst.getCount());
        }
    }
}

```

Programın çıktısı şu şekildedir:

```

Current count: 1
Current count: 2
Current count: 3

```

```
Current count: 4
Current count: 5
Current count: 6
Current count: 7
Current count: 8
Current count: 9
Current count: 10
```

**CountInst** tipinden bir nesne her oluşturulduğunda, **static** alan olan **count** artırılır. Bir nesne yok edildiğinde de **count** eksiltilir. Böylece, **count** daima o an mevcut olan nesnelerin sayısını tutar. Bu, ancak **static** bir alan kullanımıyla mümkün olabilir. Bir örnek değişkeninin bu sayıyı tutması mümkün değildir, çünkü **count** spesifik bir örnekle değil tüm sınıfla ilgilidir.

Şimdi **static** kullanan bir örneğe daha bakalım. Bu bölümün başlarında bir sınıf fabrikası kullanılarak sınıf nesnelerinin nasıl oluşturabileceğini görmüştük. O örnekte sınıf fabrikası **static** olmayan bir metottu. Dolayısıyla sadece bir nesne referansı aracılığıyla çağrılabiliyordu. Bu nedenle, fabrika metodunun çağrılabilmesi için o sınıfın varsayılan nesnesinin oluşturulması gerekliydi. Ancak, bir sınıf fabrikasını uygulamanın daha iyi bir yolu **static** metod kullanmaktır. Böylece sınıf fabrikası gereksiz bir nesne oluşturmaksızın çağrılabılır. Bu iyileştirmeyi yansitan **static** sınıf fabrikası örneği aşağıda yer almaktadır:

```
// Static bir sınıf fabrikasının kullanımı.

using System;

class MyClass {
    int a, b;

    // MyClass için bir sınıf fabrikası oluştur.
    static public MyClass factory(int i, int j) {
        MyClass t = new MyClass();

        t.a = i;
        t.b = j;

        return t; // bir nesne dondur
    }

    public void show() {
        Console.WriteLine("a and b: " + a + " " + b);
    }
}

class MakeObjects {
    public static void Main() {
        int i, j;

        // fabrikayı kullanarak nesneler oluştur
        for(i = 0, j = 10; i < 10; i++, j--) {
            MyClass ob = MyClass.factory(i, j); // bir nesne al
            ob.show();
        }
    }
}
```

```

        }
        Console.WriteLine();
    }
}

```

Bu versiyonda **factory()** şu satırdaki kod ile kendi sınıfının adı ile çağrılmaktadır:

```
MyClass ob = MyClass.factory(i, j); // bir nesne al
```

Fabrikayı kullanmak için önce **MyClass** nesnesi oluşturmaya gerek yoktur.

## static Yapılandırıcılar

Bir yapılandırıcı da **static** olarak belirtilebilir, **static** bir yapılandırıcı tipik olarak bir örneğe değil de bir sınıfa ait olan niteliklere (attributes) başlangıç değeri atamak için kullanılır. Bir başka ifade ile, **static** bir yapılandırıcı, bir sınıfa ait herhangi bir nesne oluşturulmadan önce o sınıfın bazı özelliklerine ilk değer atayabilir. İşte, basit bir örnek:

```

// static bir yapılandırıcı kullanır.
using System;

class Cons {
    public static int alpha;
    public int beta;

    // static yapılandırıcı
    static Cons() {
        alpha = 99;
        Console.WriteLine("Inside static constructor.");
    }

    // örnek yapılandırıcı
    public Cons() {
        beta = 100;
        Console.WriteLine("Inside instance constructor.");
    }
}

class ConsDemo {
    public static void Main() {
        Cons ob = new Cons();

        Console.WriteLine("Cons.alpha: " + Cons.alpha);
        Console.WriteLine("ob.beta: " + ob.beta);
    }
}

```

Programın çıktısı şöyledir:

```

Inside static constructor.
Inside instance constructor.
Cons.alpha: 99
ob.beta: 100

```

Dikkat ederseniz, **static** yapılandırıcı otomatik olarak ve örnek yapılandırıcıdan önce çağrılmaktadır. Bu, genelleştirilebilir. Tüm durumlarda **static** yapılandırıcı herhangi bir örnek yapılandırıcıdan önce çalıştırılacaktır. Ayrıca, **static** yapılandırıcılar **private** olmalıdır ve programınız tarafından çağrılamazlar.

# **OPERATÖRLERİN AŞIRI YÜKLENMESİ**

C#, bir operatörün anlamını, oluşturmuş olduğunuz sınıfı bağlı olarak tanımlamanıza imkan vermektedir. Bu işlem *operatörlerin aşırı yüklenmesi* olarak adlandırılır. Bir operatörü aşırı yükleyerek söz konusu operatörün kullanımını, oluşturduğunuz sınıf için genişletirsiniz. Operatörün etkileri tamamen sizin kontrolünüzdedir ve bir sınıfın diğerine değişiklik gösterebilir. Örneğin, bir bağlı listenin tanımlı olduğu bir sınıf, listeye bir nesne eklemek için + operatörünü kullanabilir. Bir yiğini gerçekleyen bir sınıf da yiğina bir nesne yerleştirmek için + operatörünü kullanabilir. Bir başka sınıf ise + operatörünü tümüyle farklı bir şekilde kullanabilir.

Bir operatör aşırı yüklendiğinde bu operatörün orijinal anlamı kaybolmaz. Sadece, belirli bir sınıfı bağlı olarak yeni bir işlem eklenmiş olur. Bu yüzden, örneğin bir bağlı listeyi ele alması için + operatörünü aşırı yüklemek, operatörün tamsayılarla bağlı olarak tanımlanmış anlamının (yani, toplama işlemi) değişmesine neden olmaz.

Operatörlerin aşırı yüklenmesinin sağladığı başlıca avantaj şudur: Bu işlem, yeni bir sınıf tipini programlama ortamınıza kesintisiz olarak entegre etmenize imkan verir. Tiplerin bu şekilde genişletilebilmesi, C# gibi nesne yönelimli dillerin gücünün önemli bir parçasıdır. Bir sınıf için operatörler tanımlandığı andan itibaren, C#'ın normal deyim söz dizimini kullanarak bu sınıfın nesneleri üzerinde işlem yapabilirsiniz. Hatta bir nesneyi, diğer tiplerdeki verileri içeren deyimlerde bile kullanabilirsiniz. Operatörlerin aşırı yüklenmesi, C#'ın en güçlü özelliklerinden biridir.

## Operatörlerin Aşırı Yüklenmesiyle İlgili Temel Kavramlar

Operatörlerin aşırı yüklenmesi, metodların aşırı yüklenmesiyle yakından bağlantılıdır. Bir operatörü aşırı yüklemek için bir *operatör metodu* oluşturmak amacıyla **operator** anahtar kelimesini kullanırsınız. Operatör metodu, operatörün kendi sınıfına bağlı olarak faaliyetlerini tanımlar.

İki çeşit operatör metodu vardır: tekli (unary) operatörler için ve ikili operatörler için. Her birinin genel yapısı aşağıda gösterilmiştir:

```
// Tekli bir operatörün aşırı yüklenmesinin genel yapısı
public static dönüş-tipi operator op(param-tipi operand)
{
    // işlemler
}

// İkili bir operatörün aşırı yüklenmesinin genel yapısı
public static dönüş-tipi operator op(param-tipi1 operand1, param-tipi1 operand2)
{
    // işlemler
}
```

Burada, aşırı yüklenenek operatör, söz geliş + veya /, op yerine yerleştirilir. **dönüş-tipi**, belirtilen işlem tarafından döndürulen değerin tipidir. Dönüş tipi, tercih ettiğiniz herhangi tipte olabilse bile, genellikle operatörün aşırı yüklenmekte olduğu sınıf ile aynı tiptedir. Bu korelasyon, aşırı yüklenmiş operatörün deyimlerde kullanımını kolaylaştırır. Tekli operatörler için operand, **operand** üzerinden aktarılır. İkili operatörler için ise operandlar **operand1** ve **operand2** üzerinden aktarılır.

Tekli operatörler için operand, operatörün tanımlanmakta olduğu sınıfı ile aynı tipte olmalıdır. İkili operatörler için operandlardan en az biri kendi sınıfı ile aynı tipte olmalıdır. Bu nedenle, henüz oluşturmadığınız nesneler için C# operatörlerini aşırı yükleyemezsiniz. Örneğin, + operatörünü **int** veya **string** için yeniden tanımlayamazsınız.

Bir diğer husus: Operatör parametreleri **ref** veya **out** niteliklerini kullanmamalıdır.

## İkili Operatörleri Aşırı Yüklemek

Operatörlerin aşırı yüklenme işleminin ne şekilde işlediğini anlamak için gelin, iki ikili operatörü, örneğin + ve - operatörlerini aşırı yükleyen bir örnekle başlayalım. Aşağıdaki program, üç boyutlu uzayda bir nesnenin koordinatlarını tutan **ThreeD** adında bir sınıf oluşturmaktadır. Aşırı yüklenmiş +, bir **ThreeD** nesnesinin koordinatlarını tek tek bir başka nesnenin koordinatlarına ekler. Aşırı yüklenmiş - bir nesnenin koordinatlarını bir diğerininkinden çıkartır.

```
// Operatorlerin asiri yuklenmesine bir ornek.

using System;

// Uc boyutlu bir koordinat sinifi.
class ThreeD {
    int x, y, z; // uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // Ikili + operatorunu asiri yukle
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();

        /* Bu,iki noktanin koordinatlarini toplar
           ve sonucu dondurur. */
        result.x = op1.x + op2.x; // Bunlar tamsayı toplama;
        result.y = op1.y + op2.y; // +, bunlara bagli olarak
        result.z = op1.z + op2.z; // orijinal anlamini korur

        return result;
    }

    // Ikili - operatorunu asiri yukle.
    public static ThreeD operator -(ThreeD op1, ThreeD op2)
```

```
{  
    ThreeD result = new ThreeD();  
  
    /* Operandların sırasına dikkat edin.  
     * op1, soldaki operand; op2 ise sağdaki. */  
    result.x = op1.x - op2.x; // bunlar, tamsayı çıkarmadır  
    result.y = op1.y - op2.y;  
    result.z = op1.z - op2.z;  
  
    return result;  
}  
  
// x, y, z koordinatlarını göster.  
public void show()  
{  
    Console.WriteLine(x + ", " + y + ", " + z);  
}  
}  
  
class ThreeDDemo {  
    public static void Main() {  
        ThreeD a = new ThreeD(1, 2, 3);  
        ThreeD b = new ThreeD(10, 10, 10);  
        ThreeD c = new ThreeD();  
  
        Console.Write("Here is a: ");  
        a.show();  
        Console.WriteLine();  
        Console.Write("Here is b: ");  
        b.show();  
        Console.WriteLine();  
  
        c = a + b; // a ve b'yi topla  
        Console.Write("Result of a + b: ");  
        c.show();  
        Console.WriteLine();  
  
        c = a + b + c; // a,b ve c'yi topla  
        Console.Write("Result of a + b + c: ");  
        c.show();  
        Console.WriteLine();  
  
        c = c - a; // a'yi çıkar  
        Console.Write("Result of c - a: ");  
        c.show();  
        Console.WriteLine();  
    }  
}
```

Bu program aşağıdaki çıktıyı üretir:

```

Here is a: 1, 2, 3
Here is b: 10, 10, 10
Result of a + b: 11, 12, 13
Result of a + b + c: 22, 24, 26
Result of c - a: 21, 22, 23
Result of c - b: 11, 12, 13

```

Gelin, aşırı yüklenmiş `+` operatöründen başlayarak yukarıdaki programı dikkatlice inceleyelim. `ThreeD` tipindeki iki nesne üzerine `+` operatörü uygulandığında, `operator+()`'da görüldüğü gibi bu nesnelerin her birinin koordinatlarının şiddetli toplanır. Ancak, bu metodun, operandlarının hiçbirinin değerini değiştirmedigine dikkat edin. Bunun yerine metot, işlemin sonucunu içeren `ThreeD` tipinde yeni bir nesne döndürür. `+` işleminin her iki nesnenin içeriğini değiştirmemesinin nedenini anlamak için standart aritmetik `+` işleminin şu şekilde uygulanışını ele alın: **10+12**. Bu işlemin sonucu **22**'dir, fakat ne **10** ne de **12** bu işlem sonucunda değiştirilmiştir. Aşırı yüklenmiş bir operatörün, operandlarından birinin değerini değiştirmesini önleyecek hiçbir kural mevcut olmamasına rağmen, aşırı yüklenmiş bir operatörün faaliyetlerinin operatörün alışındık anlamı ile tutarlı olması en iyisidir.

`operator+()` metodunun `ThreeD` tipinde bir nesne döndürdüğüne dikkat edin. Gerçi, metot herhangi geçerli bir C# tipini döndürebilirdi; ancak, metodun bir `ThreeD` nesnesi döndürmesi gerçeği `+` operatörünün `a+b+c` gibi bileşik deyimlerde kullanılmasına imkan vermektedir. Burada `a+b`, `ThreeD` tipinde bir sonuç üretir. Sonra bu değer `c`'ye eklenebilir. `a+b` işleminin sonucunda başka tipte bir değer üretilmiş olsaydı bu tür bir deyim çalışmazdı.

Bir başka önemli husus ise şudur: Koordinatlar `operator+()` metodunun içinde birbiri ile toplandığında, koordinatların tek tek toplanması, tamsayı toplaması olarak sonuçlanır. Bunun nedeni, her bir koordinatın, yani `x`, `y` ve `z`'nin bir tamsayı nicelik olmasıdır. `+` operatörünün `ThreeD` tipindeki nesneler için aşırı yüklenmesinin, tamsayı değerlere uygulanma şekli üzerinde hiçbir etkisi olmaz.

Simdi `operatör-()` metoduna göz atalım. `-` operatörü tipki `+` operatörü gibi çalışır; tek fark, parametre sırasının bu kez önemli olmasıdır. Toplamanın değişme özelliği olduğunu, fakat çıkarmanın bu özelliği olmadığını hatırlayın. (Yani, `A-B` ile `B-A` aynı değildir!) Tüm ikili operatörler için operatöre aktarılan ilk parametre soldaki operandı içerecektir. İkinci parametre ise sağ taraftakini içerecektir. Değişme özelliği olmayan operatörlerin aşırı yüklenmiş versiyonlarını uygularken hangi operandın solda hangisinin sağda olduğunu hatırlamalısınız.

## Tekli Operatörleri Aşırı Yüklemek

Tekli operatörler de tipki ikili operatörler gibi aşırı yüklenir. En temel fark, elbette, bu kez tek bir operandın olmasıdır. Örneğin, `ThreeD` sınıfı için tekli eksiz operatörünü aşırı yükleyen bir metot aşağıdaki gibidir:

```
// Tekli - operatorunu aşırı yükler.
public static ThreeD operator -(ThreeD op)
{
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
}
```

Burada, operandın negatif hale getirilmiş alanlarını içeren yeni bir nesne oluşturulmaktadır. Sonra da bu nesne döndürülmektedir. Dikkat ederseniz, operand değişmemiştir. Bunun nedeni yine, tekli eksi operatörünün alışık anlamını korumaktır. Örneğin, şu tür bir deyimde,

`a = -b`

**a**, **b**'nin negatif hale getirilmiş şeklini alır fakat **b** değişmez.

Yine de, bir operatör metodunun operandının içeriğini değiştirmeye gerek duyacağı iki durum mevcuttur; `++` ve `--`. Bu operatörlerin alışık anamları artırma ve eksiltme olduğu için aşırı yüklenmiş `++` veya `--` de genellikle operandını artırmalı ya da eksiltmelidir. Böylece, bu iki operatörü aşırı yüklediğinizde operand genellikle değişikliğe uğrar. Örneğin, **ThreeD** sınıfı için bir `operator++()` metodu aşağıda yer almaktadır:

```
// Tekli ++ operatorunu aşırı yükle.
public static ThreeD operator ++(ThreeD op)
{
    // ++ icin, argumanı değiştir
    op.x++;
    op.y++;
    op.z++;

    return op; // operand dondurulur
}
```

Dikkat ederseniz, **op** ile ilişkilendirilen nesne bu metot tarafından değişikliğe uğramaktadır. Böylece, `++` işlemindeki operand artırılmış olur. Değiştirilen nesne de ayrıca döndürülür. Bu, `++` operatörünün daha büyük bir deyimde kullanılması için gereklidir.

İşte, tekli `--` ve `++` operatörünü gösteren önceki örnek programın genişletilmiş bir versiyonu:

```
// Operatorlerin aşırı yüklenmesi hakkında birkac örnek daha
using System;

// Üç boyutlu koordinat sınıfı.
class ThreeD {
    int x, y, z; // üç boyutlu koordinatlar
```

```
public ThreeD() { x = y = z = 0; }
public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

// İkili + operatörünü asiri yükle
public static ThreeD operator +(ThreeD op1, ThreeD op2)
{
    ThreeD result = new ThreeD();

    /* Bu iki noktanın koordinatlarını toplar ve
       sonucu dondurur. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
}

// İkili - operatörünü asiri yükle.
public static ThreeD operator -(ThreeD op1, ThreeD op2)
{
    ThreeD result = new ThreeD();

    /* Operandların sırasına dikkat edin.
       op1 sol taraftaki operand, op2 ise sağ taraftaki. */
    result.x = op1.x - op2.x;
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
}

// Tekli - operatörünü asiri yükle.
public static ThreeD operator -(ThreeD op)
{
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
}

// Tekli ++ operatörünü asiri yükle.
public static ThreeD operator ++(ThreeD op)
{
    // ++ için, argumani degistir
    op.x++;
    op.y++;
    op.z++;

    return op;
}

// X, Y, Z koordinatlarını göster.
```

```
public void show()
{
    Console.WriteLine(x + " , " + y + " , " + z);
}
}

class ThreeDDemo {
    public static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();

        Console.Write("Here is a: ");
        a.show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.show();
        Console.WriteLine();

        c = a + b; // a ve b'yi topla
        Console.Write("Result of a + b: ");
        c.show();
        Console.WriteLine();

        c = a + b + c; //a, b ve c'yi topla
        Console.Write("Result of a + b + c: ");
        c.show();
        Console.WriteLine();

        c = c - a; // a'yi cikar
        Console.Write("Result of c - a: ");
        c.show();
        Console.WriteLine();

        c = c - b; // b'yi cikar
        Console.Write("Result of c - b: ");
        c.show();
        Console.WriteLine();

        c = -a; // c'ye -a degerini ata
        Console.Write("Result of -a: ");
        c.show();
        Console.WriteLine();

        a++; // a'yi bir artir
        Console.Write("Result of a++: ");
        a.show();
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

Here is a: 1, 2, 3

Here is b: 10, 10, 10

```

Result of a + b: 11, 12, 13
Result of a + b + c: 22, 24, 26
Result of c - a: 21, 22, 23
Result of c - b: 11, 12, 13
Result of -a: -1, -2, -3
Result of a++: 2, 3, 4

```

Bildiğiniz gibi, **++** ve **--** operatörlerinin hem önek, hem de sonek formları mevcuttur. Örneğin, hem

```

++a;
hem de
a++;

```

artırma operatörünün geçerli kullanımlarıdır. Ancak, **++** ya da **--** aşırı yüklenliğinde her iki form da aynı metodu çağırır. Böylece, aşırı yükleme söz konusu olduğunda **++** veya **--** operatörlerinin önekli veya sonekli formlarını ayırt etmek mümkün değildir.

## Standart C# Tipleri Üzerindeki İşlemleri Kontrol Altına Almak

Herhangi bir sınıf ve operatör için bir operatör metodunun kendisi aşırı yüklenebilir. Bunun en bilinen nedenlerinden biri, sınıf tipi ve diğer veri tipleri, söz geliş standart tipler arasındaki işlemlere imkan vermektedir. Örneğin, bir kez daha **ThreeD** sınıfını ele alın. Bu noktaya kadar **+** operatörünü nasıl aşırı yükleyeceğinizi görmüştünüz. Bu sayede, **+** operatörü bir **ThreeD** nesnesinin koordinatlarını bir başka nesnenin koordinatlarıyla toplar. Ancak, **ThreeD** için tanımlamak isteyebileceğiniz tek toplama yöntemi bu değildir. Örneğin, bir **ThreeD** nesnesinin her koordinatına bir tamsayı değer eklemek yararlı olabilir. Bu tür bir işlem eksenlerin çevriminde kullanılabilir. Bu tür bir işlemi gerçekleştirmek için **+** operatörünü, aşağıdaki gibi ikinci kez aşırı yüklemeniz gereklidir:

```

// nesne + int icin ikili + operatorunu asiri yukle.
public static ThreeD operator +(ThreeD op1, int op2)
{
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
}

```

İkinci parametrenin `int` tipinde olduğuna dikkat edin. Böylece, yukarıdaki metot, bir `ThreeD` nesnesinin her alanına bir tamsayı değer eklenmesine imkan vermektedir. Buna izin verilebilir, çünkü önceden açıklandığı gibi, bir ikili operatörü aşırı yüklerken operandlardan yalnızca biri, operatörün aşırı yüklenmekte olduğu sınıf ile aynı tipte olmalıdır. Diğer operand herhangi başka bir tipte olabilir.

İşte, `ThreeD`'nin iki adet aşırı yüklenmiş `+` metodu içeren bir versiyonu:

```
/* nesne + nesne ve nesne + int icin
   toplama operatorunu asiri yukle. */

using System;

// Uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // Uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // nesne + nesne icin ikili + operatorunu asiri yukle.
    public static ThreeD operator +(ThreeD op1, -ThreeD op2)
    {
        ThreeD result = new ThreeD();

        // Bu, iki noktanin koordinatlarini toplar ve sonucu dondurur.
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // nesne + int icin ikili + operatorunu asiri yukle
    public static ThreeD operator +(ThreeD op1, int op2)
    {
        ThreeD result = new ThreeD();

        result.x = op1.x + op2;
        result.y = op1.y + op2;
        result.z = op1.z + op2;

        return result;
    }

    // X, Y, Z koordinatlarini goster.
    public void show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}

class ThreeDDemo {
    public static void Main() {
```

```

ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);
ThreeD c = new ThreeD();

Console.WriteLine("Here is a: ");
a.show();
Console.WriteLine();
Console.WriteLine("Here is b: ");
b.show();
Console.WriteLine();

c = a + b; // nesne + nesne
Console.WriteLine("Result of a + b: ");
c.show();
Console.WriteLine();

c = b + 10; // nesne + int
Console.WriteLine("Result of b + 10: ");
c.show();
}
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Here is a: 1, 2, 3
Here is b: 10, 10, 10
Result of a + b: 11, 12, 13
Result of b + 10: 20, 20, 20

```

Cıktının da doğruladığı gibi, `+` operatörü iki nesneye uygulandığında nesnelerin koordinatları toplanır. `+`, bir nesne ve bir tamsayıya uygulandığında koordinatlar tamsayı değer kadar artırılır.

`+` operatörünün az önce gösterilen aşırı yüklenmiş şekli `ThreeD` sınıfına yararlı bir beceri katmış olmakla birlikte, işi tam olarak bitirmiş olmuyor. Şimdi neden böyle olduğunu açıklayalım. `operator + (ThreeD, int)` metodu şu ifadelere imkan verir:

```
ob1 = ob2 + 10;
```

Fakat ne yazık ki, şu türlere imkan vermez:

```
ob1 = 10 + ob2;
```

Bunun nedeni, tamsayı argümanın sağ taraftaki operand olan ikinci argüman olmasıdır. Problem, yukarıdaki ifadede tamsayı argümanın sola yerleştirilmesidir. Her iki ifade şeklini de mümkün kılmak için `+` operatörünü bir kez daha aşırı yüklemeniz gerekecek. Bu versiyonda ilk parametre `int` tipinde, ikinci parametre `ThreeD` tipinde olmalıdır. `operator+()` metodunun bir versiyonu `nesne + tamsayı` işlemini, diğer versiyon ise `tamsayı + nesne` işlemini ele alır. `+` operatörünü (ya da diğer ikili operatörlerden herhangi birini) bu şekilde aşırı yüklemek standart

bir tipin, operatörün sol ya da sağ tarafında bulunmasına imkan vermektedir. İşte, + operatörünü şimdilik anlatılan şekliyle aşırı yükleyen **ThreeD**'nin bir versiyonu:

```
/* nesne + nesne, nesne + int ve
   int + nesne icin + operatorunu asiri yükler. */

using System;

// uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // nesne + nesne icin ikili + operatorunu asiri yukle.
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();

        /* Bu, iki noktanin koordinatlarini toplar
           ve sonucu dondurur. */
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // nesne + int icin ikili + operatorunu asiri yukle.
    public static ThreeD operator +(ThreeD op1, int op2)
    {
        ThreeD result = new ThreeD();

        result.x = op1.x + op2;
        result.y = op1.y + op2;
        result.z = op1.z + op2;

        return result;
    }

    // int + nesne icin ikili + operatorunu asiri yukle.
    public static ThreeD operator +(int op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();

        result.x = op2.x + op1;
        result.y = op2.y + op1 ;
        result.z = op2.z + op1;

        return result;
    }

    // X, Y, Z koordinatlarini goster.
    public void show()
```

```

    {
        Console.WriteLine(x + " , " + y + " , " + z);
    }
}

class ThreeDDemo {
    public static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();

        Console.Write("Here is a: ");
        a.show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.show();
        Console.WriteLine();

        c = a + b; // nesne + nesne
        Console.Write("Result of a + b: ");
        c.show();
        Console.WriteLine();

        c = b + 10; // nesne + int
        Console.Write("Result of b + 10: ");
        c.show();
        Console.WriteLine();

        c = 15 + b; // int + nesne
        Console.Write("Result of 15 + b: ");
        c.show();
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Here is a: 1, 2, 3
Here is b: 10, 10, 10
Result of a + b: 11, 12, 13
Result of b + 10: 20, 20, 20
Result of 15 + b: 25, 25, 25

```

## İlişkisel Operatörleri Aşırı Yüklemek

`==` ya da `<` gibi ilişkisel operatörler de aşırı yüklenebilir ve bu işlemler açıkça bellidir. Genellikle, aşırı yüklenmiş bir operatör `true` veya `false` değeri döndürür. Bu, bu operatörlerin normal kullanımlarını korur ve aşırı yüklenmiş ilişkisel operatörlerin koşul ifadelerinde kullanılmasına imkan verir. Eğer farklı bir tipte bir sonuç döndürürseniz, operatörün becerisini büyük ölçüde kısıtlıyorsunuz demektir.

İşte, **ThreeD** sınıfının < ve > operatörlerini aşırı yükleyen bir versiyonu:

```
// < ve > operatorlerini asiri yukle.

using System;

// Uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // Uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // < operatorunu asiri yukle.
    public static bool operator <(ThreeD op1, ThreeD op2)
    {
        if((op1.x < op2.x) && (op1.y < op2.y) && (op1.z < op2.z))
            return true;
        else
            return false;
    }

    // > operatorunu asiri yukle.
    public static bool operator >(ThreeD op1, ThreeD op2)
    {
        if((op1.x > op2.x) && (op1.y > op2.y) && (op1.z > op2.z))
            return true;
        else
            return false;
    }

    // X, Y, Z koordinatlarini gostер.
    public void show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}

class ThreeDDemo {
    public static void Main()
    {
        ThreeD a = new ThreeD(5, 6, 7);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD(1, 2, 3);

        Console.Write("Here is a: ");
        a.show();
        Console.Write("Here is b: ");
        b.show();
        Console.Write("Here is c: ");
        c.show();

        Console.WriteLine();

        if(a > c) Console.WriteLine("a > c is true");
        if(a < c) Console.WriteLine("a < c is true");
```

```

        if(a > b) Console.WriteLine("a > b is true");
        if(a < b) Console.WriteLine("a < b is true");
    }
}

```

Bu programın çıktısı aşağıdaki gibidir:

```

Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 1, 2, 3

a > c is true
a < b is true

```

İlişkisel operatörleri aşın yükleme işlemini etkileyen önemli bir kısıtlama söz konusudur: Bu operatörleri çift çift aşırı yüklemelisiniz. Örneğin, `<` operatörünü aşırı yükleyorsanız, `>` operatörünü de aşırı yüklemelisiniz; ya da tam tersini yapmalısınız. Operatör çiftleri aşağıda gösterilmiştir:

```

==      !=
<       >
<=      >=

```

Bir diğer konu ise: `==` ve `!=` operatörlerini aşırı yükleyorsanız, genellikle `Object.Equal()` ve `Object.GetHashCode()` metodlarını devre dışı bırakmanız gereklidir. Bu metodlar ve bunları devre dışı bırakma (override) tekniği Bölüm 11'de anlatılmaktadır.

## true ve false'ı Aşırı Yüklemek

`true` ve `false` anahtar kelimeleri aşırı yükleme kapsamında tekli operatör olarak da kullanılabilirler. Bu operatörlerin aşırı yüklenmiş versiyonları, oluşturduğunuz sınıflara bağlı olarak `true` ve `false` değerlerinin alışık kullanımına imkan verirler. `true` ve `false` bir sınıf için uygulanır uygulanmaz söz konusu sınıfın nesnelerini `if`, `while`, `for` ve `do-while` ifadelerini kontrol etmek amacıyla ya da bir `?` deyiminde kullanabilirsiniz. Hatta, bu nesneleri özel mantık türlerini, söz gelişî bulanık mantık (fuzzy logic) uygulamak için bile kullanabilirsiniz.

`true` ve `false` operatörleri, çift olarak aşırı yüklenmelidir. Sadece bir tekini aşırı yükleyemezsiniz. Her ikisi de tekli operatördür ve aşağıdaki genel yapıya sahiplerdir:

```

public static bool operator true(param-tipi operand)
{
    // true veya false döndür
}

public static bool operator false(param-tipi operand)
{
    // true veya false döndür
}

```

Dikkat ederseniz, her ikisi de bir `bool` sonuç döndürmektedir.

Aşağıdaki örnek **ThreeD** sınıfı için **true** ve **false**'un nasıl uygulanabileceğini göstermektedir. Koordinatlardan en az biri sıfırdan farklıysa **ThreeD** nesnesinin doğru olduğu varsayıılır. Koordinatların üçü de sıfırsa, nesne yanlıştır. Ayrıca, kullanımını göstermek amacıyla eksiltme operatörü de uygulanmıştır.

```
// ThreeD için true ve false'i aşırı yükle.

using System;

// Üç boyutlu koordinat sınıfı.
class ThreeD {
    int x, y, z; // Üç boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // true'yu aşırı yükle.
    public static bool operator true(ThreeD op) {
        if((op.x != 0) || (op.y != 0) || (op.z != 0))
            return true; // en az bir koordinat sıfırdan farklı
        else
            return false;
    }

    // false'u aşırı yükle.
    public static bool operator false(ThreeD op) {
        if((op.x == 0) && (op.y == 0) && (op.z == 0))
            return true; // tüm koordinatlar sıfır
        else
            return false;
    }

    // Tekli - operatorunu aşırı yükle.
    public static ThreeD operator --(ThreeD op)
    {
        op.x--;
        op.y--;
        op.z--;

        return op;
    }

    // X, Y, Z koordinatlarını göster.
    public void show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}

class TrueFalseDemo {
    public static void Main() {
        ThreeD a = new ThreeD(5, 6, 7);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD(0, 0, 0);
```

```

Console.WriteLine("Here is a: ");
a.show();
Console.WriteLine("Here is b: ");
b.show();
Console.WriteLine("Here is c: ");
c.show();
Console.WriteLine();

if(a) Console.WriteLine("a is true.");
else Console.WriteLine("a is false.");

if(b) Console.WriteLine("b is true.");
else Console.WriteLine("b is false.");

if(c) Console.WriteLine("c is true.");
else Console.WriteLine("c is false.");

Console.WriteLine();

Console.WriteLine("Control a loop using a Threed object.");
do {
    b.show();
    b--;
} while(b);
}
}

```

Çıktı aşağıda gösterilmiştir:

```

Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0

a is true.
b is true.
c is false.

Control a loop using a Threed object.
10, 10, 10
9, 9, 9
8, 8, 8
7, 7, 7
6, 6, 6
5, 5, 5
4, 4, 4
3, 3, 3
2, 2, 2
1, 1, 1

```

**Threed** nesnelerin **if** ifadelerini ve **while** döngüsünü kontrol etmek için nasıl kullanıldığına dikkat edin. **if** ifadeleri söz konusu olduğunda, **Threed** nesnesi **true** kullanılarak değerlendirilir. Bu işlemin sonucu doğru ise, **if** ifadesi başarılı olur. **do-while** döngüsünde ise döngünün her iterasyonunda **b** bir kez azaltılır. Döngü, **b** doğru değerine sahip

oldukça (yani, en az bir adet sıfırdan farklı koordinat içерdiği sürece) iterasyona devam eder. **&** tamamen sıfır koordinatlarını içerdığı zaman, **true** operatörü uygulandığında **&** yanlış olarak değerlendirilir ve döngü sona erer.

## Mantıksal Operatörleri Aşırı Yüklemek

Badiğiniz gibi, C#'ta şu mantıksal operatörler tanımlıdır: **&**, **|**, **!**, **&&** ve **||**. Bunlardan yalnızca **&**, **|** ve **!** aşırı yüklenebilir. Ancak, belirli kurallara uyarak her şeye rağmen kısa devre **&&** ve **||** operatörlerinin avantajları elde edilebilir. Bu durumların her biri bu bölümde incelenmektedir.

### Mantıksal Operatörlerin Aşırı Yüklenmesine Basit Bir Yaklaşım

Gelin, en basit durumla başlayalım. Eğer kısa devre mantıksal operatörlerden yararlanmayacaksınız, tahmin edebileceğiniz gibi **&** ve **|** operatörlerini, her biri bir **bool** sonuç döndürecek şekilde aşırı yükleyebilirsiniz. Aşırı yüklenmiş **!** operatörü de bir **bool** sonuç döndürecektir.

İşte, **ThreeD** tipindeki nesneler için **!**, **&** ve **|** mantıksal operatörlerini aşırı yükleyen bir örnek. Önceki gibi, bu operatörlerin üçü de, koordinatların en az biri sıfırdan farklı olduğunda bir **ThreeD** nesnesinin **true** olduğunu varsayarlar. Koordinatların üçü de sıfırsa, nesne **false** değerine sahiptir.

```
/* ThreeD icin !, | ve & operatorlerini asiri yuklemek icin
basit bir yontem */

using System;

// Uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // Uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // | operatorunu asiri yukle.
    public static bool operator |(ThreeD op1, ThreeD op2)
    {
        if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) ||
            ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
            return true;
        else
            return false;
    }

    // & operatorunu asiri yukle.
    public static bool operator &(ThreeD op1, ThreeD op2)
```

```

{
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        (op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
        return true;
    else
        return false;
}

// Overload !.
public static bool operator !(ThreeD op)
{
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
        return false;
    else return true;
}

// X, Y, Z koordinatlarini goster.
public void show()
{
    Console.WriteLine(x + ", " + y + ", " + z);
}
}

class TrueFalseDemo {
    public static void Main() {
        ThreeD a = new ThreeD(5, 6, 7);
        ThreeD b = new TrreeD(10, 10, 10);
        ThreeD c = new TrreeD(0, 0, 0);

        Console.Write("Here is a: ");
        a.show();
        Console.Write("Here is b: ");
        b.show();
        Console.Write("Here is c: ");
        c.show();
        Console.WriteLine();

        if(!a) Console.WriteLine("a is false.");
        if(!b) Console.WriteLine("b is false.");
        if(!c) Console.WriteLine("c is false.");

        Console.WriteLine();

        if(a & b) Console.WriteLine("a & b is true.");
        else Console.WriteLine("a & b is false.");

        if(a & c) Console.WriteLine("a & c is true.");
        else Console.WriteLine("a & c is false.");

        if(a | b) Console.WriteLine("a | b is true.");
        else Console.WriteLine("a | b is false.");

        if(a | c) Console.WriteLine("a | c is true.");
        else Console.WriteLine("a | c is false.");
    }
}

```

```
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0

c is false.
```

```
a & b is true.
a & c is false.
a | b is true.
a | c is true.
```

Bu yöntemde, **&**, **|** ve **!** operatör metodlarının her biri bir **bool** sonuç döndürmektedir. Operatörler normal biçimde (yani, bir **bool** sonuç bekleyen yerlerde) kullanılacaksa bu gereklidir. Hatırlarsanız, tüm standart tipler için mantıksal bir işlemin sonucu **bool** tipinde bir değerdir. Bu nedenle, bu operatörlerin aşırı yüklenmiş versiyonlarını **bool** tipinde döndürmek mantıklı bir yaklaşımdır. Ne yazık ki, bu yöntem ancak kısa devre operatörlere ihtiyacınız olmayacaksa işe yarar.

## Kısa Devre Operatörleri Mümkün Kılmak

**&&** ve **||** kısa devre operatörlerinin kullanımını mümkün kılmak için dört kurala uymalısınız. Öncelikle, söz konusu sınıf, **&** ve **|** operatörlerini aşırı yüklemelidir. İkinci olarak, aşırı yüklenmiş **&** ve **|** metodlarının dönüş tipi, operatörlerin aşırı yüklenmekte olduğu sınıfa ait bir nesne olmalıdır. Üçüncüsü, parametrelerin her biri operatörün aşırı yüklenmekte olduğu sınıfın bir nesnesine referans olmalıdır. Dördüncüsü ise, **true** ve **false** operatörleri de söz konusu sınıf için aşırı yüklenmelidir. Bu koşullar saklandığında, kısa devre operatörler otomatik olarak kullanıma hazır hale gelir,

Aşağıdaki program, kısa devre **&&** ve **||** operatörlerini mümkün kılmak amacıyla **&** ve **|** operatörlerinin **ThreeD** sınıfı için düzgün olarak ne şekilde uygulanması gerektiğini göstermektedir.

```
/* ThreeD icin !, | ve & operatorlerini asiri yuklemenin daha iyi
bir yolu. Bu versiyon && ve || operatorlerini otomatik olarak
mumkun kilar. */

using System;

// Uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // Uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // Kisa devre kullanim icin | operatorunu asiri yukle.
}
```

```

public static ThreeD operator |(ThreeD op1, ThreeD op2)
{
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) ||
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
        return new ThreeD(1, 1, 1);
    else
        return new ThreeD(0, 0, 0);
}

// Kisa devre kullanim icin & operatorunu asiri yukle.
public static ThreeD operator &(ThreeD op1, ThreeD op2)
{
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
        return new ThreeD(1, 1, 1);
    else
        return new ThreeD(0, 0, 0);
}

// ! operatorunu asiri yukle.
public static bool operator !(ThreeD op)
{
    if(op) return false;
    else return true;
}

// true operatorunu asiri yukle.
public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
        return true; // en azindan bir koordinat sifirdan farkli
    else
        return false;
}

// false'u asiri yukle.
public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
        return true; // tum koordinatlar sifira esit
    else
        return false;
}

// X, Y, Z koordinatlarini goster.
public void show()
{
    Console.WriteLine(x + ", " + y + ", " + z);
}
}

class TrueFalseDemo {
    public static void Main() {
        ThreeD a = new ThreeD(5, 6, 7);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD(0, 0, 0);
    }
}

```

```
Console.WriteLine("Here is a: ");
a.show();
Console.WriteLine("Here is b: ");
b.show();
Console.WriteLine("Here is c: ");
c.show();
Console.WriteLine();

if(a) Console.WriteLine("a is true.");
if(b) Console.WriteLine("b is true.");
if(c) Console.WriteLine("c is true.");

if(!a) Console.WriteLine("a is false.");
if(!b) Console.WriteLine("b is false.");
if(!c) Console.WriteLine("c is false.");

Console.WriteLine();

Console.WriteLine("Use & and |");
if(a & b) Console.WriteLine("a & b is true.");
else Console.WriteLine("a & b is false.");

if(a & c) Console.WriteLine("a & c is true.");
else Console.WriteLine("a & c is false.");

if(a | b) Console.WriteLine("a | b is true.");
else Console.WriteLine("a | b is false.");

if(a | c) Console.WriteLine("a | c is true.");
else Console.WriteLine("a | c is false.");

Console.WriteLine();

// simdi kısa devre operatorleri kullan
Console.WriteLine("Use short-circuit && and ||");
if(a && b) Console.WriteLine("a && b is true.");
else Console.WriteLine("a && b is false.");

if(a && c) Console.WriteLine("a && c is true.");
else Console.WriteLine("a && c is false.");

if(a || b) Console.WriteLine("a || b is true.");
else Console.WriteLine("a || b is false.");

if(a || c) Console.WriteLine("a || c is true.");
else Console.WriteLine("a || c is false.");
}
```

Programın çıktısı aşağıdaki gibidir:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0
```

```

a is true.
b is true.
c is false.

Use & and |
a & b is true.
a & c is false.
a | b is true.
a | c is true.

Use short-circuit && and ||
a && b is true.
a && c is false.
a || b is true.
a || c is true.

```

Gelin şimdi **&** ve **|** operatörlerinin nasıl uygulandığına yakından bakalım. İlgili metodlar aşağıda gösterilmiştir;

```

// Kısa devre kullanım için | operatorunu asırı yükle.
public static ThreeD operator |(ThreeD op1, ThreeD op2)
{
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) | 
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
        return new ThreeD(1, 1, 1);
    else
        return new ThreeD(0, 0, 0);
}

// Kısa devre kullanım için & operatorunu asırı yükle.
public static ThreeD operator &(ThreeD op1, ThreeD op2)
{
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
        return new ThreeD(1, 1, 1);
    else
        return new ThreeD(0, 0, 0);
}

```

Öncelikle her ikisinin de **ThreeD** tipinde bir nesne döndürüklerine dikkat edin. Bu nesnenin nasıl üretil diligine de dikkat edin. Söz konusu işlemin sonucu **true** ise, **true** değerinde (en azından bir koordinatı sıfırdan farklı olan) bir **ThreeD** nesnesi oluşturulur ve döndürülür. Sonuç **false** ise, bu durumda **false** değerinde bir nesne oluşturulur ve döndürülür. Böylece, şu tür bir ifadede:

```

if(a & b) Console.WriteLine("a & b is true.");
else Console.WriteLine("a & b is false.");

```

**a&b** işleminin sonucu, bir **ThreeD** nesnesidir. Bu örnekte, bu nesne **true** değerine sahiptir. **true** ve **false** operatörleri tanımlı olduğu için elde edilen nesne **true** operatörüne tabi tutulur ve bir **bool** sonuç döndürülür. Bu örnekte sonuç **true**'dur ve **if** başarılı olur.

Gerekli kurallar takip edilmiş olduğu için kısa devre operatörler artık **ThreeD** nesneleri ile kullanılmaya hazırır. Bu operatörler şu şekilde çalışır: İlk operand, **operatör true** (**||** için) veya **operatör false** (**&&** için) kullanılarak test edilir. İşlemin sonucu belirlenebilirse, karşılık gelen **&** veya **|** deyiminin değeri hesaplanmaz. Aksi halde, sonucu belirlemek için karşılık gelen aşırı yüklenmiş **&** veya **|** kullanılır. Böylece, **&&** veya **||** kullanmak, yalnızca deyimin sonucu ilk operand ile belirlenemediği taktirde, karşılık gelen **&** veya **|** metodunun çağrılmasına neden olur. Örneğin, programdaki şu ifadeyi ele alın:

```
if(a || c) Console.WriteLine("a || c is true.");
```

**true** operatörü önce **a**'ya uygulanır. Bu durumda **a** doğru değerine sahip olduğu için **|** operatör metodunu kullanmaya gerek yoktur. Ancak, aynı ifade şu şekilde yazılmış olsaydı:

```
if(c || a) Console.WriteLine("c || a is true.");
```

Bu durumda **true** operatörü **c**'ye uygulanacaktı - ki, **c** bu örnekte **false** değerine sahiptir. Böylece, **a**'nın **true** olup olmadığını saptamak için (bu örnekte doğrudur) **|** operatör metodu çağrılacaktı.

Gerçi, kısa devre operatörleri mümkün kılmak için kullanılan tekniğin ilk bakışta biraz dolambaçlı olduğunu düşünebilirsiniz. Ancak biraz daha düşünürseniz bunun mantıklı olduğunu görürsünüz. **true** ve **false**'u bir sınıf için aşırı yükleyerek, derleyicinin kısa devre operatörlerini açıkça aşırı yüklemek zorunda kalmadan bu operatörlerden yararlanmasına imkan tanımış olursunuz. Ayrıca, koşul deyimlerinde nesneleri kullanma becerisine de kavuşursunuz. Genel olarak, **&** ve **|** operatörlerinin çok kısıtlı olarak uygulanmasına ihtiyacınız yoksa, bunları tam olarak uygulamanız çok daha iyi olur.

## Dönüşüm Operatörleri

Bazı durumlarda, bir sınıfın ait bir nesneyi diğer veri tiplerini içeren bir deyim içinde kullanmak isteyeceksiniz. Bazen bir veya daha fazla operatörü aşırı yüklemek bunu gerçekleştirmek için yardımcı olabilir. Ancak, diğer durumlarda ihtiyacınız olan yalnızca, söz konusu sınıf tipinden hedef tipe dönüşümüdür. Bu gibi durumları ele almak için C#, *dönüşüm operatörü* (conversion operator) denilen özel bir **operatör** metodu tipi tanımlamanıza imkan verir. Dönüşüm operatörü, sınıfınıza ait bir nesneyi bir başka tipe dönüştürür. Aslında; dönüşüm operatörü, tip ataması operatörünü aşırı yükler. Dönüşüm operatörleri, bir sınıfın ait nesnelerin diğer veri tipleriyle serbestçe bir araya getirilmesine imkan vererek sınıf tiplerinin C# programlama ortamına tamamen entegre edilmesine yardımcı olmaktadır. Diğer veri tiplerine dönüşüm tanımlı olduğu sürece bu geçerlidir.

İki çeşit dönüşüm operatörü mevcuttur: **implicit** (kapalı) ve **explicit** (açık). Her birinin genel yapısı aşağıda gösterilmiştir:

```
public static operator implicit hedef-tip (kaynak-tip v) { return değer; }
public static operator explicit hedef-tip (kaynak-tip v) { return değer; }
```

Burada ***hedef-tip***, dönüşümü gerçekleştireceğiniz hedef tip; ***kaynak-tip***, dönüştüreceğiniz tip; ***değer*** ise dönüşüm sonrasında sınıfın sahip olacağı değerdir. Dönüşüm operatörleri ***hedef-tip*** tipinde veriler döndürür ve başka tip belirleyicilerine izin verilmez.

Eğer dönüşüm operatörü ***implicit*** olarak belirtilmişse, dönüşüm otomatik olarak gerçekleştirilir (söz gelişi, bir nesne hedef tiple birlikte bir deyim içinde kullanıldığında). Dönüşüm operatörü ***explicit*** olarak belirtilmişse, dönüşüm bir tip ataması kullanıldığında çağrırlar. Aynı hedef ve kaynak tipleri için hem kapalı hem de açık dönüşüm operatörleri tanımlayamazsınız.

Dönüşüm operatörünü göz önünde canlandırmak için ***ThreeD*** sınıfı için bir dönüşüm operatörü oluşturacağız. Diyelim ki, ***ThreeD*** tipinde bir nesneyi bir tamsayı deyim içinde kullanabilmek için bir tamsayıya dönüştürmek istiyorsunuz. Ayrıca; dönüşüm, üç boyutun çarpımı kullanılarak gerçekleştirilecek. Bunu başarmak için aşağıdaki gibi bir ***implicit*** dönüşüm operatörü kullanacaksınız:

```
public static implicit operator int(ThreeD op1)
{
    return op1.x * op1.y * op1.z;
}
```

İşte, bu dönüşüm operatörünü örnekleyen bir program:

```
// Kapali donusum operatoru kullanan bir ornek.

using System;

// Uc boyutlu koordinat sinifi.
class ThreeD {
    int x, y, z; // uc boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD (int i, int j, int k) { x = i; y = j; z = k; }

    // Ikili + operatorunu asiri yukle
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();

        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // ThreeD'den int'e kapali donusum.
    public static implicit operator int(ThreeD op1)
    {
        return op1.x * op1.y * op1.z;
    }
}
```

```
// X, Y, Z koordinatlarını göster.  
public void show()  
{  
    Console.WriteLine(x + " , " + y + " , " + z);  
}  
}  
  
class ThreeDDemo {  
    public static void Main() {  
        ThreeD a = new ThreeD(1, 2, 3);  
        ThreeD b = new ThreeD(10, 10, 10);  
        ThreeD c = new ThreeD();  
        int i;  
  
        Console.WriteLine("Here is a: ");  
        a.show();  
        Console.WriteLine();  
        Console.WriteLine("Here is b: ");  
        b.show();  
        Console.WriteLine();  
  
        c = a + b; // a ve b'yi topla  
        Console.WriteLine("Result of a + b: ");  
        c.show();  
        Console.WriteLine();  
  
        i = a; // int'e donustur  
        Console.WriteLine("Result of i = a: " + i);  
        Console.WriteLine();  
  
        i = a * 2 - b; // int'e donustur  
        Console.WriteLine("result of a * 2 - b: " + i);  
    }  
}
```

Bu program aşağıdaki çıktıyı ekranda gösterir:

```
Here is a: 1, 2, 3  
Here is b: 10, 10, 10  
Result of a + b: 11, 12, 13  
Result of i = a: 6  
result of a * 2 - b: -988
```

Programdan görüldüğü gibi, bir **ThreeD** nesne, **i=a** gibi bir tamsayı deyim içinde kullanıldığından nesne üzerine dönüşüm uygulanır. Bu spesifik örnekte dönüşüm **6** değerini döndürür. Bu değer, **a**'nın içinde saklanan koordinatların çarpımıdır. Ancak, bir deyim **int**'e dönüşüm gerektirmiyorsa, dönüşüm operatörü çağrılmaz. **c=a+b** için **operator int()**'in çağrılmamasının nedeni budur.

Hatırlarsanız, farklı ihtiyaçları karşılamak için farklı dönüşüm operatörleri tanımlayabilirsiniz. **double**'a ya da **long**'a dönüştüren bir tane tanımlayabilirsiniz, örneğin. Bunların her biri otomatik ve bağımsız olarak uygulanır.

**implicit** dönüşüm operatörü; bir deyim içinde bir dönüşüm gerekiğinde, bir metoda bir nesne aktarılırken, bir atama ifadesi içinde ve de hedef tip için açık bir tip ataması kullanırken otomatik olarak uygulanır. Alternatif olarak, yalnızca açık bir tip ataması kullanıldığında çağrılmak üzere bir **explicit** dönüşüm operatörü de tanımlayabilirsiniz. **explicit** dönüşüm operatörü otomatik olarak çağrılmaz. Örneğin, önceki programın **int**'e açık dönüşüm gerçekleştirmek üzere yeniden üzerinde çalışılmış versiyonu işte şu şekildedir:

```
// Açık dönüşüm kullanır.

using System;

// Üç boyutlu koordinat sınıfı.
class ThreeD {
    int x, y, z; // Üç boyutlu koordinatlar

    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j, z = k; }

    // İkili + operatorunu asiri yükle
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();

        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // 8u artık açık.
    public static explicit operator int(ThreeD op1)
    {
        return op1.x * op1.y * op1.z;
    }

    // X, Y, Z koordinatlarını göster.
    public void show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}

class ThreeDDemo {
    public static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();
        int i;
```

```

Console.WriteLine("Here is a: ");
a.show();
Console.WriteLine();
Console.WriteLine("Here is b: ");
b.show();
Console.WriteLine();

c = a + b; // a ve b'yi topla
Console.WriteLine("Result of a + b: ");
c.show();
Console.WriteLine();

i = (int) a; // int'e açık olarak dönüştür - tip ataması gereklidir
Console.WriteLine("Result of i = a: " + i);
Console.WriteLine();

i = (int)a * 2 - (int)b; // tip atamaları gerekiyor
Console.WriteLine("result of a * 2 - b: " + i);
}
}

```

Dönüşüm operatörü artık **explicit** olarak belirtildiği için **int**'e dönüşüm bir tip ataması yoluyla açık olarak yapılmalıdır. Örneğin, aşağıdaki satırda tip atamasını kaldırırsanız, program derlenmeyecektir:

```
i = (int) a; // int'e açık olarak dönüştür - tip ataması gereklidir
```

Dönüşüm operatörleriyle ilgili birkaç kısıtlama söz konusudur:

- Dönüşüm ya hedef tipi ya da kaynak tipi sizin oluşturduğunuz bir sınıf olmalıdır. Örneğin, **double**'dan **int**'e dönüşümü yeniden tanımlayamazsınız.
- **Object**'ten veya **Object**'e dönüşüm tamlayamazsınız.
- Aynı kaynak ve hedef tipleri için hem kapalı hem de açık dönüşüm tanımlayamazsınız.
- Temel sınıfından türetilmiş sınıfa dönüşüm tanımlayamazsınız. (Temel ve türetilmiş sınıflarla ilgili açıklama için Bölüm 11'e bakın.)
- Bir arayüzden veya bir arayüze dönüşüm tanımlayamazsınız. (Arayüzlerle ilgili açıklama için Bölüm 12'ye bakın.)

Bu kurallara ek olarak, **implicit** ve **explicit** dönüşüm operatörleri arasında karar verirken normalde uymanız gereken öneriler de mevcuttur. Kullanışlı olmalarına rağmen kapalı dönüşümler yalnızca, dönüşümün doğası gereği hatasız olduğu durumlarda kullanılmalıdır. Bunu garanti etmek için kapalı dönüşümler yalnızca şu iki koşul sağlandığında oluşturulmalıdır: Birincisi, kesme, taşıma ya da işaret kaybı gibi bilgi kayıpları meydana gelmemelidir. İkincisi: dönüşüm, kural dışı bir duruma neden olmamalıdır. Eğer dönüşüm bu iki koşulu sağlayamıyorsa, bu durumda açık dönüşüm kullanmalısınız.

## Operatörleri Aşırı Yüklemekle İlgili İpuçları ve Kısıtlamalar

Aşırı yüklenmiş bir operatörün tanımlanmış olduğu sınıf çerçevesindeki işleyisi, operatörün C#'ın standart tipleri çerçevesindeki varsayılan kullanımıyla bir ilgisi olması gerekmez. Yine de, kodunuzun yapısı ve okunaklılığı açısından aşırı yüklenmiş bir operatör, mümkünse operatörün orijinal kullanımını yansıtmalıdır. Örneğin, **Threed** ile bağlantılı + operatörü, tamsayı tipleriyle bağlantılı + operatörü ile kavramsal açıdan benzerdir. + operatörünü asıl görevinden farklı olarak, örneğin bir / operatörü gibi çalışacak şekilde bir sınıfa bağlı olarak tanımlamak çok bir yarar sağlamayacaktır. Buradaki ana fikir şudur: Aşırı yüklenmiş bir operatöre istediğiniz herhangi bir anlamı verebiliyor olsanız bile, anlaşılırlık açısından en iyisi, operatörün yeni anlamının orijinal anlamıyla ilgili olmasıdır.

Operatörlerin aşırı yüklenmesiyle ilgili bazı kısıtlamalar da söz konusudur. Hiçbir operatörün öncelik sırasını değiştiremezsiniz. Operatör metodunuz bir operandı göz ardı etmeyi tercih edebilse bile, operatörün gerektirdiği operand sayısını değiştiremezsiniz. Aşırı yükleme yapamayacağınız birkaç operatör de vardır. Belki de en önemlisi, herhangi bir atama operatörünü, += gibi bileşik atamalar da dahil olmak üzere aşırı yükleyemezsiniz. Aşırı yükleme yapamayacağınız diğer operatörler şunlardır:

&&		[ ]	()	new	is
sizeof	typeof	?	-	.	=

Tip ataması () operatörünü açıkça aşırı yüklemeniz mümkün olmasa da, daha önce söz ettigimiz gibi, bu işlevi gerçekleştiren dönüşüm operatörleri oluşturabilirsiniz.

+= gibi operatörlerin aşırı yüklenememesi ciddi bir kısıtlama gibi görünebilir. Ancak, bu aslında ciddi bir kısıtlama değildir. Genel olarak, bir operatör tanımlamış iseniz ve bu operatör bir bileşik ifadede kullanılıyorsa, bu durumun sonucu olarak aşırı yüklenmiş operatör metodunuz çağrılır. Dolayısıyla += otomatik olarak **operator+()**'nin size ait olan versiyonunu kullanır. Örneğin, **Threed** sınıfı üzerinde konuştuğumuzu varsayırsak, şu sekilde bir sekans kullanırsınız:

```
Threed a = new Threed(1, 2, 3);
Threed b = new Threed(10, 10, 10);
```

```
b += a; // a ve b'yi topla
```

**Threed**'nin **operator+()**'ı otomatik olarak çağrılır, **b**'ye 11, 12, 13 koordinatı atanır.

Son bir nokta: Dizi indeksleme operatörünü [] bir operatör metodu kullanarak aşırı yükleyemeseniz bile, bir sonraki bölümde tanımlanan indeksleyiciler (indexers) oluşturabilirsiniz.

## Operatörlerin Aşırı Yüklenmesine Bir Başka Örnek

Bu bölüm boyunca operatörlerin aşırı yüklenmesini göstermek için **Threed** sınıfını kullanmaktadır. Bu bağlamda, **Threed** sınıfı çok işimize yaradı. Ancak, bu bölümde noktalamadan önce bir başka örnek üzerinden geçmek de yararlı olur. Hangi sınıf kullanılırsa kullanılsın, operatörlerin aşırı yüklenmesiyle ilgili genel prensipler aynı olmasına rağmen, aşağıdaki örnek, özellikle tiplerin genişletilebilirliği söz konusu olduğunda operatörleri aşırı yüklemenin gücünü göstermeye yardımcı olmaktadır.

Bu örnekte dört bitlik bir tamsayı tipi geliştirilmekte ve bu tip için birkaç işlem tanımlanmaktadır. Belki de biliyorsunuzdur, bilişimin ilk yıllarda dört bitlik nicelikler yaygındı, çünkü dört bit, bir byte'in yarısını simgeliyordu. Ayrıca, onaltılık bir rakamı da tutabilecek kadar büyütüldü. Dört bit, yarı byte olduğu için dört bitlik niceliklerden kimi zaman “nybble” olarak söylenir. Programcılar, kodlarını her seferinde bir nybble olarak girdikleri ön panelli makinelerin kullanıldığı günlerde nybble cinsinden düşünmek sıradan, günlük bir işt! Şimdi de yaygın olmasa da dört bitlik tip hala C#'ın diğer tamsayılarına enteresan bir katkı sağlamaktadır. Geleneksel olarak nibble, işaretsiz bir değerdir.

Aşağıdaki örnek bir nybble veri tipini gerçekleştirmek için **Nybble** sınıfını kullanmaktadır. Sınıfın depolama altyapısı olarak **int** kullanılmaktadır; fakat bu, sınıfta tutulabilecek değerleri 0 ile 15 arasına sınırlamaktadır. Programda aşağıdaki operatörler tanımlanır:

- Bir **Nybble**'ı bir **Nybble**'a eklemek
- Bir **int**'i bir **Nybble**'a eklemek
- Bir **Nybble**'ı bir **int**'e eklemek
- Büyüktür ve küçüktür
- Artırma operatörü
- **Nybble**'dan **int**'e dönüşüm
- **int**'ten **Nybble**'a dönüşüm

Bir sınıf tipinin C# tip sistemine nasıl tam olarak entegre edilebileceğini göstermesi açısından bu işlemler yeterlidir. Yine de, komple bir **Nybble** uygulaması için diğer operatörlerin tümünü tanımlamanız gerekecektir. Bunları kendi kendinize eklemeyi denemek isteyebilirsiniz.

Komple **Nybble** sınıfı, sınıfın kullanımını gösteren **Nybbiedemo** ile birlikte aşağıda gösterilmektedir:

```
// Nybble adında 4 bitlik tip olusturur.

using System;

// 4 bitlik tip
class Nybble {
    int val; // depolama altyapisi
```

```
public Nybble() { val = 0; }

public Nybble(int i) {
    val = i;
    val = val & 0xF; // alt 4 biti kaybetme
}

// Nybble + Nybble icin ikili + operatorunu asiri yukle.
public static Nybble operator +(Nybble op1, Nybble op2)
{
    Nybble result = new Nybble();

    result.val = op1.val + op2.val;
    result.val = result.val & 0xF; // alt 4 biti kaybetme

    return result;
}

// Nybble + int icin ikili + operatorunu asiri yukle.
public static Nybble operator +(Nybble op1, int op2)
{
    Nybble result = new Nybble();

    result.val = op1.val + op2;
    result.val = result.val & 0xF; // alt 4 biti kaybetme

    return result;
}

// int + Nybble icin ikili + operatorunu asiri yukle.
public static Nybble operator +(int op1, Nybble op2)
{
    Nybble result = new Nybble();

    result.val = op1 + op2.val;
    result.val = result.val & 0xF; // alt 4 biti kaybetme

    return result;
}

// ++ operatorunu asiri yukle.
public static Nybble operatör ++(Nybble op)
{
    op.val++;

    op.val = op.val & 0xF; // alt 4 biti kaybetme

    return op;
}

// > operatorunu asiri yukle.
public static bool operator >(Nybble op1, Nybble op2)
{
    if(op1.val > op2.val) return true;
```

```
        else return false;
    }

    // < operatorunu asiri yükle.
    public static bool operator <(Nybble op1, Nybble op2)
    {
        if(op1.val < op2.val) return true;
        else return false;
    }

    // Bir Nybble'i int'e donustur
    public static implicit operator int (Nybble op)
    {
        return op.val;
    }

    // Bir int'i Nybble'a donustur
    public static implicit operator Nybble (int op)
    {
        return new Nybble(op);
    }
}

class NybbleDemo {
    public static void Main()
    {
        Nybble a = new Nybble(1);
        Nybble b = new Nybble(10);
        Nybble c = new Nybble();
        int t;

        Console.WriteLine("a: " + (int) a);
        Console.WriteLine("b: " + (int) b);

        // bir if ifadesinde bir Nybble kullan
        if(a < b) Console.WriteLine("a is less than b\n");

        // iki Nybble'i topla
        c = a + b;
        Console.WriteLine("c after c = a + b: " + (int) c);

        //Bir Nybble'a bir int ekle
        a += 5;
        Console.WriteLine("a after a += 5: " + (int) a);

        Console.WriteLine();

        // Bir Nybble'i bir tamsayi deyim icinde kullan
        t = a * 2 + 3;
        Console.WriteLine("Result of a * 2 + 3: " + t);

        Console.WriteLine();

        //int atamasini ve tasmayı göster
        a = 19;
        Console.WriteLine("Result of a = 19: " + (int) a);
    }
}
```

```

        Console.WriteLine();

        // donguyu kontrol etmek icin Nybble kullan
        Console.WriteLine("Control a for loop with a Nybble.");
        for(a = 0; a < 10; a++)
            Console.Write((int) a + " ");

        Console.WriteLine();
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

a: 1
b: 10
a is less than b

c after c = a + b: 11
a after a += 5: 6

Result of a * 2 + 3: 15

Result of a = 19: 3

Control a for loop with a Nybble.
0 1 2 3 4 5 6 7 8 9

```

**Nybble**'ın işleyişinin büyük kısmı kolaylıkla anlaşılıyor olsa gerek. Yine de, üzerinde durulması gereken önemli bir nokta vardır: Bir **Nybble**'ın C# tip sistemine entegrasyonunda dönüşüm operatörleri büyük rol oynar. Dönüşümler **Nybble**'dan **int**'e ya da **int**'ten **Nybble**'a şeklinde tanımlandığı için bir **Nybble** nesnesi aritmetik deyimler içine rahatlıkla karşılaştırılabilir. Örneğin, programdaki şu deyimi ele alın:

```
t = a * 2 + 3;
```

Burada, tipki **2** ve **3** gibi **t** de bir **int**'tir, ama **a** bir **Nybble**'dır. **Nybble**'ın dahili olarak **int**'e dönüştürülmesinden dolayı, deyimdeki bu iki tip birbirile uyumludur. Bu örnekte, deyimin kalansı **int** tipinde olduğu için **a**, **int**'e kendi dönüşüm metodu tarafından dönüştürülür.

**int**'ten **Nybble**'a dönüşüm bir **Nybble** nesnesine bir **int** değer atanmasına imkan tanır. Örneğin, programdaki şu ifade

```
a = 19;
```

şu şekilde çalışır, **int**'ten **Nybble**'a dönüştüren operatör çalıştırılır. Bu, **19** değerinin alt **4** bitini (bunun değeri **3**'tür çünkü **19**, **Nybble**'in menzilinden taşar) içeren yeni bir **Nybble** nesnesinin oluşturulmasına neden olur. Bu nesne daha sonra **a**'ya atanır. Dönüşüm operatörleri olmasaydı, bu tür deyimlere izin verilmeyecekti.

**Nybble**'dan `int`'e dönüşüm ayrıca `for` döngüsünde de kullanılmaktadır. Bu dönüşüm olmadan `for` döngüsünü bu denli açık bir şekilde yazmak mümkün olmayacaktı.

10

ONUNCU BÖLÜM

# İNDEKSLEYİCİLER VE ÖZELLİKLER

Bu bölümde birbiriyile yakın ilişkisi olan iki özel tipte sınıf üyesi incelenmektedir: indeksleyiciler (*indexers*) ve özellikler (*properties*). Bunların her biri, C#'ın tip sistemine entegrasyonunu artırarak ve esnekliğini geliştirerek ait oldukları sınıfın gücünü genişletirler. İndeksleyiciler, bir nesnenin tipki bir dizi gibi indekslenebileceği bir mekanizma sağlar. Özellikler ise, bir sınıfa ait örnek verilere erişimi yönetmek için standardize edilmiş yollardan biridir. Bu iki kavram birbiriyile ilgilidir, çünkü bunların her ikisi de C#'ın bir başka özelliğine, “erişimciye” (*accessor*) dayanmaktadır.

## İndeksleyiciler

Bildiğiniz gibi, dizi indeksleme işlemi `[]` operatörü kullanılarak gerçekleştirilir. `[]` operatörünü, kendi oluşturduğunuz sınıflar için aşırı yüklemeniz mümkündür; fakat bir operatör metodu kullanamazsınız. Bunun yerine, bir *indeksleyici* tanımlarsınız. İndeksleyici, bir nesnenin tipki bir dizi gibi indekslenmesine imkan verir. İndeksleyiciler temel olarak, bir veya daha fazla kısıtlamaya tabi olan özel dizilerin oluşturulmasını desteklemek için kullanılır. Ancak; indeksleyicileri, dizi benzeri bir söz diziminin avantajlı olacağı amaçlar için kullanabilirsiniz. İndeksleyiciler bir veya daha fazla boyuta sahip olabilir. Biz tek boyutlu indeksleyiciler ile başlayacağız.

### Tek Boyutlu İndeksleyici Oluşturmak

Tek boyutlu bir indeksleyici aşağıdaki gibi bir genel yapıya sahiptir:

```
eleman-tipi this [int indeks] {
    // get erişimcisi
    get {
        // indeks ile belirtilen değeri döndür
    }

    // set erişimcisi.
    set {
        // indeks ile belirtilen değeri ayarla
    }
}
```

Burada, **eleman-tipi** indeksleyicinin temel tipidir. Böylece, indeksleyici tarafından erişilen her eleman **eleman-tipi** tipinde olacaktır. Bu, dizinin temel tipine karşılık gelir. **indeks** parametresi, erişilmekte olan elemanın indeksini alır. Bu parametrenin teknik olarak **int** tipinde olması gerekmek, fakat indeksleyiciler tipik olarak dizi indeksleme işlevi sağladıkları için, tamsayı tipi alışılmıştır.

İndeksleyicinin gövdesi içinde **get** ve **set** adında iki *erişimci* (*accessor*) tanımlanır. Erişimci, metot ile aynıdır; tek fark, erişimcinin dönüş değerinin ve parametre deklarasyonunun olmamasıdır. İndeksleyici kullanıldığında erişimciler otomatik olarak çağrılar ve her iki erişimci de parametre olarak **indeks**'i alır. Eğer indeksleyici bir atama ifadesinin sol tarafında ise, **set** erişimcisi çağrılar; bu durumda **indeks** ile belirtilen eleman ayarlanmalıdır. Aksi

halde, **get** erişimcisi çağrırlar; bu kez **indeks** ile ilişkilendirilmiş değer döndürülmelidir. **set** metodu ayrıca, belirtilen **indeks**'e atanmakta olan değeri içeren ve **value** olarak adlandırılan bir değer de alır.

İndeksleyicinin sağladığı avantajlardan biri, uygun olmayan erişimleri önleyerek bir dizinin nasıl erişildiğini tam olarak kontrol edebilmenizdir. İşte bir örnek. Aşağıdaki programda **FailSoftArray** sınıfı, sınır hatalarını yakalayan bir dizi gerçeklemektedir. Bu sayede, eğer bir dizi, sınırı dışında indekslenirse çalışma zamanında ortaya çıkan kural dışı durumlar önlenmiş olur. Bu işlem, diziyi bir sınıfın içine **private** üye olarak paketleyerek ve dizinin yalnızca indeksleyici yoluyla erişimine imkan vererek gerçekleştirilebilir. Bu yöntem sayesinde, dizinin sınırları dışında kalan erişim denemeleri, bu tür bir girişimin zarifçe sökünlendirilmesi ile önlenebilir (böylece “çarpma” yerine “yumuşak iniş” elde edilmiş olur). **FailSoftArray**, indeksleyici kullandığı için normal dizi notasyonu kullanılarak erişilebilir.

```
// "fail-soft" bir dizi olusturmak icin indeksleyici kullanir.

using System;

class FailSoftArray {
    int[] a; // temel teskil eden diziye referans

    public int Length; // Length, public uyedir

    public bool errflag; // son islemin sonucunu gosterir

    // Buyuklugu verilen diziyi yapılandir.
    public FailSoftArray(int size) {
        a = new int[size];
        Length = size;
    }

    // Bu, FailSoftArray icin bir indeksleyicidir.
    public int this[int index] {
        //Bu, get erisimcisiidir.
        get {
            if(ok(index)) {
                errflag = false;
                return a[index];
            } else {
                errflag = true;
                return 0;
            }
        }
        // Bu, set erisimcisiidir.
        set {
            if(ok(index)) {
                a[index] = value;
                errflag = false;
            }
            else errflag = true;
        }
    }
}
```

```

        }
        // indeks sinirlar icindeyse true dondur.
        private bool ok(int index) {
            if(index >= 0 & index < Length) return true;
            return false;
        }
    }

    // "fail-soft" diziyi goster.
    class FSDemo {
        public static void Main() {
            FailSoftArray fs = new FailSoftArray(5);
            int x;

            // sessiz cokusleri goster
            Console.WriteLine("Fail quietly.");
            for(int i = 0; i < (fs.Length * 2); i++)
                fs[i] = i * 10;

            for(int i = 0; i < (fs.Length * 2); i++) {
                x = fs[i];
                if(x != -1) Console.Write(x + " ");
            }
            Console.WriteLine();

            // simdi, basarisiz durumları uret
            Console.WriteLine("\nFail with error reports.");
            for(int i = 0; i < (fs.Length * 2); i++) {
                fs[i] = i * 10;
                if(fs.errflag)
                    Console.WriteLine("fs[" + i + "] out-of-bounds");
            }

            for(int i = 0; i < (fs.Length * 2); i++) {
                x = fs[i];
                if(!fs.errflag) Console.Write(x + " ");
                else
                    Console.WriteLine("fs[" + i + "] out-of-bounds");
            }
        }
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Fail quietly.
0 10 20 30 40 0 0 0 0 0

Fail with error reports.
fs[5] out-of-bounds
fs[6] out-of-bounds
fs[7] out-of-bounds
fs[8] out-of-bounds
fs[9] out-of-bounds
0 10 20 30 40 fs[5] out-of-bounds
fs[6] out-of-bounds

```

```
fs[7] out-of-bounds
fs[8] out-of-bounds
fs[9] out-of-bounds
```

İndeksleyici, dizi sınırlarının aşılmasını önlemektedir. Gelin şimdi, indeksleyicinin her parçasına daha yakından bakalım. İndeksleyici şu satırla başlar:

```
public int tnis[int index] {
```

Bu ifade, **int** tipinde elemanlar üzerinde işlem gören bir indeksleyici deklare etmektedir. İndeks, **index** üzerinden aktarılır. İndeksleyici **public**'tir. Bu sayede, kendi sınıfı dışında kalan kod tarafından kullanılmasına imkan verilir.

**get()** erişimcisi aşağıda gösterilmiştir:

```
get {
    if(ok(index)) {
        errflag = false;
        return a[index];
    }
    else {
        errflag = true;
        return 0;
    }
}
```

**get** erişimcisi dizi sınır hatalarını önler. Eğer belirtilen indeks sınırlar içindeyse, indekse karşılık gelen eleman döndürülür. Eğer indeks sınır dışına taşıyorsa, hiç bir işlem gerçekleştirilmmez ve taşma meydana gelmez. **FailSoftArray**'in bu versiyonunda **errflag** adında bir değişken, tüm işlemlerin sonucunu içerir. Bu atan, her işlemden sonra söz konusu işlemin başarılı veya başarısız olduğunu değerlendirmek için incelenebilir.

**set** erişimcisi aşağıda gösterilmiştir. Bu da sınır hatalarını önler.

```
set {
    if(ok(index)) {
        a[index] = value;
        errflag = false;
    }
    else
        errflag = true;
}
```

Burada, eğer **index** sınırlar içindeyse, **value** içinde aktarılan değer, karşılık gelen elemana atanır. Aksi halde, **errflag**'a **true** değeri verilir. Hatırlarsanız, bir erişim metodunda **value** atanmakta olan değeri içeren otomatik bir parametredir. Bunu deklare etmenize gerek yoktur (zaten edemezsınız).

İndeksleyiciler hem **get**'i hem de **set**'i desteklemek zorunda değildir. Yalnızca **get** erişimcisini uygulayarak salt okunur bir indeksleyici oluşturabilirsiniz. Yalnızca **set**'i uygulayarak da salı yazılır bir indeksleyici oluşturabilirsiniz.

## İndeksleyiciler Aşırı Yüklenebilir

Bir indeksleyici aşırı yüklenebilir. Parametresi ve indeks olarak kullanılan argümanı arasında en yakın tip eşleşmesine sahip olan versiyon, çalıştırılan versiyon olacaktır. İşte, **FailSoftArray** indeksleyicisini **double** tipinde indeksler için aşırı yükleyen bir örnek. **double** indeksleyici, indeksini en yakın tamsayı değere yuvarlamaktadır.

```
// FailSoftArray indeksleyicisini asiri yuklemek.

using System;

class FailSoftArray {
    int[] a; // temel teskil eden diziye referans

    public int Length; // Length, public uyedir

    public bool errflag; // son islemin sonucunu gosterir

    // Buyuklugu verilen diziyi yapilandir.
    public FailSoftArray(int size) {
        a = new int[size];
        Length = size;
    }

    // Bu, FailSoftArray icin bir int indeksleyicisidir.
    public int this[int index] {
        // Bu, get erisimcisiidir.
        get {
            if (ok(index)) {
                errflag = false;
                return a[index];
            } else {
                errflag = true;
                return 0;
            }
        }

        // Bu, set erisimcisiidir
        set {
            if(ok(index)) {
                a[index] = value;
                errflag = false;
            }
            else errflag = true;
        }
    }

    /* Bu, FailSoftArray icin bir baska indeksleyicidir.
       Bu indeks, double arguman alir. Sonra bu argumani en
       yakin tamsayı indekse yuvarlar. */
    public int this[double idx] {
        // Bu, get erisimcisiidir.
        get {
```

```
int index;

// en yakin tamsayıya yuvarla
if((idx - (int) idx) < 0.5) index = (int) idx;
else index = (int) idx + 1;

if(ok(index)) {
    errflag = false;
    return a[index];
} else {
    errflag = true;
    return 0;
}

// Bu, set erisimcisiidir.
set {
    int index;

    // en yakin tamsayıya yuvarla
    if((idx - (int) idx) < 0.5) index = (int) idx;
    else index = (int) idx + 1;

    if (ok(index)) {
        a[index] = value;
        errflag = false;
    }
    else errflag = true;
}
}

// indeks sinirlarin icindeyse true dondur.
private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
}

// "fail-soft" diziyi goster.
class FSDemo {
    public static void Main() {
        FailSoftArray fs = new FailSoftArray(5);

        // fs icine bazi degerler yerlestir
        for(int i = 0; i < fs.Length; i++)
            fs[i] = i;

        // simdi int ve double'larla indeksle
        Console.WriteLine("fs[1]: " + fs[1]);
        Console.WriteLine("fs[2]: " + fs[2]);

        Console.WriteLine("fs[1.1]: " + fs[1.1]);
        Console.WriteLine("fs[1.6]: " + fs[1.6]);
    }
}
```

Bu program aşağıdaki çıktıyı üretir:

```
fs[1]: 1  
fs[2]: 2  
fs[1.1]: 1  
fs[1.6]: 2
```

Cıktıdan görüldüğü gibi, **double** indeksler en yakın tamsayı değere yuvarlanır. Spesifik olarak, **1.1**, **1**'e; **1.6**, **2**'ye yuvarlanır.

Bir indeksleyiciyi bu programda gösterildiği şekilde aşırı yüklemek geçerli olsa da, bu yaygın bir kullanım değildir. Genellikle bir indeksleyici, bir sınıfın ait bir nesnenin indeks olarak kullanılmasını mümkün kılmak için, indeksin özel bazı yollardan hesaplanması ile aşırı yüklenebilir.

## İndeksleyeciler, Temel Niteliğinde Bir Dizi Gerekmez

Bir indeksleyicinin aslında bir dizi üzerinde işlem görmesi için hiç bir gereklilik olmadığını kavramak önemlidir. İndeksleyici, yalnızca indeksleyici kullanıcısına “dizi gibi” bir işlevsellik sağlamalıdır. Örneğin, aşağıdaki program 0'dan 15'e kadar 2'nin kuvvetlerini içeren ve salt okunur bir dizi gibi davranışan bir indeksleyiciye sahiptir. Ancak, gerçekte bir dizinin mevcut olmadığını dikkat edin. Bunun yerine indeksleyici, sadece verilen indeksin gerçek değerini hesaplamaktadır.

```
// Indeksleyiciler gerçek dizi üzerinde işlem görmek zorunda degildir.  
  
using System;  
  
class PwrOfTwo {  
  
    /* 0'dan 15'e kadar 2'nin kuvvetlerini içeren bir mantıksal  
    diziye erisir. */  
    public int this[int index] {  
        // 2'nin kuvvetini hesapla ve dondur.  
        get {  
            if((index >= 0) && (index < 16)) return pwr(index);  
            else return -1;  
        }  
  
        // set erisimcisi mevcut degil  
    }  
  
    int pwr(int p) {  
        int result = 1;  
  
        for(int i = 0; i < p; i++)  
            result *= 2;  
  
        return result;  
    }  
}
```

```

class UsePwrOfTwo {
    public static void Main() {
        PwrOfTwo pwr = new PwrOfTwo();

        Console.WriteLine("First 8 powers of 2: ");
        for(int i = 0; i < 8; i++)
            Console.WriteLine(pwr[i] + " ");
        Console.WriteLine();

        Console.WriteLine("Here are some errors: ");
        Console.WriteLine(pwr[-1] + " " + pwr[17]);

        Console.WriteLine();
    }
}

```

Programın çıktısı aşağıdaki gibidir:

```

First 8 powers of 2: 1 2 4 8 16 32 64 128
Here are some errors: -1 -1

```

**PwrOfTwo** için indeksleyicinin **get** erişimcisi olduğuna, ama **set** erişimcisi olmadığına dikkat edin. Önceden açıklandığı gibi bu, indeksleyicinin salt okunur olduğu anlamına gelmektedir. Böylece, bir **PwrOfTwo** nesnesi bir atama ifadesinin sağ tarafında kullanılabilir, fakat sol tarafında kullanılamaz. Örneğin, yukarıdaki programa şu ifadeyi eklemeye çalışmak işe yaramayacaktır:

```
pwr[0] = 11; // derlenmeyecektir
```

Bu ifade, derleme hatasına neden olacaktır, çünkü indeksleyici için tanımlı bir **set()** erişimcisi mevcut değildir.

İndeksleyicileri kullanmakla ilgili iki önemli kısıtlama mevcuttur. Birincisi, bir indeksleyici bir depolama yeri tanımlamadığı için indeksleyicinin ürettiği bir değer, bir metoda **ref** veya **out** parametresi olarak aktarılamaz. İkinci olarak, bir indeksleyici kendi sınıfının bir örnek üyesi olmalıdır; **static** olarak deklare edilemez.

## Çok Boyutlu İndeksleyiciler

Çok boyutlu diziler için de indeksleyiciler tanımlayabilirsiniz. Örneğin, aşağıda iki boyutlu bir “fail-soft” dizisi görürsünüz. İndeksleyicinin deklare edilme şekline özellikle dikkat edin.

```

// İki boyutlu “fail-soft” dizi.

using System;

class FailSoftArray20 {
    int[,] a; // temel teskil eden 2 boyutlu diziye referans
    int rows, cols; // boyutlar
    public int Length; // Length public uyedir
}

```

```
public bool errflag; // son islemin sonucunu gosterir

// Boyutlari verilen bir dizi yapılandır.
public FailSoftArray2D(int r, int c) {
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
}

// Bu, FailSoftArray2D icin bir indeksleyicidir.
public int this[int index1, int index2] {
    // Bu, get erisimcisiidir.
    get {
        if(ok(index1, index2)) {
            errflag = false;
            return a[index1, index2];
        } else {
            errflag = true;
            return 0;
        }
    }
}

// Bu, set erisimcisiidir.
set {
    if(ok(index1, index2)) {
        a[index1, index2] = value;
        errflag = false;
    }
    else errflag = true;
}
}

// indeksler sinirlarin icindeyse true dondur.
private bool ok(int index1, int index2) {
    if(index1 >= 0 & index1 < rows & index2 >= 0 &
       index2 < cols) return true;

    return false;
}
}

// 2 boyutlu indeksleyiciyi goster.
class TwoDIndexerDemo {
    public static void Main() {
        FailSoftArray2D fs = new FailSoftArray2D(3, 5);
        int x;

        // sessiz cokusleri goster
        Console.WriteLine("Fail quietly.");
        for(int i = 0; i < 6; i++)
            fs[i, i] = i*10;

        for(int i = 0; i < 6; i++) {
            x = fs[i, i];
        }
    }
}
```

```

        if(x != -1) Console.WriteLine(x + " ");
    }
    Console.WriteLine();

    // simdi, basarisiz durumlari uret
    Console.WriteLine("\nFail with error reports.");
    for(int i = 0; i < 6; i++) {
        fs[i, i] = i * 10;
        if(fs.errflag)
            Console.WriteLine("fs[" + i + ", " + i + "] out-of-bounds");
    }

    for(int i = 0; i < 6; i++) {
        x = fs[i, i];
        if(!fs.errflag) Console.WriteLine(x + " ");
        else
            Console.WriteLine("fs[" + i + ", " + i + "] out-of-
                           bounds");
    }
}
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Fail quietly.
0 10 20 0 0 0

Fail with error reports.
fs[3, 3] out-of-bounds
fs[4, 4] out-of-bounds
fs[5, 5] out-of-bounds
0 10 20 fs[3, 3] out-of-bounds
fs[4, 4] out-of-bounds
fs[5, 5] out-of-bounds

```

## Özellikler

Bir başka sınıf üyesi tipi ise *özelliktir* (property). Özellik, bir alanı bu alana erişen metot ile birleştirir. Elinizdeki kitaptaki önceki bazı örneklerde gösterilmiş olduğu gibi, sık sık bir nesnenin kullanıcıları için kullanıma hazır bir alan oluşturmak istersiniz; ama aynı zamanda, bu alan için izin verilen işlemler üzerindeki kontrolü de devamlı kılmak istersiniz. Örneğin, söz konusu alana atanabilecek değerlerin aralığını sınırlamak isteyebilirsiniz. Bu amacı, bir **private** değişken ve bu değişkenin değerine erişmek için gerekli metotlar aracılığıyla gerçekleştirmemiz mümkün olsa da, özelliklerden yararlanmak çok daha iyi ve çok daha verimli bir yaklaşımındır.

Özellikler indeksleyicilere benzer. Bir özellik, bir isim ve bununla birlikle **get** ve **set** erişimcilerini içerir. Erişimciler, bir değişkenin değerini almak ve ayarlamak için kullanılır. Bir özelliğin en temel faydası, özelliğin isminin deyimlerde ve atamalarda normal bir değişken gibi kullanılabilmesidir; aslında **get** ve **set** erişimcileri otomatik olarak çağrılır. Bu, bir indeksleyicinin **get** ve **set** metodlarının otomatik olarak kullanılması ile aynıdır.

Bir özelliğin genel yapısı şu şekildedir:

```
tip isim {
    get {
        // get erişimci kodu
    }

    set {
        // set erişimci kodu
    }
}
```

Burada **tip**, özelliğin tipini belirler; örneğin, **tip**, **int** olabilir. **isim** ise özelliğin ismidir. Özellik bir kez tanımlandıktan sonra özellik isminin her kullanımı, ilgili erişimcinin çağrılmalarıyla sonuçlanır. **set** erişimcisi, otomatik olarak, özelliğe atanmakta olan değeri içeren **value** adında bir parametre alır.

Özelliklerin depolama konumları tanımlamadığını kavramak önemlidir. Böylece, özellik bir alana erişimi yönetir. Ancak, özelliğin kendisi bu alanı sağlamaz. Alan, özellikten bağımsız olarak belirtilmelidir.

İşte, **prop** adında bir alana erişmek için **myprop** adında bir özellik tanımlayan basit bir örnek. Bu örnekte özellik, yalnızca pozitif değerlerin atanmasını mümkün kılıyor.

```
// Basit bir ozellik ornegi,
using System;

class SimpProp {
    int prop; // myprop ile kontrol edilmekte olan alan

    public SimpProp() { prop = 0; }

    /*Bu, private ornek degiskeni prop'a erisimi destekleyen bir
     ozelliktir. Bu ozellik yalnızca pozitif degerlere izin
     verir.
    public int myprop {
        get {
            return prop;
        }
        set {
            if(value >= 0) prop = value;
        }
    }
}

// Bir ozellik goster.
class PropertyDemo {
    public static void Main() {
        SimpProp ob = new SimpProp();

        Console.WriteLine("Original value of ob.myprop: " + ob.myprop);
```

```

    ob.myprop = 100; // değer ata
    Console.WriteLine("Value of ob.myprop: " + ob.myprop);

    // prop'a negatif değerler atayamazsınız
    Console.WriteLine("Attempting to assign -10 to ob.myprop");
    ob.myprop = -10;
    Console.WriteLine("Value of ob.myprop: " + ob.myprop);
}
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Original value of ob.myprop: 0
Value of ob.myprop: 100
Attempting to assign -10 to ob.myprop
Value of ob.myprop: 100

```

Gelin bu programı şimdi dikkatlice inceleyelim. Programda **prop** adında bir **private** alan ve **myprop** adında **prop**'a erişimi kontrol eden bir özellik tanımlanır. Önceden açıklandığı üzere, özelliğin kendisi bir depolama yeri tanımlamaz. Özellik basitçe, bir alana erişimi kontrol eder. Böylece, temel niteliginde bir alan olmadan özellik kavramı diye bir şey olmaz. Üstelik, **prop**, **private** olduğu için yalnızca **myprop** aracılığıyla erişilebilir.

**myprop** özelliği **public** olarak belirtildiği için, kendi sınıfının dışında kalan kod tarafından da erişilebilir. Bu akla yatkındır, çünkü **myprop**, özel olan **prop**'a erişim sağlamaktadır. **get** erişimcisi yalnızca **prop**'un değerini döndürür, **set** erişimcisi ise, **prop**'un değerini sadece ve sadece değer pozitif olduğunda ayarlar. Böylece, **myprop** özelliği **prop**'un hangi değerlere sahip olabileceğini kontrol eder. İşte özellikler bu nedenle önemlidir.

**myprop** ile tanımlanan özelliğin tipine okunur-yazılır özellik denir, çünkü bu tür bir özellik, temel oluşturan alanın okunmasına ve yazılmasına imkan verir. Ancak, salt okunur ve salt yazıılır özellikler oluşturmak da mümkündür. Salt okunur bir özellik oluşturmak için yalnızca **get** erişimcisini tanımlayın. Salt yazıılır bir özellik tanımlamak için ise yalnızca **set** erişimcisi tanımlayın.

“**fail-soft**” dizisinin sınıfını daha da geliştirmek için özellik kullanabilirsiniz. Bildiğiniz gibi, tüm diziler kendileriyle ilişkilendirilmiş bir **Length** özelliğine sahiptir. Şimdiye kadar **FailSoftArray** sınıfında bu amaç için yalnızca **Length** adında bir **public** tamsayı alanı kullanıldı. Gerçi, bu iyi bir uygulama değildi, çünkü bu, **Length**'e “**fail-soft**” dizisinin uzunluğundan başka bazı değerler atanmasına da imkân vermektedir. (Örneğin, kötü niyetli bir programcı bu değeri kasten bozabilir.) **FailSoftArray**'in aşağıdaki versiyonunda gösterildiği gibi, **Length**'ı salt okunur bir özelliğe dönüştürerek bu durumu düzeltebiliriz:

```

// FailSoftArray'a Length özelliğini ekler.

using System;

class FailSoftArray {
    int[] a; // temel niteliginde diziye referans
}

```

```
int len; // dizinin uzunlugu - Length ozelligine dayanir

public bool errflag; // son islemin sonucunu gosterir

// Buyuklugu verilen diziyi yapislandir.
public FailSoftArray(int size) {
    a = new int[size];
    len = size;
}

// Salt okunur Length ozelligi.
public int Length {
    get {
        return len;
    }
}

// Bu, FailSoftArray icin bir indeksleyicidir.
public int this [int index] {
    // Bu, get erisimcisiidir.
    get {
        if(ok(index)) {
            errflag = false;
            return a[index];
        } else {
            errflag = true;
            return 0;
        }
    }

    // Bu, set erisimcisiidir.
    set {
        if(ok(index)) {
            a[index] = value;
            errflag = false;
        }
        else errflag = true;
    }
}

// indeks sinirlarin icindeyse true dondur.
private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
}

// Gelistirilmis "fail-soft" dizisini goster.
class ImprovedFSDemo {
    public static void Main() {
        FailSoftArray fs = new FailSoftArray(5);
        int x;

        // Length okunabilir
        for(int i = 0; i < fs.Length; i++)
```

```

        fs[i] = i * 10;

        for(int i = 0; i < fs.Length; i++) {
            x = fs[i];
            if (x != -1) Console.WriteLine(x + " ");
        }
        Console.WriteLine();

        // fs.Length = 10; // Hata, gecersiz!
    }
}

```

Artık `Length` veri saklamak için `private` değişken `len`'i kullanan bir özelliktir. `Length`, sadece bir `get` erişicisini tanımlamaktadır. Bu da `Length`'in salt okunur olduğu anlamına gelmektedir. Dolayısıyla, `Length`'ı okumak mümkündür, ancak onun değerini değiştirmek mümkün değildir. Bunu kendi kendinize kanıtlamak isterseniz, kaynak kodunda aşağıdaki satırın başındaki açıklama işaretini kaldırın:

```
// fs.Length = 10; // Hata, gecersiz!
```

Açıklama işaretini kaldırıldıktan sonra programı derlemeye çalıştığınızda, `Length`'in salt okunur olduğuna dair bir hata mesajı alacaksınız.

`Length` özellüğünün eklenmesi `FailSoftArray`'ı geliştirirse de, özelliklerin sağlayabileceği tek gelişme bundan ibaret değildir. Kendisine erişimin salt okunur olarak sınırlanması gereken `errflag` üyesi de bir özellüğe dönüştürmek için çok uygun bir adaydır. `FailSoftArray` üzerinde yapacağımız son geliştirmeler aşağıda gösterilmiştir. Burada `Error` adını verdigimiz ve veri tutmak için orijinal `errflag` değişkenini kullanan bir özellik oluşturmaktayız:

```

// errflag'i bir ozellige donusturur.

using System;

class FailSoftArray {
    int[] a; // temel teskil eden diziye referans
    int len; // dizinin uzunlugu

    bool errflag; // artik private olarak kullaniyoruz

    // Buyuklugu verilen diziyi yapislandir.
    public FailSoftArray(int size) {
        a = new int[size];
        len = size;
    }

    // Salt okunur Length ozelligi.
    public int Length {
        get {
            return len;
        }
    }
}

```

```

// Salt okunur Error ozelligi.
public bool Error {
    get {
        return errflag;
    }
}

// Bu, FailSoftArray icin bir indeksleyicidir.
public int this[int index] {
    // Bu, get erisimcisiidir.
    get {
        if(ok(index)) {
            errflag = false;
            return a[index];
        } else {
            errflag = true;
            return 0;
        }
    }
}

// Bu, set erisimcisiidir.
set {
    if(ok(index)) {
        a[index] = value;
        errflag = false;
    }
    else errflag = true;
}
}

// indeks sinirlarin icindeyse true dondur.
private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
}
}

// Gelistirilmis "fail-soft" diziyi göster.
class FinalFSDemo {
    public static void Main() {
        FailSoftArray fs = new FailSoftArray(5);

        // Error ozelligini kullan
        for(int i = 0; i < fs.Length + 1; i++) {
            fs[i] = i * 10;
            if(fs.Error)
                Console.WriteLine("Error with index " + i);
        }
    }
}

```

**Error** özelliğinin oluşturulmasıyla birlikte, **FailSoftArray**'de iki değişiklik yapılması gerekmıştır: Birincisi; artık **Error** özelliğinin altındaki veri deposu olarak kullanılan

**errflag**, **private** hale getirilmiştir. Bu nedenle **errflag**'a artık doğrudan erişim mümkün değildir. İkinci olarak; salt okunur bir **Error** özelliği eklenmiştir. Artık hataları yakalamak isteyen programların bu **Error** özelliğini sorgulamaları gerekecektir. **Main()**'de bu durumun bir uygulamasına yer verilmiş, kasıtlı olarak sınır hatası yapılmış ve bu hatayı yakalamak için **Error** özelliğini kullanılmıştır.

## Özelliklerle İlgili Kısıtlamalar

Özelliklerle ilgili bazı önemli kısıtlamalar söz konusudur. Birincisi, bir özellik bir veri depolama yeri tanımlamadığı için onun bir metoda **ref** ya da **out** parametresi olarak aktarılması mümkün değildir. İkincisi, bir özelliği aşırı yükleyemezsiniz. (Aynı değişkene erişen iki farklı özelliğiniz olabilir ama bu pek alışındık bir durum değildir.) Son olarak, derleyici tarafından böyle bir kural zorla uygulanmasa da, bir özellik **get** erişimcisi çağrıldığında ilgili olduğu değişkenin durumunu değiştirmemelidir. Bir başka ifade ile **get** işleminin bozucu olmaması gereklidir.

## İndeksleyiciler ve Özelliklerin Kullanımı

Onceki örnekler, indeksleyiciler ve özelliklerle ilgili temel mekanizmayı bize gösterdiler, ama bunların tam gücünü henüz ortaya koymadılar. Bölümü tamamlamak üzere yapacağımız örnekte, **RangeArray** adlı bir sınıf geliştireceğiz. Bu sınıf, indeks aralığı programcı tarafından geliştirilen bir dizi tanımlamak için indeksleyiciler ve özellikler kullanmaktadır.

Bildiğiniz gibi, C#'ta dizi indeksleri daima sıfırdan başlar. Ancak, bazı uygulamalarda dizi indekslerinin herhangi bir noktadan başlatılabilmesi yararlı olmaktadır. Örneğin, bazı durumlarda indeksi **1**'den başlatmak kolaylık sağlayabilir. Başka durumlarda da, örneğin **-5**'ten **5**'e giden bir indeks aralığı kullanmak için negatif indekslere izin vermek yarar sağlayabilir. Burada geliştirilen **RangeArray** sınıfı, bu ve diğer indeksleme türlerinin tanımlanmasına ve kullanımına olanak sağlamaktadır.

**RangeArray**'i kullanarak şöyle bir kod yazabilirsiniz:

```
RangeArray ra = new RangeArray(-5, 10);
// indeksleri -5'ten 10'a giden bir dizi

for(int i = -5; i <= 10; ra[i] = i;
// indeks -5'ten 10'a gitmektedir
```

Tahmin edebileceğiniz gibi, ilk satırda indeksleri **-5**'ten **10**'a (uç noktalar dahil) giden **RangeArray** adlı dizi yapılandırılmaktadır. İlk argüman, başlangıç indeksini belirlemektedir. İkinci argüman da bitiş indeksini vermektedir. **ra** bir kez bu şekilde kurulduktan sonra, ona **-5** ila **10** arasında değişen indekslerle referansta bulunmak olanaklı olmaktadır.

Aşağıda bu dizinin kullanımını gösteren **RangeArrayDemo** sınıfı ile birlikte **RangeArray** sınıfının tamamına yer verilmektedir. **RangeArray**, burada uygulandığı biçimde **int**

değerlerinden oluşan dizileri desteklemektedir ama siz kodu değiştirerek, diğer herhangi bir veri tipine de uyarlayabilirsiniz.

```
/* Indeks araligi varsayılandan farklı bir dizi sınıfı
   oluşturur. RangeArray sınıfı, indekslemenin sıfırdan
   farklı bir değerden başlamasına olanak sağlar. Bir
   RangeArray oluşturduğunda, baslangic ve bitis
   indekslerini de belirlersiniz. Indeks olarak negatif
   sayılar da kullanılabilir. Örneğin, indeksleri -5'ten 5'e, 1'den 10'a ya da 50'den 56'ya giden
   diziler oluşturabilirsiniz.
*/
using System;

class RangeArray {
    // private veriler
    int[] a; // temel teskil eden diziye referans
    int lowerBound; // en küçük indeks
    int upperBound; // en büyük indeks

    // özelliklerin verileri
    int len; // Length özelliğine temel teskil eden değişken
    bool errflag; // outcome'a temel teskil eden değişken

    // Diziyi verilen sınırlarla yapılandırır.
    public RangeArray(int low, int high) {
        highn++;
        if (high <= low) {
            Console.WriteLine("Invalid Indices");
            high = 1; // güvenlik için minimal bir dizi oluştur
            low = 0;
        }
        a = new int[high - low];
        len = high - low;

        lowerBound = low;
        upperBound = --high;
    }

    // Salt okunur Length özelliği.
    public int Length {
        get {
            return len;
        }
    }

    // Salt okunur Error özelliği.
    public bool Error {
        get {
            return errflag;
        }
    }

    // RangeArray'in indeksleyicisi.
```

```

public int this[int index] {
    // Bu, get erisimcisisidir.
    get {
        if(ok(index)) {
            errflag = false;
            return a[index - lowerBound];
        } else {
            errflag = true;
            return 0;
        }
    }

    // Bu, set erisimcisisidir.
    set {
        if(ok(index)) {
            a[index - lowerBound] = value;
            errflag = false;
        }
        else errflag = true;
    }
}

// indeks sinirlar icindeyse true dondur.
private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
}
}

// Indeks-aralik dizisini goster.
class RangeArrayDemo {
    public static void Main() {
        RangeArray ra = new RangeArray(-5, 5);
        RangeArray ra2 = new RangeArray(1, 10);
        RangeArray ra3 = new RangeArray(-20, -12);

        // ra'yi goster
        Console.WriteLine("Length of ra: " + ra.Length);

        for(int i = -5; i <= 5; i++)
            ra[i] = i;

        Console.Write("Contents of ra: ");
        for(int i = -5; i <= 5; i++)
            Console.Write(ra[i] + " ");

        Console.WriteLine("\n");

        // ra2'yi goster
        Console.WriteLine("Length of ra2: " + ra2.Length);

        for(int i = 1; i <= 10; i++)
            ra2[i] = i;

        Console.Write("Contents of ra2: ");
    }
}

```

```

        for(int i = 1; i <= 10; i++)
            Console.WriteLine(ra2[i] + " ");

        Console.WriteLine("\n");

        // ra3'u göster
        Console.WriteLine("Length of ra3: " + ra3.Length);

        for(int i = -20; i <= -12; i++)
            ra3[i] = i;

        Console.Write("Contents of ra3: ");
        for(int i = -20; i <= -12; i++)
            Console.Write(ra3[i] + " ");

        Console.WriteLine("\n");
    }
}

```

Programın çıktısı şöyledir:

```

Length of ra: 11
Contents of ra: -5 -4 -3 -2 0 1 2 3 4 5

Length of ra2: 10
Contents of ra2: 1 2 3 4 5 6 7 8 9 10

Length of ra3: 9
Contents of ra3: -20 -19 -18 -17 -16 -15 -14 -13 -12

```

Program çıkışının da doğruladığı gibi, **RangeArray** tipinden nesneler, sıfırdan başlamayan indekslere de sahip olabilmektedirler. Şimdi gelin, **RangeArray**'in nasıl uygulandığına biraz daha yakından bakalım:

**RangeArray**, aşağıdaki **private** örnek değişkenlerini tanımlayarak başlar:

```

// private veriler
int[] a; // temel teskil eden diziye referans
int lowerBound; // en küçük indeks
int upperBound; // en büyük indeks

// özelliklerin verileri
int len; // Length özelliğine temel teskil eden değişken
bool errflag; // outcome'a temel teskil eden değişken

```

Temel oluşturan diziye **a** ile referans verilir. Bu dizi için gerekli bellek alanı **RangeArray** yapılandırıcısı tarafından ayrıılır. Dizi indekslerinin alt limiti **lowerBound** değişkeninde, üst limiti de **upperBound** değişkeninde tutulur. Ardından, **Length** ve **Error** özelliklerini destekleyen örnek değişkenler deklare edilir.

**RangeArray** yapılandırıcısı aşağıda gösterilmiştir:

```
// Diziyi verilen sınırlarla yapılandır.
```

```

public RangeArray (int low, int high) {
    high++;
    if (high <= low) {
        Console.WriteLine("Invalid Indices");
        high = 1; // guvenlik icin minimal bir dizi olustur
        low = 0;
    }
    a = new int[high - low];
    len = high - low;

    lowerBound = low;
    upperBound = --high;
}

```

Bir **RangeArray**, indeks alt sınırı **low** üzerinden, üst sınırı da **high** üzerinden aktarılırak yapılandırılır. Daha sonra dizinin büyüklüğünü hesaplamak için **high** değerine artırma uygulanır. Bunun nedeni belirtilen indeks değerlerinin her iki uçtaki sınırları da içine almasıdır. Ardından üst sınırın alt sınırдан büyük olduğunun sağlanması yapılmaktadır. Üst sınır alt sınırдан büyük değilse, hata mesajı verilir ve tek elemanlı bir dizi oluşturulur. **Length** özelliğine temel oluşturan **len** değişkeni, dizideki elemanların sayısına eşitlenir. Son olarak da **lowerBound** ve **upperBound**'a değer ataması yapılır.

**RangeArray**; **Length** ve **Error** özelliklerinin uygulanmasını aşağıda gösterildiği şekilde gerçekleştirir:

```

// Salt okunur Length ozelligi.
public int Length {
    get {
        return len;
    }

    // Salt okunur Error ozelligi.
    public bool Error {
        get {
            return errflag;
        }
    }
}

```

Bu özellikler, **FailSoftArray** tarafından kullanılan özelliklere benzerdir ve aynı şekilde çalışırlar.

Daha sonra **RangeArray**, indeksleyicisini aşağıda gösterildiği gibi uygular:

```

// RangeArray icin indeksleyici.
public int this[int index] {
    // Bu, get erisimcisiidir.
    get {
        if(ok(index)) {
            errflag = false;
            return a[index - lowerBound];
        } else {
            errflag = true;
            return 0;
        }
    }
}

```

```
        }
    }

    // 8u, set erisimcisiidir.
    set {
        if(ok(index)) {
            a[index - lowerBound] = value;
            errflag = false;
        }
        else errflag = true;
    }
}
```

Bu indeksleyici, **FailSoftArray** tarafından kullanılana benzemekle birlikte, önemli bir farkı vardır. Dikkat ederseniz, **a**'yı indeksleyen deyim şu şekildedir:

```
index - lowerBound
```

Yukarıdaki ifade, **index** ile aktarılan indeks değerini **a** üzerinde kullanılmaya uygun, sıfır tabanlı bir indekse çevirir. Bu deyim, **lowerBound** pozitif te olsa, negatif te olsa, sıfır da olsa çalışır.

**ok()** metodu aşağıda gösterilmiştir:

```
// indeks sinirlarin icindeyse true dondur.
private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
}
```

**ok()** metodu da **FailSoftArray** tarafından kullanılan metoda benzer. Tek farkı, burada dizinin indeks aralığının lowerBound ve upperBound'a bakarak kontrol edilmesidir.

**RangeArray**, indeksleyicileri ve özelliklerini kullanarak oluşturabileceğiniz kişisel dizi tiplerinden sadece birini gösterir. Kuşkusuz birkaç başka tip oluşturmak ta mümkündür. Örneğin, gereksinime göre genişleyen ve daralan dinamik diziler, ilişkisel diziler ve seyrek diziler oluşturabilirsiniz. Alıştırma olarak bu dizi tiplerinden birini oluşturmayı denemek isteyebilirsiniz.

ONBİRİNCİ BÖLÜM

---

11

## KALITIM

Kalıtım, nesne yönelimli programmanın üç temel prensibinden biridir, çünkü kalıtım, hiyerarşik sınıflandırma oluşumlarına olanak tanır. Kalıtım kullanarak, birbiriyile bağlantılı bir grup ögenin ortak özelliklerini tanımlayan genel bir sınıf oluşturabilirsiniz. Bu sınıf daha sonra diğer, daha spesifik sınıflara kalıtım yoluyla aktarılır ve her sınıf kendisine özgü özellikleri buna ekler.

C# dilinde kalıtım yoluyla aktarılan sınıfa *temel sınıf* denir. Kalıtım işlemini gerçekleştiren sınıf *türetilmiş sınıf* olarak adlandırılır. Bu yüzden, türetilmiş sınıf temel sınıfın özelleştirilmiş bir versiyonudur. Türetilmiş sınıf, temel sınıf tarafından tanımlanan tüm değişken, metod, özellik, operatör ve indeksleyicileri kalıtım yoluyla elde eder ve kendisine özgü elemanları da ayrıca ekler.

## Kalıtımın Temel Unsurları

C#, bir sınıfın kendi deklarasyonuna bir başka sınıfı dahil etmesine olanak tanıyarak kalıtımı destekler. Bu, bir türetilmiş sınıf deklare edilirken temel sınıf ta belirtilerek gerçekleştirilir. Gelin şimdi bir örnekle başlayalım. Aşağıdaki **TwoDShape** adındaki sınıf, iki boyutlu “genel” bir şeklin niteliklerini tanımlamaktadır. Söz konusu şekil bir kare, dikdörtgen, üçgen vs olabilir.

```
// İki boyutlu nesneler için bir sınıf.
class TwoDShape {
    public double width;
    public double height;

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }
}
```

**TwoDShape**, iki boyutlu nesnelerin özel tiplerini tarif eden sınıflar için bir temel sınıf olarak (yani, bir başlangıç noktası olarak) kullanılabilir. Örneğin, aşağıdaki program **Triangle** adında bir sınıf türetmek için **TwoDShape**'ı kullanmaktadır, **Triangle**'ın deklare edilme şekline özellikle dikkat edin.

```
// Basit bir sınıf hiyerarsisi.

using System;

// İki boyutlu nesneler için bir sınıf.
class TwoDShape {
    public double width;
    public double height;

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }
}
```

```
// Triangle, TwoDShape'den türetilir.
class Triangle : TwoDShape {
    public string style; // ucgenin sekli

    // ucgenin alanini dondur.
    public double area() {
        return width * height / 2;
    }

    // ucgenin seklini goster.
    public void showStyle() {
        Console.WriteLine("Triangle is " + style);
    }
}

class Shapes {
    public static void Main() {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "isosceles";

        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "right";

        Console.WriteLine("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Area is " + t1.area());

        Console.WriteLine();

        Console.WriteLine("Info for t2: ");
        t2.showStyle();
        t2.ShowDim();
        Console.WriteLine("Area is " + t2.area());
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Info for t1:
Triangle is isosceles
Width and height are 4 and 4
Area is 8

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48
```

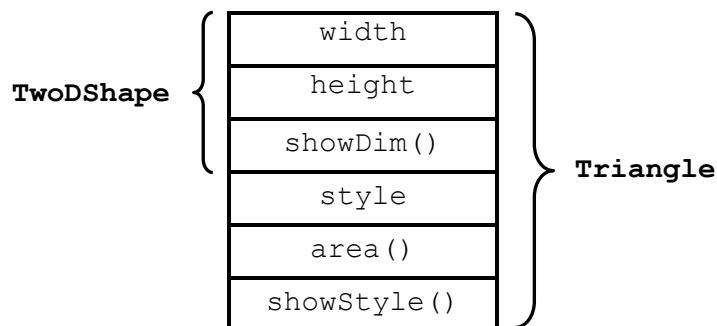
**Triangle** sınıfı, **TwoDShape**'in belirli bir tipini oluşturur. Bu örnekte bu tip bir üçgendir. **Triangle** sınıfı **TwoDShape**'in tümünü içerir; ayrıca, **style** alanını, **area()** ve **showStyle()** metodlarını da ekler. Üçgenin şekli **style** içinde saklanır; **area()**, üçgenin alanını hesaplar ve döndürür; **showStyle()** ise üçgenin şeklini ekranda gösterir.

**TwoDShape**'i kalıtım yoluyla elde etmek (devralmak) için **Triangle**'da kullanılan söz dizimine dikkat edin:

```
class Triangle : TwoDShape {
```

Bu söz dizimi genelleştirilebilir. Ne zaman bir sınıf diğerini katılım yoluyla elde etse, temel sınıfın ismi türetilmiş sınıfın ismini takip eder ve bu ikisi, iki nokta ile birbirinden ayrılır C#'ta bir sınıfı kalıtım yoluyla aktarmak için kullanılan söz dizimi dikkat çeken olçüde basit ve kullanışlıdır.

**Triangle**, kendisinin temel sınıfı olan **TwoDShape**'in tüm üyeleri içerdığı için **area()**'nın içindeki **width** ve **height**'a erişebilir. Ayrıca, **Main()** içinde, **t1** ve **t2** nesneleri de **width** ve **height**'a sanki bunlar **Triangle**'ın birer parçası olmuş gibi referansta bulunabilirler. Şekil 11.1'de **TwoDShape**'in **Triangle**'a nasıl dahil edildiği kavramsal olarak gösteriliyor.



**ŞEKİL 11.1:** *Triangle sınıfının kavramsal gösterimi*

**TwoDShape**, **Triangle** için bir temel olsa bile, ayını zamanda tamamen bağımsız, kendi başına varolan bir sınıfıtır. Bir türetilmiş sınıf için temel sınıf niteliğinde olmak, temel sınıfın kendi başına kullanılmayacağı anlamına gelmez. Örneğin, aşağıdaki kod parçası tamamen geçerlidir;

```
TwoDShape shape = new TwoDShape();

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Kuşkusuz, **TwoDShape**'e ait bir nesne ne **TwoDShape**'ten türetilen sınıflar hakkında ne de bu sınıflara erişimler hakkında bir bilgiye sahip değildir.

Bir temel sınıfın kalıtım yoluyla aktarıldığı sınıf deklârasyonunun genel yapısı aşağıdaki gibidir:

```
class türetilmiş-sınıf-ismi : temel-sınıf-ismi {
    // sınıfın gövdesi
}
```

Oluşturduğunuz her türetilmiş sınıf için yalnızca bir adet temel sınıf belirtebilirsiniz. C#, tek bir türetilmiş sınıf içine birden fazla temel sınıfı kalıtım yoluyla aktarmayı desteklemez. (*Bu, birden fazla temel sınıfı kalıtım yoluyla aktarabildiğiniz C++'taki yapıdan farklıdır. C++ kodunu C#'a dönüştürürken bu konuya dikkat edin.*) Ancak, bir türetilmiş sınıfın bir başka türetilmiş sınıfın temel sınıfı olmasını sağlayan bir kalıtım hiyerarşisi oluşturabilirsiniz. Kuşkusuz hiçbir sınıf dolaylı veya dolaysız olarak kendisinin temel sınıfı olamaz.

Kalıtımın başlıca avantajı şudur: Bir grup nesnenin ortak niteliklerini tanımlayan bir temel sınıf oluşturduktan sonra bu sınıf, çok daha spesifik olan türetilmiş sınıflardan islenilen sayıda oluşturmak için kullanılabilir. Türetilmiş sınıfların her biri, kendisine özgü sınıfılandırmayı tam olarak biçimlendirebilir. Örneğin, dikdörtgenleri bir sınıf içine paketleyen ve `TwoDShape`'ten türetilmiş bir başka sınıf, şu şekildedir:

```
// Dikdortgenler icin TwoDShape'ten türetilmis bir sınıf.
class Rectangle : TwoDShape {
    // Eger dortgen bir kareyse true dondur.
    public bool isSquare() {
        if(width == height) return true;
        return false;
    }

    // Dikdortgenin alanini dondur.
    public double area() {
        return width * height;
    }
}
```

`Rectangle` sınıfı `TwoDShape`'i içerir; ayrıca, dikdörtgenin kare olup olmadığını belirleyen `isSquare()` metodunu ve dikdörtgenin alanını hesaplayan `area()` metodunu da ekler.

## Üye Erişimi ve Kalıtım

Bölüm 8'de öğrendiğiniz gibi, sınıf üyelerinin izinsiz kullanımlarını veya gereksiz yere kurcalanmalarını önlemek için sınıf üyeleri genellikle `private` olarak deklare edilir. Bir sınıfı kalıtım yoluyla aktarmak, bu `private` erişim kısıtlamasını *çiğnemez*. Böylece, bir türetilmiş sınıf kendi temel sınıfının tüm üyelerini içerse dahi, temel sınıfın `private` olarak deklare edilmiş üyelerine erişemez. Örneğin, eğer `width` ve `height`, `TwoDShape` içinde, aşağıda gösterildiği gibi, `private` olarak deklare edilseydi, `Triangle`'ın bunlara erişmesi mümkün olmayacaktı:

```
// private üyeler kalitim yoluyla aktarilamaz.
```

```
// Bu ornek derlenmeyecektir.
using System;

// Iki boyutlu nesneler icin bir sinif.
class TwoDShape {
    double width; // artik ozel
    double height; // artik ozel

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }
}

// Triangle, TwoDShape'ten turetilir.
class Triangle : TwoDShape {
    public string style; // ucgenin sekli

    // Ucgenin alanini dondur.
    public double area() {
        return width * height / 2; // Hata, private uyeye erisemez
    }

    // ucgenin seklini goster.
    public void showStyle() {
        Console.WriteLine("Triangle is " + style);
    }
}
```

**Triangle** sınıfı derlenmeyecektir, çünkü **area()** metodu içindeki **width** ve **height**'a yönelik referanslar erişimin çiğnenmesine neden olur. **width** ve **height** şimdi **private** oldukları için artık yalnızca kendi sınıflarının diğer üyeleri tarafından erişilirler. Türetilmiş sınıflar bunlara erişim hakkına sahip değildir.

**NOT**

Bir sınıfın **private** üyesi kendi sınıfında **private** olarak kalacaktır. Böyle bir **private** üye, türetilmiş sınıflar da dahil olmak üzere kendi sınıfı dışında kalan kod tarafından erişilemez.

İlk bakışta, türetilmiş sınıfların, temel sınıfların **private** üyelerine erişimlerinin olmamasını ciddi bir kısıtlama gibi düşünebilirsiniz, çünkü bu durum **private** üyelerin kullanımlarına birçok yerde engel olacaktır. Ancak, bu doğru değildir, çünkü C# çeşitli çözümler sağlamaktadır. Bunlardan biri, **protected** (*korumalı*) üyeler kullanmaktadır. **protected** üyeleri bir sonraki bölümde anlatılmaktadır. İkinci bir çözüm, **private** verilere erişim sağlamak amacıyla **public** özellikler ya da metodlar kullanmaktadır. Önceki bölümlerde gördüğünüz gibi, C# programcıları genellikle bir sınıfın **private** üyelerine metodlar aracılığıyla ya da söz konusu üyeleri özellik haline getirerek erişim izni verirler. İşte aşağıda **TwoDShape** sınıflarının **width** ve **height** üyeleri özellik haline getirilerek yeniden yazılmış versiyonunu görmektesiniz:

```
// private uyeleri ayarlamak ve almak icin ozellik kullanir.

using System;

// Iki boyutlu nesneler icin bir sinif.
class TwoDShape {
    double pri_width; // artik ozel
    double pri_height; // artik ozel

    // width ve height icin ozellikler
    public double width {
        get { return pri_width; }
        set { pri_width = value; }
    }

    public double height {
        get { return pri_height; }
        set { pri_height = value; }
    }

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }
}

// Ucgenler icin TwoDShape'ten turetilmis bir sinif.
class Triangle : TwoDShape {
    public string style; // ucgenin sekli

    // ucgenin alanini dondur.
    public double area() {
        return width * height / 2;
    }

    // Ucgenin seklini goster.
    public void showStyle() {
        Console.WriteLine("Triangle is " + style);
    }
}

class Shapes2 {
    public static void Main() {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "isosceles";

        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "right";

        Console.WriteLine("Info for t1: ");
    }
}
```

```
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Area is " + t1.area());

        Console.WriteLine();

        Console.WriteLine("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        Console.WriteLine("Area is " + t2.area());
    }
}
```

Temci ve türetilmiş sınıflara referansta bulunurken kimi zaman *üst sınıf* (*superclass*) ve *alt sınıf* (*subclass*) terimleri kullanılır. Bu terimler Java programlamasından gelmektedir. Java'da üst sınıf olarak adlandırılan, C#'ta temel sınıf olarak adlandırılmaktadır. Java'da alt sınıf denilen ise, C#'ta türetilmiş sınıf olarak kullanılır. Her iki terim kümesini çoğunlukla her iki dilin sınıflarına uygulanırken göreceksiniz, fakat elinizdeki kitapta standart C# terimleri kullanılmaya devam edecektir. C++'ta da temel sınıf / türetilmiş sınıf terminolojisi kullanılmaktadır.

## protected Erişim Kullanımı

Az önce açıklandığı gibi, bir temel sınıfın bir **private** üyesi bir türetilmiş sınıf tarafından erişilemez. Buradan söyle bir sonuç çıkmaktadır: Bir türetilmiş sınıfın temel sınıfın içindeki bazı üyelerere erişebilmesini istiyorsanız, türetilmiş sınıfın **public** olması gereklidir. Kuşkusuz söz konusu üyeyi **public** hale getirmek, ayrıca bu üyeyi kodun diğer bölümlerinin erişimi için de **public** hale getirir. Oysa, istenilen bu olmayabilir. Neyse ki, bu çıkarım doğru değildir, çünkü C# bir *korumalı* üye oluşturmanıza olanak tanır. Bir korumalı üye sınıf hiyerarşisi içinde **public**'tir ama hiyerarşi dışında **private**'tir.

Korumalı üye **protected** erişim niteleyicisi kullanılarak oluşturulur. Bir sınıfın bir üyesi **protected** olarak deklere edilirse, söy konusu üye önemli bir istisna haricinde **private**'tir. **protected** üye kalıtım yoluyla aktarıldığı zaman, istisnai durum ortaya çıkar. Bu durumda, temel sınıfın **protected** üyesi türetilmiş sınıfın **protected** üyesi halini alır ve böylece türetilmiş sınıf tarafından erişilebilir. Bundan dolayı, **protected** niteleyicisini kullanarak, kendi sınıflarında **private** olan fakat yine de kalıtım yoluyla aktarılabilen ve bir türetilmiş sınıf tarafından erişilebilen sınıf üyeleri oluşturabilirsiniz.

Aşağıda, **protected** niteleyicisini kullanan basit bir örnek görünsünüz;

```
// protected'i tanitir.

using System;

class B {
    protected int i, j; // B'de private, fakat D tarafından erisilebilir

    public void set(int a, int b) {
        i = a;
```

```

        j = b;
    }

    public void show() {
        Console.WriteLine(i + " " + j);
    }
}

class D : B {
    int k; // private

    // D, B'nin i ve j üyelerine erişebilir
    public void setk() {
        k = i * j;
    }

    public void showk() {
        Console.WriteLine(k);
    }
}

class ProtectedDemo {
    public static void Main() {
        D ob = new D();

        ob.set(2, 3); // OK, D tarafından bilinir
        ob.show(); // OK, D tarafından bilinir

        ob.setk(); // OK, D'nin bir parçası
        ob.showk(); // OK, D'nin bir parçası
    }
}

```

Bu örnekte **B**, kalıtım yoluyla **D**'ye aktarıldığı için ve **B**'nin içinde **i** ve **j protected** olarak deklare edildikleri için, **setk()** metodu bunlara erişemez. Eğer **i** ve **j**, **B** tarafından **private** olarak tanımlanmış olsalardı **D**'nin bunlara erişimi olmayacağı ve program derlenmeyecekti.

Tıpkı **public** ve **private** gibi, hangi sayıda kalıtım katmanı dahil edilmiş olursa olsun **protected** statüsü üye ile birlikte kalır. Bu yüzden, bir türetilmiş sınıf bir başka türetilmiş sınıf için temel sınıf olarak kullanılırken, ilk türetilmiş sınıf tarafından kalıtım yoluyla elde edilen ilk temel sınıfın herhangi bir **protected** üyesi, ikinci türetilmiş sınıf'a da aynen **protected** olarak kalıtım yoluyla aktarılır.

## Yapılandırıcılar ve Kalıtım

Bir hiyerarşi içinde, hem temel sınıfların hem de türetilmiş sınıfların kendi yapılandırıcılarına sahip olmaları mümkündür. Bu ortaya önemli bir soru atar: Türetilmiş bu sınıfın bir nesnesini yapılandırmaktan hangi yapılandırıcı sorumludur? Temel sınıf içindeki mi, türelmiş sınıf içindeki mi, yoksa her ikisi de mi? Yanıt: Nesnenin temel sınıf'a ait parçaları temel sınıfın yapılandırıcısı tarafından, türetilmiş sınıf'a ait parçaları türetilmiş sınıfın

yapılandırıcı tarafından yapılandırılır. Bu akla yatkındır, çünkü temel sınıfın, türetilmiş sınıf içindeki öğeler ve öğelere erişim hakkında bilgisi yoktur. Bu nedenle, her iki sınıfın yapılandırıcıları ayrı olmalıdır. Önceki örnekler C# tarafından otomatik olarak oluşturulan varsayılan yapılandırıcılara dayanıyordu; dolayısıyla, bu bir sorun değildi. Ancak, pratikte birçok sınıfın yapılandırıcısı vardır. Burada, bu durumun nasıl kontrol altına alındığını göreceksiniz.

Yalnızca türetilmiş sınıfın içinde bir yapılandırıcı tanımlanırsa, söz konusu süreç basittir ve sadece türetilmiş sınıfın nesnesini yapılandırmaktan ibarettir. Nesnenin temel sınıfla ilgili parçası varsayılan yapılandırıcı tarafından otomatik olarak yapılandırılır. Örneğin, işte size, **Triangle**'ın üzerinde yeniden çalışılmış bir versiyonu. Bu kez **Triangle** içinde bir yapılandırıcı tanımlanmaktadır. Ayrıca, **style** da **private** hale getirilmiştir çünkü **style**'ın değeri artık yapılandırıcı tarafından verilmektedir.

```
// Triangle'a bir yapılandırıcı eklemek.

using System;

// İki boyutlu nesneler için bir sınıf.
class TwoDShape {
    double pri_width; // private
    double pri_height; // private

    // width ve height için özellikler.
    public double width {
        get { return pri_width; }
        set { priwidth = value; }
    }

    public double height {
        get { return pri_height; }
        set { pri_height = value; }
    }

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }
}

// Üçgenler için TwoDShape'ten türetilen bir sınıf.
class Triangle : TwoDShape {
    string style; // özel

    // Yapılandırıcı
    public Triangle(string s, double w, double h) {
        width = w; // temel sınıfı ilk kullanıma hazırla
        height = h; // temel sınıfı ilk kullanıma hazırla

        style = s; // türetilmiş sınıfı ilk kullanıma hazırla
    }
}
```

```

// Ucgenin alanini dondur.
public double area() {
    return width * height / 2;
}

// Ucgenin seklini goster.
public void showStyle() {
    Console.WriteLine("Triangle is " + style);
}
}

class Shapes3 {
    public static void Main() {
        Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
        Triangle t2 = new Triangle("right", 8.0, 12.0);

        Console.WriteLine("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Area is " + t1.area());

        Console.WriteLine();

        Console.WriteLine("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        Console.WriteLine("Area is " + t2.area())
    }
}

```

Bu örnekte **Triangle**'ın yapılandırıcısı, kalıtım yoluyla kendi **style** alanlarını da birlikte getiren **TwoDShape**'in üyelerini ilk kullanıma hazırlar.

Hem temel sınıfta hem de türetilmiş sınıfta yapılandırıcı tanımlanınca söz konusu süreç biraz daha karmaşık hale gelir, çünkü bu kez hem temel sınıfın hem de türetilmiş sınıfın yapılandırıcıları çalıştırılmalıdır. Bu durumda C#'ın bir başka anahtar kelimesini kullanmalısınız: **base**. **base**'in iki kullanım şekli vardır. Birinci kullanımı, temel sınıfın yapılandırıcısını çağırmaktadır. İkinci kullanımı ise; temel sınıfın, türetilmiş sınıfın bir üyesinden gizlediği üyesine erişmek içindir. Bu bölümde birinci kullanımını inceleyeceğiz.

## Temel Sınıfın Yapılandırıcılarını Çağırma

Türetilmiş bir sınıf kendi temel sınıfı içinde tanımlanmış bir yapılandırıcıyı çağrılabılır. Bunun için, türetilmiş sınıfın yapılandırıcı deklarasyonunun genişletilmiş bir şekli ve **base** anahtar kelimesi kullanılmalıdır. Bu genişletilmiş deklarasyonun genel yapısı aşağıdaki gibidir:

```
türetilmiş-yapılandırıcı (parametre-listesi) : base(arg-listesi) {
    // yapılandırıcının gövdesi
}
```

Burada **arg-listesi**, temel sınıf içindeki yapılandırmaların tarafından ihtiyaç duyulan herhangi bir argümanı belirtmektedir. İki noktanın konumuna dikkat edin.

**base**'in nasıl kullanıldığını anlamak için **TwoDShape**'in aşağıdaki programdaki versiyonunu ele alın. **TwoDShape**'de bu kez **width** ve **height** özelliklerine ilk değer atayan bir yapılandırıcı tanımlanmaktadır.

```
// TwoDShape'e yapılandırıcılar ekle.  
  
using System;  
  
// İki boyutlu nesneler için bir sınıf.  
class TwoDShape {  
    double pri_width; // private  
    double pri_height; // private  
  
    // TwoDShape için yapılandırıcı.  
    public TwoDShape(double w, double h) {  
        width = w;  
        height = h;  
    }  
  
    // width ve height için özellikler.  
    public double width {  
        get { return pri_width; }  
        set { pri_width = value; }  
    }  
  
    public double height {  
        get { return pri_height; }  
        set { pri_height = value; }  
    }  
  
    public void showDim() {  
        Console.WriteLine("Width and height are " + width +  
                          " and " + height);  
    }  
}  
  
// Üçgenler için TwoDShape'ten türetilen bir sınıf.  
class Triangle : TwoDShape {  
    string style; // özel  
  
    // Temel sınıf yapılandırıcısını çağır.  
    public Triangle(string s, double w, double h) : base(w, h) {  
        style = s;  
    }  
  
    // Üçgenin alanını dondur.  
    public double area() {  
        return width * height / 2;  
    }  
  
    // Üçgenin şeklini göster.  
    public void showStyle() {  
        Console.WriteLine("Triangle is " + style);  
    }  
}
```

```

    }

    class Shapes4 {
        public static void Main() {
            Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
            Triangle t2 = new Triangle("right", 8.0, 12.0);

            Console.WriteLine("Info for t1: ");
            t1.showStyle();
            t1.showDim();
            Console.WriteLine("Area is " + t1.area());

            Console.WriteLine();

            Console.WriteLine("Info for t2: ");
            t2.showStyle();
            t2.showDim();
            Console.WriteLine("Area is " + t2.area());
        }
    }
}

```

Bu örnekte **Triangle()**, **base**'i **w** ve **h** parametreleriyle çağırır. Bu, **TwoDShape()** yapılandırıcısının çağrılmamasına neden olur. **TwoDShape()** yapılandırıcısı bu parametre değerlerini kullanarak **width** ve **height**'a ilk değer atar. **Triangle** artık bu değerlere kendisi ilk değer atamaz, **Triangle** yalnızca kendisine ait bir değer olan **style**'a ilk değer atar. Bu sayede, **TwoDShape** kendi alt nesnesini istediği gibi yapılandırmakta özgür kalır. Üstelik **TwoDShape**, mevcut türetilmiş sınıflardan hangilerinin bilgisi olmadığı hakkında da bir işlevsellik ekleyebilir; böylece, mevcut kodun çökmesi önlenmiş olur.

Temel sınıf tarafından tanımlanan herhangi bir yapılandırıcı **base** tarafından çağrılabılır. Çalıştırılacak olan yapılandırıcı, argümanları eşleşen yapılandırıcı olacaktır. Örneğin, işte size **TwoDShape** ve **Triangle**'ın her ikisinin de genişletilmiş versiyonları. Aşağıdaki örnekte **TwoDShape** ve **Triangle**, hem varsayılan hem de tek argümanlı yapılandırıcı içerirler.

```

// TwoDShape'e biraz daha yapılandırıcı eklemek.

using System;

class TwoDShape {
    double pri_width; // private
    double pri_height; // private

    // varsayılan yapılandırıcı.
    public TwoDShape() {
        width = height = 0.0;
    }

    // TwoDShape için yapılandırıcı.
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
}

```

```
// Esit width ve height degerleri olan bir nesne yapislandir.
public TwoDShape(double x) {
    width = height = x;
}

// width ve height icin ozellikler.
public double width {
    get { return pri_width; }
    set { pri_width = value; }
}

public double height {
    get { return pri_height; }
    set { pri_height = value; }
}

public void showDim() {
    Console.WriteLine("Width and height are " + width +
                      " and " + height);
}

// Ucgenler icin TwoDShape'ten turetilen bir sinif.
class Triangle : TwoDShape {
    string style; // ozel

    /* Varsayılan yapislandirici. Bu, otomatik olarak TwoDShape'in
       varsayılan yapislandiricisini cagirir. */
    public Triangle() {
        style = "null";
    }

    // Uc arguman alan yapislandirici.
    public Triangle(string s, double w, double h) : base(w, h) {
        style = s;
    }

    // Ikizkenar bir ucgen yapislandir.
    public Triangle(double x) : base(x) {
        style = "isosceles";
    }

    // Ucgenin alanini dondur.
    public double area() {
        return width * height / 2;
    }

    // Ucgenin seklini goster.
    public void showStyle() {
        Console.WriteLine("Triangle is " + style);
    }
}

class Shapes5 {
```

```

public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle("right", 8.0, 12.0);
    Triangle t3 = new Triangle(4.0);

    t1 = t2;

    Console.WriteLine("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Area is " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Area is " + t2.area());

    Console.WriteLine();

    Console.WriteLine("Info for t3: ");
    t3.showStyle();
    t3.showDim();
    Console.WriteLine("Area is " + t3.area());

    Console.WriteLine();
}
}

```

Bu versiyonun çıktısı aşağıdaki gibidir.

```

Info for t1:
Triangle is right
Width and height are 8 and 12
Area is 48

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48

Info for t3:
Triangle is isosceles
Width and height are 4 and 4
Area is 8

```

Gelin, **base**'in ardında yatan en önemli kavramları yeniden gözden geçirelim. Bir türetilmiş sınıf **base** kelimesi kullanılarak belirtilince, kendisinin doğrudan türetildiği temel sınıfın yapılandırıcısını çağrıiyorudur. Bu, birden fazla seviyeye sahip hiyerarşiler için de geçerlidir. Temel sınıfın yapılandırıcısına argüman aktarmak için, bunları **base**'in argümanları olarak belirtmelisiniz. Eğer **base** kelimesi mevcut değilse, bu durumda **base** sınıfının varsayılan yapılandırıcısı otomatik olarak çağrılır,

## Kalıtım ve İsim Gizleme

Bir türetilmiş sınıfın temel sınıfındaki bir üye ile aynı ismi taşıyan bir üye tanımlaması mümkün değildir. Böyle bir durum söz konusu olunca temel sınıfın üyesi türetilmiş sınıf içinde gizlenmiş olur. Bu, C#'ta teknik olarak bir hata olmasa da derleyici bir uyarı mesajı verecektir. Bu bir ismin gizlenmeyeceği gerçekine işaret edecektir. Eğer niyetiniz temel sınıf üyesini gizlemek ise, bu durumda bu uyarıyı önlemek için türetilmiş sınıf üyesinin öncesinde **new** anahtar kelimesini kullanmalısınız. **new**'un bu kullanımının bir nesne örneği oluştururkenki kullanımından tamamen ayrı ve de farklı olduğunu kavrayın.

İşte, isim gizleme ile ilgili bir örnek:

```
// Kalitim baglantili isim gizleme ile ilgili bir ornek.

using System;

class A {
    public int i = 0;
}

// Bir turetilmis sinif olustur.
class B : A {
    new int i; // bu i, A'nin icindeki i'yi gizliyor

    public B(int b) {
        i = b; // B'nin icindeki i
    }

    public void show() {
        Console.WriteLine("i in derived class: " + i);
    }
}

class NameHiding {
    public static void Main() {
        B ob = new B(2);

        ob.show();
    }
}
```

Öncelikle, **B**'nin içinde **i** deklare edilirken **new**'un kullanımına dikkat edin. Aslında bu ifade, derleyiciye şunu anlatıyor: **A** temel sınıfı içindeki **i**'yi gizleyen **i** adında yeni bir değişkenin oluşturulmakta olduğunun farkındasınız. **new**'u dışında bırakırsanız, bir uyarı mesajı üretilir.

Bu programdan elde edilen çıktı aşağıdaki gibidir;

```
i in derived class: 2
```

**B**, **i** adında kendi örnek değişkenini tanımladığı için **A**'nın içindeki **i**'yi gizlemektedir. Bu yüzden, **show()** metodu **B** tipinde bir nesne üzerinde çağrırlrsa **i**'nin **A** içinde tanımlanan değeri değil de, **B** tarafından tanımlanan değeri ekranda gösterilir.

## Gizli Bir İsmeye Erişmek İçin base Kullanımı

**base**'in biraz **this** gibi davranışın ikinci bir şekli de vardır. Bunun **this**'ten tek farkı, bunun her zaman kendisinin kullanılmakta olduğu türetilmiş sınıfın türediği temel sınıfına referansta bulunmasıdır. Bu kullanımın genel yapısı şu sekildedir:

```
base.üye
```

Burada **üye**, bir metod ya da bir örnek değişken olabilir. **base**'in bu tür kullanımı, bir türetilmiş sınıf içindeki üye isminin temel sınıf içindeki aynı isimli üyeyi gizlediği durumlar için çok uygundur. Bir önceki örnekteki sınıf hiyerarşisinin şu versiyonunu ele alın:

```
//İsim gizliliğinin üstesinden gelmek için base kullanmak.

using System;

class A {
    public int i = 0;
}

// Türetilmiş bir sınıf oluştur.
class B : A {
    new int i; // bu i, A'nın içindeki i'yi gizliyor

    public B(int a, int b) {
        base.i = a; // bu, A'nın içindeki i'yi ortaya çıkarıyor
        i = b; // B'nin içindeki i
    }

    public void show() {
        // bu, A'nın içindeki i'yi gösterir.
        Console.WriteLine("i in base class: " + base.i);

        // bu, B'nin içindeki i'yi gösterir.
        Console.WriteLine("i in derived class: " + i);
    }
}

class UncoverName {
    public static void Main() {
        B ob = new B(1, 2);

        ob.show();
    }
}
```

Bu program aşağıdaki çıktıyi ekranda gösterir:

```
i in base class: 1
```

i in derived class: 2

B'nin içindeki i örnek değişkeni A'nın içindeki i'yi gizliyor olsa da base, temel sınıf içinde tanımlanmış i'ye erişim imkanı verir.

Gizlenmiş metodlar da base kullanılarak çağrılabılır. Şu örneği ele alın:

```
// Gizlenmis bir metot cagir.

using System;

class A {
    public int i = 0;

    // A'nin icindeki show()
    public void show() {
        Console.WriteLine("i in base class: " + i);
    }
}

// Türetilmis bir sinif olustur.
class B : A {
    new int i; // bu i, A'nin icindeki i'yi gizler

    public B(int a, int b) {
        base.i = a; // bu i, A'nin icindeki i'yi ortaya cikarir.
        i = b; // B'nin icindeki i
    }

    // Bu, A'nin icindeki show()'u gizler.
    // new kullanimina dikkat edin.
    new public void show() {
        base.show(); // bu, A'nin icindeki show()'u cagirir

        // bu, B'nin icindeki i'yi gosterir.
        Console.WriteLine("i in derived class: " + i);
    }
}

class UncoverName {
    public static void Main() {
        B ob = new B(1, 2);

        ob.show();
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
i in base class: 1
i in derived class: 2
```

Gördüğünüz gibi base.show(), show()'un temel sınıf içinde yer alan versiyonunu çağırmaktadır.

Bir diğer husus: Bu programda, **A**'nın içindeki **show()**’u gizleyen **show()** adında yeni bir metot oluşturulmakta olduğunu bildiğinizi derleyiciye anlatmak için **new** kullanıldığına dikkat edin.

## Çok Katmanlı Hiyerarşi Oluşturmak

Şu ana dek, yalnızca bir temel sınıf ve bir türemiş sınıfından oluşan basit sınıf hiperarşilerini kullanmaktayız. Ancak, istediğiniz kadar çok sayıda kalıtım katmanı içeren hiperarşiler inşa edebilirsiniz. Önceden de bahsedildiği gibi, bir türetilmiş sınıfı bir başkasının temel sınıfı olarak kullanmak tamamen geçerlidir. Örneğin, **A**, **B** ve **C** adında üç sınıfın verildiğini varsayıarak **C**, **B**'den; **B** de **A**'dan türetilerebilir. Bu tür bir durum söz konusu olduğunda, türetilmiş sınıfların her biri kendi temel sınıflarının tümünde yer alan özelliklerin tümünü kalıtım yoluyla elde eder. Bu örnekte **C**, **B** ve **A**'nın tüm özelliklerine kalıtım yoluyla sahip olur.

Çok katmanlı hiperarşinin nasıl kullanışlı olabileceğini anlamak için, aşağıdaki programı ele alın. Bu programda **Triangle** adındaki türetilmiş sınıf, **ColorTriangle** adında türetilmiş sınıfı oluşturmak için temel sınıf olarak kullanılmaktadır. **ColorTriangle**, **Triangle** ve **TwoDShape**'in tüm özelliklerini kalıtım yoluyla elde ettiği gibi, üçgenin rengini tutan **color** adında bir de alan eklemektedir.

```
// Çok katmanlı bir hiperarsı.

using System;

class TwoDShape {
    double pri_width; // private
    double pri_height; // private

    // Varsayılan yapılandırıcı.
    public TwoDShape() {
        width = height = 0.0;
    }

    // TwoDShape için yapılandırıcı.
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Eşit width ve height değerlerine sahip bir nesne yapılandırır
    public TwoDShape(double x) {
        width = height = x;
    }

    // width ve height için özellikler.
    public double width {
        get { return pri_width; }
        set { pri_width = value; }
    }
}
```

```
public double height {
    get { return pri_height; }
    set { pri_height = value; }
}

public void showDim() {
    Console.WriteLine("Width and height are " + width +
                      " and " + height);
}

// Ucgenler icin TwoDShape'ten turetilen bir sinif.
class Triangle : TwoDShape {
    string style; // ozel

/* Varsayılan yapılandırıcı.
   Bu, TwoDShape'in varsayılan yapılandırıcısını çağırır. */
public Triangle() {
    style = "null";
}

// Yapılandırıcı
public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
}

// Bir ikizkenar ucgen yapılandır.
public Triangle(double x) : base(x) {
    style = "isosceles";
}

// ucgenin alanını dondur.
public double area() {
    return width * height / 2;
}

// Ucgenin seklini göster.
public void showStyle() {
    Console.WriteLine("Triangle is " + style);
}
}

// Triangle'i genişlet.
class ColorTriangle : Triangle {
    string color;

    public colorTriangle(string c, string s, double w,
                         double h) : base(s, w, h) {
        color = c;
    }

    // Rengi göster.
    public void showColor() {
        Console.WriteLine("Color is " + color);
    }
}
```

```

class Shapes6 {
    public static void Main() {
        ColorTriangle t1 =
            new ColorTriangle("Blue", "right", 8.0, 12.0);
        ColorTriangle t2 =
            new ColorTriangle("Red", "isosceles", 2.0, 2.0);

        Console.WriteLine("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        t1.showColor();
        Console.WriteLine("Area is " + t1.area());

        Console.WriteLine();

        Console.WriteLine("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        t2.showColor();
        Console.WriteLine("Area is " + t2.area());
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Info for t1;
Triangle is right
Width and height are 8 and 12
Color is Blue
Area is 48

```

```

Info for t2;
Triangle is isosceles
Width and height are 2 and 2
Color is Red
Area is 2

```

Kalımdan ötürü **ColorTriangle**, önceden tanımlı **Triangle** ve **TwoDShape** sınıflarından yararlanabilir. **ColorTriangle**'ın yalnızca kendisine özgü, spesifik uygulama için gerek duyduğu ekstra enformasyonu eklemesi yeterlidir. Bu, kalıtımın önemini bir parçasıdır; kodun yenide kullanımına olanak tanır.

Bu örnek, bir diğer önemli hususu daha açıklamaktadır: **base** daima kendisine en yakın temel sınıf içindeki yapılandırıcıyla ilişkilidir. **ColorTriangle** içindeki **base**, **Triangle** içindeki yapılandırıcıyı çağırır. **Triangle** içindeki **base** ise, **TwoDShape** içindeki yapılandırıcıyı çağırır. Bir sınıf hiyerarşisinde eger bir temel sınıf yapılandırıcısı parametre gerektirirse, tüm türetilmiş sınıflar bu parametreleri “yükarıya doğru” aktarmalıdır. Türetilmiş sınıfın kendisi parametre gerektirsin ya da gerektirmesin, bu durum geçerlidir.

## Yapılandırıcılar Ne Zaman Çağrılır?

Kalıtım ve sınıf hiyerarşileri yukarıda ele alınırken önemli bir soru aklınıza takılmış olabilir: Türetilmiş bir sınıf oluşturulunca önce kimin yapılandırıcısı çalıştırılır; türetilmiş sınıfın yapılandırıcısı mı, yoksa temel sınıf tarafından tanımlanan yapılandırıcı mı? Örneğin, **B** adında bir türetilmiş sınıf ve **A** adında bir temel sınıf verildiğini varsayalım. **A**'nın yapılandırıcısı **B**'ninkinden önce mi çağrılır, yoksa sonra mı? Bu sorunun yanıtı şudur: Bir sınıf hiyerarşisinde yapılandırıcılar, temel sınıfından türetilmiş sınıfa doğru türetilme sırasına göre çağrırlar. Üstelik, **base** kullanılsın ya da kullanılmamasın bu sırada aynıdır. Eğer **base** kullanılmazsa, temel sınıfların her birinin varsayılan (parametresiz) yapılandırıcısı çalıştırılacaktır. Aşağıdaki program yapılandırıcıların çalıştırılma sırasını göstermektedir:

```
// Yapilandiricilarin ne zaman cagrilacagini gosterir.

using System;

// Bir temel sinif olustur.
class A {
    public A() {
        Console.WriteLine("Constructing A.");
    }
}

// A'dan turetilen bir sinif olustur.
class B : A {
    public B() {
        Console.WriteLine("Constructing B.");
    }
}

// B'den turetilen bir sinif olustur.
class C : B {
    public C() {
        Console.WriteLine("Constructing C.");
    }
}

class orderOfConstruction {
    public static void Main() {
        C c = new C();
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Constructing A
Constructing B
Constructing C
```

Gördüğünüz gibi, yapılandırıcılar türetilme sırasına göre çağrılmaktadır.

Üzerinde biraz düşünürseniz, yapılandırıcıların türetilme sırasına göre çağrımlarının akla yatkın olduğunu görürsünüz. Temel sınıfın türetilmiş sınıflar hakkında bir bilgisi olmadığı için temel sınıfın gerektirdiği herhangi bir ilk kullanıma hazırlama işlemi, türetilmiş sınıflar tarafından gerçekleştirilen ilk kullanıma hazırlama işleminden ayrıdır ve muhtemelen de ön şart niteliğindedir. Bu nedenle, ilk önce gerçeklenmelidir.

## Temel Sınıf Referansları ve Türetilmiş Nesneler

Bildiğiniz gibi, C# veri tiplerine sıkı sıkıya bağlı bir dildir. Standart dönüşümler ve basit tiplere uygulanan otomatik terfiler bir yana, tip uyumluluğu kesinlikle zorlanır. Bu yüzden, bir sınıf tipindeki bir referans değişkeni normalde bir başka sınıf tipindeki nesneye referansta bulunamaz. Örneğin, aşağıdaki programı ele alın:

```
// Bu program derlenmeyecektir.

class X {
    int a;

    public X(int i) { a = i; }
}

class Y {
    int a;

    public Y(int i) { a = i; }
}

class IncompatibleRef {
    public static void Main() {
        X x = new X(10);
        X x2;
        Y y = new Y(5);

        x2 = x; // OK, her ikisi de aynı tipte
        x2 = y; // Hata, aynı tipte degiller
    }
}
```

Bu örnekte, **x** ve **y** sınıfları fiziksel olarak aynı olmalarına rağmen, bir **y** nesnesini bir **x** referans değişkenine atamak mümkün değildir, çünkü tipleri farklıdır. Genel olarak, bir nesne referans değişkeni yalnızca kendi tipindeki nesnelere referansta bulunabilir.

Ancak, C#'ın tipler üzerindeki katı baskısının önemli bir istisnası mevcuttur. Bir temel sınıfı ait referans değişkenine, bu temel sınıfından türetilmiş herhangi bir sınıfı ait bir nesnenin referansı değer olarak atanabilir. İşte bir örnek:

```
// temel sınıf referansı, türetilmiş sınıf nesnesine referansta bulunabilir.

using System;
```

```
class X {
    public int a;

    public X(int i) {
        a = i;
    }
}

class Y : X {
    public int b;

    public Y(int i, int j) : base(j) {
        b = i;
    }
}

class BaseRef {
    public static void Main() {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);

        x2 = x; // OK, her ikisi de aynı tipte
        Console.WriteLine("x2.a: " + x2.a);

        x2 = y; // bu da tamam cunku Y, X'ten türetiliyor
        Console.WriteLine("x2.a: " + x2.a);

        // x referansları yalnızca X'in üyeleri hakkında bilgi sabibi
        x2.a = 19; // OK
        // x2.b = 27; // Hata, X'in b adında bir üyesi yok
    }
}
```

Bu örnekte **y** artık **x**'ten türetilmektektir. Böylece, **x2**'ye bir **y** nesnesini gösteren referans değer olarak atanabilir.

Hangi üyelerin erişilebileceğini belirleyen referans değişkeninin tipidir; referans değişkeninin referansta bulunduğu nesnenin tipi değildir. Bu ayrimı kavramak önemlidir. Yani, bir türetilmiş sınıf nesnesine referans olarak bir temel sınıf referans değişkeni atanırken söz konusu nesnenin yalnızca temel sınıf tarafından tanımlanan parçalarına erişim iznine sahip olacaksınız. **x2**'nin bir **y** nesnesine referansta bulunmasına rağmen **b** değerine erişemiyor olmasının nedeni budur. Bu akla yatkındır, çünkü türetilmiş sınıfın kendi içinde temel sınıfa neler eklendiğinden, temel sınıfın haberi yoktur. Programın son satırının açıklama şeklinde programdan çıkarılması da işte bu sebeptendir.

Yukarıda anlatılanlar, deyim yerindeyse bir parça “uçuk” görünüyor olsa da, bunların bazı önemli pratik uygulamaları mevcuttur. Bu uygulamalardan biri burada anlatılmaktadır. Diğer ise bu bölümün ileriki sayfalarında, sanal metotlar anlatılırken ele alınacaktır.

Türetilmiş sınıf referanslarına temel sınıf değişkenlerinin atandığı önemli yerlerden biri, bir sınıf hiyerarşisi içinde yapılandırıcılar çağrıldığı zamandır. Bildiğiniz gibi, bir sınıfın

kendisine ait bir nesneyi parametre olarak alan bir yapılandırıcı tanımlaması yaygın bir uygulamadır. Bu sayede, söz konusu sınıfın bir nesnenin kopyasını çıkarmasına olanak tanınmış olur. Bu tür bir sınıfın türetilen sınıflar, bu özellikten yararlanabilirler. Örneğin, **TwoDShape** ve **Triangle**'ın aşağıdaki versiyonlarını ele alın. Bu sınıfların her ikisi de parametre olarak bir nesne alan yapılandırıcı eklemektedir.

```
// Bir türetilmiş sınıf referansını bir temel sınıf referansına aktarmak.

using System;

class TwoDShape {
    double pri_width; // özel
    double pri_height; // özel

    // Varsayılan yapılandırıcı.
    public TwoDShape() {
        width = height = 0.0;
    }

    // TwoDShape için yapılandırıcı.
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Eşit width ve height değerlerine sahip bir nesne yapılandırır.
    public TwoDShape(double x) {
        width = height = x;
    }

    // Bir nesneden bir başka nesne yapılandırır.
    public TwoDShape(TwoDShape ob) {
        width = ob.width;
        height = ob.height;
    }

    // width ve height için özellikler.
    public double width {
        get { return pri_width; }
        set { pri_width = value; }
    }

    public double height {
        get { return pri_height; }
        set { pri_height = value; }
    }

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                          " and " + height);
    }
}

// Üçgenler için TwoDShape'ten türetilen bir sınıf.
```

```
class Triangle : TwoDShape {
    string style; // ozel

    // Varsayılan yapılandırıcı.
    public Triangle() {
        style = "null";
    }

    // Triangle için yapılandırıcı.
    public Triangle(string s, double w, double h) : base(w, h) {
        style = s;
    }

    // Bir ikizkenar üçgen yapılandır.
    public Triangle(double x) : base(x) {
        style = "isosceles";
    }

    // Bir nesneden bir başka nesne yapılandır.
    public Triangle(Triangle ob) : base(ob) {
        style = ob.style;
    }

    // Üçgenin alanını dondur.
    public double area() {
        return width * height / 2;
    }

    // Üçgenin şeklini göster.
    public void showStyle() {
        Console.WriteLine("Triangle is " + style);
    }
}

class Shapes7 {
    public static void Main() {
        Triangle t1 = new Triangle("right", 8.0, 12.0);

        // t1'in kopyasını çıkart
        Triangle t2 = new Triangle(t1);

        Console.WriteLine("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Area is " + t1.area());

        Console.WriteLine();

        Console.WriteLine("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        Console.WriteLine("Area is " + t2.area());
    }
}
```

Bu programda **t2**, **t1** kullanılarak kurulmuştur. Bu nedenle, her ikisi de aynıdır. Çıktı aşağıda gösterildiği gibidir:

```
Info for t1:  
Triangle is right  
Width and height are 8 and 12  
Area is 48
```

```
Info for t2:  
Triangle is right  
Width and height are 8 and 12  
Area is 48
```

Bu **Triangle** yapılandırıcısına özellikle dikkat edin:

```
// Bir nesneden bir başka nesne yapılandır.  
public Triangle(Triangle ob) : base(ob) {  
    style = ob.style;  
}
```

Bu yapılandırıcı, **Triangle** tipinde bir nesne alır ve bu nesneyi (**base** aracılığıyla) aşağıda gösterilen **TwoDShape** yapılandırıcısına aktarır:

```
// Bir nesneden bir başka nesne yapılandır.  
public TwoDShape(TwoDShape ob) {  
    width = ob.width;  
    height = ob.height;  
}
```

Buradaki en önemli nokta, **TwoDShape()**'in bir **TwoDShape()** nesnesi bekliyor olmasıdır. Ancak; **Triangle()**, bir **Triangle** nesnesi aktarır. Önceden açıklandığı gibi, bu kod çalışır, çünkü temel sınıf tipinde bir referans, bir türetilmiş sınıf nesnesine referansta bulunabilir. Bu nedenle, **TwoDShape()** tipinde bir referansı, **TwoDShape**'ten türetilmiş bir sınıfın nesnesine aktarmak kesinlikle kabul edilebilir bir yaklaşımdır. **TwoDShape()** yapılandırıcısı, türetilmiş sınıf nesnesinin yalnızca **TwoDShape** üyesi olan kısımlarına ilk değer atadığı için, söz konusu nesnenin türetilmiş sınıflar tarafından eklenmiş başka üyeleri de içerebilecek olması önemli değildir.

## Sanal Metotlar ve Devre Dışı Bırakma (Overriding)

*Sanal metot*, temel sınıf içinde **virtual** olarak deklare edilen ve bir veya daha fazla türetilmiş sınıf içinde yeniden tanımlanan bir metottur. Böylece, her türetilmiş sınıf bir sanal metodun kendine özgü bir versiyonuna sahip olabilir. Sanal metotlar ilginçtir, çünkü sanal metodlardan biri bir temel sınıf referansı tarafından çağrıldığında ortaya çıkan durum ilginçtir. Bu durumda, referans tarafından *referansta bulunan* nesnenin *tipine* bağlı olarak C#, söz konusu metodun hangi versiyonunun çağrılacağını belirler ve bu belirleme *çalışma zamanında* gerçekleştirilir. Yani, farklı nesnelere referansta bulunuyorsa sanal metodun farklı versiyonları çalıştırılır. Bir başka deyişle, sanal metodun hangi versiyonunun çalıştırılacağını belirleyen referansta bulunan nesnenin tipidir (referansın tipi değildir). Dolayısıyla, eğer bir

temel sınıf bir sanal metot içeriyorsa ve bu temel sınıfın başka sınıflar türetilmişse, bu durumda bir temel sınıf referansı aracılığıyla farklı nesne tiplerine referansta bulunulurken sanal metodun farklı versiyonları çalıştırılır.

Bir temel sınıf içinde bir metodu sanal olarak deklare etmek için metodun deklarasyonunun önüne **virtual** anahtar kelimesini yerleştirirsiniz. Bir sanal metot bir türetilmiş sınıf tarafından yeniden tanımlanırken **override** nitelikini kullanılır. Bu nedenle, bir sanal metodu bir türetilmiş sınıf içinde yeniden tanımlama işlemine *metodu devre dışı bırakma* (*method overriding*) denir. Bir metodu devre dışı bırakırken devre dışı bırakılan metodun tip imzası, devre dışı bırakılmakta olan sanal metot ile aynı olmalıdır. Ayrıca, sanal metodlar **static** veya **abstract** (bu bölüm içinde ele alınacaktır) olarak belirtilemezler.

Metotları devre dışı bırakma özelliği C#'ın en güçlü kavramlarından birinin temelini oluşturmaktadır: *dinamik metot dağıtım* (*dynamic method dispatch*). Dinamik metot dağıtım, devre dışı bırakılan bir fonksiyona yapılan çağrıının derleme zamanında değil, çalışma zamanında çözüldüğü bir mekanizmadır. Dinamik metot dağıtım önemlidir; çünkü dinamik metot dağıtım; C#'ın programın çalışması sırasında çok biçimliliği nasıl uyguladığının bir göstergesidir.

İşte size, sanal metodları ve devre dışı bırakma özelliğini gösteren bir örnek:

```
// Sanal bir metot gösterir.

using System;

class Base {
    // Temel bir sınıf içinde sanal bir metot oluşturur.
    public virtual void who() {
        Console.WriteLine("who() in Base");
    }
}

class Derived1 : Base {
    // who() metodunu türetilmiş sınıfından devre dışı bırak.
    public override void who() {
        Console.WriteLine("who() in Derived1");
    }
}

class Derived2 : Base {
    // who() metodunu bir başka türetilmiş sınıfından devre dışı bırak.
    public override void who() {
        Console.WriteLine("who() in Derived2");
    }
}

class OverrideDemo {
    public static void Main() {
        Base baseOb = new Base();
        Derived1 dOb1 = new Derived1();
        Derived2 dOb2 = new Derived2();
```

```

        Base baseRef; // bir temel sınıf referansı

        baseRef = baseOb;
        baseRef.who();

        baseRef = dOb1;
        baseRef.who();

        baseRef = dOb2;
        baseRef.who();
    }
}

```

Programın çıktısı aşağıdaki gibidir:

```

who() in Base
who() in Derived1
who() in Derived2

```

Bu program **Base** adında bir temel sınıf ve **Derived1** ve **Derived2** adında iki türetilmiş sınıf oluşturur. **Base**, **who()** adında bir metot deklare eder ve türetilmiş sınıflar bu metodu devre dışı bırakır. **Main()** metodu içinde **Base**, **Derived1** ve **Derived2** tipinde nesneler deklare edilir. Ayrıca, **baseRef** adında **Base** tipinde bir referans da deklare edilmektedir. Program daha sonra her tipten nesneye yapılan referansları **baseRef**'e atar ve **who()**'yu çağrırmak için bu referansları kullanır. Çıktıdan görüldüğü gibi, **who()**'nun hangi versiyonunun çalıştırılacağı çağrı sırasında kendisine referansta bulunan nesnenin tipi ile belirlenir. Bu versiyon belirleme işleminde **baseRef**'in sınıf tipi baz alınmaz.

Sanal metodları devre dışı bırakmak gereklidir. Eğer bir türetilmiş sınıf, mevcut bir sanal metodun kendisine ait bir versiyonunu sağlamıyorsa, bu durumda temel sınıf içindeki metod kullanılır. Örneğin:

```

/* Bir sanal metod devre dışı bırakılmayınca
   temel sınıf içindeki metod kullanılır. */

using System;

class Base {
    // Temel sınıf içinde sanal metod oluştur.
    public virtual void who() {
        Console.WriteLine("who() in Base");
    }
}

class Derived1 : Base {
    // who()'yu turetilmis sınıf içinden devre dışı bırak.
    public override void who() {
        Console.WriteLine("who() in Derived1");
    }
}

class Derived2 : Base {

```

```

    // Bu sinif who()'yu devre disi bırakmaz.
}

class NoOverrideDemo {
    public static void Main() {
        Base baseOb = new Base();
        Derived1 dOb1 = new Derived1();
        Derived2 dOb2 = new Derived2();

        Base baseRef; // bir temel sinif referansi

        baseRef = baseOb;
        baseRef.who();

        baseRef = dOb1;
        baseRef.who();

        baseRef = dOb2;
        baseRef.who(); // Base'deki who()'yu cagir
    }
}

```

Bu programdan ekle edilen çıktı aşağıda gösterilmiştir:

```

who() in Base
who() in Derived1
who() in Base

```

Bu örnekte **Derived2**, **who()**'yu devre dışı bırakmaz. Böylece **who()**, **Derived2** nesnesi üzerinde çağrıldığı zaman **Base** içindeki **who()** çalıştırılır.

Çok katmanlı bir hiyerarşi söz konusu olduğunda, eğer bir türetilmiş sınıf bir sanal metodu devre dışı bırakmıyorsa, bu durumda hiyerarşinin yukarılarına doğru ilerlenir ve metodun ilk kez devre dışı bırakıldığı yerdeki metot çalıştırılır. Örneğin;

```

/* Cok katmanli bir hiyerarside,
hiyerarsinin yukarilarina dogru ilerlerken
devre disi bırakilan ilk sanal metot
calistirilacak olan metottur. */

using System;

class Base {
    // Temel sinif icinde bir sanal metot olustur.
    public virtual void who() {
        Console.WriteLine("who() in Base");
    }
}

class Derived1 : Base {
    // who()'yu turetilmis sinif icinden devre disi birak.
    public override void who() {
        Console.WriteLine("who() in Derived1");
    }
}

```

```

    }

    class Derived2 : Derived1 {
        // Bu sınıf da who()'yu devre dışı bırakmıyor.
    }

    class Derived3 : Derived2 {
        // Bu sınıf who()'yu devre dışı bırakmıyor.
    }

    class NoOverrideDemo2 {
        public static void Main() {
            Derived3 dOb = new Derived3();

            Base baseRef; // bir temel sınıf referansı

            baseRef = dOb;
            baseRef.who(); // Derived1'in içindeki who()'yu çağır
        }
    }
}

```

Çıktı şu şekildedir:

```
who() in Derived1
```

Bu örnekte **Base** kalıtım yoluyla **Derived1**'e, **Derived1** kalıtım yoluyla **Derived2**'ye, **Derived2** ise kalıtım yoluyla **Derived3**'e aktarılır. Çıktının da doğruladığı gibi **who()**, **Derived3** veya **Derived2** tarafından devre dışı bırakılmadığı için **who()**'nun çalıştırılan versiyonu **Derived1** içinde **who()**'yu devre dışı bırakan versiyondur, çünkü **who()**'nun rastlanan ilk versiyonu budur.

Bir diğer husus: Özellikler de **virtual** anahtar kelimesi ile değiştirilebilir ve **override** kullanılarak devre dışı bırakılabilirler.

## Devre Dışı Bırakılan Metotlara Neden Gerek Var?

Devre dışı bırakılan metotlar, C#'ın çalışma zamanında gerekli olan çok biçimliliği desteklemesine olanak tanır. Çok biçimlilik nesne yönelimli programlama için gereklidir, çünkü çok biçimlilik bir yandan bir genel sınıfın tüm türevlerinde ortak olacak metotları belirtmeye olanak tanırken, öte yandan türetilmiş sınıflara da bu metotların bir kısmının ya da tümünün spesifik uygulamalarını tanımlamalarına imkan verir. Devre dışı bırakılan metotlar, C#'ın “tek arayüz, çok metot” çok biçimlilik özelliğini uygulamanın bir başka yoludur.

Çok biçimliliği başarıyla uygulamanın anahtarı bir ölçüde, temel ve türetilmiş sınıfların hiyerarşî içinde daha az uzmanlık alanından daha yoğun uzmanlık alanına doğru ilerlediğini kavramaktır. Doğru kullanıldığında bir temel sınıf, türetilmiş sınıfın direkt olarak kullanabileceğî öğelerin tümünü sağlar. Ayrıca, türetilmiş sınıfın kendi başına uygulaması gereken metotları da tanımlar. Bu, türetilmiş sınıfın kendi metotlarını tanımlama esnekliğini

sunarken, aynı zamanda tutarlı bir arayüzü de mecbur kılmayı sürdürür. Böylece, kalıtımı devre dışı bırakılan metodlarla birleştirerek bir temel sınıf, tüm türetilmiş sınıfları tarafından kullanılacak olan metodların genel yapısını tanımlayabilir.

## Sanal Metotları Uygulamak

Sanal metodların gücünü daha iyi kavramak için bu metodları **TwoDShape** sınıfına uygulayacağız. Önceki örneklerde, **TwoDShape**'ten türetilen her sınıf, **area()** adında bir metod tanımlamaktaydı. Bu kullanım bize bir fikir verir: **area()**'yı **TwoDShape** sınıfının bir sanal metod yapmak çok daha iyi olabilir. Bu sayede, türetilmiş sınıfların her biri **area()**'yı devre dışı bırakabilir ve sınıf içinde belirtilen şeklin alanının nasıl hesaplandığını da tanımlayabilir. Aşağıdaki program bunu gerçekleştirir. Kolaylık sağlamak için **TwoDShape**'e ayrıca bir de isim özelliği eklenmektedir. (Bu, sınıfları göstermeyi kolaylaştırır.)

```
// Sanal metodlar ve çokbılıklılık kullanır.

using System;

class TwoDShape {
    double pri_width; // private
    double pri_height; // private
    string pri_name; // private

    // Varsayılan yapılandırıcı.
    public TwoDShape() {
        width = height = 0.0;
        name = "null";
    }

    // Parametreli yapılandırıcı
    public TwoDShape(double w, double h, string n) {
        width = w;
        height = h;
        name = n;
    }

    // Eşit width ve height değerine sahip bir nesne yapılandırı.
    public TwoDShape(double x, string n) {
        width = height = x;
        name = n;
    }

    // Bir nesneden bir başka nesne yapılandırı.
    public TwoDShape(TwoDShape ob) {
        width = ob.width;
        height = ob.height;
        name = ob.name;
    }

    // width, height ve name için özellikler
    public double width {
```

```
        get { return pri_width; }
        set { pri_width = value; }
    }

    public double height {
        get { return pri_height; }
        set { pri_height = value; }
    }

    public string name {
        get { return pri_name; }
        set { pri_name = value; }
    }

    public void showDim() {
        Console.WriteLine("Width and height are " + width +
                           " and " + height);
    }

    public virtual double area() {
        Console.WriteLine("area() must be overridden");
        return 0.0;
    }
}

// Ucgenler icin TwoDShape'ten turetilen bir sinif.
class Triangle : TwoDShape {
    string style; // ozel

    // Varsayılan yapilandirici.
    public Triangle() {
        style = "null";
    }

    // Triangle icin yapilandirici.
    public Triangle(string s, double w, double h) :
        base(w, h, "triangle") {
        style = s;
    }

    // ikizkenar bir ucgen yapilandir.
    public Triangle(double x) : base(x, "triangle") {
        style = "isosceles";
    }

    // Bir nesneden baska bir nesne yapilandir.
    public Triangle(Triangle ob) : base(ob) {
        style = ob.style;
    }

    // Triangle icin area()'yi devre disi birak.
    public override double area() {
        return width * height / 2;
    }
}
```

```
// Üçgenin şeklini göster.
public void showStyle() {
    Console.WriteLine("Triangle is " + style);
}

// Dikdörtgenler için TwoDShape'ten türetilmiş bir sınıf.
class Rectangle : TwoDShape {
    // Rectangle için yapılandırıcı.
    public Rectangle(double w, double h) :
        base(w, h, "rectangle") { }

    // Bir kare yapılandırılmıştır.
    public Rectangle(double x) : base(x, "rectangle") { }

    // Bir nesneden bir başka nesne yapılandırılmıştır.
    public Rectangle(Rectangle ob) : base(ob) { }

    // Dikdörtgen bir kareyse true dondur.
    public bool isSquare() {
        if(width == height) return true;
        return false;
    }

    // Rectangle için area()'yi devre dışı bırakır.
    public override double area() {
        return width * height;
    }
}

class DynShapes {
    public static void Main() {
        TwoDShape[] shapes = new TwoDShape[5];

        shapes[0] = new Triangle("right", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);
        shapes[4] = new TwoDShape(10, 20, "generic");

        for(int i = 0; i < shapes.Length; i++) {
            Console.WriteLine("object is " + shapes[i].name);
            Console.WriteLine("Area is " + shapes[i].area());

            Console.WriteLine();
        }
    }
}
```

Programın çıktısı aşağıdaki gibidir:

```
object is triangle
Area is 48

object is rectangle
```

```
Area is 100  
  
object is rectangle  
Area is 40  
  
object is triangle  
Area is 24.5  
  
object is generic  
area() must be overridden  
Area is 0
```

Gelin, bu programı yakından inceleyelim. Öncelikle, daha önce açıklandığı gibi, `area()` metodu `TwoDShape` içinde `virtual` olarak deklare edilir ve `Triangle` ve `Rectangle` tarafından devre dışı bırakılır. `TwoDShape` içinde `area()`, kullanıcıya bu metodun bir türetilmiş sınıf tarafından devre dışı bırakılması gerektiği bilgisini veren bir yer tutucu görevini görür. `area()`'nın her devre dışı bırakılışında türetilmiş sınıf içine paketlenmiş nesnenin tipine uygun bir uygulama gerçekleştirilir. Yani, örneğin bir elips sınıfını uyguluyor olsaydınız, `area()` elipsin alanını hesaplamak durumunda olurdu.

Yukarıdaki programda bir başka önemli özellik daha mevcuttur. Dikkat ederseniz, `Main()` içinde `shapes`, `TwoDShape` nesnelerinden oluşan bir dizi olarak deklare edilir. Ancak, bu dizinin elemanlarına `Triangle`, `Rectangle` ve `TwoDShape` referansları atanır. Bu geçerli bir kullanımdır, çünkü bir temel sınıf referansı bir türetilmiş sınıf nesnesine referansta bulunabilir. Program daha sonra dizi üzerinde ilerler ve nesnelerin her biriyle ilgili bilgileri ekranda gösterir. Oldukça basit olmasına karşın bu program, hem kalıtımın, hem de metodları devre dışı bırakma özelliğinin gücünü ortaya koyar. Temel sınıf referans değişkeninde saklanan nesnenin tipi, programın çalışması sırasında belirlenir ve buna göre hareket edilir.

Eğer `TwoDShape`'ten bir nesne türetilmişse, bu nesnenin alanı `area()` çağrılarak elde edilebilir. Ne tür bir şekil kullanılsa kullanılsın bu işlemin arayüzü aynıdır.

## Özet Sınıfların Kullanımı

Bazen, şöyle bir temel sınıf oluşturmak istersiniz: O sınıf sadece kendisinden türetilmiş sınıfların genelleştirilmiş bir şekli olmalı, detaylar her bir türetilmiş sınıf tarafından doldurulmalıdır. Bu tür bir sınıf, türetilmiş sınıfların uygulamaları gereken metodların özünü belirler, fakat bu metodlardan biri ya da daha fazlası için kendisi bir uygulama sağlamaz. Temel sınıf, anlamlı bir metod uygulaması oluşturmadığı zaman böyle bir durumla karşılaşılabilir. `TwoDShape`'in yukarıdaki örnekte kullanılan versiyonunda bu tür bir durum söz konusudur. `area()`'nın tanımı yalnızca bir yer belirteci olmaktan ibarettir. `area()`, hiçbir nesne tipinin alanını hesaplayıp, ekranda göstermez.

Kendi sınıf kütüphanelerinizi oluşturdukça göreceksiniz ki, bir metodun kendi temel sınıfı kapsamında anlamlı bir tanımının olmaması yaygın bir kullanım değildir. Bu durumu iki

şekilde kontrol altına alabilirsiniz. Birincisi, önceki örnekte gösterildiği gibi, metodun bir uyarı mesajı vermesi yeterlidir. Bu yöntem belirli durumlarda -örneğin, hata ayıklama sırasında- yararlı olsa da, bu yöntemi kullanmak genellikle uygun değildir. Türetilmiş sınıfın bir anlam taşımıası için türetilmiş sınıf tarafından devre dışı bırakılması gereken metodlara sahip olabilirsiniz. **Triangle** sınıfını ele alın. **area()** tanımlanmazsa bu sınıfın bir anlamı kalmaz. Bu durumda, türetilmiş sınıfın gerçekten gerekli metodların tümünü devre dışı bırakmasını garanti edecek bir çözüme ihtiyacınız olur. Bu probleme C#'ın getirdiği çözüm *özet metot* kullanımızdır.

Özet metot **abstract** tip niteleyicisi kullanılarak oluşturulur. Özet metodun gövdesi yoktur; bu nedenle, özet metodlar temel sınıf tarafından gerçeklenmez. Yani, türetilmiş sınıf, özet sınıfı devre dışı bırakmalıdır, temel sınıf içinde tanımlanmış olan versiyonunu kullanamaz. Büyük olasılıkla siz de tahmin etmişsinizdir; özet metodlar otomatik olarak sanaldır ve **virtual** niteleyicisini kullanmaya gerek yoktur. Aslında, **virtual** ve **abstract** niteleyicilerini birlikte kullanmak hatalıdır.

Özet metodları deklare etmek için aşağıdaki genel yapıyı kullanın:

```
abstract tip isim(parametre-listesi);
```

Gördüğünüz gibi, metod gövdesi mevcut değildir. **abstract** niteleyicisi yalnızca normal metodlarla kullanılabilir. **static** metodlara uygulanamaz. Özellikler de özet olabilir.

Bir veya daha fazla özet metod içeren bir sınıf da özet olarak deklare edilmelidir. Bunun için **class** deklarasyonundan önce **abstract** belirleyicisini kullanmalısınız. Özet bir sınıf komple bir uygulama tanımlamadığından dolayı, özet bir sınıfa ait nesneler mevcut olamaz. Böylece, **new** kullanılarak özet bir sınıfa ait bir nesne oluşturmaya çalışmak derleme sırasında hataya karşılaşmaya neden olur.

Bir özet sınıf kalıtım yoluyla bir türetilmiş sınıfa aktarılınca, söz konusu türetilmiş sınıf temel sınıf içinde yer alan özet metodların tümünü uygulamak zorundadır. Eğer uygulamazsa, türetilmiş sınıf da ayrıca **abstract** olarak belirtilmelidir. Böylece; **abstract** niteliği, özet metodların tümü tamamen uygulanana kadar kalıtım yoluyla aktarılır.

Özet sınıf kullanarak **TwoDShape** sınıfını geliştirebilirsiniz. Tanımlanmamış iki boyutlu bir şeklin alanını ifade etmek için, anlamlı bir kavram mevcut değildir. Bu nedenle, önceki programın aşağıdaki versiyonunda **area()**, **TwoDShape** içinde **abstract** olarak deklare edilmektedir. **TwoDshape** de **abstract** olarak deklare edilir. Bu, kuşkusuz, **TwoDShape**'ten türetilen tüm sınıfların **area()**'yı devre dışı bırakması gerektiği anlamına gelmektedir.

```
// Ozet bir sinif olustur.  
  
using System;  
  
abstract class TwoDShape {  
    double pri_width; // private  
    double pri_height; // private
```

```
string pri_name; // private

// Varsayılan yapılandırıcı.
public TwoDShape() {
    width = height = 0.0;
    name = "null";
}

// Parametreli yapılandırıcı.
public TwoDShape(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
}

// Esit width ve height değerlerine sahip bir nesne yapılandırır.
public TwoDShape(double x, string n) {
    width = height = x;
    name = n;
}

// Bir nesneden başka bir nesne yapılandırır.
public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
}

// width, height ve name için özellikler.
public double width {
    get { return pri_width; }
    set { pri_width = value; }
}

public double height {
    get { return pri_height; }
    set { pri_height = value; }
}

public string name {
    get { return pri_name; }
    set { pri_name = value; }
}

public void showDim() {
    Console.WriteLine("Width and height are " + width +
                      " and " + height);
}

// Artık area() özettir.
public abstract double area();
}

// Üçgenler için TwoDShape'ten türetilen bir sınıf.
class Triangle : TwoDShape {
```

```
string style; // ozel

// Varsayılan yapılandırıcı.
public Triangle() {
    style = "null";
}

// Triangle için yapılandırıcı.
public Triangle(string s, double w, double h) :
    base(w, h, "triangle") {
    style = s;
}

// Bir ikizkenar üçgen yapılandır.
public Triangle(double x) : base(x, "triangle") {
    style = "isosceles";
}

// Bir nesneden başka bir nesne yapılandır.
public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
}

// Triangle için area()'yi devre dışı bırak.
public override double area() {
    return width * height / 2;
}

// Üçgenin şeklini göster.
public void showStyle() {
    Console.WriteLine("Triangle is " + style);
}
}

// Dörtgenler için TwoDShape'ten turetilen bir sınıf.
class Rectangle : TwoDShape {
    // Rectangle için yapılandırıcı.
    public Rectangle(double w, double h) :
        base(w, h, "rectangle") { }

    // Bir kare yapılandır.
    public Rectangle(double x) : base(x, "rectangle") { }

    // Bir nesneden başka bir nesne yapılandır.
    public Rectangle(Rectangle ob) : base(ob) { }

    // Dikdörtgen kare ise true dondur.
    public bool isSquare() {
        if(width == height) return true;
        return false;
    }

    // Rectangle için area()'yi devre dışı bırak.
    public override double area() {
        return width * height;
    }
}
```

```

        }

    }

    class AbsShape {
        public static void Main() {
            TwoDShape[] shapes = new TwoDShape[4];

            shapes[0] = new Triangle("right", 8.0, 12.0);
            shapes[1] = new Rectangle(10);
            shapes[2] = new Rectangle(10, 4);
            shapes[3] = new Triangle(7.0);

            for(int i = 0; i < shapes.Length; i++) {
                Console.WriteLine("object is " + shapes[i].name);
                Console.WriteLine("Area is " + shapes[i].area());

                Console.WriteLine();
            }
        }
    }
}

```

Programdan görüldüğü üzere, tüm türetilmiş sınıfların **area()**'yı devre dışı bırakmaları gerekmektedir (veya **abstract** olarak da deklare edilebilirler). Bunu kendi kendinize ispatlamamanız için **area()**'yı devre dışı bırakmayan bir türetilmiş sınıf oluşturmayı deneyin. Bu durumda derleyici hatası ile karşılaşırınsınız. Elbette, **TwoDShape** tipinde bir nesne referansı oluşturmak hala mümkündür. Programda da böyle yapılmıştır. Ancak, **TwoDShape** tipinde nesne deklare etmek artık mümkün değildir. Bundan dolayı, **Main()**'de **shapes** dizisi 4 elemandan oluşan şekilde kısaltılmıştır ve genel bir **TwoDShape** nesnesi artık oluşturulmaz.

Bir diğer husus: Buna rağmen **TwoDShape**'in içinde **showDim()** metodunun yer aldığına ve **abstract** ile nitelenmediğine dikkat edin. Bir özet sınıf, türetilmiş bir sınıfın “olduğu gibi” kullanmakta serbest olduğu somut metotlar içerebilir. Bu, tamamen kabul edilebilir, hatta, oldukça da yaygın bir kullanımıdır. Yalnızca **abstract** olarak deklare edilen sınıflar türetilmiş sınıflar tarafından devre dışı bırakılmalıdır.

## Kalıtımı Önlemek İçin sealed Kullanmak

Kalıtım her ne kadar güçlü ve kullanışlı olsa da kimi zaman bunu önlemek isteyeceksiniz. Örneğin, özelleştirilmiş bir donanım aygıtinın, söz gelişî bir tıbbi monitörün ilk kullanıma hazırlanması ile ilgili adımları bir araya getiren bir sınıfınız olabilir. Böyle bir durumda, sınıfınızı kullanan kullanıcıların monitörün ilk kullanıma hazırlanma şeklini değiştirebilmelerini istemezsiniz. Bu, aygıtın büyük olasılıkla hatalı olarak hazırlanmasına neden olacaktır. Sebep ne olursa olsun C#'ta **sealed** anahtar kelimesi kullanılarak bir sınıfın kalıtım yoluyla aktarılması kolaylıkla önlenir.

Bir sınıfın kalıtım yoluyla aktarılmasını önlemek için söz konusu sınıfın deklarasyonunun önüne **sealed** anahtar kelimesini yerleştirmelisiniz. Tahmin edebileceğiniz gibi, bir sınıfı hem

**abstract** hem de **sealed** olarak deklare etmek kurallara aykırıdır; çünkü özet sınıf kendi başına tam değildir, tam olarak uygulanabilmek için kendisinden türetilen sınıflara güvenir.

İşte, bir **sealed** sınıf örneği:

```
sealed class A {  
    // ...  
}  
  
// Asagidaki sınıf kurallara aykırıdır.  
class B : A { // HATA! A sınıfından bir sınıf turetemezsiniz.  
    // ...  
}
```

Açıklamalarda da ifade edildiği gibi, **A**'nın kalıtım yoluyla **B**'ye aktarılması kurallara aykırıdır, çünkü **A**, **sealed** olarak deklare edilmiştir.

## object Sınıfı

C#'ta **object** olarak adlandırılan özel bir sınıf tanımlıdır. **object**, tüm diğer sınıflar ve tüm diğer tipler (değer tipleri de dahil olmak üzere) için kapalı bir temel sınıf görevini görür. Bir başka deyişle, tüm diğer tipler **object**'ten türetilir. Bu, **object** tipindeki bir referans değişkeninin diğer herhangi tipteki bir nesneye referansta bulunabilmesi anlamına gelir. Ayrıca, diziler de sınıf olarak gerçeklendikleri için, **object** tipindeki bir değişken herhangi bir diziye de referansta bulunabilir. Teknik olarak, C#'taki **object** ismi aslında .NET Framework sınıf kütüphanesinin bir parçası olan **System.Object** için verilen bir başka isimdir.

**object** sınıfı Tablo 11.1'de gösterilen metotları tanımlar. Yani, bu metotlar her nesne için kullanıma hazırlıdır.

Bu metotlardan birkaçını biraz daha açıklamak gereklidir. **Equals (object)** metodu, kendisini çağıran nesnenin, ona aktarılan argümanla aynı nesneye referansta bulunup bulunmadığını saptar. (Bir başka ifade ile, iki referansın aynı olup olmadığını kontrol eder.) Bu metodu, kendi oluşturduğunuz sınıflarda devre dışı bırakabilirsiniz. Bu sayede, “eşitliğin” bir sınıfta ne anlama geldiğini kendinize göre tanımlama olanağı bulursunuz. Örneğin, **Equals (object)**'i, iki nesneyi içerik bakımından karşılaştıracak biçimde tanımlayabilirsiniz. **Equals (object, object)** metodu, sonucu hesaplamak için **Equals (object)**'i kullanır.

**GetHashCode ()** metodu, çağıran nesneyle ilişkili “hash” kodunu döndürür. Bu kod, depolanmış nesnelere erişmek için “hashing” yönteminden yararlanan herhangi bir algoritma tarafından kullanılabilir.

Bölüm 9'da bahsedildiği gibi, **==** operatörünü aşırı yüklerseniz, **Equals (object)** ve **GetHashCode ()**'u devre dışı bırakmanız gerekecektir, çünkü çoğu zaman **==** operatörü ile **Equals (object)**'in aynı sonucu vermesini isteyeceksinizdir. **Equals ()** devre dışı bırakıldığında iki metodun uyumlu olabilmesi için **GetHashCode ()**'un da devre dışı bırakılması gereklidir.

## TABLO 11.1: Object Sınıfının Metotları

Metot	Amaç
<code>public virtual bool Equals(object ob)</code>	Metodu çağrıran nesne ile <b>ob</b> tarafından referansta bulunan nesnenin aynı olup olmadığını belirler.
<code>public static bool Equals(object ob1, object ob2)</code>	<b>ob1</b> 'in <b>ob2</b> ile aynı olup olmadığını belirler.
<code>protected Finalize()</code>	Anlamsız verilerin toplanması işleminden önce tüm faaliyetleri sonlandırır. C#'ta <b>Finalize()</b> bir yok edici aracılığıyla erişilir.
<code>public virtual int GetHashCode()</code>	Metodu çağrıran nesne ile ilişkili "hash" kodunu döndürür.
<code>public Type GetType()</code>	Programın çalışması sırasında bir nesnenin tipini elde eder.
<code>protected object MemberwiseClone()</code>	Nesnenin "siğ kopyasını" oluşturur. (Üyeler kopyalanır ama üyelerin referansta bulunduğu nesneler kopyalanmaz.)
<code>public static bool ReferenceEquals(object ob1, object ob2)</code>	<b>ob1</b> ve <b>ob2</b> 'nin aynı olup olmadığını belirler.
<code>public virtual string ToString()</code>	Nesneyi tarif eden bir karakter katarı döndürür.

**ToString()** metodu, üzerinden çağrıldığı nesnenin tanımını içeren bir karakter katarı döndürür. Bu metot aynı zamanda, bir nesne **WriteLine()** ile yazıldığında da otomatik olarak çağrılır. Pek çok sınıf, bu metodu devre dışı bırakır. Bu sayede, bir sınıfı, oluşturdukları nesne tiplerine spesifik olarak uydurma olanağı bulurlar. Örneğin:

```
// ToString()'i tanitir.

using System;

class MyClass {
    static int count = 0;
    int id;

    public MyClass() {
        id = count;
        count++;
    }

    public override string ToString() {
        return "MyClass object #" + id;
    }
}
```

```

class Test {
    public static void Main() {
        MyClass ob1 = new MyClass();
        MyClass ob2 = new MyClass();
        MyClass ob3 = new MyClass();

        Console.WriteLine(ob1);
        Console.WriteLine(ob2);
        Console.WriteLine(ob3);
    }
}

```

Programın sıklısı şu şekildedir:

```

MyClass object #0
MyClass object #1
MyClass object #2

```

## Kutulama ve Kutudan Çıkarma

Daha önce de açıkladığımız gibi; değer tipleri de dahil olmak üzere tüm C# tipleri **object**'ten türetilirler. Bu nedenle, **object** tipine yapılan bir referans, değer tipleri dahil olmak üzere tüm tiplere referansta bulunmak üzere kullanılabilir. Bir **object** referansı bir değer tipine referansta bulunduğuanda *kutulama (boxing)* denilen bir süreç işler. Kutulama, bir değer tipinin değerinin bir nesne örneğinde depolanmasına neden olur. Dolayısıyla, bir değer tipi bir nesnenin içine “kutulanır”. Bu nesne, diğer herhangi bir nesne gibi kullanılabilir. Kutulama her zaman otomatik olarak yapılır. Sizin tek yapmanız gereken bir **object** referansına bir değer atamaktır. Gerisini C# halleder.

*Kutudan çıkışma (unboxing)* ise bir nesneden bir değer okuma işlemidir. Bu işlem, **object** referansından istenen değer tipine bir tip ataması kullanılarak gerçekleştirilir.

Kutulama ve kutudan çıkışma işlemlerini gösteren bir örnek aşağıda verilmiştir.

```

// Basit bir kutulama/kutudan cikarma ornegi.

using System;

class BoxingDemo {
    public static void Main() {
        int x;
        object obj;

        x = 10;
        obj = x; // x'i bir nesneye kutula

        int y = (int)obj; // obj'i kutudan cikararak bir int'e ata
        Console.WriteLine(y);
    }
}

```

Bu program, **10** değerini ekrana getirir. Dikkat ederseniz, **x** değeri sadece onu bir **object** referansı olan **obj**'a atayarak kutulanmıştır, **obj**'dan **int**'e tip ataması kullanılarak **obj**'daki tamsayı değer geri alınmıştır.

Gelin şimdi de kutulamanın bir diğer ve daha ilginç örneğine bakalım. Bu örnekte **int**, bir **object** parametresi kullanan **sqr()** metoduna argüman olarak aktarılmaktadır.

```
//Kutulama, deger aktarilirken de meydana gelir.

using System;

class BoxingDemo {
    public static void Main() {
        int x;

        x = 10;
        Console.WriteLine("Here is x: " + x);

        // X, sqr()'e aktarildiginda otomatik olarak kutulanir.
        x = BoxingDemo.sqr(x);
        Console.WriteLine("Here is x squared: " + x);
    }

    static int sqr(object o) {
        return (int)o * (int)o;
    }
}
```

Programın çıktısı aşağıdadır:

```
Here is x: 10
Here is x squared: 100
```

Burada **x**'in değeri **sqr()**'e aktarıldığından otomatik olarak kutulanmaktadır.

Kutulama ve kutudan çıkarma işlemleri, C#'ın tip sisteminin tümüyle tek standarda uyumsunu sağlar. Tüm tipler **object**'ten türetilir. Herhangi bir tipe yapılacak bir referansa bir **object** referansı atanabilir. Kutulama/kutudan çıkarma, bu işin değer tipleriyle ilgili kısmını otomatik olarak halleder. Daha da ötesi, tüm tipler **object**'ten türetildikleri için, tiplerin tümü **object**'in metodlarına erişebilir. Örneğin, aşağıda verdiğimiz biraz şartsızı programa bakalım:

```
// Kutulama, metodlari bir deger uzerinden cagirmaya olanak verir!

using System;

class MethOnValue {
    public static void Main() {
        Console.WriteLine(10.ToString());
    }
}
```

Bu program, **10** değerini ekrana getirir, çünkü **ToString()** metodu, üzerinden çağrıldığı nesnenin karakter katarı cinsinden ifade edilmiş biçimini döndürür. Bu durumda **10**'un karakter cinsinden ifadesi **10**'dur.

## object Genel Bir Veri Tipi midir?

**object** tüm diğer tiplerin temel sınıfı olduğu ve kutulama/kutudan çıkarma işlemleri değer tipleri için otomatik olarak işlediği için, **object**'i genel bir veri tipi olarak kullanmak mümkündür. Örnek olarak, bir **object** dizisi oluşturan ve çeşitli diğer tiplerden verileri bu dizinin elemanlarına atayan aşağıdaki programı inceleyelim:

```
// Genel bir dizi olusturmak icin object' in kullanimi.

using System;

class GenericDemo {
    public static void Main() {
        object[] ga = new object[10];

        // int'leri depola
        for(int i = 0; i < 3; i++)
            ga[i] = i;

        // double'lari depola
        for(int i = 3; i < 6; i++)
            ga[i] = (double) i / 2;

        // iki string, bir bool bir de char depola
        ga[6] = "Generic Array";
        ga[7] = true;
        ga[8] = 'X';
        ga[9] = "end";

        for(int i = 0; i < ga.Length; i++)
            Console.WriteLine("ga[" + i + "]: " + ga[i] + " ");
    }
}
```

Programın çıktısı şöyledir:

```
ga[0]: 0
ga[1]: 1
ga[2]: 2
ga[3]: 1.5
ga[4]: 2
ga[5]: 2.5
ga[6]: Generic Array
ga[7]: True
ga[8]: X
ga[9]: end
```

Bu programdan görüldüğü üzere, bir **object** referansını herhangi bir veri tipine referansta bulunmak için kullanmak mümkündür. Dolayısıyla, bu programın kullandığı türden bir **object** dizisine herhangi bir veri türünden değer atanabilir. Bu, **object** dizisinin esasen genel bir liste olduğu anlamına gelir. Kavramı genişletirsek, örneğin, **object** referanslarının tutulduğu bir yığın sınıfı oluşturmanın ne kadar kolay olduğu anlaşılır. Bu sayede yığında her türlü veri tipinden eleman tutulabilir.

**object**'in genel tip olması güçlü bir özellikleştir ve bazı durumlarda gayet etkin biçimde kullanılabilir. Ancak **object**'i C#'ın diğer durumlarda güçlü olan tip kontrollerini aşmanın bir yolu olarak görmenz hata olur. Genel olarak, bir **int** tutmanız gerekiyorsa, bir **int** değişkeni kullanın. Karakter katarı depolamak için bir **string** referansından yararlanın. Diğer tipler için de benzer şekilde davranışın. **object**'in genel olma niteliğini sadece özel durumlarda kullanın.

# ARAYÜZLER, YAPILAR VE NUMARALANDIRMALAR

Bu bölümde C#'ın en önemli özelliklerinden biri olan arayüzler ele alınmaktadır. *Arayüz*, (*interface*) bir sınıf tarafından uygulanacak olan bir grup metodu tanımlar. Arayüzün kendisi herhangi bir metodu uygulamaz. Yani; arayüz, uygulamayı gerekli kılmadan işlevselligi tarif eden tamamen mantıksal bir özelliktir.

Ayrıca, C#'ın iki veri tipi daha bu bölümde ele alınmaktadır: Yapılar ve numaralandırmalar. *Yapı* (*structure*), sınıfı benzer. Tek fark yapıların referans tipi yerine değer tipi olarak ele alınmasıdır. *Numaralandırmalar* (*enumerations*), isimlendirilmiş tamsayı sabitlerden oluşan listelerdir. Yapılar ve numaralandırmalar, C#'ın programlama ortamının zenginliğine katkıda bulunurlar.

## Arayüzler

Nesne yönelimli programlamada bir sınıfın ne yapması gerektiğini tanımlamak, fakat bunu nasıl yapacağını tanımlamamak kimi zaman yararlı olabilir. Bunun bir örneğini görmüştünüz: Özet metot. Özet metot, bir metot için bir imza tanımlar, ama söz konusu metodun nasıl uygulanacağını tarif etmez. Türetilmiş metot, kendi temel sınıfı tarafından tanımlanan özet metodların her biri için kendi uygulamasını sağlamalıdır. Böylece; özet metot, metot için gerekli *arayüzü* belirtirken metodun nasıl *uygulanacağını* hakkında bilgi vermez. Özet sınıflar ve metodlar kullanışlı olmakla birlikte, bu kavramı bir adım daha ileriye taşımak mümkündür. C#'ta **interface** anahtar kelimesini kullanarak bir sınıfın arayüzü ile uygulamasını birbirinden tamamen ayırlırsınız.

Arayüzler söz dizimsel olarak özet sınıflara benzerler. Ancak, bir arayüz içindeki metodların hiçbirinin gövdesi yoktur. Yani, bir arayüz hiç bir şekilde bir uygulama sağlamaz. Arayüz, ne yapılması gerektiğini belirtir ama, nasıl yapılacağını belirtmez. Arayüz bir kez tanımlandıktan sonra, herhangi bir sayıda sınıf bunu uygulayabilir. Ayrıca, bir sınıf da herhangi bir sayıda arayüzü uygulayabilir.

Bir sınıf, bir arayüz uygulamak amacıyla, arayüz tarafından tarif edilen metodlar için gövdeler (uygulamalar) sağlamalıdır. Her sınıf kendi uygulaması ile ilgili ayrıntıları belirlemekte özgürdür. Böylece, iki sınıf aynı metot grubunu destekliyor olmalarına rağmen, aynı arayüzü farklı yollardan uygulayabilir. Bu yüzden, arayüz hakkında bilgi sahibi olan kod, her iki sınıfın nesnelerini de kullanabilir, çünkü bu nesneler için arayüz aynıdır. Arayüz kavramını sağlamakla C#, çok biçimliliğin “tek arayüz, birden fazla metot” özelliğinden tam olarak yararlanmanıza olanak tanır.

Arayüzler **interface** anahtar kelimesi kullanılarak deklare edilirler. Arayüz deklarasyonunun sadeleştirilmiş bir versiyonu şu şekildedir:

```
interface isim {
    dönüş-tipi metot-ismi1(param-listesi);
    dönüş-tipi metot-ismi2(param-listesi);
    // ...
    dönüş-tipi metot-ismiN(param-listesi);
}
```

Arayüzün ismi ***isim*** ile belirtilir. Metotlar yalnızca dönüş tipleri ve imzaları kullanılarak deklare edilirler. Bunlar özellikle özet metotlardır. Daha önce açıklandığı gibi, bir **interface** içindeki metotların hiçbirini bir uygulama içermez. Böylece, bir **interface** içeren sınıfların her biri metotların tümünü uygulamalıdır. Bir arayüz içindeki metotlar kapalı olarak **public**'tir; açıkça bir erişim belirleyicisinin kullanılmasına izin verilmez.

İşte bir **interface** örneği. Bu örnekte, bir sayı serisi üreten bir sınıf'a bir arayüz belirtilmektedir.

```
public interface ISeries {
    int getNext(); // seri icindeki bir sonraki sayiyi dondur
    void reset(); // yeniden basla
    void setStart(int x); // baslangic degerini ayarla
}
```

Bu arayüzün adı **Iseries**'tir. **I** öneki gerekli olmasa da, programcıların birçoğu arayüzleri sınıflardan ayırmak için arayüzleri önekli olarak kullanırlar. **Iseries**, **public** olarak deklare edilmektedir, böylece herhangi bir program içindeki herhangi bir sınıf tarafından uygulanabilecektir.

Metot imzalarına ek olarak arayüzler; özellikler, indeksleyiciler ve olaylar için de imza deklare edebilirler. Olaylar Bölüm 15'te anlatılacaktır; bu bölümde yalnızca metotlar, özellikler ve indeksleyicilerle ilgileneceğiz. Arayüzler, veri üyeleri içeremezler. Ayrıca, yapılandırıcı, yok edici veya operatör metotlarını da tanımlayamazlar. Üstelik, üyelerinin hiçbirini **static** olarak deklare edilemez.

## Arayüzleri Uygulamak

Bir **interface** bir kez tanımlandıktan sonra, bir ya da daha fazla sınıf bu arayüzü uygulayabilir. Bir arayüzü uygulamak için sınıf isminden sonra, aynen temel sınıf belirtilir gibi arayüzün ismi belirtilir. Bir arayüz uygulayan bir sınıfın genel şekli aşağıdaki gibidir:

```
class sınıf-ismi : arayüz-ismi {
    // sınıf gövdesi
}
```

Uygulanmakta olan arayüzün ismi, **arayüz-ismi** içinde belirtilmektedir.

Bir sınıf bir arayüzü uygularken arayüzün bütününu uygulamalıdır. Örneğin, hangi kısımların uygulanacağını seçip alamaz.

Sınıflarda birden fazla arayüz uygulanabilir. Birden fazla arayüz uygulamak için, arayüzler virgül ile birbirinden ayrılır. Temel sınıf kalıtım yoluyla bir sınıf'a aktarılabilir ve söz konusu sınıf da bir veya daha fazla arayüz uygulayabilir. Bu durumda, temel sınıfın ismi virgül ile ayrılan listenin en başında yer almmalıdır.

Bir arayüzü uygulayan metotlar, **public** olarak deklare edilmelidir. Bunun nedeni şudur: Bir arayüz içindeki metotlar kapalı olarak **public**'tir; bundan dolayı, bu metotların

uygulamaları da **public** olmalıdır. Ayrıca, uygulamayı gerçekleştirmekte olan metodun tip imzası da arayüzün tanımında belirtilen tip imzası ile birebir eşlenmelidir.

İşte, daha önce gösterilen **ISeries** arayüzünün uygulandığı bir örnek. Bu programda **ByTwos** adında bir sınıf oluşturulmaktadır. **ByTwos**, her biri bir öncekinden iki büyük değere sahip bir sayı serisi üretir.

```
// Iseries uygulanır.
class ByTwos : ISeries {
    int start;
    int val;

    public ByTwos() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = start;
    }
}
```

Gördüğünüz gibi; **ByTwos**, **ISeries** tarafından tanımlanan üç metodu da uygular. Önceden açıklandığı üzere, bu gereklidir. Çünkü bir sınıf, bir arayüzü kısmen uygulayamaz.

İşte, **ByTwos**'u gösteren bir sınıf:

```
// ByTwos arayuzunu goster.

using System;

class SeriesDemo {
    public static void Main() {
        ByTwos ob = new ByTwos();

        for(int i = 0; i < 5; i++)
            Console.WriteLine("Next value is " + ob.getNext());

        Console.WriteLine("\nResetting");
        ob.reset();
        for(int i = 0; i < 5; i++)
            Console.WriteLine("Next value is " + ob.getNext());

        Console.WriteLine("\nStarting at 100");
    }
}
```

```

        ob.setStart(100);
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Next value is " + ob.getNext());
    }
}

```

**SeriesDemo**'yu derlemek için **ISeries**, **ByTwos** ve **SeriesDemo**'yu içeren dosyaları derlemeye dahil etmelisiniz. Derleyici, çalıştırılabilir nihai dosyayı oluşturmak için bu üç dosyayı otomatik olarak derleyecektir. Söz gelişi, bu dosyalara **ISeries.cs**, **ByTwos.cs** ve **SeriesDemo.cs** isimlerini vermişseniz, aşağıdaki komut satırı sorunsuz derlenecektir:

```
csc SeriesDemo.cs ISeries.cs ByTwos.cs
```

Visual Studio IDE'yi kullanıyorsanız, bu üç dosyanın tümünü C# projenizin içine eklemek yeterlidir. Bir diğer husus da şudur: Bu üç sınıfı aynı dosya içine koymak da tamamen geçerli bir yöntemdir.

Bu programın çıktısı aşağıda gösterilmiştir:

```

Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

```

```

Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

```

```

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110

```

Sınıfların kendilerine ait ek üyeleri tanımlamak için arayüzleri uygulamaları hem izin verilen hem de yaygın bir yaklaşımdır. Örneğin, **ByTwos**'un aşağıdaki versiyonu, önceki değeri döndüren **getPrevious()** adında bir metot ekler:

```

// ISeries'i uygular ve getPrevious() metodunu ekler.
class ByTwos : ISeries {
    int start;
    int val;
    int prev;

    public ByTwos() {
        start = 0;
        val = 0;
        prev = -2;
    }

    public int getPrevious() {
        return prev;
    }

    public void setStart(int start) {
        this.start = start;
    }

    public int getNext() {
        return val;
    }

    public void increment() {
        prev = val;
        val += 2;
    }
}

```

```

    }

    public int getNext() {
        prev = val;
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
        prev = start - 2;
    }

    public void setStart(int x) {
        start = x;
        val = start;
        prev = val - 2;
    }

    // ISeries tarafından belirtilmeyen bir metot.
    public int getPrevious() {
        return prev;
    }
}

```

Dikkat ederseniz, **getPrevious()**'ı eklemek, **ISeries** tarafından tanımlanan metodların uygulamalarında bir değişiklik yapmayı gerektirir. Ancak, bu metodların arayüzü değişmeden kaldığı için yapılan değişiklik problem çıkarmaz ve önceden mevcut olan kodun çökmesine neden olmaz. Bu, arayüzlerin sağladığı avantajlardan biridir.

Önceki açıklandığı gibi, bir **interface** herhangi bir sayıda sınıf tarafından uygulanabilir. Örneğin, işte size, bir asal sayı serisi üreten **Primes** adında bir sınıf. Bu örnekteki **ISeries** uygulamasının **ByTwos** tarafından sağlanan uygulamaktan esasen farklı olduğuna dikkat edin.

```

// Bir asal sayı serisi uygulamak için ISeries kullanır.
class Primes : ISeries {
    int start;
    int val;

    public Primes() {
        start = 2;
        val = 2;
    }

    public int getNext() {
        int i, j;
        bool isprime;

        val++;
        for(i = val; i < 1000000; i++) {
            isprime = true;
            for(j = 2; j < (i/j + 1); j++) {

```

```

        if((i%j == 0) {
            isprime = false;
            break;
        }
    }
    if(isprime) {
        val = i;
        break;
    }
}
return val;
}

public void reset() {
    val = start;
}

public void setstart(int x) {
    start = x;
    val = start;
}
}

```

Buradaki kilit nokta şudur: **ByTwos** ve **Primes** tamamen ilgisiz sayı dizileri üretiyor olsa bile, her ikisi de **ISeries**'i uygulamaktadır. Önceden açıklandığı gibi bir arayüz, uygulama hakkında bilgi vermez. Bu yüzden her sınıf, arayüzü kendi kullanımına uygun bir biçimde uygulamakta serbesttir.

## Arayüz Referanslarının Kullanımı

Arayüz tipinde bir referans değişkeni deklare edebildiğinizi öğrenince oldukça şaşırabilirsiniz. Bir başka deyişle, arayüz referans değişkeni oluşturabilirsiniz. Bu tür bir değişken kendi arayüzünyi uygulayan herhangi bir nesneye referansta bulunabilir. Bir arayüz referansı aracılığıyla, bir nesne üzerinden bir metot çağrıdığınızda aslında, söz konusu nesne tarafından uygulanan metot çağrılmış olur. Bu süreç, Bölüm 11'de anlatılan, türetilmiş bir sınıf nesnesine erişmek için bir temel sınıf referansı kullanmaya benzer.

Aşağıdaki örnek bir arayüz referansının kullanımını göstermektedir. Bu örnekte, **ByTwos** ve **Primes**'in her ikisinin de nesneleri üzerinden metot çağrıları yapmak için aynı arayüz referans değişkeni kullanılmaktadır.

```

// Arayuz referanslarini gosterir.

using System;

// Arayuzu tanimla.
public interface ISeries {
    int getNext(); // serideki bir sonraki sayiyi dondur
    void reset(); // yeniden basla
    void setStart(int x); // baslangic degerini ayarla
}

```

```
// Cift sayilar serisi uretmek icin ISeries kullan.
class ByTwos : ISeries {
    int start;
    int val;

    public ByTwos() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = start ;
    }
}

// Asal sayilar serisi uretmek icin ISeries kullan.
class Primes : ISeries {
    int start;
    int val;

    public Primes() {
        start = 2;
        val = 2;
    }

    public int getNext() {
        int i, j;
        bool isprime;

        val++;
        for(i = val; i < 1000000; i++) {
            isprime = true;
            for(j = 2; j < (i/j + 1); j++) {
                if((i%j) == 0) {
                    isprime = false;
                    break;
                }
            }
            if(isprime) {
                val = i;
                break;
            }
        }
        return val;
    }
}
```

```
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = start;
    }
}

class SeriesDemo2 {
    public static void Main() {
        ByTwos twoOb = new ByTwos();
        Primes primeOb = new Primes();
        ISeries ob;

        for(int i = 0; i < 5; i++) {
            ob = twoOb;
            Console.WriteLine("Next ByTwos value is " +
                ob.getNext());
            ob = primeOb;
            Console.WriteLine("Next prime number is " +
                ob.getNext());
        }
    }
}
```

Program çıktısı aşağıda gösterilmiştir:

```
Next ByTwos value is 2
Next prime number is 3
Next ByTwos value is 4
Next prime number is 5
Next ByTwos value is 6
Next prime number is 7
Next ByTwos value is 8
Next prime number is 11
Next ByTwos value is 10
Next prime number is 13
```

**Main()**'de **ob**, **ISeries** arayüzüne bir referans olarak deklare edilmektedir. Bunun anlamı şudur: **ISeries**'i uygulayan herhangi bir nesneye atıfta bulunan referansları saklamak için **ob** kullanılabilir. Bu örnekte **ob**, her ikisi de **ISeries**'i uygulayan ve sırasıyla **ByTwos** ve **Primes** tipinde nesneler olan **twoOb** ve **primeOb**'a referansta bulunmak için kullanılmaktadır.

Bir diğer husus: Bir arayüz referans değişkeni, yalnızca kendi arayüz deklarasyonunda deklare edilen metodlar hakkında bilgi sahibidir. Bu nedenle, söz konusu nesne tarafından desteklenme ihtimali olan diğer değişken ve metodlara erişmek için arayüz referansı kullanılamaz.

## Arayüz Özellikleri

Tıpkı metodlar gibi, özellikler de bir arayüz içinde gövdesiz olarak belirtilebilir. Özellikler genel olarak şu şekilde belirtilir:

```
// arayüz özelliği
tip isim {
    get;
    set;
}
```

Kuşkusuz, salt okunur veya salt yazılmış özellikler için sırasıyla, yalnızca **get** veya **set** mevcut olacaktır.

**ISeries** arayüzüünün ve **ByTwos** sınıfının, seri içindeki bir sonraki elemanı elde etmek ve ayarlamak için bir özellik kullanılarak yeniden yazılmış versiyonları aşağıdaki gibidir:

```
// Bir arayuz içinde bir ozellik kullan.

using System;

public interface ISeries {
    // bir arayuz ozelligi
    int next {
        get; // seri icindeki bir sonraki sayiyi dondur
        set; // bir sonraki sayiyi ayarla
    }
}

// ISeries'i uygula.
class ByTwos : ISeries {
    int val;

    public ByTwos() {
        val = 0;
    }

    // degeri al veya ayarla
    public int next {
        get {
            val += 2;
            return val;
        }
        set {
            val = value;
        }
    }
}

// Bir arayuz ozelligi goster.
class SeriesDemn3 {
    public static void Main() {
        ByTwos ob = new ByTwos();
```

```

// seriye bir ozellik aracılığıyla eriş
for(int i = 0; i < 5; i++)
    Console.WriteLine("Next value is " + ob.next);

Console.WriteLine("\nStarting at 21");
ob.next = 21;
for(int i = 0; i < 5; i++)
    Console.WriteLine("Next value is " + ob.next);
}
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

```

```

Starting at 21
Next value is 23
Next value is 25
Next value is 27
Next value is 29
Next value is 31

```

## Arayüz İndeksleyiciler

Bir arayüz, bir indeksleyici belirtebilir. Bir arayüz içinde deklare edilen bir indeksleyici genel olarak şu şekildedir:

```

// arayüz indeksleyicisi
eleman-tipi this[int indeks] {
    get;
    set;
}

```

Önceki gibi, salt okunur veya salt yazılar indeksleyiciler için yalnızca **get** veya **set** mevcut olacaktır.

İşte, **ISeries**'in bir başka versiyonu daha. **ISeries**'e bu kez, seri içindeki **i**'nci elemanı döndüren salt okunur bir indeksleyici eklenmektedir.

```

// Bir arayuze bir indeksleyici ekle.

using System;

public interface ISeries {
    // arayuz ozelligi
    int next {
        get; // seri icindeki bir sonraki sayiyi dondur
        set; // bir sonraki sayiyi ayarla
    }
}

```

```
    }

    // arayuz indeksleyicisi
    int this[int index] {
        get; // seri icinde belirtilen sayiyi dondur
    }
}

// ISeries'i uygula.
class ByTwos : ISeries {
    int val;

    public ByTwos() {
        val = 0;
    }

    // ozellik kullanarak degeri al ya da ayarla.
    public int next {
        get {
            val += 2;
            return val;
        }
        set {
            val = value;
        }
    }
}

// indeks kullanarak bir deger al
public int this[int index] {
    get {
        val = 0;
        for(int i = 0; i < index; i++)
            val += 2;
        return val;
    }
}
}

// Bir arayuz indeksleyicisi goster.
class SeriesDemo4 {
    public static void Main() {
        ByTwos ob = new ByTwos();

        // ozellik aracılığıyla seriye eris
        for(int i = 0; i < 5; i++)
            Console.WriteLine("Next value is " + ob.next);

        Console.WriteLine("\nStarting at 21");
        ob.next = 21;
        for(int i = 0; i < 5; i++)
            Console.WriteLine("Next value is " + ob.next);

        Console.WriteLine("\nResetting to 0");
        ob.next = 0;
```

```

    // indeksleyici aracılığıyla seride eriş.
    for(int i = 0; i < 5; i++)
        Console.WriteLine("Next value is " + ob[i]);
    }
}

```

Bu programdan elde edilen çıktı aşağıdaki gibidir:

```

Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

```

```

Starting at 21
Next value is 23
Next value is 25
Next value is 27
Next value is 29
Next value is 31

```

```

Resetting to 0
Next value is 0
Next value is 2
Next value is 4
Next value is 6
Next value is 8

```

## Arayüzler Kalıtım Yoluyla Aktarılabilir

Bir arayüz kalıtım yoluyla bir başkasına aktarılabilir. Söz dizimi kalıtımın aktarıldığı sınıflar için aynıdır. Bir arayüz kalıtım yoluyla bir başka arayüze aktarılıyorsa, bu arayüzü uygulayan bir sınıf, arayüz kalıtım zinciri içinde tanımlanan tüm üyeler için gerekli uygulamaları da sağlamalıdır. Aşağıda bunun bir örneği yer alıyor:

```

// Bir arayuz kalitim yoluyla bir baskasina aktarilabilir.

using System;

public interface A {
    void meth1();
    void meth2();
}

// B artık meth1() ve meth2()'yi icerir .. meth3()'u ekler.
public interface B : A {
    void meth3();
}

// Bu sinif, A ve B'nin tumunu uygulamalidir.
class MyClass : B {
    public void meth1() {
        Console.WriteLine("Implement meth1().");
    }
}

```

```

    }

    public void meth2() {
        Console.WriteLine("Implement meth2().");
    }

    public void meth3() {
        Console.WriteLine("Implement meth3().");
    }
}

class IFExtend {
    public static void Main() {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}

```

Deneysel amaçla **MyClass**'ın içindeki **meth1()** ile ilgili uygulamayı kaldırmayı denemek isteyebilirsiniz. Fakat bu, derleme hatasına neden olacaktır. Önceden de bildirildiği gibi, herhangi bir arayüzü uygulamakta olan bir sınıf, diğer arayızlarından kalıtım yoluyla aktarılanlar da dahil olmak üzere, söz konusu arayüz tarafından tanımlanan metodların tümünü uygulamalıdır.

## Arayüz Kalıtımı İle Birlikte İsim Gizleme

Kalıtım yoluyla bir arayüz bir diğerine aktarılırken, türetilmiş arayüz içinde temel arayüz tarafından tanımlanmış bir üyesi gizleyen bir başka üye deklare etmek mümkündür. Bu durum, türetilmiş arayüz içindeki üye, temel arayüz içindekiyle aynı imzaya sahip olduğu zaman söz konusu olur. Türetilmiş arayüz içindeki üyesi **new** kullanarak değiştirmediniz sürece bu durum bir uyarı mesajına neden olacaktır.

## Açık Uygulamalar

Bir arayüze ait bir üyesi uygularken üyenin ismini arayüzün ismi ile birlikte tam olarak belirtmek mümkündür. Bu şekilde davranışın bir *açık arayüz üye uygulamasına* (explicit interface member implementation) ya da kısaca *açık uygulamaya* (explicit implementation) neden olur. Örneğin, aşağıdaki kod parçasının verildiğini varsayıarak:

```

interface IMyIF {
    int myMeth(int x);
}

```

aşağıda gösterilen **IMyIF**'i uygulamak kurallara uygundur:

```

class MyClass : IMyIF {
    int IMyIF.myMeth(int x) {

```

```

        return x / 3;
    }
}

```

Gördüğünüz gibi, **IMyIF**'in **myMeth()** üyesi uygulanırken üyenin tam ismi, üyenin arayüzünün ismi ile birlikte belirtilir.

Bir arayüz üyesi için açık bir uygulama oluşturmak için iki sebebiniz olabilir. Birincisi; bir metodu, tam olarak belirtilmiş bir isim kullanarak uyguladığınızda; sınıfın dışında kalan kod tarafından açıkça görülmeyen, özel bir uygulama olarak nitelendirebileceğimiz bir uygulama sağlıyorsunuz demektir. İkincisi; bir sınıfın, içinde isimleri ve tip imzaları aynı olan metodların deklare edildiği iki arayüz uygulaması mümkündür. Isimleri tam olarak belirtmek bu durumdaki belirsizliği ortadan kaldırır. Gelin şimdi her ikisini de birer örnekte görelim.

## Özel Bir Uygulama Geliştirmek

Aşağıdaki program **IEven** adında bir arayüz içermektedir. **IEven**, bir sayının tek ya da çift olup olmadığını belirleyen **isEven()** ve **isOdd()** adında iki metod tanımıştır. Daha sonra **MyClass**, **IEven**'ı uygular. Bunu gerçekleştirdikten sonra ise **isOdd()**'u açık olarak uygular.

```

// Bir arayuz uyesinin acik olarak uygulanmasi.

using System;

interface IEven {
    bool isOdd(int x);
    bool isEven(int x);
}

class MyClass : IEven {
    // acik uygulama
    bool IEven.isOdd(int x) {
        if((x%2) != 0) return true;
        else return false;
    }

    // normal uygulama
    public bool isEven(int x) {
        IEven o = this; // cagiran nesneye referans

        return !o.isOdd(x);
    }
}

class Demo {
    public static void Main() {
        MyClass ob = new MyClass();
        bool result;

        result = ob.isEven(4);
        if(result) Console.WriteLine("4 is even.");
        else Console.WriteLine("3 is odd.");
    }
}

```

```

        // result = ob.isOdd(); // Hata. Metot açıkça ortada değildir!
    }
}

```

`isOdd()` açıkça uygulandığı için `MyClass`'ın dışında kullanılamaz. Bu, `isOdd()`'un uygulanışını etkin biçimde özel kılmaktadır. `MyClass` içinde `isOdd()` yalnızca bir arayüz referansı aracılığıyla erişilebilir. `isEven()`'in uygulanışı sırasında `o` aracılığıyla çağrılmamasının sebebi de budur.

## Belirsizliği Ortadan Kaldırmak İçin Açık Uygulamaların Kullanımı

Şimdi de, iki arayüz uygulaması içeren ve her iki arayüz içinde `meth()` adında bir metot deklare edilen bir örneği inceleyelim. Bu duruma özgü belirsizliği ortadan kaldırmak için bu örnekte açık uygulama kullanılıyor.

```

// Belirsizligi ortadan kaldırmak icin acik uygulama kullan.

using System;

interface IMyIF_A {
    int meth(int x);
}

interface IMyIF_B {
    int meth(int x);
}

// MyClass her iki arayuzu de uygular.
class MyClass : IMyIF_A, IMyIF_B {

    // iki meth()'i de açıkça uygula
    int IMyIF_A.meth(int x) {
        return x + x;
    }

    int IMyIF_B.meth(int x) {
        return x * x;
    }

    // meth()'i bir arayuz referansı aracılığıyla çağır
    public int methA(int x) {
        IMyIF_A a_ob;
        a_ob = this;
        return a_ob.meth(x); // IMyIF_A'yi çağırır
    }

    public int methB(int x) {
        IMyIF_B b_ob;
        b_ob = this;
        return b_ob.meth(x); // IMyIF_B'yi çağırır
    }
}

```

```
        }  
    }  
  
    class FQIFNames {  
        public static void Main() {  
            MyClass ob = new MyClass();  
  
            Console.WriteLine("Calling IMyIF_A.meth(): ");  
            Console.WriteLine(ob.methA(3));  
  
            Console.WriteLine("Calling IMyIF_B.meth(): ");  
            Console.WriteLine(ob.methB(3));  
        }  
    }  

```

Bu programın çıktısı aşağıdaki gibidir:

```
Calling IMyIF_A.meth(): 6  
Calling IMyIF_B.meth(): 9
```

Programa bakarak öncelikle **meth()**'in imzasının hem **IMyIF\_A**'da hem de **IMyIF\_B**'de aynı olduğuna dikkat edin. Böylece, **MyClass** bu iki arayüzü uygularken, her birinin ismini bu işlem sırasında tam olarak belirterek her birini ayrı ayrı açık olarak uygulamalıdır. Açık olarak uygulanan bir metot yalnızca bir arayüz referansı üzerinden çağrılabılır. Bu nedenle; **methA()**, **IMyIF\_A** için; **methB()** ise **IMyIF\_B** için bir referans oluşturmaktadır. Metotlar daha sonra bu referanslar aracılığıyla **meth()**'i çağrırlar; bu sayede, belirsizliği de ortadan kaldırılmış olurlar.

## Bir Arayüz ve Özет Sınıf Arasında Tercihte Bulunmak

C# programlamanın daha zorlayıcı bölümlerinden biri, uygulamayı değil de işlevselligi tarif etmek istediğinizde bir arayüz tanımlayacağınız ya da bir özel sınıf kullanacağınız yeri ve zamanı bilmenizdir. Genel kural şudur: Eğer bir kavramı “nasıl yapar?” sorusuna yanıt aramaya gerek kalmadan “ne yapar?” sorusuna verilecek yanıt ile tam olarak tarif edebiliyorsanız bir arayüz kullanmalısınız. Eğer biraz uygulama ayrıntılarını da eklemeniz gerekiyorsa, bu durumda kavramınızı bir özet sınıf ile simgelemeniz gerekecektir.

## .NET Standart Arayüzleri

.NET Framework, bir C# programının kullanabileceği çok sayıda arayüz tanımlamaktadır. Örneğin, nesneler arasında bir sıralama ilişkisi gerekince nesnelerin karşılaştırılmasını sağlayan **CompareTo()** metodu **System.IComparable** tarafından tanımlanır. Arayüzler ayrıca gruplar halindeki nesneler için çeşitli depolama tipleri (söz gelişi, yiğinlar ve kuyruklar) sağlayan **Collections** sınıflarının da önemli bir bölümünü oluştururlar. Örneğin, **System.Collections.ICollection** tüm koleksiyonlarda ortak olan işlevselligi tanımlar. **System.Collections\_IEnumerator** ise bir koleksiyon içindeki elemanların üzerinde art arda gezinmek için bir yöntem sunmaktadır. Bunlara ve diğer arayzlere Kısım II'de göz atacağız.

## Arayüzlerle İlgili Bir Örnek Çalışma

Daha da ilerlemeden önce bir arayüz kullanan bir başka örnek üzerinde çalışmak yararlı olacaktır. Bu bölümde **ICipher** adında bir arayüz oluşturacağız. **ICipher**'ın içinde şifreleme için kullanılan karakter katarlarını destekleyen metotlar belirtilecektir. Bu amaç için bir arayüz kullanmak akla yatkındır, çünkü şifrelemenin “ne” ile ilgili kısmını “nasıl” ile ilgili kısmından tam olarak ayırmak mümkündür.

**ICipher** arayüzü aşağıda gösterilmiştir:

```
// Bir şifreleme arayuzu.
public interface ICipher {
    string encode(string str);
    string decode(string str);
}
```

**ICipher** arayüzünde iki metot belirtilmektedir: **encode()**, bir karakter katarım şifrelemek için kullanır ve **decode()**, bir karakter katarının şifresini çözmek için kullanılır. Diğer hiçbir ayrıntı belirtilmez. Bu, bu metotları uygulayan sınıfların şifreleme için herhangi bir yöntemi kullanmakta özgür oldukları anlamını taşır. Örneğin; bir sınıf, kullanıcı tarafından belirtilen bir anahtarı baz alarak bir karakter katarını şifreleyebilir, İkinci bir sınıf şifre koruması (password protection) kullanabilir. Üçüncü, bitleri manipüle etme esasına dayanan bir şifreleme yöntemini tercih edebilir; bir diğeri ise basit bir şekilde karakterlerin yerlerini değiştiren bir kod kullanabilir. Şifreleme için hangi yaklaşım kullanılsın, bir karakter katarını şifrelemek ve şifre çözmek için kullanılan arayüz aynıdır. Şifreleme ile ilgili uygulamanın herhangi bir bölümünü belirtmeye gerek yoktur. Bu nedenle, arayüz bunu simgeleyen mantıksal bir seçenektedir.

Şimdi de **ICipher**'ı uygulayan iki sınıfa göz atalım. İlkî, her karakteri bir konum yukarıya kaydırarak bir karakter katarını şifreleyen **SimpleCipher**'dır. Örneğin; **A**, **B** olur; **B**, **C** olur vs. İlkicisi ise; bir anahtar gibi davranışan 16 bitlik bir değer ile her karaktere XOR uygulayarak bir karakter katarını şifreleyen **BitCipher**'dır.

```
/* Her karakteri 1 konum yukarıya kaydırarak
bir mesajı şifreleyen ICipher'in basit bir uygulaması.
Böylece; A, B olur vs. */

class SimpleCipher : ICipher {

    // Orijinal metin verilince şifrelenmiş karakter katarını dondurur.
    public string encode(string str) {
        string ciphertext = "";
        for(int i = 0; i < str.Length; i++)
            ciphertext = ciphertext + (char) (str[i] + 1);

        return ciphertext;
    }
}
```

```

/* Sifrelenmis metin verilince sifresi cozulmus karakter
   katarini dondurur. */
public string decode(string str) {
    string plaintext = "";

    for(int i = 0; i < str.Length; i++)
        plaintext = plaintext + (char) (str[i] - 1);

    return plaintext;
}

/* ICipher'in bu uygulamasi bit manipulasyonu ve anahtar
   kullanir. */
class BitCipher : ICipher {
    ushort key;

    // BitCiphers'i kurarken bir anahtar belirt.
    public BitCipher (ushort k) {
        key = k;
    }

    // Orijinal metin verilince sifrelenmis karakter katarini dondur.
    public string encode(string str) {
        string ciphertext = "";

        for(int i = 0; i < str.Length; i++)
            ciphertext = ciphertext + (char) (&str[i] ^ key);

        return ciphertext;
    }

    /* Sifrelenmis metin verilince sifresi cozulmus karakter
       katarini dondur. */
    public string decode(string str) {
        string plaintext = "";

        for(int i = 0; i < str.Length; i++)
            plaintext = plaintext + (char) (str[i] ^ key);

        return plaintext;
    }
}

```

Gördüğünüz gibi, **SimpleCipher** ve **BitCipher** uygulamada farklılık gösteriyor olsa da, her ikisi de **ICipher** arayüzüni uygulamaktalar. Aşağıdaki program **SimpleCipher** ve **BitCipher**'ı göstermektedir:

```

// ICipher'i gosterir.

using System;

class ICipherDemo {

```

```

public static void Main() {
    ICipher ciphRef;
    BitCipher bit = new BitCipher(27);
    SimpleCipher sc = new SimpleCipher();

    string plain;
    string coded;

    // oncelikle ciphRef, basit sifreleyiciye referansta bulunur
    ciphRef = sc;

    Console.WriteLine("Using simple cipher.");

    plain = "testing";
    coded = ciphRef.encode(plain);
    Console.WriteLine("Cipher text: " + coded);

    plain = ciphRef.decode(coded);
    Console.WriteLine("Plain text: " + plain);

    /* artik, ciphRef'in bit tabanlı sifrelemeye referansta
       bulunmasına izin ver */
    ciphRef = bit;
    Console.WriteLine("\nUsing bitwise cipher.");

    plain = "testing";
    coded = ciphRef.encode(plain);
    Console.WriteLine("Cipher text: " + coded);

    plain = ciphRef.decode(coded);
    Console.WriteLine("Plain text: " + plain);
}
}

```

Çıktı aşağıda gösterilmiştir:

```

Using simple cipher.
Cipher text: uftujoh
Plain text: testing

```

```

Using bitwise cipher.
Cipher text: o-horu|
Plain text: testing

```

Şifreleme arayüzü oluşturmanın bir avantajı, şifreleme ne şekilde uygulanırsa uygulansın, arayüzü uygulayan sınıfların her birinin aynı şekilde erişilmeleridir. Örneğin, aşağıdaki programı ele alın. Bu programda, listelenmemiş telefon numaralarını şifreli biçimde saklayan UnlistedPhone adında bir sınıf yer almaktadır. İsimlerin ve numaraların şifreleri gerektiğinde otomatik olarak çözülür.

```

// ICipher kullanır.

using System;

```

```

// Listelenmemis telefon numaralarini saklamak icin bir sinif.
class UnlistedPhone {
    string pri_name; // isim ozelligini destekler
    string pri_number; // numara ozelligini destekler

    ICipher crypt; // sifreleme nesnesine referans

    public UnlistedPhone(string name, string number, ICipher c)
    {
        crypt = c; // sifrelame nesnesini sakla

        pri_name = crypt.encode(name);
        pri_number = crypt.encode(number);
    }

    public string Name {
        get {
            return crypt.decode(pri_name);
        }
        set {
            pri_name = crypt.encode(value);
        }
    }

    public string Number {
        get {
            return crypt.decode(pri_number);
        }
        set {
            pri_number = crypt.encode(value);
        }
    }
}

// UnlistedPhone'i goster
class UnlistedDemo {
    public static void Main() {
        UnlistedPhone phone1 =
            new UnlistedPhone("Tom", "555-3456", new BitCipher(27));
        UnlistedPhone phone2 =
            new UnlistedPhone("Mary", "555-8391", new BitCipher(9));

        Console.WriteLine("Unlisted number for " + phone1.Name +
                          " is " + phone1.Number);

        Console.WriteLine("Unlisted number for " + phone2.Name +
                          " is " + phone2.Number);
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Unlisted number for Tom is 555-3456
Unlisted number for Mary is 555-8891

```

**UnlistedPhone**'un nasıl uygulandığına yakından bakın. Dikkat ederseniz, **UnlistedPhone** üç alan içerir. İlk ikisi, isim ve telefon numaralarını saklayan özel değişkenlerdir. Üçüncüsü ise **ICipher** nesnesine bir referanstır. Bir **UnlistedPhone** nesnesi kurulunca bu üç referans bu nesneye aktarılır. İlk ikisi, isim ve telefon numarasını tutan karakter katarlarına referanstır. Üçüncüsü, isim ve numarayı şifrelemek için kullanılan şifreleme nesnesine referanstır. Şifreleme nesnesinin referansları **crypt** içinde saklanır. **ICipher** arayüzüne uyguladığı sürece her tipten şifreleme nesnesi kabul edilebilir. Bu örnekte **BitCipher** kullanılmıştır. Böylece; **UnlistedPhone**, **crypt** referansı aracılığıyla bir **BitCipher** nesnesi üzerinde **encode()** ve **decode()** metodlarını çağırabilir.

Şimdi de, **Name** ve **Number** özelliklerinin nasıl çalıştığını dikkat edin. **set** işlemi gerçekleştirken **crypt** nesnesi üzerinde **encode()** çağrılarak isim veya numara otomatik olarak şifrelenir. **get** işlemi gerçekleştirken ise, **decode()** çağrılarak isim ve numara otomatik olarak çözülür. Ne **Name**, ne de **Number**, altta yatan şifreleme metodu hakkında spesifik bir bilgiye sahip değildir. Bunlar yalnızca şifreleme metodunun işlevselligine, şifreleme metodunun arayüzü aracılığıyla erişirler.

Şifreleme arayüzü **ICipher** tarafından standardize edildiği için **UnlistedPhone**'un kodunun içindeki çalışmaların hiçbirini değiştirmeden şifreleme nesnesini değiştirmek mümkündür. Örneğin, aşağıdaki program **UnlistedPhone** nesnelerini yapılandırırken **BitCipher** yerine **SimpleCipher** kullanır. Söz konusu tek değişiklik, **UnlistedPhone** için yapılandırıcıya aktarılan şifreleme nesnesindedir.

```
// Bu versiyon SimpleCipher kullanır.

using System;

class UnlistedDemo {
    public static void Main() {

        // simdi, BitCipher yerine SimpleCipher kullan
        UnlistedPhone phone1 =
            new unlistedPhone("Tom", "555-3456", new SimpleCipher());
        UnlistedPhone phone2 =
            new UnlistedPhone("Mary", "555-8891", new SimpleCipher());

        Console.WriteLine("Unlisted number for " + phone1.Name +
                           " is " + phone1.Number);

        Console.WriteLine("Unlisted number for " + phone2.Name +
                           " is " + phone2.Number);
    }
}
```

Bu programın da gösterdiği gibi, **BitCipher** ve **SimpleCipher**'ın her ikisi de **ICipher**'i uyguladıkları için **UnlistedPhone** nesnelerini kurarken ikisinden biri kullanılabilir.

Son bir husus: **UnlistedPhone**'un uygulaması, bir arayüz referansı aracılığıyla arayüzleri uygulayan nesnelere erişmenin gücünü de göstermektedir. Şifreleme nesnesine **ICipher** referans değişkeni aracılığıyla referansta bulunulduğu için **ICipher** arayüzünyi uygulayan herhangi bir nesne bu amaçla kullanılabilir. Bu durum, **UnlistedPhone**'un kodunun hiçbir bölümünün değiştirilmesine neden olmadan spesifik şifreleme uygulamasının problemsiz ve güvenli olarak değiştirilmesini mümkün kılar. Eğer **UnlistedPhone** bunun yerine belirli tipte bir şifreleme nesnesini **crypt** tipindeki veriler için donanımsal olarak kodlamış olsayıdı, söz gelişî **BitCipher** gibi, bu durumda farklı bir şifreleme yöntemi istendiğinde **UnlistedPhone**'un kodunun değiştirilmesi gerekecekti.

## Yapılar

Bildiğiniz gibi sınıflar, referans tipidir. Bunun anlamı şudur: Sınıf nesneleri bir referans aracılığıyla erişilirler. Bu durum, doğrudan erişilen değer tiplerinden farklılık gösterir. Yine de, kimi zaman bir nesnenin, değer tiplerine erişir şekilde doğrudan erişilebilir olması yararlı olacaktır. Bunun nedenlerinden biri verimliliktir. Sınıf nesnelerine bir referans aracılığıyla erişmek her erişime ilave yük bindirir. Ayrıca bellek alanı da harcar. Çok küçük nesneler için bu ekstra alan önemli olabilir. Bu sorunları dile getirmek için C#, yapıları önermektedir. *Yapı* (*structure*), sınıf ile aynıdır. Tek fark, yapının bir referans tipi yerine bir değer tipinde olmasıdır.

Yapılar **struct** anahtar kelimesi kullanılarak deklare edilir ve söz dizimsel olarak sınıflara benzer. Bir **struct** genel olarak şu şekildedir;

```
struct isim : arayüzler {  
    // üye deklarasyonları  
}
```

Yapının ismi **isim** ile belirtilir.

Yapılar, diğer yapıları veya sınıfları kalıtım yoluyla aktaramaz ya da diğer yapılar veya sınıflar için temel sınıf olarak kullanılamazlar. (Diğer tüm C# tiplerinde olduğu gibi yapılar da kuşkusuz **object**'ı kalıtım yoluyla aktarırlar.) Bununla birlikte, bir yapı bir ya da daha fazla arayüzü uygulayabilir. Bu arayüzler, yapının isminden sonra virgül ile ayrılmış bir liste halinde belirtilir. Tıpkı sınıflar gibi, yapı üyeleri de metot, alan, indeksleyici, özellik, operatör metodlarını ve olayları kapsar. Yapılar ayrıca yapılandırmacıları da tanımlar, fakat yok ediciler, yapılarda tanımlanmaz. Ancak, bir yapı için varsayılan (parametresiz) yapılandırmacı tanımlayamazsınız. Bunun nedeni, varsayılan yapılandırmacının tüm yapılar için otomatik olarak tanımlı olmasıdır; bu varsayılan yapılandırmacı değiştirilemez. Yapılar kalıtımı desteklemedikleri için yapı üyeleri **abstract**, **virtual** ya da **protected** olarak belirtilemezler.

Bir yapı nesnesi, bir sınıf nesnesi ile aynı şekilde **new** kullanılarak oluşturulabilir; fakat bu gerekli değildir. **new** kullanılırca belirtilen yapılandırmacı çağrırlır. **new** kullanılmiyorsa nesne bu durumda da oluşturulur fakat nesneye ilk değer atanmaz. Böylece, ilk değer atama işlemlerini elle gerçekleştirmeniz gerekecektir.

İşte bir örnek, bu örnekte bir kitap hakkında bilgileri tutmak için bir yapı kullanılmaktadır:

```
// Bir yapı göster.

using System;

// Bir yapı tanımla.
struct Book {
    public string author;
    public string title;
    public int copyright;

    public Book(string a, string t, int c) {
        author = a;
        copyright = c;
    }
}

// Book yapısını göster.
class StructDemo {
    public static void Main() {
        Book book1 = new Book("Saim Mehmet Ozturk",
                              "C# A Beginner's Guide", 2001);
        // açık yapılandırıcı
        Book book2 = new Book(); // varsayılan yapılandırıcı
        Book book3; // yapılandırıcı yok

        Console.WriteLine(book1.title + " by " + book1.author +
                          ", (c) " + book1.copyright);
        Console.WriteLine();

        if(book2.title == null)
            Console.WriteLine("book2.title is null.");
        // şimdi book2'ye biraz bilgi ver
        book2.title = "Brave New World";
        book2.author = "Aldous Huxley";
        book2.copyright = 1932;
        Console.Write("book2 now contains: ");
        Console.WriteLine(book2.title + " by " + book2.author +
                          ", (c) " + book2.copyright);

        Console.WriteLine();

        // Console.WriteLine(book3.title); // hata, önce ilk değer atamalı
        book3.title = "Red Storm Rising";

        Console.WriteLine(book3.title); // şimdi oldu
    }
}
```

Bu programın çıktısı aşağıdaki gibidir:

```
C# A Beginner's Guide by Saim Mehmet Ozturk, (c) 2001
book2.title is null.
```

book2 now contains: Brave New World by Aldous Huxley, (c) 1932

Red Storm Rising

Programdan görüldüğü gibi, bir yapıya iki şekilde ilk değer atanabilir: Bir yapılandıracı çağrı makamıyla **new** kullanılır ya da basitçe bir nesne deklare ederek bu işlem gerçekleştirilebilir. **new** kullanılırsa yapının alanlarına ilk değerleri atanacaktır. Bunun için, tüm alanlara ilk değer olarak, alanların varsayılan değerlerini atayan bir varsayılan yapılandıracı kullanılabilir ya da kullanıcı tarafından tanımlanan bir yapılandıracı da kullanılabilir. **book3**'te söz konusu olduğu gibi eğer **new** kullanılmazsa, bu durumda nesneye ilk değer atanmaz; nesnenin alanları nesnenin kullanımından önce ayarlanmalıdır.

Bir yapıyı bir diğerine atarken söz konusu nesnenin kopyası çıkartılır. Bu, yapıları sınıflardan ayıran önemli bir özelliklektir. Elinizdeki kitabın daha önceki sayfalarında açıkladığı gibi, bir sınıf referansını bir diğerine atarken atama operatörünün sol tarafında bulunan referansın referansta bulunduğu nesneyi değiştirmeniz yeterlidir. Bir yapı değişkenini bir başkasına atarken ise atama operatörünün sağındaki nesnenin kopyasını çıkartırsınız. Örneğin, aşağıdaki programı ele alın:

```
// Bir struct'i kopyala.

using System;

// Bir yapı tanımla.
struct MyStruct {
    public int x;
}

// Yapının atanısını göster.
class StructAssignment {
    public static void Main() {
        MyStruct a;
        MyStruct b;

        a.x = 10;
        b.x = 20;

        Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
        a = b;
        b.x = 30;

        Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
a.x 10, b.x 20
a.x 20, b.x 30
```

Cıktıdan da görüldüğü üzere, aşağıdaki atama işleminden sonra birer yapı değişkeni olan **a** ve **b** halen ayrı ve farklıdır:

```
a = b;
```

Yani; **a**, **b**'nin değerinin bir kopyasına referansta bulunmaktan başka hiçbir şekilde **b** ile ilişkili değildir ya da **b**'ye başka hiçbir şekilde referansta bulunamaz, **a** ve **b** sınıf referansları olmuş olsalardı, böyle bir durum söz konusu olmayacağı. Örneğin, yukarıdaki programın sınıflı versiyonu şu şekildedir:

```
// Bir sınıf kopyala.

using System;

// Bir yapı tanımla.
class MyClass {
    public int x;
}

// Şimdi bir sınıf nesnesine atama yapılışını göster.
class ClassAssignment {
    public static void Main() {
        MyClass a = new MyClass();
        MyClass b = new MyClass();

        a.x = 10;
        b.x = 20;

        Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

        a = b;
        b.x = 30;

        Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
    }
}
```

Bu versiyondan elde edilen çıktı şöyledir:

```
a.x 10, b.x 20
a.x 30, b.x 30
```

Gördüğünüz gibi, **b**'nin **a**'ya atanmasından sonra her iki değişken de aynı nesneye “başlangıçta **b** tarafından referansta bulunan nesneye” referansta bulunmaktadır.

## **Yapılara Neden Gerek Vardır?**

Bu noktada, **struct**'un C#'ta neden yer aldığı merak ediyor olabilirsiniz. Öyle ya; **struct**, **class**'ın daha az yetenekli bir versiyonu gibi görünüyor. Bunun yanıtını verimlilik ve performansta aramalısınız. Yapılar bir değer tipi oldukları için, bir referans aracılığıyla değil de doğrudan kendi üzerinde işlem görürler. Böylece, bir **struct** ayrı bir referans değişkeni gerektirmez. Bu, bazı durumlarda daha az bellek kullanıldığı anlamına gelir. Üstelik, bir **struct** doğrudan erişilebildiği için, sınıf nesnelerinin erişimlerine özgü performans kaybına da maruz kalmaz. Sınıflar bir referans tipidir. Bu nedenle, sınıf nesnelerine yapılan tüm

erişimler bir referans aracılığıyla olmalıdır. Bu tür bir dolaylı erişim, erişimlerin her birine ek bir yük bindirir. Yapılar böyle bir yükün altına girmezler. Genel olarak, birbirile bağıntılı küçük bir veri grubunu saklamak istiyorsanız, fakat kalıtima ihtiyacınız yoksa ya da referans tiplerinin sağladığı diğer avantajlardan yararlanmak istemiyorsanız, **struct** sizin için çok daha uygun bir tercih olacaktır.

Bir yapının uygulamada nasıl kullanılabileceğini gösteren işte bir başka örnek. Aşağıdaki program bir e-ticaret işlem kaydını canlandırmaktadır. Her işlem, paketin numarasını ve uzunluğunu içeren bir paket başlığına sahiptir. Bunun peşinden hesap numarası ve işlem yapılacak miktar gelir. Paket başlığı kendi kendisini içeren bilgi birimi olduğu için bir yapı olarak organize edilmiştir. Bu yapı daha sonra bir işlem kaydını ya da bir başka tipte bilgi paketini oluşturmak amacıyla kullanılabilir.

```
// Verileri gruplamak icin yapilar uygundur.

using System;

// Bir paket yapisi tanimla.
struct PacketHeader {
    public uint packNum; // paket numarasi
    public ushort packLen; // paketin uzunlugu
}

// PacketHeader'i kullanarak bir e-ticaret islem kaydi olustur.
class Transaction {
    static uint transacNum = 0;

    PacketHeader ph; // PacketHeader'i Transaction'a dahil et
    string accountNum;
    double amount;

    public Transaction(string acc, double val) {
        // paket basligi olustur
        ph.packNum = transacNum++;
        ph.packLen = 512; // keyfi bir uzunluk
        accountNum = acc;
        amount = val;
    }

    // Islemi simule et.
    public void sendTransaction() {
        Console.WriteLine("Packet #: " + ph.packNum +
                           ", Length: " + ph.packLen +
                           ",\n Account #: " + accountNum +
                           ", Amount: {0:C}\n", amount);
    }
}

// Paketi goster.
class PacketDemo {
    public static void Main() {
        Transaction t = new Transaction("31243", -100.12);
```

```

        Transaction t2 = new Transaction("AB4655", 345.25);
        Transaction t3 = new Transaction("8475-09", 9800.00);

        t.sendTransaction();
        t2.sendTransaction();
        t3.sendTransaction();
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Packet #: 0, Length: 512,
Account #: 31243, Amount: ($100.12)

Packet #: 1, Length: 512,
Account #: AB4655, Amount: $345.25

Packet #: 2, Length: 512,
Account #: 8475-09, Amount: $9,800.00

```

**PacketHeader, struct** için iyi bir tercihtir, çünkü küçük miktarda veri içermektedir, üstelik kalıtım ve hatta metot bile kullanmamaktadır. Bir yapı olarak **PacketHeader**, bir referansın getirdiği ek yükün altına girmez. Oysa bir sınıf bu yükü taşımak zorundadır. Yani, herhangi bir tipte işlem kaydı, **PacketHeader**'i verimliliğini etkilemeden kullanabilir.

İlginc bir nokta, C++'in da yapıları içermesi ve **struct** anahtar kelimesini kullanmasıdır. Ancak, C# ve C++ yapıları aynı değildir. C++'ta **struct** bir sınıf tipi tanımlar. Bu nedenle, C++'ta **struct** ve **class** hemen hemen eşdeğerdir. (Aradaki fark, her ikisinin üyelerinin varsayılan erişimlerinden kaynaklanmasıdır. **class** için varsayılan üye erişimi **private**'tir, **struct** için ise **public**'tir.) C#'ta ise **struct** bir değer tipi tanımlar; **class** ise bir referans tipi tanımlar.

## Numaralandırmalar

*Numaralandırma* (*enumeration*), isimlendirilmiş tamsayı sabitlerinden oluşan bir kümedir. **enum** anahtar kelimesi bir numaralandırılmış tip deklare eder. Numaralandırmanın şekli genel olarak şöyledir:

```
enum isim { numaralandırma listesi };
```

Burada, numaralandırmaya verilen tip ismi **isim** ile belirtilir. **numaralandırma listesi**, virgülle ayrılan tanımlayıcıların bir listesidir.

İşte bir örnek; aşağıdaki kod parçası, çeşitli elma tiplerini numaralandıran **apple** adında bir numaralandırma tanımlamaktadır:

```
enum apple { Jonathan, GoldenDel, RedDel, Winsap,
             Cortland, McIntosh };
```

Numaralandırmayı anlamak için en önemli hususlardan biri şudur: Her sembol bir tamsayı değere karşılık gelir ve her sembole kendisinden hemen önce gelen sembolün değerinden bir

fazla değer verilir. İlk numaralandırma simbolünün varsayılan değeri 0'dır. Böylece, **Jonathan**'ın değeri 0, **GoldenDel**'in değeri 1'dir vs.

Bir tamsayının kullanılabileceği her yerde bir numaralandırma sabiti de kullanılabilir. Ancak, bir **enum** tipi ile standart tamsayı tipler arasında kapalı dönüşümler tanımlı değildir. Bu nedenle, açık tip atamaları kullanılmalıdır. Ayrıca, iki numaralandırma tipi arasında dönüşüm gerçekleştirirken bir tip ataması kullanılmalıdır.

Bir numaralandırmanın üyeleri, nokta operatörü üzerinden üyelerin tip isimleri aracılığıyla erişilir. Örneğin, şu ifade

```
Console.WriteLine(apple.RedDel + " has the value " +  
    (int)apple.RedDel);
```

aşağıdaki çıktıyı gösterir:

```
RedDel has the value 2
```

Çıktıdan görüldüğü gibi, numaralandırılmış bir değer ekranda gösterilirken bu değerin ismi kullanılır. Bunun tamsayı değerini elde etmek için ise **int**'e dönüşüm sağlayan bir tip ataması kullanılmalıdır. (Bu davranış C#’ın ilk versiyonlarına göre farklılık gösterir. İlk versiyonlarda, numaralandırılmış bir değerin ismi yerine tamsayı karşılıkları ekranda gösterilirdi.)

İşte size **apple**'ın numaralandırılışını gösteren bir program:

```
// Bir numaralandırma gösterir.  
  
using System;  
  
class EnumDemo {  
    enum apple { Jonathan, GoldenDel, RedDel, Winsap,  
        Cortland, McIntosh };  
  
    public static void Main() {  
        string[] color = {  
            "Red",  
            "Yellow",  
            "Red",  
            "Red",  
            "Red",  
            "Red",  
            "Reddish Green"  
        };  
  
        apple i; // bir enum değişkeni deklare et  
  
        // enum üzerinde gezinmek için i kullan  
        for(i = apple.Jonathan; i <= apple.McIntosh; i++)  
            Console.WriteLine(i + " has value of " + (int)i);  
  
        Console.WriteLine();  
  
        // bir diziyi indekslemek için numaralandırma kullan
```

```

        for(i = apple.Jonathan; i <= apple.McIntosh; i++)
            Console.WriteLine("Color of " + i + " is " +
                color[(int)i]);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Jonathan has value of 0
GoldenDel has value of 1
RedDel has value of 2
Winsap has value of 3
Cortland has value of 4
McIntosh has value of 5

Color of Jonathan is Red
Color of GoldenDel is Yellow
Color of RedDel is Red
Color of Winsap is Red
Color of Cortland is Red
Color of McIntosh is Reddish Green

```

**for** döngülerinin **apple** tipinde bir değişken tarafından nasıl kontrol edildiğine dikkat edin. Numaralandırma tamsayı tipinde olduğu için, bir numaralandırma değeri bir tamsayının kullanılabileceği her yerde kullanılabilir. **apple** içindeki numaralandırılmış değerler sıfırdan başladığından ötürü, bu değerler elmanın rengini elde etmek amacıyla **color**'u indekslemek için kullanılabilir. Numaralandırma değeri **color** dizisini indekslemek için kullanılıncı bir tip dönüşümüne gerek olduğuna dikkat edin. Önceden de bahsedildiği gibi, tamsayılar ve numaralandırma tipleri arasında kapalı bir dönüşüm tanımlı değildir. Açıkça bir tip ataması gereklidir.

Bir diğer husus da şudur: Numaralandırılmış değerler birer tamsayı oldukları için, bir **switch** ifadesini kontrol etmek amacıyla bir numaralandırılmış tip kullanabilirsiniz. Bunun bir örneğini az sonra göreceksiniz.

## Bir Numaralandırmaya İlk Değer Atamak

Bir ilk değer ataması kullanarak bir ya da daha fazla sembolün değerini belirtebilirsiniz. Bunun için sembolün peşinden eşittir işareti ve bir tamsayı değer kullanın. İlk değer atamalarının peşinden gelen sembollere bir önceki atanan değerden daha büyük değerler atanır. Örneğin, aşağıdaki kod, **RedDel**'e 10 değerini atar:

```
enum apple { Jonathan, GoldenDel, RedDel = 10, Winsap,
    Cortland, McIntosh };
```

Şimdi bu simgelerin değerleri aşağıdaki gibidir:

Jonathan	0
GoldenDel	1
RedDel	10
Winsap	11
Cortland	12

McIntosh 13

## Bir Numaralandırmanın Temel Tipini Belirtmek

Numaralandırmaların varsayılan tipi `int`'tir, fakat `char` tipi haricinde herhangi bir tamsayı tipinde bir numaralandırma oluşturabilirsiniz. `int` dışında bir tip belirtmek için temel tipi numaralandırma isminden sonra yazın ve bu ikisini, iki nokta işaretleri ile birbirinden ayırin. Örneğin, aşağıdaki ifade `apple`'ı, `byte`'ı temel alan bir numaralandırma haline sokmaktadır:

```
enum apple : byte { Jonathan, GoldenDel, RedDel, Winsap,  
                     Cortland, McIntosh };
```

Artık, örneğin `apple.Winsap`, `byte` cinsinden bir niceliktir.

## Numaralandırmaların Kullanımı

İlk bakışta numaralandırmaları C#'ın ilginç ancak nispeten önemsiz bir parçası olarak düşünebilirsiniz. Fakat bu, söz konusu değildir. Programınız bir ya da daha fazla özelleştirilmiş sembole gerek duyduğunda numaralandırmalar çok kullanışlıdır. Örneğin, bir fabrikadaki taşıma bandını kontrol eden bir program yazdığınıza hayal edin. Şu komutları parametre olarak kabul eden `conveyor()` adında bir metot oluşturabilirsiniz: başla, dur, ileri ve geri. `conveyor()`'a, mesela başla için `1`, dur için `2` gibi tamsayı değerler aktarmak yerine - ki, bu hataya açık bir yaklaşımındır - bu değerlere birer sözcük atayan bir numaralandırma oluşturabilirsiniz. Bu yaklaşımın bir örneği şu şöyledir:

```
// Bir taşıma bandını canlandırır.  
  
using System;  
  
class ConveyorControl {  
    // taşıma bandı komutlarını numaralandır  
    public enum action { start, stop, forward, reverse };  
  
    public void conveyor(action com) {  
        switch(com) {  
            case action.start:  
                Console.WriteLine("Starting conveyor.");  
                break;  
            case action.stop:  
                Console.WriteLine("Stopping conveyor.");  
                break;  
            case action.forward:  
                Console.WriteLine("Moving forward.");  
                break;  
            case action.reverse:  
                Console.WriteLine("Moving backward.");  
                break;  
        }  
    }  
}
```

```
class ConveyorDemo {  
    public static void Main() {  
        ConveyorControl c = new ConveyorControl();  
  
        c.conveyor(ConveyorControl.action.start);  
        c.conveyor(ConveyorControl.action.forward);  
        c.conveyor(ConveyorControl.action.reverse);  
        c.conveyor(ConveyorControl.action.stop);  
    }  
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Starting conveyor.  
Moving forward.  
Moving Backward.  
Stopping conveyor.
```

**conveyor()**, **action** tipinde bir argüman aldığı için, yalnızca **action** tarafından tanımlanan değerler bu metoda aktarılabilir. Örneğin aşağıda, **conveyor()**'a 22 değerini aktarmak için bir girişimde bulunulmaktadır:

```
c.conveyor(22); // Hata!
```

Bu ifade derlenmeyecektir, çünkü **int**'ten **action**'a önceden tanımlanmış bir dönüşüm söz konusu değildir. Bu durum, **conveyor()**'a geçersiz komutların aktarılmasına engel olur. Kuşkusuz, tip ataması yoluyla dönüşümü zorlayabilirsiniz, fakat bu önceden tasarlanmış bir hareket olmalıdır ve kazara hatalı kullanımlara yol açmamalıdır. Ayrıca, komutlar numara yerine isim ile belirtildikleri için, **conveyor()**'u kullananların kazara yanlış bir değer aktarmaları pek de olası bir durum değildir.

Bu örnekle ilgili dikkat çekmemiz gereken bir başka husus daha vardır: **switch** ifadesini kontrol etmek amacıyla bir numaralandırılmış tip kullanıldığına dikkat edin. Önceden de bahsedildiği gibi, numaralandırmalar tamsayı tipler olarak ele alındıklarından dolayı, **switch** ifadesinde kullanılmak üzere tamamen uygundurlar.

# KURAL DIŞI DURUM YÖNETİMİ

*Kural dışı durum (exception)*, programın çalışması sırasında ortaya çıkan bir hatadır. C#'ın kural dışı durum yönetimi alt sistemini kullanarak programın çalışması sırasında ortaya çıkan hataları yapısal ve kontrollü bir şekilde ele alabilirsiniz. Kural dışı durum yönetimi için C#'ın uyguladığı yaklaşım, C++ ve Java tarafından kullanılan metodların bir harmanıdır ve bu metodlar üzerine bir ilerlemedir. Yani bu konu, bu dillerden herhangi biriyle ilgili bilgisi olan okuyuculara tanındık gelecektir. C#'ın kural dışı durum yönetimini eşsiz kıyan, kural dışı durum yönetiminin temiz ve anlaşılır olarak uygulanmasıdır.

Kural dışı durum yönetiminin başlıca avantajı, daha önce büyük programlara “elle” girilmek zorunda olan hata yönetimi kodunun büyük bir kısmını otomatize etmesidir. Örneğin, kural dışı durum yönetimi mevcut olmayan bir bilgisayar dilinde, bir metod başarısız olduğunda hata kodları döndürülmelidir ve metodun her çağrılarında bu değerler elle kontrol edilmelidir. Bu yaklaşım hem sıkıcı hem de hataya açiktır. Kural dışı durum yönetimi, programınızda bir hata ortaya çıktığında otomatik olarak çalıştırılan *kural dışı durum yöneticisi (exception handler)* adında bir kod bloğu tanımlamanıza imkan vererek hata yönetimi verimliliğini artırır. Spesifik işlemlerin ya da metod çağrılarının başarılı ya da başarısız olduğunu elle kontrol etmeye gerek yoktur. Eğer bir hata meydana gelmişse söz konusu hata, kural dışı durum yöneticisi tarafından işleme alınacaktır.

Kural dışı durum yönetiminin önemli olmasının bir başka sebebi, yaygın program hataları için C#'ta standart kural dışı durumlar tanımlanmış olmasıdır. Sıfır'a bölme ya da değer aralığı dışına çıkan indeks, yaygın programlama hataları arasında yer almaktadır. Bu tür hatalara yanıt vermek için programınız bu tür kural dışı durumları gözlemeli ve kontrol altına almalıdır.

Son tahlilde, başarılı bir C# programcısı olmak için, C#'ın kural dışı durum yönetim alt sistemini yönlendirmeyi tam anlamıyla becerebiliyor olmalısınız.

## System.Exception Sınıfı

C#'ta kural dışı durumlar sınıflarla simgelenir. Tüm kural dışı durum sınıfları, **System** isim uzayının bir parçası olan standart **Exception** sınıfından türetilmelidir. Böylece, tüm kural dışı durumlar **Exception**'ın birer alt sınıfıdır.

**SystemException** ve **ApplicationException**, **Exception**'dan türetilir. Bunlar C#'ta tanımlı olan iki genel kategoriyi desteklerler: C#'ın çalışma zamanı sistemi (yani, Common Language Runtime) tarafından üretilenler ve uygulama programları tarafından üretilenler. Ne **SystemException** ne de **ApplicationException**, **Exception**'a bir şey eklemez. Bunlar sadece, iki farklı kural dışı durum hiyerarşisinin en üstünde yer alan durumları tanımlarlar.

C#'ta, **SystemException**'dan türetilen birkaç standart kural dışı durum tanımlanmıştır. Örneğin, sıfır'a bölme denemesinde bulunulduğunda **DivideByzeroException** olarak bilinen bir kural dışı durum üretilir. Bu bölümde daha sonra göreceğiniz gibi, kendi kural dışı durum sınıfınızı **ApplicationException**'dan türeterek kendiniz oluşturabilirsiniz.

# Kural Dışı Durum Yönetimiyle İlgili Esaslar

C#'ta kural dışı durum yönetimi dört anahtar kelime aracılığıyla gerçekleştirilir: **try**, **catch**, **throw** ve **finally**. Bu anahtar kelimeler, birinin kullanımının diğerinin kullanımını gerektirdiği birbiriyle bağlantılı bir alt sistem oluştururlar. Bu bölüm boyunca bu anahtar kelimelerin biri ayrıntılarıyla incelenecektir. Ancak, incelemeye başlamadan önce, bunların her birinin kural dışı durum yönetiminde nasıl bir rol üstlendiği hakkında genel bilgi sahibi olmak yararlıdır. Şimdi bunların nasıl çalışıklarına kısaca bir göz atalım.

Kural dışı durumları izlemek istediğiniz program ifadeleri bir **try** bloğu içinde yer alır. Eğer bir **try** bloğu içinde bir kural dışı durum meydana gelirse, söz konusu kural dışı durum fırlatılmış olur. Kodunuz bu kural dışı durumu **catch** kullanarak yakalayabilir ve bunu mantıklı bir şekilde kontrol altına alır. Sistem tarafından üretilen kural dışı durumlar, otomatik olarak C# çalışma zamanı sistemi tarafından fırlatılır. Kural dışı bir durumu elle fırlatmak için **throw** anahtar kelimesini kullanın. **try** bloğundan çıktıktan sonra kesinlikle çalıştırılması gereken bir kod **finally** bloğuna yerleştirilir.

## try ve catch'in Kullanımı

Kural dışı durum yönetiminin özünde **try** ve **catch** yer almaktadır. Bu anahtar kelimeler birlikte çalışırlar; **catch** olmadan **try**'ı tek başına kullanamazsınız; **try**'sız da **catch** olmaz. **try/catch** kural dışı durum yönetim bloklarının genel yapısı şu şekildedir:

```
try {  
    // hataları izleyen kod bloğu  
}  
catch (KuralDışıTip1 exOb) {  
    // KuralDışıTip1 için yönetici  
}  
catch (KuralDışıTip2 exOb) {  
    // KuralDışıTip2 için yönetici  
}  
.  
.  
.
```

Burada **KuralDışıTip**, meydana gelen kural dışı durumun tipidir. Kural dışı bir durum fırlatıldığında, karşılık gelen **catch** ifadesi tarafından yakalanır; söz konusu **catch** ifadesi kural dışı durumu işler. Genel yapıdan da görüldüğü gibi, bir **try** ile ilişkili birden fazla **catch** ifadesi olabilir. Hangi **catch** ifadesinin çalıştırılacağını kural dışı durumun tipi belirler. Yani, **catch** ifadesinde belirtilen kural dışı durum tipi, kural dışı durum ile eşleşirse, söz konusu **catch** ifadesi çalıştırılır (diğerlerinin tümü atlanır). Kural dışı bir durum yakalandığında **exOb** bunun değerini alacaktır.

Aslında **exOb**'u belirtmek istege bağlıdır. Eğer kural dışı durum yöneticisinin kural dışı durum nesnesine erişmesi gerekmiyorsa (ki, bu sıkça rastlanılan bir durumdur), **exOb**'u

belirtmeye gerek yoktur. Bu nedenle, bu bölümdeki örneklerin birçoğunda **exOb** belirtilmeyecektir.

İşte önemli bir husus: Eğer hiç kural dışı durum fırlatılmamışsa, **try** bloğu normal olarak sona erer ve tüm **catch** ifadeleri de pas geçilir. Program, son **catch** ifadesinin peşinden gelen ifade ile devam eder. Yani, **catch** ifadeleri yalnızca bir kural dışı durum fırlatıldığında çalıştırılırlar.

## Basit Bir Kural Dışı Durum Örneği

Aşağıda, bir kural dışı durumun nasıl izleneceğini ve yakalanacağını gösteren bir örnek yer almaktadır. Bildiğiniz gibi, bir diziyi sınırları dışında indekslemeye çalışmak hatalı bir durumdur. Böyle bir durum meydana gelince, C# çalışma zamanı sistemi C#'ta tanımlı, standart bir kural dışı durum olan **IndexOutOfRangeException**'ı fırlatır. Aşağıdaki program kasıtlı olarak bu tür bir kural dışı durum oluşturur ve sonra da onu yakalar:

```
//Kural disi durum yonetimini gosterir.

using System;

class ExcDemo1 {
    public static void Main() {
        int[] nums = new int[4];

        try {
            Console.WriteLine("Before exception is generated.");

            /* Indeksin sinir disinda oldugunu gosteren bir kural
               disi durum uret */
            for(int i = 0; i < 10; i++) {
                nums[i] = i;
                Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
            }

            Console.WriteLine("this won't be displayed");
        }
        catch (IndexOutOfRangeException) {
            // kural disi durumu yakala
            Console.WriteLine("Index out-of-bounds!");
        }
        Console.WriteLine("After catch statement.");
    }
}
```

Bu program aşağıdaki çıktıyi ekranda gösterir:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Index out-of-bounds!
```

After catch statement.

Dikkat ederseniz, `nums` dört elemandan oluşan bir int dizisidir. Ancak, `for` döngüsü `nums`'ı 0'dan 9'a kadar indekslemeye çalışmaktadır. Bu durum, indeks değeri olarak 4'e erişilmeye çalışıldığında `IndexOutOfRangeException`'ın meydana gelmesine neden olur.

Oldukça kısa bir program olmasına rağmen yukarıdaki program, kural dışı durum yönetimiyle ilgili birkaç önemli noktaya ışık tutmaktadır. Öncelikle, hata durumu olup olmadığını izlemek istediğiniz kod, bir `try` bloğu içine yerleştirilir. İkincisi, bir kural dışı durum ortaya çıktığında (bu örnekte, `for` döngüsü içinde `nums`'ı sınırları dışına taşıacak şekilde indekslemeye çalışmak kural dışı duruma neden olur), söz konusu kural dışı durum `try` bloğundan fırlatılır ve `catch` ifadesi tarafından yakalanır. Bu aşamada kontrol `catch`'e aktarılır ve `try` bloğu sona erer. Yani, `catch` çağrılmaz. Bunun yerine, programın çalışması `catch`'ten itibaren devam eder. Böylece, indeks sınır dışına taşındıktan sonra, takip eden `WriteLine()` ifadesi asla çalıştırılmayacaktır. `catch` ifadesi çalıştırıldıktan sonra program kontrolü `catch`'i takip eden ifadelerle sürecekdir. Yani, programın çalışmasının normal olarak devam edebilmesi için kural dışı duruma neden olan problemi çözme görevi, kural dışı durum yöneticisine aittir.

`catch` ifadesinde hiç parametre belirtilmemiğine dikkat edin. Daha önce bahsedildiği gibi, yalnızca kural dışı durum nesnesine erişim gerekliliğinde bir parametreye ihtiyaç duyulur. Bazı durumlarda kural dışı durum nesnesinin değeri, hata hakkında ek bilgi elde etmek amacıyla kural dışı durum yöneticisi tarafından kullanılabilir. Fakat, birçok durumda yalnızca kural dışı bir durumun meydana geldiğini bilmek yeterlidir. Bu nedenle, yukarıdaki programda olduğu kural dışı durum yöneticisinde `catch` parametresinin eksik oluşu, olağandışı bir durum değildir.

Daha önce açıklandığı gibi, `try` bloğu tarafından hiç kural dışı bir durum fırlatılmamışsa, `catch` ifadelerinden hiçbirini çalıştırılmayacaktır ve program kontrolü `catch` ifadesinin peşinden devam edecektir. Bunu doğrulamak için yukarıdaki örnekteki `for` döngüsünü şu halden

```
for(int i = 0; i < 10; i++) {  
    // ...  
    // suna değiştirin:  
  
    for(int i = 0; i < nums.Length; i++) {  
        // ...  
    }
```

Artık döngü hiçbir zaman `nums`'ın sınırları dışına çıkmaz. Böylece, hiç kural dışı bir durum ortaya çıkmaz ve `catch` bloğu çalıştırılmaz.

## İkinci Bir Kural Dışı Durum Örneği

`try` bloğu içinde çalıştırılan kodun tümünün kural dışı durumlar açısından izlendiğini kavramak önemlidir. `try` bloğu içinden çağrılan bir metot tarafından üretilebilecek kural dışı durumlar da buna dahildir. `try` bloğu içinden çağrılan bir metot tarafından fırlatılan bir kural

dışı durum söz konusu **try** bloğu tarafından yakalanabilir. Elbette burada, metodun kendisinin bu kural dışı durumu yakalamadığını varsayıyoruz. Örneğin, şu geçerli bir programdır:

```
/* Kural disi bir durum bir metot tarafından uretilebilir ve
   bir baska metot tarafından yakalanabilir. */

using System;

class ExcTest {
    // Kural disi bir durum uret.
    public static void genException() {
        int[] nums = new int[4];

        Console.WriteLine("Before exception is generated.");

        /* indeksin sinir disinda oldugunu gösteren
           bir kural disi durum uret. */
        for(int i = 0; i < 10; i++) {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }

        Console.WriteLine("this won't be displayed");
    }
}

class ExcDemo2 {
    public static void Main() {
        try {
            ExcTest.genException();
        }
        catch (IndexOutOfRangeException) {
            // kural disi durumu yakala
            Console.WriteLine("Index out-of-bounds!");
        }
        Console.WriteLine("After catch statement.");
    }
}
```

Bu program aşağıdaki çıktıyı üretir. Bu çıktı, daha önce gösterilen, programın ilk versiyonundan elde edilen çıktıının aynısıdır:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Index out-of-bounds!
After catch statement.
```

**genException()**, **try** bloğu içinden çağrıldığı için bunun ürettiği (ve yakalamadığı) kural dışı durum **Main()** içindeki **catch** tarafından yakalanır. Ancak, **genException()** eğer kural dışı durumu yakalamiş olsaydı, **Main()**’e tekrar geri aktarılmayacaktı. Bunu kavrayın.

## Yakalanmayan Kural Dışı Durumların Sonuçları

Önceki programda olduğu gibi, C#'ın standart kural dışı durumlarından birini yakalamanın sağladığı bir yan fayda da vardır: Bu durum, programın normal dışı olarak sona ermesini öner. Kural dışı bir durum fırlatıldığında bir yerlerdeki bir parça kod tarafından bu kural dışı durum yakalanmalıdır. Genel olarak, eğer programınız kural dışı bir durumu yakalamazsa, bu kural dışı durum C# çalışma zamanı sistemi tarafından yakalanacaktır. Buradaki sorun, çalışma zamanı sisteminin hatayı rapor edecek ve programı sonlandıracak olmasıdır. Söz geliş, bu örnekte indeksin sınır dışına taşmasından kaynaklanan kural dışı durum program tarafından yakalanmamaktadır:

```
// Birakin C# calisma zamanı sistemi hatayı kontrol altına alsin.

using System;

class NotHandled {
    public static void Main() {
        int[] nums = new int[4];

        Console.WriteLine("Before exception is generated.");

        /* Indeksin sinir disinda oldugunu gosteren
           bir kural disi durum uret. */
        for(int i = 0; i < 10; i++) {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }
    }
}
```

Dizi indeks hatası meydana gelince, programın çalışması durdurulur ve aşağıdaki hata mesajı ekranda gösterilir:

```
Unhandled Exception: System.IndexOutOfRangeException:
Index was outside the bounds of the array.
at NotHandled.Main()
```

Hataları ayıklarken bu tür bir mesajın sizin için yararlı olmasının yanı sıra, bu en azından, başkalarının görmesini istemediğiniz bir mesaj da olabilir. Programınızın kural dışı durumları kendisinin kontrol altına alması işte bu sebepten önemlidir.

Daha önce bahsedildiği gibi, kural dışı durumun tipi **catch** ifadesinde belirtilen tip ile eşlenmelidir. Eğer eşlenmiyorsa, kural dışı durum yakalanmayacaktır. Örneğin, aşağıdaki program **DivideByZeroException**'ı (C#'ın standart kural dışı durumlarından bir başkası) bir **catch** ifadesinde kullanarak bir dizinin sınır hatalarını yakalamaya çalışmaktadır. Dizinin sınırı aşıldığı zaman **IndexOutOfRangeException** üretilir, fakat bu, **catch** ifadesi tarafından yakalanmayacaktır. Bu durum, programın normal olmayan biçimde sona ermesiyle sonuçlanır.

```
// Bu calismayacaktir!
```

```
using System;

class ExcTypeMismatch {
    public static void Main() {
        int[] nums = new int[4];

        try {
            Console.WriteLine("Before exception is generated.");

            /* Indeksin sinir disinda oldugunu gosteren
               bir kural disi durum uret. */
            for(int i = 0; i < 10; i++) {
                nums[i] = i;
                Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
            }
            Console.WriteLine("this won't be displayed");
        }
        /* DivideByZeroException ile bir dizinin sinir hatalarini
           yakalayamassiniz. */
        catch(DivideByZeroException) {
            // kural disi durumu yakala
            Console.WriteLine("Index out-of-bounds!");
        }
        Console.WriteLine("After catch statement.");
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Before exception is generated.
```

```
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
```

```
Unhandled Exception: System.IndexOutOfRangeException:
    Index was outside the bounds of the array.
    at ExcTypeMismatch.Main()
```

Çıktının da gösterdiği gibi, **DivideByZeroException** için kullanılan bir **catch** ifadesi **IndexOutOfRangeException** kural dışı durumunu yakalamayacaktır.

## Kural Dışı Durumlar, Hataları Zarifçe Kontrol Altına Almanıza Olanak Tanır

Kural dışı durum yönetiminin sağladığı kilit avantajlardan biri, programınızın hatalara tepki vermesini ve daha sonra çalışmasını sürdürmesini mümkün kılmıştır. Örneğin, bir dizinin elemanlarını bir başka dizinin elemanlarına bölen aşağıdaki örneği ele alın. Eğer sıfıra bölme meydana gelirse DivideByZeroException üretilir. Programda bu kural dışı durum, hatanın rapor edilmesi ve ardından çalışmanın sürdürülmesiyle kontrol altına alınır. Böylece,

sıfır böölme girişimi, programın çalışması sırasında beklenmedik şekilde ortaya çıkan ve programın sona ermesiyle sonuçlanan bir hataya neden olmaz. Bunun yerine, hata zarifce kontrol altına alınır ve bu sayede programın çalışmasını sürdürmesine olanak tanınmış olur.

```
// Hatayı zarifce kontrol altına alır ve çalışmayı sürdürür.

using System;

class ExcDemo3 {
    public static void Main() {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i = 0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i] / denom[i]);
            }
            catch (DivideByZeroException) {
                // kural dışı durumu yakala.
                Console.WriteLine("Can't divide by Zero!");
            }
        }
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

Bu örnek bir başka önemli hususa işaret ediyor: Bir kural dışı durum bir kez kontrol altına alındıktan sonra sistemden dışarı çıkarılır. Bu yüzden, programda, döngünün her tekrarında **try** bloğuna yeniden girilir, çünkü önceki kural dışı durumların her biri kontrol altına alınmıştır. Bu sayede, programınızın tekrarlanan hataları kontrol altına alması mümkün olur.

## Birden Fazla catch İfadesinin Kullanımı

Bir **try** bloğu ile birden fazla **catch** ifadesini ilişkilendirebilirsiniz. Aslında bu yaygın bir yaklaşımındır. Ancak, her bir **catch**, farklı tipte bir kural dışı durumu yakalamalıdır. Örneğin, aşağıda gösterilen program hem dizi sınırlarından kaynaklanan hataları, hem de sıfır böölme hatalarını yakalar:

```
// Birden fazla catch ifadesi kullanır.

using System;
```

```

class ExcDemo4 {
    public static void Main() {
        // Bu ornekte numer, denom'dan daha uzundur.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i = 0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i] / denom[i]);
            }
            catch (DivideByZeroException) {
                // kural disi durumu yakala
                Console.WriteLine("Can't divide by Zero!");
            }
            catch (IndexOutOfRangeException) {
                // kural disi durumu yakala
                Console.WriteLine("No matching element found.");
            }
        }
    }
}

```

Bu program aşağıdaki çıktıyı üretir:

```

4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.

```

Cıktının da doğruladığı gibi, **catch** ifadelerinin her biri kendisine karşılık gelen kural dışı durum tipine tepki verir.

**catch** deyimleri genel olarak programda yer aldıkları sıra ile kontrol edilirler. Yalnızca eşlenen ifade gerçekleşir. Diğer **catch** bloklarının hiç biri dikkate alınmaz.

## Kural Dışı Durumların Tümünü Yakalamak

Bazen hangi tipte olursa olsun kural dışı durumların tümünü yakalamak isteyeceksiniz, Bunu gerçekleştirmek için parametresi belirtilmeyen bir **catch** ifadesi kullanın. Bu, bir “tümünü yakala” yöneticisi olacaktır. Bu tür bir yönetici, kural dışı durumların tümünün programınız tarafından kontrol altına alındığını garanti etmek istediğinizde kullanılabilir. Örneğin, aşağıdaki programda tek **catch** ifadesi “tümünü yakala” türündendir. Bu ifade, program tarafından üretilen **IndexOutOfRangeException** ve **DivideByZeroException**'in her ikisini de yakalar.

```
// "Tumunu yakala" catch ifadesi kullanır.
```

```

using System;

class ExcDemo5 {
    public static void Main() {
        // Bu ornekte numer, denom'dan daha uzundur.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i = 0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i] / denom[i]);
            }
            catch {
                Console.WriteLine("Some exception occurred.");
            }
        }
    }
}

```

Çıktı aşağıda gösterilmiştir:

```

4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
Some exception occurred.
128 / 8 is 16
Some exception occurred.
Some exception occurred.

```

“Tümünü yakalayan” bir **catch** kullanırken bir konunun hatırda tutulması gereklidir: Bu tür bir **catch, catch** sekansı içindeki son **catch** ifadesi olmalıdır.

## try Bloklarını Kümelemek

Bir **try** bloğu bir başkasının içine yerleştirilip, kümelenebilir. İçteki **try** bloğu tarafından üretilen ancak bu **try** ile ilişkili bir **catch** tarafından yakalanmayan bir kural dışı durum, dışındaki **try** bloğuna aktarılır. Örneğin, aşağıdaki programda **IndexOutOfRangeException**, içteki **try** bloğu tarafından değil, dışındaki **try** bloğu tarafından yakalanmaktadır:

```

// Ic ice kumelenmis try blogu kullanir.

using System;

class NestTrys {
    public static void Main() {
        // Bu ornekte numer, denom'dan daha uzundur.

        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
    }
}

```

```

try {
    // distaki try
    for(int i = 0; i < numer.Length; i++) {
        try { // kumelenmiş try
            Console.WriteLine(numer[i] + " / " +
                               denom[i] + " is " +
                               numer[i] / denom[i]);
        }
        catch (DivideByZeroException) {
            // kural dışı durumu yakala
            Console.WriteLine("Can't divide by Zero!");
        }
    }
    catch (IndexOutOfRangeException) {
        // kural dışı durumu yakala

        Console.WriteLine("No matching element found.");
        Console.WriteLine("Fatal error -- program terminated.");
    }
}
}

```

Programdan elde edilen çıktı aşağıdaki gibidir:

```

4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error -- program terminated.

```

Bu örnekte, içteki **try** tarafından kontrol altına alınabilecek bir kural dışı durum - sıfır'a bölme hatası - programın çalışmasına imkan verir. Ancak, dışındaki **try** tarafından dizi sınırlarıyla ilgili bir hata yakalanır ve bu, programın sonlandırılmasına neden olur.

Yukarıdaki program önemli bir noktaya ışık tutmaktadır, ki bu durum genelleştirilebilir. Genellikle kümelenmiş **try** blokları, farklı hata kategorilerinin farklı şekillerde kontrol altına alınmalarına olanak tanımak için kullanılır. Bazı hata tipleri bir faciadır; üstelik bu tür hatalar düzeltilemeyebilir. Bazıları ise önemsizdir ve anında kontrol altına alınabilir. Programcıların birçoğu en şiddetli hataları yakalamak için dışındaki **try** bloğunu kullanırlar; böylece, içteki **try** bloklarının daha az öneme sahip hataları kontrol altına almalarına imkan vermiş olurlar. Dışındaki **try** bloğunu ayrıca, içteki blok tarafından kontrol altına alınmayan hatalar için “tümünü yakalayan” bir blok olarak da kullanabilirsiniz.

## Bir Kural Dışı Durum Fırlatmak

Önceki örnekler C# tarafından otomatik olarak üretilen kural dışı durumları yakalamışlardı. Ancak, **throw** ifadesini kullanarak bir kural dışı durumu elle fırlatmak da mümkündür. Bunun genel yapısı aşağıda gösterilmiştir:

```
throw exceptOb;
```

**exceptOb**, **Exception** sınıfından türetilmiş bir kural dışı durum sınırlarına ait bir nesne olmalıdır.

İşte, **DivideByZeroException**'ını elle fırlatarak **throw** ifadesini gösteren bir örnek:

```
// Bir kural dışı durumu elle fırlatmak.  
  
using System;  
  
class ThrowDemo {  
    public static void Main() {  
        try {  
            Console.WriteLine("Before throw.");  
            throw new DivideByZeroException();  
        }  
  
        catch (DivideByZeroException) {  
            // kural dışı durumu yakala  
            Console.WriteLine("Exception caught.");  
        }  
  
        Console.WriteLine("After try/catch block.");  
    }  
}
```

Programın çıktısı aşağıdaki gibidir:

```
Before throw.  
Exception caught.  
After try/catch block.
```

**throw** ifadesi içinde **new** kullanılarak **DivideByZeroException**'in nasıl oluşturulduğuna dikkat edin. **throw**'un bir nesne fırlattığını hatırlınızdan çıkarmayın. Bu nedenle, **throw**'un fırlatması için bir nesne oluşturmalısınız - yani, yalnızca bir tip fırlatamazsınız. Bu örnekte bir **DivideByZeroException** nesnesi oluşturmak için varsayılan yapılandırıcı kullanılmaktadır, fakat diğer yapılandırıcılar da kural dışı durumlar için kullanılmak üzere hazırlıdır.

Çoğunlukla fırlatacağınız kural dışı durumlar, sizin oluşturduğunuz kural dışı durum sınıflarının örnekleri olacaktır. Bu bölümün ileriki sayfalarında göreceğiniz gibi, kendi kural dışı durum sınıflarınızı oluşturmak kodunuz içindeki hataları, programınızın genel kural dışı durum yönetim stratejisinin bir parçası olarak kontrol altına almanıza imkan verir.

## Kural Dışı Durumu Yeniden Fırlatmak

Bir **catch** ifadesi tarafından yakalanan bir kural dışı durum, dışındaki **catch** tarafından da yakalanabilsin diye yeniden fırlatılabilir. Kural dışı bir durumu yeniden fırlatmanın en olası nedeni, birden fazla kural dışı durum yöneticisinin söz konusu kural dışı duruma erişmesine imkan vermektedir. Söz geliş, bir kural dışı durum yöneticisi bir kural dışı durumun belki bir yönünü idare etmektedir; ikinci bir yönetici ise bir diğer yönyle ilgilenmektedir. Bir kural dışı durumu yeniden fırlatmak için, açıkça bir kural dışı durumu belirtmeden yalnızca **throw**'u belirtmeniz yeterlidir. Yani, **throw**'un şu şeklini kullanırsınız:

```
throw ;
```

Hatırlarsanız, bir kural dışı durumu yeniden fırladığınızda bu durum aynı **catch** ifadesi tarafından yeniden yakalanmayacaktır; bir sonraki **catch** ifadesine kayacaktır.

Aşağıdaki program, bir kural dışı durumu yeniden fırlatmayı örneklemektedir. Bu kez program, **IndexOutOfRangeException**'ını yeniden fırlatmaktadır.

```
// Kural disi bir durumu yeniden fırlat.

using System;

class Rethrow {
    public static void genException() {
        // Bu ornekte numer, denom'dan daha uzundur.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i = 0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i] / denom[i]);
            }
            catch (DivideByZeroException) {
                // kural disi durumu yakala
                Console.WriteLine("Can't divide by Zero!");
            }
            catch (IndexOutOfRangeException) {
                // kural disi durumu yakala
                Console.WriteLine("No matching element found.");
                throw; // kural disi durumu yeniden fırlat
            }
        }
    }

    class RethrowDemo {
        public static void Main() {
            try {
                Rethrow.genException();
            }
        }
    }
}
```

```

        }
        catch (IndexOutOfRangeException) {
            // kural disi durumu yeniden yakala
            Console.WriteLine("Fatal error - " +
                "program terminated.");
        }
    }
}

```

Bu programda sıfır bölmeye hataları yerel olarak **genException()** tarafından kontrol altına alınmaktadır, fakat dizi sınırlarıyla ilgili bir hata yeniden fırlatılmıştır. Bu örnekte **IndexOutOfRangeException**, **Main()** tarafından kontrol altına alınmıştır.

## **finally'nin Kullanımı**

Kimi zaman **try/catch** bloğundan çıktıktan sonra çalıştırılmak üzere bir kod bloğu tanımlamak isteyeceksiniz. Örneğin, bir kural dışı durum, mevcut metodun vaktinden önce dönmesine neden olarak metodu sona erdiren bir hataya yol açabilir. Ancak, söz konusu metod kapatılması gereken bir dosya ya da ağ bağlantısı açmış olabilir. Bu tür durumlara programlamada sıkça rastlanır. C# bu tür durumları kontrol alıma almak için kullanışlı bir yöntem sunmaktadır: **finally**.

Btr **try/catch** bloğundan çıktıktan sonra çalıştırılacak kod bloğunu belirtmek için **try/catch** sekansının sonuna bir **finally** bloğu yerleştirin. **finally** içeren bir **try/catch** bloğunun genel şekli aşağıdaki gibidir:

```

try {
    // hataları izleyen kod bloğu
}
catch (KuralDışıTip1 exOb) {
    // KuralDışıTip1 için yönetici
}
catch (KuralDışıTip2 exOb) {
    // KuralDışıTip2 için yönetici
}
.
.
.
finally {
    // finally kodu
}

```

**finally** bloğu, program kontrolü **try/catch** bloğunu terk eder etmez, bu duruma hangi şartlar neden olursa olsun gerçekleşecektir. Yani, **try** bloğu normal biçimde sona erse de ya da buna bir kural dışı durum neden olsa da, çalıştırılacak en son kod **finally** tarafından tanımlanandır. Ayrıca, **try** bloğu içindeki herhangi bir kod ya da buna karşılık gelen **catch** ifadelerinden herhangi biri söz konusu metottan dönüyorsa **finally** bloğu gerçekleşir.

İşte bir **finally** örneği:

```

// finally kullanır.

using System;

class UseFinally {
    public static void genException(int what) {
        int t;
        int[] nums = new int[2];

        Console.WriteLine("Receiving " + what);
        try {
            switch(what) {
                case 0:
                    t = 10 / what; // sıfıra bolme hatası uret
                    break;
                case 1:
                    nums[4] = 4; // dizi indeks hatası uret.
                    break;
                case 2:
                    return; // try blogundan don.
            }
        }
        catch (DivideByZeroException) {
            // kural disi durumu yakala
            Console.WriteLine("Can't divide by Zero!");
            return; // return from catch
        }
        catch (IndexOutOfRangeException) {
            // kural disi durumu yakala
            Console.WriteLine("No matching element found.");
        }
        finally {
            Console.WriteLine("Leaving try.");
        }
    }
}

class FinallyDemo {
    public static void Main() {
        for(int i = 0; i < 3; i++) {
            UseFinally.genException(i);
            Console.WriteLine();
        }
    }
}

```

İşte, programın ürettiği çıktı:

```

Receiving 0
Can't divide by Zero!
Leaving try.

```

```

Receiving 1
No matching element found.
Leaving try.

```

```
Receiving 2
Leaving try.
```

Çıktıdan da görüldüğü gibi, **try** bloğu ne şekilde terk edilirse edilsin **finally** bloğu her koşulda gerçekleşmektedir.

## Kural Dışı Durumlara Yakından Bir Bakış

Şimdiye kadar kural dışı durumları yakalamaktayız, fakat kural dışı durum nesnesinin kendisi ile ilgili henüz hiç bir şey yapmadık. Önceden de bahsedildiği gibi, **catch** ifadesi bir kural dışı durum tipi ve parametresi belirtmenize imkan vermektedir. Söz konusu parametre kural dışı durum nesnesini alır. Tüm kural dışı durumlar **Exception**'dan türetildiği için tüm kural dışı durumlar **Exception**'ın tanımladığı üyeleri desteklerler. Bu bölümde **Exception**'ın en yararlı üyelerinden ve yapılandırıcılarından birkaçını inceleyeceğiz ve **catch** parametresinin uygun bir kullanımını göreceğiz.

**Exception**'da birkaç özellik tanımlıdır. Bunlardan en ilginç üçü **Message**, **StackTrace** ve **TargetException**'dır. Bunların tümü salt okunurdur. **Message**, hatanın özünü tarif eden bir karakter katarı içerir. **StackTrace**, kural dışı duruma yol açan çağrıların yığısını (stack) tutan bir karakter katarı içerir. **TargetException** ise kural dışı durumun hangi metot tarafından üretildiğini belirten bir nesne alır.

**Exception**'da ayrıca birkaç metot da tanımlıdır. Bunların içinde en sık kullanacağınız metot, söz konusu kural dışı durumu tarif eden bir karakter katarı döndüren **ToString()**'dır. Bir kural dışı durum **WriteLine()** üzerinden gösterilince **ToString()** otomatik olarak çağrılır, örneğin.

Aşağıdaki program bu özellikleri ve metotları göstermektedir:

```
// Exception üyelerinin kullanımı.

using System;

class ExcTest {
    public static void genException() {
        int[] nums = new int[4];

        Console.WriteLine("Before exception is generated.");

        /* Indeksin sınır dışında olduğunu gösteren
           bir kural dışı durum üret. */
        for(int i = 0; i < 10; i++) {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }

        Console.WriteLine("this won't be displayed");
    }
}
```

```

    }

    class UseExcept {
        public static void Main() {

            try {
                ExcTest.genException();
            }
            catch (IndexOutOfRangeException exc) {
                // kural disi durumu yakala
                Console.WriteLine("Standard message is: ");
                Console.WriteLine(exc); // ToString()'i cagirir
                Console.WriteLine("stack trace: " + exc.StackTrace);
                Console.WriteLine("Message: " + exc.Message);
                Console.WriteLine("TargetSite: " + exc.TargetSite);
            }
            Console.WriteLine("After catch statement.");
        }
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Standard message is:
System.IndexOutOfRangeException: Index was outside the bounds of
the array.
   at ExcTest.genException()
   at UseExcept.Main()
Stack trace:      at ExcTest.genException()
   at UseExcept.Main()
Message:  Index was outside the bounds of the array.
TargetSite: Void genException()
After catch statement.

```

**Exception**'da dört yapılandırıcı tanımlıdır. Bunlardan en sık kullanılan ikisi aşağıda gösterilmiştir:

```

Exception()
Exception(string str)

```

İlki varsayılan yapılandırıcıdır. İkincisi ise, söz konusu kural dışı durumla ilişkili **Message** özelliğini belirtir. Kendi kural dışı sınıflarınızı oluştururken bu yapılandırıcıların her ikisini de uygulamalısınız.

## Sıkça Kullanılan Kural Dışı Durumlar

**System** isim uzayında birkaç tane standart, derleyiciyle birlikle gelen kural dışı durum tanımlıdır. Bunların tümü **SystemException**'dan türetilmiştir, çünkü çalışma zamanında hatalar ortaya çıkınca bu kural dışı durumlar Common Language Runtime tarafından üretilirler,

C#'ta tanımlı standart kural dışı durumların daha sıkça kullanılanlarından birkaçı Tablo 13.1'de gösterilmektedir.

**TABLO 13.1: System İsim Uzayında Tanımlı, Sıkça Kullanılan Kural Dışı Durumlar**

Kural Dışı Durum	Anlamı
<b>ArrayTypeMismatchException</b>	Depolanmakta olan değerin tipi, dizinin tipiyle uyumsuzdur.
<b>DivideByZeroException</b>	Sıfıra bölme girişimi.
<b>IndexOutOfRangeException</b>	Dizi indeksi sınır dışına taşmıştır.
<b>InvalidCastException</b>	Programın çalışması sırasında bir tip ataması geçerli değildir.
<b>OutOfMemoryException</b>	Yeterince serbest bellek olmadığı için <b>new</b> çağrıları başarısız olmuştur.
<b>OverflowException</b>	Aritmetik taşıma meydana gelmiştir.
<b>NullReferenceException</b>	Null referans – yani, bir nesneye atıfta bulunmayan bir referans – üzerinde işlem yapılmaya çalışılmıştır.
<b>StackOverflowException</b>	Yığın taşmıştır.

Tablo 13.1'deki kural dışı durumların birçoğu kendi kendilerini açıklayacak kadar nettir. Yalnız, bunun tek istisnası **NullReferenceException** olabilir. Bu kural dışı durum, bir null referans sanki bir nesneye referansta bulunuyormuş gibi kullanılmaya çalışıldığında fırlatılır - söz gelişi, bir null referans üzerinden bir metodu çağrımayı denediğinizde... *Null referans*, herhangi bir nesneye işaret etmeyen bir referanstır. Null referans oluşturmanın bir yolu, söz konusu referansa **null** anahtar kelimesini kullanarak açıkça bir değer atamaktır. Null referanslar daha az belli olan diğer yöntemlerle de ortaya çıkabilir. İşte, **NullReferenceException**'ı gösteren bir program:

```
// NullReferenceException'ı kullanır.

using System;

class X {
    int x;
    public X(int a) {
        x = a;
    }

    public int add(X o) {
        return x + o.x;
    }
}

// NullReferenceException'ı göster.
class NREDemo {
```

```

public static void Main() {
    X p = new X(10);
    X q = null; // q'ya açıkça null değeri ataniyor
    int val;

    try {
        val = p.add(q); // bu, kural dışı bir duruma yol acacaktır
    }
    catch (NullReferenceException) {
        Console.WriteLine("NullReferenceException!");
        Console.WriteLine("fixing...\n");

        // simdi, bunu düzelt
        q = new X(9);
        val = p.add(q);
    }

    Console.WriteLine("val is {0}", val);
}
}

```

Programın çıktısı aşağıdaki gibidir:

```

NullReferenceException!
fixing...

val is 19

```

Programda **X** adında bir sınıf oluşturulur, **X**'in içinde **x** adında bir üye ve **add()** metodу tanımlanır. **add()**, kendisini çağıran nesnenin **x**'ini, parametre olarak aktarılan nesnenin **x**'i ile toplar. **Main()**'de iki **X** nesnesi oluşturulur. Bunlardan ilkine, yani **p**'ye ilk değer atanır. İkincisine, yani **q**'ya ilk değer atanmaz. Bunun yerine, **q**'ya açıkça **null** değeri atanır. Sonra, **q** argüman olacak şekilde **p.meth()** çağrıılır. **q** bir nesneye referansta bulunmadığı için **q.x** değerini elde etmeye yönelik bir girişimde bulunulunca **NullReferenceException**'ı üretilir.

Enteresan bir kural dışı durum ise, sistem yiğinında taşıma meydana geldiğinde fırlatılan **StackOverflowException**'ıdır. Böyle bir durum, yinelenen bir metot kontrolden çıktığında ortaya çıkabilir. Yinelenmeyi fazlaıyla kullanan bir program bu kural dışı durumu izlemek isteyebilir, böylece bu tür bir kural dışı durum meydana geldiğinde uygun davranışa bulunabilir. Yine de dikkatli olun. Bu kural dışı durum fırlatıldığında sistem yiğinını tamamen tüketilmiş olduğundan ötürü, yinelenen çağrılarından geri dönmeye başlamak genellikle yapılacak en iyi harekettir.

## Kural Dışı Durum Sınıflarını Türetmek

C#'ta tanımlı standart kural dışı durumlar sıkça rastlanan hataların birçoğunu kontrol altına alıyor olsa da, C#'ın kural dışı durum yönetim mekanizması yalnızca bu hatalarla sınırlı değildir. Aslında, kural dışı durumlara yönelik C#'ın sahip olduğu gücün bir parçası, sizin oluşturduğunuz kural dışı durumları kontrol altına alma becerisinden kaynaklanmaktadır. Kendi kodunuzdaki hataları kontrol altına almak için kendi oluşturduğunuz kural dışı durumları

kullanabilirsiniz. Kural dışı bir durum oluşturmak kolaydır. Yapmanız gereken yalnızca **Exception** sınıfından türetilen bir sınıf tanımlamaktır. Genel bir kural olarak, sizin tarafınızdan tanımlanan kural dışı durumlar **ApplicationException**'dan türetilmelidir, çünkü uygulamıyla bağlantılı kural dışı durumlar için ayrılmış hiyerarşi budur. Türetilmiş sınıflarınızın gerçekte bir şeyle uygulaması gerekmez - bunları kural dışı durum olarak kullanmanıza imkan veren bu kural dışı durumların tip sistemi içinde var olmalıdır.

Sizin oluşturduğunuz kural dışı durum sınıfları, **Exception** tarafından tanımlanan ve sizin oluşturduğunuz sınıfların kullanımına sunulan özelliklere ve metodlara otomatik olarak sahip olacaktır. Kuşkusuz bu üyelerin birini ya da daha fazlasını kendi oluşturduğunuz kural dışı durum sınıflarında devre dışı bırakabilirsiniz.

İşte size, kendi oluşturduğumuz bir kural dışı durum tipinden yararlanan bir örnek. Bölüm 10'un sonunda **RangeArray** adında bir dizi sınıfı geliştirilmiştir. Hatırlayacağınız gibi **RangeArray**, başlangıç ve bitiş indeksleri kullanıcı tarafından belirtilen tek boyutlu **int** dizilerini destekler. Söz gelisi, **-5** ile **27** arasında uzanan bir dizi **RangeArray** için kusursuz uygunluktadır. Bölüm 10'da eğer indeks sınır dışına taşmışsa, **RangeArray** tarafından tanımlanan özel bir hata değişkeninin değeri ayarlanıyordu. Bu, söz konusu hata değişkeninin her işlemden sonra **RangeArray**'i kullanan kod tarafından kontrol edilmesi gereği anlamına geliyordu. Bu tür bir yaklaşım elbette hatalara açık ve biçimsızdır. Çok daha iyi bir tasarım, bir aralık hatası ortaya çıktığında **RangeArray**'in kendi oluşturduğumuz bir kural dışı durum fırlatmasını sağlamaktır. **RangeArray**'in aşağıdaki versiyonda gerçekleştirilen tamamen budur:

```
// RangeArray hataları için ismarlama Exception kullanır.

using System;

// RangeArray için kural dışı bir durum oluştur.
class RangeArrayException : ApplicationException {
    // Standard yapılandırıcıları uygula
    public RangeArrayException() : base() { }
    public RangeArrayException(string str) : base(str) { }

    // RangeArrayException için ToString'i devre dışı bırak.
    public override string ToString() {
        return Message;
    }
}

// RangeArray'in geliştirilmiş bir versiyonu.
class RangeArray {
    // private veriler
    int[] a; // temel oluşturan diziye referans
    int lowerBound; // en küçük indeks
    int upperBOund; // en büyük indeks

    int len; // Length özelliği için temel olarak kullanılan değişken
    // Büyüklüğü verilen diziyi kur.
```

```

public RangeArray(int low, int high) {
    high++;
    if(high <= low) [
        throw new RangeArrayException("Low index not less
                                         than high.");
    ]
    a = new int[high - low];
    len = high - low;

    lowerBound = low;
    upperBound = --high;
}

// Salt okunur Length ozelligi.
public int Length {
    get {
        return len;
    }
}

// Bu, RangeArray icin indeksleyicidir.
public int this[int index] {
    // Bu, get erisimcisisidir.
    get {
        if(ok(index)) {
            return a[index - lowerBound];
        } else {
            throw new RangeArrayException("Range Error.");
        }
    }

    // Bu, set erisimcisisidir.
    set {
        if(ok(index)) {
            a[index - lowerBound] = value;
        }
        else throw new RangeArrayException("Range Error.");
    }
}

// Indeks sinirlar icindeyse true dondur.
private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
}
}

// Indeks-araklik dizisini goster.
class RangeArrayDemo {
    public static void Main() {
        try {
            RangeArray ra = new RangeArray(-5, 5);
            RangeArray ra2 = new RangeArray(1, 10);

            // ra'yi goster

```

```
Console.WriteLine("Length of ra: " + ra.Length);

for(int i = -5; i <= 5; i++)
    ra[i] = i;

Console.Write("Contents of ra: ");
for(int i = -5; i <= 5; i++)
    Console.Write(ra[i] + " ");

Console.WriteLine("\n");

// ra2'yi goster
Console.WriteLine("Length of ra2: " + ra2.Length);

for(int i = 1; i <= 10; i++)
    ra2[i] = i;

Console.Write("Contents of ra2: ");
for(int i = 1; i <= 10; i++)
    Console.Write(ra2[i] + " ");

Console.WriteLine("\n");

} catch (RangeArrayException exc) {
    Console.WriteLine(exc);
}

// Simdi birkac hata goster.
Console.WriteLine("Now generate some range errors.");

// Gecersiz bir yapılandırıcı kullan.
try {
    RangeArray ra3 = new RangeArray(100, -10); // Hata

} catch (RangeArrayException exc) {
    Console.WriteLine(exc);
}

// Gecersiz bir indeks kullan.
try {
    RangeArray ra3 = new RangeArray(-2, 2);

    for(int i = -2; i <= 2; i++)
        ra3[i] = i;

    Console.Write("Contents of ra3: ");
    for(int i = -2; i <= 10; i++) // aralık hatası oluştur
        Console.Write(ra3[i] + " ");

} catch (RangeArrayException exc) {
    Console.WriteLine(exc);
}
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Length of ra: 11
Contents of ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Length of ra2: 10
Contents of ra2: 1 2 3 4 5 6 7 8 9 10

Now generate some range errors.
Low index not less than high.
Contents of ra3: -2 -1 0 1 2 Range Error.
```

Bir aralık hatası ortaya çıktığında **RangeArray**, **RangeArrayException** tipinde bir nesne fırlatır. Bu sınıf **ApplicationException**'dan türetilmiştir. Önceden açıklandığı gibi, sizin oluşturduğunuz bir kural dışı durum sınıfı normal olarak **ApplicationException**'dan türetilmelidir. Bir **RangeArray**'ın yapılandırılması sırasında aralık hatasının ortaya çıkabileceğine dikkat edin. Bu tür kural dışı durumları yakalamak, programda da gösterildiği gibi **RangeArray** nesnelerinin **try** bloğu içinde kurulmalarını gerektirir. **RangeArray**, hataları rapor etmek için bir kural dışı durum kullanarak artık tipki C#'ın standart tiplerinden biri gibi davranmaktadır; bu nedenle, programın kural dışı durum yönetim mekanizmasına tam olarak entegre edilebilir.

Biraz daha ilerlemeden önce, bu programı biraz daha denemek isteyebilirsiniz. Örneğin, **ToString()**'in devre dışı bırakıldığı ifadeyi açıklama haline büründürerek programdan çıkarmayı deneyin ve sonuçları gözleyin. Ayrıca, varsayılan yapılandırıcıyı kullanarak bir kural dışı durum oluşturmaya çalışın ve C#'ın ürettiği varsayılan mesajı gözlemleyin.

## Türetilmiş Sınıflarla İlgili Kural Dışı Durumları Yakalamak

Temel ve türetilmiş sınıfları içeren kural dışı durum tiplerini yakalamaya çalışırken **catch** ifadelerini nasıl sıraladığınıza dikkat etmeniz gereklidir, çünkü bir temel sınıfı karşılık gelen bir **catch** ifadesi aynı zamanda temel sınıfın türetilmiş herhangi bir sınıf ile de eşlenecektir. Örneğin, tüm kural dışı durumların temel sınıfı **Exception** olduğu için **Exception**'ı yakalamak tüm olası kural dışı durumları da yakalamak demektir. Elbette, **catch**'i argümansız kullanmak, daha önceden de bahsedildiği gibi kural dışı durumların tümünü yakalamak için daha net bir yöntemdir. Bununla birlikte, türetilmiş sınıflarla ilgili kural dışı durumları yakalamak diğer bağlamlarda, özellikle kendi kural dışı durumlarınızı oluşturduğunuzda çok önemli olmaktadır.

Eğer hem temel sınıf tipindeki hem de türetilmiş sınıf tipindeki kural dışı durumları yakalamak istiyorsanız, **catch** sekansı içine önce türetilmiş sınıfı yerleştirin. Bu şekilde yapmazsanız, temel sınıf ayrıca türetilmiş sınıfların tümünü yakalayacaktır. Bu kuralı uygulamak için sizin bir şeyler yapmanız gereklidir, kural kendiliğinden uygulanır; çünkü, önce temel sınıfı yerleştirmek, türetilmiş sınıfı karşılık gelen **catch** ifadesi asla

gerçeklenmeyeceği için erişilemeyecek kodun oluşmasına neden olabilir. C#'ta erişilemeyen **catch** ifadesi bir hatadır.

Aşağıdaki program **ExceptA** ve **ExceptB** adında iki kural dışı durum sınıfı oluşturmaktadır. **ExceptA**, **ApplicationException**'dan türetilmektedir. **ExceptB** ise **ExceptA**'dan türetilmektedir. Program daha sonra her iki tipte bir kural dışı durum fırlatmaktadır.

```
// Turetilmis kural disi durumlar temel sinifla ilgili kural disi
// durumlardan once gelmelidir.

using System;

// Kural disi bir durum olustur.
class ExceptA : ApplicationException {
    public ExceptA() : base() { }
    public ExceptA(string str) : base(str) { }

    public override string ToString() {
        return Message;
    }
}

// ExceptA'dan turetilen kural disi bir durum olustur
class ExceptB : ExceptA {
    public ExceptB() : base() { }
    public ExceptB(string str) : base(str) { }

    public override string ToString() {
        return Message;
    }
}

class OrderMatters {
    public static void Main() {
        for(int x = 0; x < 3; x++) {
            try {
                if(x == 0) throw new ExceptA("Caught an
                    ExceptA exception");
                else if(x == 1) throw new ExceptB("Caught an ExceptB
                    exception");
                else throw new Exception();
            }
            catch (ExceptB exc) {
                // kural disi durumu yakala
                Console.WriteLine(exc);
            }
            catch (ExceptA exc) {
                // kural disi durumu yakala
                Console.WriteLine(exc);
            }
            catch (Exception exc) {
                Console.WriteLine(exc);
            }
        }
    }
}
```

```
        }  
    }  
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Caught an ExceptA exception  
Caught an ExceptB exception  
System.Exception: Exception of type System.Exception was thrown.  
at OrderMatters.Main()
```

**catch** ifadelerinin sırasına dikkat edin. Bu ifadeler yalnızca bu şekilde sıralanabilir. **ExceptB**, **ExceptA**'dan türetildiği için **ExceptB** için catch ifadesi **ExceptA**'ya karşılık gelenden önce gelmelidir. Aynı şekilde, **Exception** (tüm kural dışı durumların temel sınıfı) için **catch** de en son olarak yer almalıdır. Bunu kendi kendinize kanıtlamak için **catch** ifadelerini yeniden düzenlemeyi deneyin. Bu şekilde davranışın derleme hatasıyla sonuçlanacaktır.

Kural dışı durumlar kategorisini bütünüyle yakalamak, temel sınıf'a ait **catch** ifadesinin uygun kullanımlarından biridir. Örneğin, diyalim ki bir aygit için birtakım kural dışı durumlar oluşturuyorsunuz. Eğer kural dışı durumların tümünü ortak bir temel sınıfından türetirseniz, ne tür bir problemin ortaya çıktığını tam olarak bilmeleri gerekmeyen uygulamaların temel sınıfla ilgili kural dışı durumu yakalamaları yeterlidir. Böylece kodun gereksiz yere tekrarlanması önlenmiş olur.

## checked ve unchecked Kullanımı

C#'ta ekstra bir özellik, aritmetik hesaplamalarda taşıma ile ilgili kural dışı durumların üretilmesiyle bağlantılıdır. Bildiginiz gibi, bazı aritmetik hesaplama türlerinde hesaplamaya katılan veri tiplerinin aralığını aşan bir sonucun üretilmesi mümkün değildir. Böyle bir durum ortaya çıkınca sonucun *taştığı* (overflow) söylenir. Örneğin, aşağıdaki sekansı ele alın:

```
byte a, b, result;  
a = 127;  
b = 127;  
  
result = (byte) (a * b);
```

Burada **a** ve **b**'nin çarpımı **byte** değerinin menzilini aşar. Böylece sonuç, sonucun tipini aşmaktadır.

C#, **checked** ve **unchecked** anahtar kelimelerini kullanarak taşıma durumunda kodunuzun kural dışı bir durum fırlatıp fırlatmayıacağını belirlemenize imkan verir. Bir deyimin taşımaya karşı kontrol edileceğini belirtmek için **checked** kullanın. Taşmanın dikkate alınmayacağı belirtmek için **unchecked** kullanın. Bu örnekte sonuç, deyimin hedef tipine sığacak şekilde kesiliyor.

**checked** anahtar kelimesi aşağıda gösterilen iki genel yapıya sahiptir. Bunlardan biri belirli bir deyimi kontrol eder ve buna **checked**'in *operatör formu* denir, Diğer ise bir ifade bloğunu kontrol eder.

```
checked (deyim)

checked {
    // kontrol edilecek ifadeler
}
```

Burada deyim kontrol edilmekte olan deyimdir. Eğer kontrol edilen bir deyim taşırsa bir **OverflowException** fırlatılır.

**unchecked** anahtar kelimesinin de aşağıdaki gibi iki genel yapısı mevcuttur. Bunlardan biri, belirli bir deyimin taşmasını dikkate almayan operatör formudur. Diğer ise, bir ifade bloğunun taşmasını dikkate almaz.

```
unchecked (deyim)

unchecked {
    // taşmanın dikkate alınmayacağı ifadeler
}
```

Burada **deyim**, taşıma durumu kontrol edilmemekte olan deyimdir. Eğer kontrol edilmeyen bir deyim taşırsa, taşan kısmın kesilmesi söz konusu olacaktır. İşte, **checked** ve **unchecked**'ı birlikte gösteren bir program:

```
// checked ve unchecked kullanır.

using System;

class CheckedDemo {
    public static void Main() {
        byte a, b;
        byte result;

        a = 127;
        b = 127;

        try {
            result = unchecked((byte)(a * b));
            Console.WriteLine("Unchecked result: " + result);

            result = checked((byte)(a * b));
            // Bu, kural dışı duruma neden olur
            Console.WriteLine("Checked result: " + result);
        }
        catch (OverflowException exc) {
            // kural dışı durumu yakala
            Console.WriteLine(exc);
        }
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Unchecked result: 1
System.OverflowException: Arithmetic operation resulted in an
overflow.
at CheckedDemo.Main()
```

Apaçık görüldüğü üzere, kontrol edilmeyen deyim sonucun kesilmesine neden olmuştur. Kontrol edilen deyimi ise bir kural dışı duruma yol açmıştır.

Yukarıdaki program tek bir deyim için **checked** ve **unchecked**'in kullanımını göstermiştir. Aşağıdaki program ise **checked** ve **unchecked**'in bir ifade bloğu ile nasıl kullanıldığını göstermektedir.

```
// checked ve unchecked'in ifade bloklarıyla birlikte kullanımı.

using System;

class CheckedBlocks {
    public static void Main() {
        byte a, b;
        byte result;

        a = 127;
        b = 127;

        try {
            unchecked {
                a = 127;
                b = 127;
                result = unchecked((byte)(a * b));
                Console.WriteLine("Unchecked result: " + result);

                a = 125;
                b = 5;
                result = unchecked((byte)(a * b));
                Console.WriteLine("Unchecked result: " + result);
            }
            checked {
                a = 2;
                b = 7;
                result = checked((byte)(a * b)); // bu, tamam
                Console.WriteLine("Checked result: " + result);

                a = 127;
                b = 127;
                result = checked((byte)(a * b));
                // bu, kural dışı duruma neden olur
                Console.WriteLine("Checked result: " + result);
            }
        }
        catch (OverflowException exc) {
            // kural dışı durumu yakala
            Console.WriteLine(exc);
        }
    }
}
```

```
        }  
    }  
}
```

Programın çıktısı aşağıdaki gibidir:

```
Unchecked result: 1  
Unchecked result: 113  
Checked result: 14  
System.OverflowException: Arithmetic operation resulted in an overflow.  
at CheckedBlocks.Main()
```

Gördüğünüz gibi; kontrol edilmeyen blok, sonucun kesilmesine neden olan bir taşmayla sonuçlanmaktadır. Kontrol edilen blok içinde bir taşma meydana gelince kural dışı bir durum ortaya çıkmıştır.

**checked** veya **unchecked** kullanmaya gerek duyabilecek olmanızın bir nedeni şudur; Taşmanın kontrollü ya da kontrollsüz olması durumu, bir derleyici seçeneği ayarlanarak ve programın çalışma ortamının kendisi tarafından belirlenir. Bu nedenle, bazı program türleri için en iyisi, taşma kontrol durumunu açıkça belirtmektir.

## I/O KULLANIMI

Elinizdeki kitabın ilk bölümlerinde C# I/O (giriş/çıkış) sisteminin bazı parçaları, söz gelisi `Console.WriteLine()` kullanılmıştı; fakat, bu bölümlerde pek fazla özel açıklama sağlanmamıştı. C# I/O sistemi sınıf hiyerarşisi üzerine kurulduğu için öncelikle sınıfları, kalıtımı ve kural dışı durumları ele almadan C# I/O sistemini teorisi ve ayrıntılarıyla sunmak mümkün olmayacaktı. Şimdi C#'ın I/O'ya yönelik yaklaşımını ayrıntılarıyla incelemenin vakti geldi. C#, .NET Framework tarafından tanımlanan I/O sistemini ve ilgili sınıfları kullandığından dolayı, I/O'nun C# bünyesinde ele alınması aynı zamanda .NET I/O sisteminin de genel olarak ele alınması demektir.

Bu bölümde C#'ın hem konsol I/O, hem de dosya I/O'su ile ilgili yaklaşımları incelenmektedir. C#'ın I/O sisteminin oldukça büyük olduğu konusunda sizi önceden uyardı. Bu bölümde en önemli ve en çok kullanılan özellikler anlatılmaktadır.

## C#'ın I/O Sistemi, Akışlar Üzerine Kurulmuştur

C# programları I/O'yı akışlar aracılığıyla gerçekleştirirler. *Akış (stream)*, bir bilgi üreten ya da tüketen bir soyutlamadır. Bir akış, C# I/O sistemi tarafından fiziksel bir aygıta bağlanır. Tüm akışlar, bağlı oldukları fiziksel cihazlar farklılık gösterse dahi, aynı şekilde davranışmalıdır. Böylece, I/O sınıfları ve metodları pek çok tipte cihaza uygulanabilir. Örneğin, konsola bir şeyler yazdırınmak için kullandığınız metodların aynıları, disk dosyasına yazdırınmak için de kullanılabilir.

### Byte Akışları ve Karakter Akışları

En alt düzeyde, tüm C# I/O Sistemleri byte'lar üzerinde işlem görür. Bu akla yatkındır, çünkü I/O işlemleri açısından düşünüldüğünde aygıtların birçoğu byte tabanlıdır. Yine de, biz insanoğlu iletişim için karakterleri kullanmayı sıkça tercih ederiz. C#'ta `char` tipinin 16 bit, `byte` tipinin ise 8 byte olduğunu hatırlayın, Eğer ASCII karakter setini kullanıyorsanız `char` ve `byte` arasında kolaylıkla dönüşüm yapabilirsiniz; `char` değerinin üst (high-order) byte'ını dikkate almamanız yeterlidir. Fakat bu yaklaşım geriye kalan Unicode karakterleri için işe yaramaz, çünkü bu karakterler için her iki byte'ı da gereklidir. Bu nedenle, byte akışları karakter tabanlı I/O işlemlerini kontrol altına almak için tam olarak uygun değildir. Bu problemi çözmek amacıyla C#'ta bir byte akışını, karakter akışına dönüştüren birkaç sınıf tanımlıdır. Bu sınıflar, `byte`'tan `char`'a ve `char`'dan `byte`'a dönüşümleri sizin adınıza otomatik olarak ele alırlar.

### Önceden Tanımlı Akışlar

`System` isim uzayını kullanan programların tümünün kullanımına hazır olan, önceden tanımlı üç akış mevcuttur. Bu akışlar `Console.In`, `Console.Out` ve `Console.Error` adındaki özellikler aracılığıyla ortaya çıkar. `Console.Out` standart çıktı akışıyla ilgilidir. Varsayılan çıktı akışı konsoldur. `Console.WriteLine()`'ı çağrıdığınızda örneğin, enformasyon otomatik olarak `Console.Out`'a gönderilir. `Console.In`, standart giriş ile

ilgilidir. Varsayılan standart giriş, klavyedir. `Console.Error`, standart hata akışı ile ilgilidir. Bunun için de varsayılan akış konsoldur. Ancak, bu akışlar, uyumlu herhangi bir I/O cihazına yönlendirilebilirler. Standart akışlar karakter akışlarıdır. Bu sayede; bu akışlar, karakterleri okur ve yazarlar.

## Akış Sınıfları

C#'ta hem byte akışı hem de karakter akışı sınıfları tanımlıdır. Bununla birlikte, karakter akışı sınıfları gerçekte, alta yatan byte akışını bir karakter akışına dönüştüren ve dönüşümleri otomatik olarak ele alan bir ambalajdan ibarettir. Bu nedenle, karakter akışları mantıksal olarak ayrı olmakla birlikte byte akışları üzerine kurulurlar.

Tüm akış sınıfları `System.IO` isim uzayı içinde tanımlıdır. Bu sınıfları kullanmak için aşağıdaki ifadeyi genellikle programınızın en üstüne yakın bir yerlere yerlestireceksiniz:

```
using System.IO;
```

Konsol giriş ve çıkışları için `System.IO`'yu belirtmek zorunda olmamanızın nedeni `Console` sınıfının `System` isim uzayı içinde tanımlı olmasıdır.

## Stream Sınıfı

C# akışlarının özünde `System.IO.Stream` yer alır. `Stream`, byte akışını simgeler, ayrıca tüm diğer akış sınıfları için temel sınıf işlevini görür. `Stream` aynı zamanda özettir; yani, bir `Stream` nesnesi örnekleyemezsiniz. `Stream`, birtakım standart akış işlemleri tanımlar. Tablo 14.1'de `Stream` tarafından tanımlanmış sıkça kullanılan birkaç metot gösterilmektedir.

**TABLO 14.1: Stream Tarafından Tanımlanan Metotlardan Bazıları**

Metot	Açıklama
<code>void Close()</code>	Akışı kapatır.
<code>void Flush()</code>	Akışın içeriğini fiziksel cihaza yazar.
<code>int ReadByte()</code>	Girdinin bir sonraki uygun byte'inin tamsayı gösterimi döndürür. Dosyanın sonuna ulaşılinca <b>-1</b> döndürür.
<code>int Read(byte[] tampon, int öteleme, int byteSayısı)</code>	<code>tampon[öteleme]</code> 'den başlayarak <code>tampon</code> 'a <code>byteSayısı</code> kadar byte okumaya çalışır; başarıyla okunan byte sayısını döndürür.
<code>long seek(long öteleme, SeekOrigin orijin)</code>	Akış içindeki mevcut konumu, belirtilen <code>orijin</code> 'den belirtilen <code>öteleme</code> kadar ileriye ayarlar.
<code>void WriteByte(byte b)</code>	Bir çıktı akışına tek bir byte yazar.
<code>void Write(byte[] tampon, int öteleme, int byteSayısı)</code>	<code>tampon[öteleme]</code> 'den başlayarak <code>tampon</code> dizisinden <code>byteSayısı</code> kadar byte'tan oluşan bir alt aralığı yazar.

Genel olarak, eğer bir I/O hatası meydana gelirse, Tablo 14.1'de gösterilen metodlar bir **IOException** fırlatacaklardır. Eğer geçersiz bir işlem gerçekleştirilmeye çalışılıyorsa, örneğin salt okunur bir akışa yazmaya çalışmak gibi, bir **NotSupportedException** fırlatılır.

**Stream** tarafından verileri okumak ve yazmak için metodlar tanımlandığına dikkat edin. Ancak, akışların tümü bu işlemlerin her ikisini de desteklemeyeceklerdir, çünkü salt okunur ya da salt yazılır akışlar açmak mümkündür. Ayrıca, **Seek()** üzerinden konum talepleri de akışların tümü tarafından desteklenmeyecektir. Bir akışın becerilerini belirlemek için **Stream** özelliklerinden bir veya daha fazlasını kullanacaksınız. Bu özellikler Tablo 4.2'de gösterilmiştir. Akışın uzunluğunu ve mevcut konumunu içeren **Length** ve **Position** özellikleri de ayrıca gösterilmiştir.

## Byte Stream Sınıfları

**Stream**'den türetilen üç somut byte akış sınıfı aşağıda gösterilmiştir:

Akış Sınıfı	Açıklama
<b>BufferedStream</b>	Bir byte akışını sarar ve tampon bellek ekler. Tampon bellek birçok durumda performans artışı sağlar.
<b>FileStream</b>	Dosya I/O işlemleri için tasarlanmış bir byte akışı.
<b>MemoryStream</b>	Depolama için bellek kullanan bir byte akışı.

Kendi akış sınıflarınızı türetmeniz de mümkündür. Yine de, uygulamaların büyük çoğunluğu için standart akışlar yeterli olacaktır.

**TABLO 14.2: Stream Tarafından Tanımlanan Özellikler**

Metot	Açıklama
<b>bool CanRead</b>	Eğer akış okunabiliyorsa bu özellik doğru değerine sahiptir. Bu salt okunur bir özelliklektir.
<b>bool CanSeek</b>	Eğer akış, konum taleplerini destekliyorsa bu özellik doğru değerine sahiptir. Bu salt okunur bir özelliklektir.
<b>bool CanWrite</b>	Eğer akış yazılabiliyorsa bu özellik doğru değerine sahiptir. Bu salt okunur bir özelliklektir.
<b>long Length</b>	Bu özellik, akışın uzunluğunu içerir. Bu salt okunur bir özelliklektir.
<b>long Position</b>	Bu özellik, akışın mevcut konumunu simgeler. Bu okunur/yazılır bir özelliklektir.

## Karakter Akışını Saran Sınıflar

Bir karakter akışı oluşturmak için bir byte akışını C#'ın karakter akışı ambalajlarından birinin içine sarın. Karakter akışı hiyerarşisinin en üstünde birer özet sınıf olan **TextReader** ve **TextWriter** yer almaktadır. **TextReader** girdiyi; **TextWriter** ise çıktıyi ele alır. Bu iki özet sınıf tarafından tanımlanan metodlar, bu sınıfların tüm alt sınıflarının kullanımına hazırlıdır. Böylece bu sınıflar, tüm karakter akışlarının sahip olacağı minimal bir I/O fonksiyon kümesi oluşturmuş olurlar.

**TextReader** içindeki girdi metodları Tablo 14.3'te gösterilmiştir. Genel olarak, bu metodlar hata durumunda bir **IOException** fırlatabilirler. (Bazları diğer türde kural dışı durumları da fırlatabilirler.) Bir metin satırını bütünüyle okuyan ve bunu bir **string** olarak döndüren **ReadLine()** özellikle ilginçtir. Bu metod, içinde gömülü boşluklar olan girdileri okurken işe yarar.

**TABLO 14.3: TextReader Tarafından Tanımlanan Girdi Metotları**

Metot	Açıklama
<b>void Close()</b>	Girdi kaynağını kapatır.
<b>int Peek()</b>	Girdi akışında bir sonraki karakteri alır fakat karakteri girdiden çıkarmaz. Eğer hiç karakter mevcut değilse <b>-1</b> döndürür.
<b>int Read()</b>	Metodu çağrıran girdi akışındaki bir sonraki uygun karakterin tamsayı gösterimini döndürür. Akışın sonuna ulaşılınca <b>-1</b> döndürür.
<b>int Read(char[] tampon, int öteleme, int karakterSayısı)</b>	<b>tampon[öteleme]</b> 'den başlayarak <b>tampon'a karakterSayısı</b> kadar karakter okumaya çalışır; başarıyla okunan karakter sayısını döndürür.
<b>int ReadBlock(char[] tampon, int öteleme, int karakterSayısı)</b>	<b>tampon[offset]</b> 'ten başlayarak <b>tampon'a karakterSayısı</b> kadar karakter okumaya çalışır; başarıyla okunan karakter sayısını döndürür.
<b>string ReadLine()</b>	Bir sonraki metin satırını okur ve bunu bir karakter katarı olarak döndürür. Dosya sonundan okumaya çalışıldığında <b>null</b> döndürülür.
<b>string ReadToEnd()</b>	Bir akışta geriye kalan karakterlerin tümünü okur ve bunları bir karakter katarı olarak döndürür.

**TextReader**, tüm standart tiplerin çıktılarını alan **Write()** ve **WriteLine()**'ın çeşitli versiyonlarını tanımlamaktadır. Örneğin, bunların aşırı yüklenmiş birkaç versiyonu şöyledir:

Metot	Açıklama
<code>void Write(int değer)</code>	Bir <code>int</code> yaz.
<code>void Write(double değer)</code>	Bir <code>double</code> yaz.
<code>void Write(bool değer)</code>	Bir <code>bool</code> yaz.
<code>void WriteLine(string değer)</code>	Ardından yeni bir satır gelen bir <code>string</code> yaz.
<code>void WriteLine(uint değer)</code>	Ardından yeni bir satır gelen bir <code>uint</code> yaz.
<code>void WriteLine(char değer)</code>	Ardından yeni bir satır gelen bir karakter yaz.

`Write()` ve `WriteLine()`'a ek olarak `TextWriter` ayrıca aşağıda gösterilen `Close()` ve `Flush()` metodlarını da tanımlamaktadır:

```
virtual void Close()
virtual void Flush()
```

`Flush()`, çıktı tamponunda kalan verilerin fiziksel ortama yazılmasına neden olur. `Close()` akışı kapatır.

`TextReader` ve `TextWriter` sınıfları, aşağıdaki tabloda gösterilenler de dahil karakter tabanlı birkaç akış sınıfı tarafından uygulanır. Böylece, bu akışlar `TextReader` ve `TextWriter` tarafından belirtilen metotları ve özellikleri sağlamış olurlar.

Akış Sınıfı	Açıklama
<code>StreamReader</code>	Bir byte akışından karakterleri okur. Bu sınıf, bir byte girdi akışını sarar.
<code>StreamWriter</code>	Bir byte akışına karakterleri yazar. Bu sınıf, bir byte çıktı akışını sarar.
<code>StringReader</code>	Bir karakter katarından karakterleri okur.
<code>StringWriter</code>	Bir karakter katarına karakterleri yazar.

## İkili Akışlar

Byte akışlarına ve karakter akışlarına ek olarak C#'ta ikili verileri doğrudan okumak ve yazmak için kullanılabilecek iki adet ikili akış sınıfı da tanımlıdır. Bu akışlar `BinaryReader` ve `BinaryWriter` olarak adlandırılır. Bu bölümün ileriki sayfalarında ikili dosyaların I/O işlemleri ele alınırken bunları daha yakından inceleyeceğiz.

Artık C# I/O sisteminin genel yapısını kavradığınıza göre, bölümün geri kalanında konsol I/O işlemlerinden başlayarak C# I/O sisteminin çeşitli parçalarını ayrıntılı olarak inceleyeceğiz.

## Konsol I/O

Konsol I/O; `Console.In`, `Console.Out` ve `Console.Error` standart akışları aracılığıyla gerçekleştirilir. Konsol I/O'yu Bölüm 2'den beri kullanmaktasınız; bu nedenle, buna zaten aşinasınız. Konsol I/O'nun ilave bazı becerilere sahip olduğunu daha sonra göreceksiniz.

Konuya başlamadan önce bu kilapta daha önce bahsedilen bir hususu vurgulamak önemlidir: C#'ın gerçek uygulamalarının birçoğu, metin tabanlı konsol programları olmayacağıdır. Bunun yerine; bu uygulamalar, grafik tabanlı programlardan oluşacak ya da kullanıcı ile etkileşim için penceredeli bir arayüzü temel alan bileşenler içerecektir. Bu nedenle, C#'ın konsol girdi ve çıktısına dayanan I/O sistemi yaygın olarak kullanılmaz. Metin tabanlı programlar, kısa yardımcı programlar ve bazı bileşen tipleri için öğretici örnekler olarak kusursuz olsalar da, gerçek dünyaya ait uygulamaların pek çoğu için uygun değildirler.

### Konsol Girdisini Okumak

`Console.In`, `TextReader`'ın bir örneğidir; buna erişmek için `TextReader` tarafından tanımlanan metodları ve özellikleri kullanabilirsiniz. Bununla birlikte, genellikle, `Console.In`'den otomatik olarak okuyan ve `Console` tarafından sağlanan metodları kullanacaksınız. `Console`'da iki girdi metodu tanımlıdır: `Read()` ve `ReadLine()`.

Tek bir karakter okumak için `Read()` metodunu kullanın:

```
static int Read()
```

`Read()`, konsoldan okunan bir sonraki karakteri döndürür. `Read()`, kullanıcı bir tuşa basana kadar bekler ve sonra sonucu döndürür. Karakter `int` olarak döndürülür; bu değer, tip ataması yoluyla `char`'a dönüştürülmelidir. Hata durumunda `Read()`, `-1` döndürür. Bu metod, başarısızlık durumunda bir `IOException` fırlatacaktır. Konsol girdileri, satır satır tampona yüklenir; bu nedenle, tuşladığınız herhangi bir karakterin programınıza gönderilmesinden önce ENTER tuşuna basmalısınız.

İşte, `Read()` kullanarak klavyeden bir karakter okuyan bir program:

```
// Klavyeden bir karakter okur.

using System;

class KbIn {
    public static void Main() {
        char ch;

        Console.Write("Press a key followed by ENTER: ");
        ch = (char) Console.Read(); // bir karakter al
        Console.WriteLine("Your key is: " + ch);
    }
}
```

```
    }  
}
```

İste, programın çalışmasına bir örnek:

```
Press a key followed by ENTER: t  
Your Key is: t
```

**Read()**'in satır satır tampona okuması kimi zaman sinir bozucu da olabilir. ENTER tuşuna bastığınızda carriage return ve linefeed sekansı girdi akışına eklenir. Üstelik, bu karakterler, siz onları okuyana kadar girdi tamponunda asılı kalırlar. Bu nedenle, bazı uygulamalarda bir sonraki girdi işleminden önce bunları (okuyarak) çıkarmanız gerekebilir.

Bir karakter katarını okumak için **ReadLine()** metodunu kullanın. Metot şu şekildedir:

```
static string ReadLine()
```

**ReadLine()**, siz ENTER tuşuna basana kadar karakterleri okur ve okunan karakterleri bir **string** nesnesi olarak döndürür. Başarısızlık durumunda bu metot bir **IOException** fırlatacaktır.

İste, **ReadLine()** kullanarak **Console.In**'den bir karakter katarı okumayı gösteren bir program:

```
// ReadLine() kullanarak konsoldan girdi al.  
  
using System;  
  
class ReadString {  
    public static void Main() {  
        string str;  
  
        Console.WriteLine("Enter some characters.");  
        str = Console.ReadLine();  
        Console.WriteLine("You entered: " + str);  
    }  
}
```

İste, programın çalışmasına bir örnek:

```
Enter some characters.  
This is a test.  
You entered: This is a test.
```

**Console** metodları **Console.In**'den okumanın en kolay yolu olmalarına rağmen, altta yatan **TextReader** üzerinden de metodları çağırabilirsiniz. Örneğin, işte yukarıdaki programın **TextReader** tarafından tanımlanmış metodlar kullanılarak yeniden yazılmış versiyonu:

```
// Dogrudan Console.In kullanarak klavyeden bir karakter katari oku.  
  
using System;  
  
class ReadChars2 {
```

```

public static void Main() {
    string str;

    Console.WriteLine("Enter some characters.");
    str = Console.In.ReadLine();

    Console.WriteLine("You entered: " + str);
}
}

```

`ReadLine()`'ın şimdi doğrudan `Console.In` üzerinden nasıl çağrıldığına dikkat edin. Buradaki temel nokta şudur: `Console.In`'in altında yatan `TextReader` tarafından tanımlanmış metodlara erişmeniz gerekiyorsa, bu metodları bu örnekte gösterildiği gibi çağıracaksınız.

## Konsol Çıktısı Yazmak

`Console.Out` ve `Console.Error`, `TextWriter` tipinde nesnelerdir. Konsol çıktısı en kolay şekilde, sizin önceden aşina olduğunuz `Write()` ve `WriteLine()` ile gerçekleştirilebilir. Standart tiplerin her birinin çıktısını almak için bu metodların versiyonları mevcuttur. `Console`, `Write()` ve `WriteLine()`'ın kendi versiyonlarını tanımlar; böylece bu metodlar, bu kitapta başından beri yapmakta olduğunuz gibi, doğrudan `Console` üzerinden çağrılabılır. Yine de, tercih ederseniz `Console.Out` ve `Console.Error`'nın altında yatan `TextWriter` üzerinden de bu metodları (ve diğerlerini) çağrılabilsiniz.

İşte, `Console.Out` ve `Console.Error`'a yazmayı gösteren bir program:

```

// Console.Out ve Console.Error'a yaz.

using System;

class ErrOut {
    public static void Main() {
        int a = 10, b = 0;
        int result;

        Console.Out.WriteLine("This will generate an exception.");
        try {
            result = a / b; // Kural disi bir durum uret
        } catch(DivideByZeroException exc) {
            Console.Error.WriteLine(exc.Message);
        }
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

This will generate an exception.
Attempted to divide by zero.

```

Programlamaya yeni başlayanlar bazen `Console.Error`'ı ne zaman kullanacaklarını karıştırırlar. Hem `Console.Out` hem de `Console.Error` çıktılarını konsola gönderdiklerine göre niye iki farklı isim kullanılıyor? Bu sorunun yanıtı, standart akışların diğer aygıtlara yönlendirilebilir olması çerçeğinde yatıyor. Örneğin, `Console.Error` ekran yerine disk dosyasına yazacak şekilde yönlendirilebilir. Böylece, konsol çıktısını etkilemeden hata çıktısını örneğin bir günlük (log) dosyasına yönlendirmeniz mümkündür. Tam tersi durumda, eğer konsol çıktısı yönlendirilmiş ve hata çıktısı yönlendirilmemişse hata mesajları konsolda, görülebilecekleri yerde gözükeceklerdir. Yönlendirmeleri daha sonra, dosya I/O işlemleri anlatıldıkten sonra inceleyeceğiz.

## FileStream ve Byte Yönetimli Dosya I/O İşlemleri

C# dosyaları okumanıza ve dosyalara yazmanızı imkan veren sınıflar sağlar. Elbette, disk dosyaları en yaygın dosya tiplerindendir. İşletim sistemi düzeyinde tüm dosyalar byte tabanlıdır. Tahmin edeceğiniz gibi, C# dosyalardan okumak ve dosyalara yazmak için metodlar içerir. Böylece, byte akışları kullanarak dosyalardan okumak ve dosyalara yazmak çok yaygındır. C# ayrıca byte tabanlı bir dosya akışını karakter tabanlı bir nesne içine sarmaniza da olanak tanır. Karakter tabanlı dosya işlemleri metinlerin saklanması sırasında işe yarar. Karakter akışları bu bölümün içinde ileriki sayfalarda ele alınmaktadır. Byte yönetimli I/O ise burada anlatılmaktadır.

Bir dosyaya ilişirilmiş byte tabanlı bir akış oluşturmak için `FileStream` sınıfını kullanacaksınız. `FileStream`, `Stream`'den türetilmiştir ve `Stream`'in tüm işlevselligine sahiptir.

Hatırlarsanız, `FileStream` de dahil olmak üzere akış sınıfları `System.IO` içinde tanımlanmaktadır. Bu nedenle genellikle, akış sınıflarını kullanan bir programın başına yakın bir yererde aşağıdaki ifadeyi dahil etmelisiniz.

```
using System.IO;
```

### Bir Dosyayı Açımak ve Kapatmak

Bir dosyaya bağlı bir byte akışı oluşturmak için bir `FileStream` nesnesi oluşturun. `FileStream`'de birkaç yapılandırıcı tanımlıdır. Belki de en yaygın olarak kullanılanı aşağıda gösterilmiştir:

```
FileStream(string dosyaadi, FileMode kip)
```

Burada `dosyaadi`, açılacak dosyanın adını belirtir. `dosyaadi`, dizin yolunun tam ifadesini içerebilir. `kip` parametresi, dosyanın nasıl açılacağını belirtir. `kip`, `FileMode` numaralandırması tarafından tanımlanan değerlerden biri olmalıdır. Bu değerler Tablo 14.4'te gösterilmiştir. Bu yapılandırıcı dosyayı okuma/yazma erişimine açar.

Dosayı açmaya çalışırken başarısız olunursa bir kural dışı durum fırlatılacaktır. Eğer dosya mevcut olmadığı için açılamıyorsa, **FileNotFoundException** fırlatılacaktır. Eğer dosya bir I/O hatasından dolayı açılamıyorsa, **IOException** fırlatılacaktır. Diğer olası kural dışı durumlar şunlardır: **ArgumentNullException** (dosya ismi mevcut olmadığında), **ArgumentException** (kip parametresi geçersiz olduğunda), **SecurityException** (kullanıcının erişim hakları olmadığından) ve **DirectoryNotFoundException** (belirtilen dizin geçersiz olduğunda).

**TABLO 14.4: FileMode Değerleri**

Değer	Açıklama
<b>FileMode.Append</b>	Çıktı dosyasının sonuna eklenir.
<b>FileMode.Create</b>	Yeni bir çıktı dosyası oluşturur. Aynı isimde önceden mevcut herhangi bir dosya yok edilecektir.
<b>FileMode.CreateNow</b>	Yeni bir çıktı dosyası oluşturur. Dosya önceden mevcut olmamalıdır.
<b>FileMode.Open</b>	Önceden mevcut bir dosyayı açar.
<b>FileMode.OpenOrCreate</b>	Dosya mevcutsa dosyayı açar ya da dosya önceden mevcut değilse dosyayı oluşturur.
<b>FileMode.Truncate</b>	Önceden mevcut bir dosyayı açar, fakat uzunluğunu sıfırı indirir.

Aşağıda, **test.dat** dosyasını girdi için açmayı gösteren bir yöntem sunuluyor:

```
FileStream fin;

try {
    fin = new FileStream("test.dat", FileMode.Open);
}
catch(FileNotFoundException exc) {
    Console.WriteLine(exc.Message);
    return;
}
catch {
    Console.WriteLine("Cannot open file.");
    return;
}
```

Bu örnekte ilk **catch** ifadesi dosyanın bulunamadığını belirten hatayı yakalıyor. “Tümünü yakala” türünde bir ifade olan ikinci **catch** ise, diğer olası dosya hatalarını kontrol altına alıyor. Her bir hatayı ayrı ayrı da kontrol edebilirdiniz. Bu durumda, meydana gelen problemler de daha spesifik olarak rapor edilmiş olurdu. İşleri basit tutmak açısından bu kitaptaki örnekler yalnızca **FileNotFoundException** ile **IOException**'ı yakalayacaklardır, fakat sizin gerçek kodunuzun, koşullara bağlı olarak diğer olası kural dışı durumları da kontrol altına alması gerekebilir.

Az önce anlatılan **FileStream** yapılandırıcısı, önceden de bahsedildiği gibi, okuma/yazma erişimine sahip olan bir dosyayı açar. Eğer erişimi yalnızca okuma ya da yazma olarak sınırlamak isterseniz, bu durumda öncekinin yerine şu yapılandırıcıyı kullanın:

```
FileStream(String dosyaadi, FileMode kip, FileAccess nasıl)
```

Önceki gibi burada da **dosyaadi**, açılacak dosyanın adını, **kip** ise dosyanın nasıl açılacağını belirtir. **nasıl** üzerinden aktarılan değer de, dosyanın nasıl erişileceğini belirler. Bu değer, **FileAccess** numaralandırması tarafından tanımlanan değerlerden biridir. **FileAccess** tarafından tanımlanan değerler aşağıda gösterilmiştir:

<code>FileAccess.Read</code>	<code>FileAccess.Write</code>	<code>FileAccess.ReadWrite</code>
------------------------------	-------------------------------	-----------------------------------

Örneğin, şu ifade salt okunur bir dosya açar:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,  
FileAccess.Read);
```

Bir dosyayla işiniz bittiğinde **Close()** metodunu çağırarak bunu kapatmalısınız. **Close()** genel olarak şu şekildedir:

```
void Close()
```

Bir dosyayı kapatmak, dosyaya tahsis edilen sistem kaynaklarını serbest bırakır; bu sayede, bu kaynakların başka bir dosya tarafından kullanılabilmesi mümkün olur. **Close()**, bir **IOException** fırlatabilir.

## Bir FileStream'den Byte'lar Okumak

**FileStream**'de bir dosyadan byte byte okuyan iki metot tanımlıdır: **ReadByte()** ve **Read()**. Bir dosyadan tek bir byte okumak için **ReadByte()**'ı kullanın. **ReadByte()** genel olarak şu şekildedir:

```
int ReadByte()
```

**ReadByte()**, her çağrılarında dosyadan tek bir byte okur ve bunu bir tamsayı değer olarak döndürür. Dosyanın sonuna ulaşınca **-1** döndürür. Olası kural dışı durumlar şunlardır: **NotSupportedException** (akış girdi için açılmamışsa) ve **ObjectDisposedException** (akış kapatılmışsa).

Bir byte bloğu okumak için **Read()** kullanın. Bunun genel şekli şöyledir:

```
int Read(byte[] tampon, int öteleme, int byteSayısı)
```

**Read()**, **tampon[öteleme]**'den başlayarak **tampon** içine **byteSayısı** kadar byte okumaya çalışır. Başarıyla okunan byte sayısı döndürülür. I/O hatası meydana geldiğinde bir **IOException** fırlatılır. Diğer tipten kural dışı durumlardan birkaçı da mümkündür. Okuma, akış tarafından desteklenmediğinde fırlatılan **NotSupportedException** da bunlardan biridir.

Aşağıdaki program, ismi komut satırı argümanı olarak belirtilen bir metin dosyasına girdi girmek ve dosyanın içeriğini göstermek için **ReadByte()** kullanmaktadır. Bu program çalışmaya başladığında ortaya çıkabilecek iki hatayı kontrol altına alan **try/catch** bloklarına dikkat edin: Belirtilen dosya mevcut olmayabilir ya da kullanıcı dosya ismini dahil etmeyi unutabilir. Komut satırı argümanlarını her kullanışınızda bu yöntemin aynısını kullanabilirsiniz.

```
/* Bir metin dosyasini gosterir.

Bu programi kullanmak icin, gormek istediginiz
dosyanin adini belirtin. Ornegin, TEST:CS
adinda bir dosyayı gormek icin asagidaki
komut satirini kullanin.

ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
    public static void Main(string[] args) {
        int i;
        FileStream fin;
        try {
            fin = new FileStream(args[0], FileMode.Open);
        } catch(FileNotFoundException exc) {
            Console.WriteLine(exc.Message);
            return;
        } catch(IndexOutOfRangeException exc) {
            Console.WriteLine(exc.Message + "\nUsage: ShowFile File");
            return ;
        }

        // EOF'a rastlayana kadar byte'lari oku
        do {
            try {
                i = fin.ReadByte();
            } catch(Exception exc) {
                Console.WriteLine(exc.Message);
                return ;
            }
            if(i != -1) Console.Write((char) i);
        } while(i != -1);

        fin.Close();
    }
}
```

## Bir Dosyaya Yazmak

Bir dosyaya bir byte yazmak için **WriteByte()** metodunu kullanın. Bunun en basit şekli aşağıda gösterilmiştir:

```
void WriteByte(byte değer)
```

Bu metod, **değer** tarafından belirtilen byte’ı dosyaya yazar. Yazma sırasında eğer bir hata meydana gelirse, bir **IOException** fırlatılır. Eğer söz konusu akış, çıktı için açılmamışsa, **NotSupportedException** fırlatılır.

**Write()**’ı çağrıarak bir dosyaya bir byte dizisi yazabilirsiniz. Bu durum aşağıda gösterilmiştir:

```
void Write(byte[] tampon, int offset, int byteSayısı)
```

**Write()**, **tampon[öteleme]**’den başlayarak **tampon** dizisinden **byteSayısı** kadar byte’ı dosyaya yazar. Yazma sırasında bir hata ortaya çıkarsa bir **IOException** fırlatılır. Eğer söz konusu akış, çıktı için açılmamışsa bir **NotSupportedException** fırlatılır. Diğer kural dışı durumlar da ayrıca mümkündür.

Muhtemelen biliyorsunuzdur, dosya çıktısı gerçekleştirilirken genellikle çıktı, asıl fiziksel cihaza hemen yazılmaz. Bunun yerine, veriler bir kerede tümdeň yazılabilecek büyülüge erişene kadar işletim sistemi tarafından bir tamponda depolanır. Bu, sistemin verimliliğini artırır. Örneğin, disk dosyaları sektör sektör düzenlenmiştir. Bir sektörün uzunluğu yaklaşık **128** byte civarındadır; daha büyük de olabilir. Çıktı genellikle bütün bir sektör bir kerede yazılabilecek hale gelene kadar tamponlanır. Ancak, tampon dolu olsun ya da olmasın verilerin fiziksel cihaza yazılmasını istiyorsanız, **Flush()**’ı şu şekilde çağırabilirsiniz:

```
void Flush()
```

Başarısızlık durumunda bir **IOException** fırlatılır.

Bir çıktı dosyası ile işiniz bitliğinde dosyayı **Close()** kullanarak kapatmanız gerektiğini hatırlamalısınız. Bu şekilde davranış olarak disk tamponunda kalan çıktıların gerçekten diske yazılmasını garanti etmiş olursunuz. Bu nedenle, bir dosyayı kapatmadan önce **Flush()**’ı çağrımanın pek bir esprisi yoktur.

İşte, bir dosyaya yazan basit bir örnek program:

```
// Dosyaya yazar.

using System;
using System.IO;

class WriteToFile {
    public static void Main(string[] args) {
        FileStream fout;

        // çıktı dosyası aç
        try {
            fout = new FileStream("test.txt", FileMode.Create);
        } catch(IOException exc) {
            Console.WriteLine(exc.Message +
                "\nError Opening Output File");
        }
    }
}
```

```

        return;
    }

    // Harfi dosyaya yaz
    try {
        for(char c = 'A'; c <= 'Z'; c++)
            fout.WriteByte((byte) c);
    } catch(IOException exc) {
        Console.WriteLine(exc.Message + "File Error");
    }

    fout.Close();
}
}

```

Program öncelikle çıktı için **test.txt** adında bir dosya açar. Sonra, bu dosyaya büyük harfleri yazar. Son olarak, dosyayı kapatır. Olası hataların **try/catch** blokları tarafından nasıl kontrol altına alındığını dikkat edin. Programın çalışmasından sonra **test.txt** aşağıdaki çıktıyi içerecektir:

ABCDEFGHIJKLMNPQRSTUVWXYZ

## Dosya Kopyalamak İçin FileStream Kullanmak

**FileStream** tarafından kullanılan byte tabanlı I/O'nun sağladığı avantajlardan biri, bunu -sadece metin içeren dosyalar üzerinde değil - her çeşit dosya üzerinde kullanılabiliyor olmanızdır. Örneğin, aşağıdaki program, çalıştırılabilir dosyalar da dahil olmak üzere her tür dosyanın kopyasını çıkartmaktadır. Kaynak ve hedef dosyaların isimleri komut satırında belirtilmektedir.

```

/* Dosyayı kopyalar.

Bu programı kullanmak için, kaynak ve
hedef dosyaların isimlerini belirtin.
Ornegin, FIRST.DAT adında bir dosyayı
SECOND.DAT adında bir dosyaya kopyalamak için
Asagidaki komut satirini kullanin.

CopyFile FIRST.DAT SECOND.DAT
 */

using System;
using System.IO;

class CopyFile {
    public static void Main(string[] args) {
        int i;
        FileStream fin;
        FileStream fout;

        try {
            // girdi dosyasını aç
            try {

```

```
        fin = new FileStream(args[0], FileMode.Open);
    } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message + "\nInput File
                                Not Found");
        return;
    }

    // çıktı dosyasını aç
    try {
        fout = new FileStream(args[1], FileMode.Create);
    } catch(IOException exc) {
        Console.WriteLine(exc.Message + "\nError Opening
                                Output File");
        return;
    }
} catch(IndexOutOfRangeException exc) {
    Console.WriteLine(exc.Message + "\nUsage:
                                CopyFile From To");
    return;
}

// Dosyayı kopyala
try {
    do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte)i);
    } while(i != -1);
} catch(IOException exc) {
    Console.WriteLine(exc.Message + "File Error");
}

fin.Close();
fout.Close();
}
}
```

## Karakter Tabanlı Dosya I/O İşlemleri

Byte tabanlı dosya yönetimi oldukça yaygınmasına rağmen C# ayrıca karakter tabanlı akışları da sağlamaktadır. Karakter tabanlı akışların avantajı, bunların doğrudan Unicode karakterleri üzerinde işlem görmeleridir. Yani, Unicode metinlerini saklamak istiyorsanız, karakter akışları kesinlikle sizin için en iyi tercih olacaktır. Genel olarak, karakter tabanlı dosya işlemlerini gerçekleştirmek için **FileStream**'ı ya bir **StreamReader** ya da **StreamWriter** içine yerleştireceksiniz. Bu sınıflar, byte akışını otomatik olarak bir karakter akışına dönüştürürler. Tersi de geçerlidir.

İşletim sistemi düzeyinde bir dosyanın birtakım byte'lardan oluştuğunu hatırlınızdan çıkarmayın. **StreamReader** ya da **StreamWriter** kullanmak bu gerceği değiştirmeyecektir,

**StreamWriter**, **TextWriter**'dan türetilmiştir. **StreamReader** ise **TextReader**'dan türetilmiştir. Böylece, **StreamWriter** ve **StreamReader** kendi temel sınıfları tarafından tanımlanan özelliklere ve metodlara erişim hakkında sahiplerdir.

## StreamWriter Kullanımı

Karakter tabanlı bit çıktı akışı oluşturmak için bir **Stream** nesnesini (mesela, bir **FileStream**) bir **StreamWriter** içine yerleştirin. **StreamWriter**'da birkaç yapılandırıcı tanımlıdır. Bunların en popülerlerinden biri aşağıda gösterilmiştir:

```
StreamWriter(Stream akış)
```

Burada **akış**, açık bir akışın ismidir. Bu yapılandırıcı, eğer akış, çıktı için açık değilse bir **ArgumentException** fırlatır, **akış**, **null** olduğunda ise bir **ArgumentNullException** fırlatılır. Bir kez oluşturulduktan sonra bir **StreamWriter** karakterlerin byte'a dönüşümünü otomatik olarak ele alır.

Aşağıda, klavyeden girilen bir satır metni okuyup, bunu **test.txt** adındaki bir dosyaya yazan “klavyeden diske” işlevine sahip basit bir yardımcı program yer almaktadır:

```
/* Bir StreamWriter'i gösteren, "klavyeden diske"
   işlevine sahip basit bir yardımcı program. */

using System;
using System.IO;

class KtoD {
    public static void Main() {
        string str;
        FileStream fout;

        try {
            fout = new FileStream("test.txt", FileMode.Create);
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message + "Cannot open file.");
            return;
        }
        StreamWriter fstr_out = new StreamWriter(fout);

        Console.WriteLine("Enter text ('stop' to quit).");
        do {
            Console.Write(": ");
            str = Console.ReadLine();

            if(str != "stop") {
                str = str + "\r\n"; // yeni bir satır ekle
                try {
                    fstr_out.Write(str);
                } catch(IOException exc) {
                    Console.WriteLine(exc.Message + "File Error");
                    return ;
                }
            }
        }
    }
}
```

```

        }
    }
} while(str != "stop");

fstr_out.Close();
}
}

```

Bazı durumlarda bir dosyayı StreamWriter kullanarak doğrudan açmak çok daha kullanışlı olabilir. Bunun için aşağıdaki yapılandırıcılarından birini kullanın:

```
StreamWriter(string dosyaadı)
StreamWriter(string dosyaadı, bool eklemeİşareti)
```

Burada ***dosyaadı***, açılacak dosyanın adını belirtir. ***dosyaadı***, tüm dizin yolunu tam olarak içerebilir. İkinci formda, eğer ***eklemeİşareti true*** ise çıktı, mevcut dosyanın sonuna eklenir. Aksi halde çıktı, belirtilen dosyanın üzerine yazar. Her iki durumda da, eğer dosya mevcut değilse oluşturulur. Ayrıca, bir I/O hatası durumunda her ikisi de bir **IOException** fırlatır. Diğer kural dışı durumlar da mümkündür.

Aşağıda, “klavyeden diske” işlevine sahip programın, çıktı dosyasını açmak için bir **StreamWriter** kullanacak şekilde yeniden yazılmış bir versiyonu yer almaktadır:

```
// StreamWriter kullanarak bir dosya ac.

using System;
using System.IO;

class KtoD {
    public static void Main() {
        string str;
        StreamWriter fstr_out;

        // StreamWriter kullanarak dosyayı doğrudan ac.
        try {
            fstr_out = new StreamWriter("test.txt");
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message + "Cannot open file.");
            return;
        }

        Console.WriteLine("Enter text ('stop' to quit).");
        do {
            Console.Write(": ");
            str = Console.ReadLine();

            if(str != "stop") {
                str = str + "\r\n"; // yeni bir satır ekle
                try {
                    fstr_out.Write(str);
                } catch(IOException exc) {
                    Console.WriteLine(exc.Message + "File Error");
                }
            }
        }
    }
}
```

```

        return ;
    }
}
} while(str != "stop");

fstr_out.Close();
}
}
}

```

## StreamReader Kullanımı

Karakter tabanlı bir girdi akışı oluşturmak için bir byte akışını bir **StreamReader** içine yerleştirin. **StreamReader**'da birkaç yapılandırıcı tanımlıdır. Sıkça kullanılan bir yapılandırıcı aşağıda gösterilmiştir:

```
StreamReader(Stream akış)
```

Burada **akış**, açık bir akışın ismidir. Bu yapılandırıcı, **akış null** ise bir **ArgumentNullException** fırlatır. Bir kez oluşturuluktan sonra bir **StreamReader**, byte'ların karaktere dönüşümünü otomatik olarak ele alır.

Aşağıda, **test.txt** adında bir metin dosyasını okuyup, dosya içeriğini ekranda gösteren bir “diskten ekrana” işlevine sahip basit bir yardımcı program oluşturulmaktadır. Böylece bu program, önceki bölümde gösterilen “klavyeden diske” işlevine sahip yardımcı programı tamamlamaktadır:

```

/* Bir FileReader'i gosteren "diskten ekrana"
   islevine sahip basit bir yardimci program. */

using System;
using System.IO;

class DtoS {
    public static void Main() {
        FileStream fin;
        string s;

        try {
            fin = new FileStream("test.txt", FileMode.Open);
        }
        catch(FileNotFoundException exc) {
            Console.WriteLine(exc.Message + "Cannot open file.");
            return ;
        }

        StreamReader fstr_in = new StreamReader(fin);

        // Dosyayı satır satır oku.
        while((s = fstr_in.ReadLine()) != null) {
            Console.WriteLine(s);
        }
    }
}

```

```

        fstr_in.Close();
    }
}

```

Dosyanın sonunun nasıl belirlendiğine dikkat edin. `ReadLine()` tarafından döndürülen referans `null` ise, dosyanın sonuna erişilmiş demektir.

`StreamWriter`'daki gibi, bazı durumlarda bir dosyayı `StreamReader` kullanarak doğrudan açmak size daha kolay gelebilir. Bunun için aşağıdaki yapılandırıcıyı kullanın:

```
StreamReader(string dosyaadı)
```

Burada `dosyaadı`, açılacak dosyanın ismini belirtir. `dosyaadı`, dosyanın dizin yolunu tam olarak da içerebilir. Dosya mevcut olmalıdır. Eğer değilse, bir `FileNotFoundException` fırlatılır. Eğer `dosyaadı null` değerine sahipse, bir `ArgumentNullException` fırlatılır. `dosyaadı` boş bir karakter katarından ibaret ise `ArgumentException` fırlatılır.

## Standart Akışları Yönlendirmek

Daha önce bahsedildiği gibi, standart akışlar - örneğin `Console.In` - yönlendirilebilir. En yaygın yönlendirme, bir dosyaya yapılır. Standart bir akış yönlendirildiği zaman girdi ve/veya çıktı, varsayılan cihazlar atlanarak otomatik olarak yeni akışa yöneltilir. Standart akışları yönlendirerek programınız bir disk dosyasından komutlar okuyabilir, kütük dosyaları oluşturabilir, hatta bir ağ bağlantısından girdi bile okuyabilir.

Standart akışların yönlendirilmesi iki şekilde gerçekleştirilebilir: Birincisi, bir programı komut satırından çalıştırığınızda `Console.In` ve/veya `Console.Out`'a yönlendirmek için sırasıyla < ve > operatörlerini kullanabilirsiniz. Örneğin, aşağıdaki programı inceleyelim:

```

using System;

class Test {
    public static void Main() {
        Console.WriteLine("This is a test.");
    }
}

```

Programı aşağıdaki şekilde çalıştmak, `log` adında bir dosyaya "This is a test." satırının yazılmasına neden olacaktır.

```
Test > log
```

Girdi de aynı şekilde yönlendirilebilir. Girdi yönlendirildiğinde şunu hatırlamalısınız: Girdi kaynağı olarak belirtmiş olduğunuz kaynağın programın taleplerini karşılamak için yeterince girdi içerdiginden emin olmalısınız. Eğer içermiyorsa program asılı kalacaktır.

Komut satırını yönlendirme operatörleri olan < ve >, C#'ın bir parçası değildirler; işletim sistemi tarafından sağlanırlar. Böylece, eğer ortamınız I/O yönlendirmesini destekliyorsa (Windows'ta söz konusu olduğu gibi), programınızda herhangi bir değişiklik yapmadan

standart girdi ve standart çıktıyi yönlendirebilirsiniz. Yine de, standart akışları yönlendirebilmeniz için sizin kontrolünüzde olan ikinci bir yol da mevcuttur. Bunun için **Console**'un üyesi olan ve aşağıda gösterilen **SetIn()**, **SetOut()** ve **SetError()** metodlarını kullanacaksınız:

```
static void SetIn (TextReader girdi)
static void SetOut(TextWriter çıktı)
static void SetError(TextWriter çıktı)
```

Böylece, girdiyi yönlendirmek için istenilen akışı belirterek **SetIn()**'ı çağırın. **TextReader**'dan türetildiği sürece herhangi bir girdi akışını kullanabilirsiniz. Çıktıyı bir dosyaya yönlendirmek için bir **StreamWriter** içine yerleştirilmiş bir **FileStream** belirtin. Aşağıdaki program bunun bir örneğini göstermektedir:

```
// Console.Out'a yönlendir.

using System;
using System.IO;

class Redirect {
    public static void Main() {
        StreamWriter log_out;

        try {
            log_out = new StreamWriter("logfile.txt");
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message + "Cannot open file.");
            return;
        }

        // Standart çıktıyi kutuk dosyasına yönlendir.
        Console.SetOut(log_out);
        Console.WriteLine("This is the start of the log file.");

        for(int i = 0; i < 10; i++) Console.WriteLine(i);

        Console.WriteLine("This is the end of the log file.");
        log_out.Close();
    }
}
```

Bu programı çalıştırığınızda ekranda hiç çıktı görmeyeceksiniz, ama **logfile.txt** aşağıdakileri içerecektir:

```
This is the start of the log file.
0
1
2
3
4
5
6
```

```

7
8
9
This is the end of the log file.

```

Kendi kendinize diğer standart akışları yönlendirmeyi deneyebilirsiniz.

## İkili Verileri Okumak ve Yazmak

Şimdiye dek byte’ları veya karakterleri okuyup yazmaktayız. Fakat, diğer tipteki verileri okumak ve yazmak da mümkündür - hatta, yaygındır. Söz geliş, **int**, **double** ve **short**’ları içeren bir dosya oluşturmak isteyebilirsiniz. C#’ın standart tiplerinin ikili (binary) değerlerini okumak ve yazmak için **BinaryReader** ve **BinaryWriter** kullanacaksınız. Bu tür verilerin insanların okuyacağı türden metin formunda değil de dahili, ikili biçim kullanılarak okunduğunu ve yazıldığını anlamak önemlidir.

### Binary Writer

**BinaryWriter**, ikili verilerin yazılmasını yöneten bir byte akışının elrafındaki ambalajdır. Bunun en sık kullanılan yapılandırıcısı aşağıda gösterilmiştir:

```
BinaryWriter(Stream çıkış)
```

Burada **çıkış**, verilerin yazılacağı akıştır. Bir dosyaya çıktı yazmak amacıyla bu parametre için **FileStream** tarafından oluşturulan nesneyi kullanabilirsiniz. Eğer **çıkış** null ise bir **ArgumentNullException** fırlatılır. Eğer **çıkış** yazma işlemi için henüz açılmamışsa **ArgumentException** fırlatılır.

**BinaryWriter**, C#’ın tüm standart tiplerini yazabilen metodları tanımlamaktadır. Bunlardan birkaçı Tablo 14.5’té gösterilmiştir. Bir **string**’in, uzunluk belirleyicisini de içeren dahili biçimini kullanılarak yazıldığına dikkat edin. **BinaryWriter**’da ayrıca standart **Close()** ve **Flush()** metodları da tanımlanmaktadır. Bunlar önceden anlatıldıkları gibi çalışmaktadır.

**TABLO 14.5: BinaryWriter Tarafından Tanımlanan Sıkça Kullanılan Çıktı Metotları**

Metot	Açıklama
<b>void Write(sbyte değer)</b>	İşaretli bir byte yazar.
<b>void Write(byte değer)</b>	İşaretsiz bir byte yazar.
<b>void Write(byte[] tampon)</b>	Bir byte dizisi yazar.
<b>void Write(short değer)</b>	Bir kısa tamsayı yazar.
<b>void Write(ushort değer)</b>	Bir işaretetsiz kısa tamsayı yazar.
<b>void Write(int değer)</b>	Bir tamsayı yazar.
<b>void Write(uint değer)</b>	Bir işaretetsiz tamsayı yazar.

<code>void Write(long değer)</code>	Bir uzun tamsayı yazar.
<code>void Write(ulong değer)</code>	Bir işaretetsiz uzun tamsayı yazar.
<code>void Write(float değer)</code>	Bir <code>float</code> yazar.
<code>void Write(double değer)</code>	Bir <code>double</code> yazar.
<code>void Write(char değer)</code>	Bir karakter yazar.
<code>void Write(char[] tampon)</code>	Bir karakter dizisi yazar.
<code>void Write(string değer)</code>	Bir <code>string</code> 'i dahili gösterimini kullanarak yazar; dahili gösterim, bir uzunluk belirleyicisi içerir.

## Binary Reader

`BinaryReader`, ikili verileri okuma işlemini kontrol altına alan bir byte akışının etrafındaki ambalajdır. Bunun en yaygın kullanılan yapılandırıcısı aşağıda gösterilmiştir:

`BinaryReader(Stream girdiAkışı)`

Burada `girdiAkısı` verilerin okunduğu akıştır. Bir dosyadan okumak amacıyla bu parametre için `FileStream` tarafından oluşturulmuş nesneyi kullanabilirsiniz. Eğer `girdiAkısı null` ise bir `ArgumentNullException` fırlatılır. Eğer `girdiAkısı` okuma işlemi için henüz açılmamışsa, `ArgumentException` fırlatılır.

`BinaryReader`, C#'ın tüm basit tiplerini okumak için metotlara sahiptir. Bunlardan en yaygın olarak kullanılanlar Tablo 14.6'da gösterilmiştir. `ReadString()`'in, depolanmış bir karakter katarını dahili biçimini kullanarak okuduğuna dikkat edin. Dahili biçim, uzunluk belirleyicisini de içermektedir. Akışın sonuna ulaşınca tüm metotlar bir `EndOfStreamException` fırlatırlar. Hata durumunda ise bir `IOException` fırlatırlar. `BinaryReader` ayrıca `Read()`'in üç versiyonunu da tanımlamaktadır. Bu versiyonlar şunlardır:

Metot	Açıklama
<code>int Read()</code>	Metodu çağrıran girdi akışındaki bir sonraki mevcut karakteri simgeleyen bir tamsayı döndürür. Dosyanın sonuna ulaşınca <code>-1</code> döndürür.
<code>int Read(byte[] tampon, int öteleme, int no)</code>	<code>tampon[öteleme]</code> 'den başlayarak <code>tampon'a</code> <code>no</code> kadar byte okumaya çalışır. Başarıyla okunan byte sayısını döndürür.
<code>int Read(char[] tampon, int öteleme, int no)</code>	<code>tampon[öteleme]</code> 'den başlayarak <code>tampon'a</code> <code>no</code> kadar karakter okumaya çalışır. Başarıyla okunan karakter sayısını döndürür.

Bu metotlar başarısızlık durumunda bir `IOException` fırlatırlar.

Standart `Close()` metodu da ayrıca tanımlanmaktadır.

**TABLO 14.6: BinaryReader Tarafından Tanımlanan Sıkça Kullanılan Girdi Metotları**

Metot	Açıklama
<code>bool ReadBoolean()</code>	Bir <code>bool</code> okur.
<code>byte ReadByte()</code>	Bir <code>byte</code> okur.
<code>sbyte ReadSByte()</code>	Bir <code>sbyte</code> okur.
<code>byte[] ReadBytes(int no)</code>	<code>no</code> adet byte okur ve bunları bir dizi olarak döndürür.
<code>char ReadChar()</code>	Bir <code>char</code> okur.
<code>char[] ReadChars(int no)</code>	<code>no</code> adet karakter okur ve bunları bir dizi olarak döndürür.
<code>double ReadDouble()</code>	Bir <code>double</code> okur.
<code>float ReadSingle()</code>	Bir <code>float</code> okur.
<code>short ReadInt16()</code>	Bir <code>short</code> okur.
<code>int ReadInt32()</code>	Bir <code>int</code> okur.
<code>long ReadInt64()</code>	Bir <code>long</code> okur.
<code>ushort ReadUInt16()</code>	Bir <code>ushort</code> okur.
<code>uint ReadUInt32()</code>	Bir <code>uint</code> okur.
<code>ulong ReadUInt64()</code>	Bir <code>ulong</code> okur.
<code>string ReadString()</code>	Dahili, ikili biçimde simgelenen bir <code>string</code> okur. Biçim, uzunluk belirleyicisi içerir. Bu metot yalnızca bir <code>BinaryWriter</code> kullanılarak yazılmış bir karakter katarını okumak için kullanılmalıdır.

## İkili I/O İşlemlerinin Gösterilmesi

Aşağıda, `BinaryReader` ve `BinaryWriter`'ı gösteren bir program yer almaktadır. Program, çeşitli tipten verileri bir dosyaya yazar ve sonra bunları dosyadan geri okur.

```
// İkili verileri yaz ve sonra geri oku.

using System;
using System.IO;

class RWData {
    public static void Main() {
        BinaryWriter dataOut;
        BinaryReader dataIn;

        int i = 10;
        double d = 1023.56;
        bool b = true;
```

```
try {
    dataOut = new BinaryWriter(new FileStream("testdata",
                                              FileMode.Create));
}
catch(IOException exc) {
    Console.WriteLine(exc.Message + "\nCannot open file.");
    return;
}

try {
    Console.WriteLine("Writing " + i);
    dataOut.Write(i);

    Console.WriteLine("Writing " + d);
    dataOut.Write(d);

    Console.WriteLine("Writing " + b);
    dataOut.Write(b);

    Console.WriteLine("Writing " + 12.2 * 7.4);
    dataOut.Write(12.2 * 7.4);
}
catch(IOException exc) {
    Console.WriteLine(exc.Message + "\nWrite error.");
}

dataOut.Close();

Console.WriteLine();

// Simdi, bunlari geri oku.
try {
    dataIn = new BinaryReader(new FileStream("testdata",
                                              FileMode.Open));
}
catch(FileNotFoundException exc) {
    Console.WriteLine(exc.Message + "\nCannot open file.");
    return;
}

try {
    i = dataIn.ReadInt32();
    Console.WriteLine("Reading " + i);

    d = dataIn.ReadDouble();
    Console.WriteLine("Reading " + d);

    b = dataIn.ReadBoolean();
    Console.WriteLine("Reading " + b);

    d = dataIn.ReadDouble();
    Console.WriteLine("Reading " + d) ;
}
catch(IOException exc) {
```

```
        Console.WriteLine(exc.Message + "Read error.");
    }

    dataIn.Close();
}
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Writing 10
Writing 1023.56
Writing True
Writing 90.28

Reading 10
Reading 1023.56
Reading True
Reading 90.28
```

Bu program tarafından oluşturulan **testdata** dosyasını incelerseniz, dosyanın insanların okuyabileceği gibi bir metin değil, ikili veriler içerdigini fark edeceksiniz.

İşte, ikili I/O'nun ne kadar güçlü olduğunu gösteren daha pratik bir örnek. Aşağıdaki program çok basit bir envanter programını uygulamaktadır. Envanterdeki her kalem için program; malın ismini, stokta kaç adet olduğunu ve maliyetini saklar. Sonra program, kullanıcıyı bir malın ismini girmesi için yönlendirir. Daha sonra veri tabanını arar. Eğer söz konusu mal veri tabanında bulunursa, ilgili envanter bilgileri ekranda gösterilir.

```
/* Basit bir envanter programini uygulamak
   icin BinaryReader ve BinaryWriter kullanir. */

using System;
using System.IO;

class Inventory {
    public static void Main() {
        BinaryWriter dataOut;
        BinaryReader dataIn;

        string item; // malin ismi
        int onhand; // stoktaki mal miktarı
        double cost; // maliyet

        try {
            dataOut = new
                BinaryWriter(new FileStream("inventory.dat",
                    FileMode.Create));
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message + "\nCannot open file.");
            return;
        }

        // Dosyaya bazi envanter verilerini yaz.
```

```
try {
    dataOut.Write("Hammers");
    dataOut.Write(10);
    dataOut.Write(3.95);

    dataOut.Write("Screwdrivers");
    dataOut.Write(18);
    dataOut.Write(1.50);

    dataOut.Write("Pliers");
    dataOut.Write(5);
    dataOut.Write(4.95);

    dataOut.Write("Saws");
    dataOut.Write(8);
    dataOut.Write(8.95);
}
catch(IOException exc) {
    Console.WriteLine(exc.Message + "\nWrite error.");
}

dataOut.Close();

Console.WriteLine();

// Simdi, envanter dosyasini okuma icin ac.
try {
    dataIn = new
        BinaryReader(new FileStream("inventory.dat",
            FileMode.Open));
}
catch(FileNotFoundException exc) {
    Console.WriteLine(exc.Message + "\nCannot open file.");
    return;
}

// Kullanicinin girdigi mali ara.
Console.Write("Enter item to lookup: ");
string what = Console.ReadLine();
Console.WriteLine();

try {
    for(;;) {
        // Envanter girdilerinden birini oku.
        item = dataIn.ReadString();
        onhand = dataIn.ReadInt32();
        cost = dataIn.ReadDouble();

        /* Malin istenilenle eslenip eslenmedigine bak.
           Eger esleniyorsa, ilgili bilgileri ekranda
           goster. */
        if(item.CompareTo(what) == 0) {
            Console.WriteLine(onhand + " " + item + " on hand. "
                + "Cost: {0:C} each", cost);
            Console.WriteLine("Total value of {0}: {1:C}.",

```

```

                item, cost * onhand);
            break;
        }
    }
}
catch(EndOfStreamException) {
    Console.WriteLine("Item not found.");
}
catch(IOException exc) {
    Console.WriteLine(exc.Message + "Read error.");
}

dataIn.Close();
}
}

```

İşte programın örnek bir çalışması:

```

Enter item to lookup: Screwdrivers
18 Screwdrivers on hand. Cost: $1.50 each
Total value of Screwdrivers: $27.00

```

Programda envanter bilgilerinin ikili formatta nasıl saklandığına dikkat edin. Böylece, stoktaki mal miktarı ve maliyet, insanların okuyabildiği metin tabanlı eşdeğerleri yerine ikili biçimleri kullanılarak saklanmıştır. Bu sayede, verileri insanların okuyabildiği şekilde çevirmek zorunda kalmadan nümerik veriler üzerinde işlem gerçekleştirmek mümkün olur.

Envanter programında ilginç olan diğer bir husus daha vardır. Dosyanın sonunun nasıl tespit edildiğine dikkat edin. İkili girdi metodları akışın sonuna ulaşıldığında bir **EndOfStreamException** fırlattıkları için program istenilen öğeyi bulana kadar ya da bu kural dışı durum üretilene kadar dosyayı okur. Böylece, dosyanın sonunu tespit etmek için özel bir mekanizmaya gerek yoktur.

## Rasgele Erişimli Dosyalar

Bu aşamaya kadar *sıralı dosyaları* (*sequential files*) kullanılmıştır. Sıralı dosyalar, bir byte ardından bir diğer byte şeklinde, tamamen doğrusal olarak erişilen dosyalardır. Ancak; C#, bir dosyanın içeriğine rasgele erişmenize de imkan vermektedir. Bunun için **FileStream** tarafından tanımlanan **Seek()** metodunu kullanacaksınız. Bu metod, dosya konum belirtecini (buna dosya işaretçisi de denir) dosya içinde herhangi bir noktaya ayarlamانızı olanak tanır.

**Seek()** metodu aşağıda gösterilmiştir:

```
long Seek(long yeniKonum, SeekOrigin orijin)
```

Burada **yeniKonum**, dosya işaretçisinin yeni konumunu, **orijin** ile belirtilen konumdan byte cinsinden uzaklık olarak belirtir. **orijin**, **SeekOrigin** numaralandırması tarafından tanımlanan şu değerlerden biri olacaktır:

Değer	Anlamı
<b>SeekOrigin.Begin</b>	Dosyanın başından başlayarak ara.
<b>SeekOrigin.Current</b>	Mevcut konumdan başlayarak ara.
<b>SeekOrigin.End</b>	Dosyanın sonundan başlayarak ara.

**Seek()** çağrıldıktan sonra bir sonraki okuma ya da yazma işlemi yeni dosya konumunda gerçekleştirilecektir. Arama sırasında bir hata meydana gelirse bir **IOException** fırlatılır. Eğer söz konusu akış, konum taleplerini desteklemiyorsa, bir **NotSupportedException** fırlatılır.

İşte, rasgele erişimle ilgili I/O işlemlerini gösteren bir örnek. Bu program bir dosyaya büyük harflerle alfabeyi yazar ve sonra bunu dosyadan sırasız olarak geri okur.

```
// Rasgele erisimi gosterir.

using System;
using System.IO;

class RandomAccessDemo {
    public static void Main() {
        FileStream f;
        char ch;

        try {
            f = new FileStream("random.dat", FileMode.Create);
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        // Alfabeyi yaz.
        for(int i = 0; i < 26; i++) {
            try {
                f.WriteByte((byte)('A' + i));
            }
            catch(IOException exc) {
                Console.WriteLine(exc.Message);
                return;
            }
        }

        try {
            // Simdi, belirli degerleri geri oku
            f.Seek(0, SeekOrigin.Begin); // ilk byte'i ara
            ch = (char) f.ReadByte();
            Console.WriteLine("First value is " + ch);

            f.Seek(1, SeekOrigin.Begin); // ikinci byte'i ara
            ch = (char) f.ReadByte();
            Console.WriteLine("Second value is " + ch);
        }
    }
}
```

```

        f.Seek(4, SeekOrigin.Begin); // 5'inci byte'i ara
        ch = (char) f.ReadByte();
        Console.WriteLine("Fifth value is " + ch);

        Console.WriteLine();

        // Simdi, bir baska deger daha oku.
        Console.WriteLine("Here is every other value: ");
        for(int i = 0; i < 26; i += 2) {
            f.Seek(i, SeekOrigin.Begin); // i'nici byte'i ara
            ch = (char) f.ReadByte();
            Console.Write(ch + " ");
        }
    }
    catch(IOException exc) {
        Console.WriteLine(exc.Message);
    }

    Console.WriteLine();
    f.Close();
}
}

```

Programın çıktısı aşağıdaki gibidir:

```

First value is A
Second value is B
Fifth value is E

Here is every other value:
A C E G I K M O Q S U W Y

```

## MemoryStream'in Kullanımı

Kimi zaman, doğrudan bir aygıtta girdi okumak veya doğrudan bir aygıta çıktı yazmak yerine, girdileri bir diziden okumak ya da çıktıları bir diziye yazmak kullanışlıdır. Bunu gerçekleştirmek için **MemoryStream**'ı kullanacaksınız. **MemoryStream**, girdi ve/veya çıktı için bir byte dizisi kullanan **Stream**'in bir uygulamasıdır. Bunu tanımlayan yapılandırıcılarından biri şu şekildedir:

```
MemoryStream(byte[] tampon)
```

Burada **tampon**, I/O taleplerinin kaynağı ve/veya hedefi olarak kullanılacak olan bir byte dizisidir. Bu yapılandırıcı tarafından oluşturulan akış, okunabilir ve yazılabilir; üstelik **seek()**'i de destekler. **tampon**'u, **tampon**'a yönlendireceğiniz çıktıları tutabilecek büyüklikte yapmanız gerektiğini hatırlamalısınız.

Aşağıda, **MemoryStream**'in kullanımını gösteren bir program yer almaktadır:

```
// MemoryStream'i gösterir.
```

```

using System;
using System.IO;

class MemStrDemo {
    public static void Main() {
        byte[] storage = new byte[255];

        // Bellek tabanlı bir akış oluştur.
        MemoryStream memstrm = new MemoryStream(storage);

        // memstrm'i bir okuyucu ve yazıcı içine yerlestir.
        StreamWriter memwtr = new StreamWriter(memstrm);
        StreamReader memrdr = new StreamReader(memstrm);

        // memwtr aracılığıyla storage'a yaz.
        for(int i = 0; i < 10; i++)
            memwtr.WriteLine("byte [" + i + "]: " + i);

        // en sona bir nokta koy
        memwtr.Write('.');

        memwtr.Flush();

        Console.WriteLine("Reading from storage directly: ");

        // storage'in içerigini doğrudan göster.
        foreach(char ch in storage) {
            if(ch == '.') break;
            Console.Write(ch);
        }

        Console.WriteLine("\nReading through memrdr: ");

        // Akış okuyucusunu kullanarak memstrm'den oku.
        memstrm.Seek(0, SeekOrigin.Begin); // dosya işaretçisini
                                         sifırla

        string str = memrdr.ReadLine();
        while(str != null) {
            str = memrdr.ReadLine();
            if(str.CompareTo(".") == 0) break;
            Console.WriteLine(str);
        }
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Reading from storage directly:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5

```

```
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9

Reading through memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

Programda **storage** adında bir byte dizisi oluşturulmaktadır. Bu dizi daha sonra **memstrm** adında bir **MemoryStream** için alta yatan bir depolama olarak kullanılmaktadır. **memstrm**'den **memrdr** adında bir **StreamReader** ve **memwtr** adında bir **StreamWriter** oluşturulur. **memwtr** kullanılarak çıktı, bellek tabanlı akışa yazılır. Çıktı yazıldıktan sonra **memwtr** üzerinden **flush()**'ın çağrıldığına dikkat edin. **memwtr**'in tamponunda kalan içeriğin alta yatan diziye gerçekten yazılması için bu gereklidir. Sonra, alta yatan byte dizisinin içeriği **foreach** döngüsü kullanılarak elle gösterilir. Daha sonra **Seek()** kullanılarak dosya işaretçisi, akışın başlangıcını gösterecek şekilde sıfırlanır ve **memrdr** kullanılarak bellek akışı okunur.

Bellek tabanlı akışlar programlamada oldukça kullanışlıdır. Örneğin, çıktıyı ihtiyaç olana kadar dizide saklayarak karmaşık çıktılar oluşturabilirsiniz. Bu teknik, özellikle Windows gibi bir GUI ortamı için programlama yaparken işe yarar. Ayrıca bir standart akışı da bir diziden okuyacak şekilde yönlendirebilirsiniz. Bu örneğin, test bilgilerini bir programa beslemek için yararlı olabilir.

## StringReader ve StringWriter Kullanımı

Bazı uygulamalarda bellek tabanlı I/O işlemlerini gerçekleştirirken alta yatan depolama olarak bir byte dizisi yerine bir **string** kullanmak çok daha kolay olabilir. Böyle bir durum söz konusu olduğunda **StringReader** ve **StringWriter** kullanın. **StringReader**, **TextReader**'dan; **StringWriter** da **TextWriter**'dan türetilmiştir. Böylece, bu akışların söz konusu iki sınıf tarafından tanımlanan metodlara erişimleri vardır. Örneğin, bir **StringReader** üzerinden **ReadLine()**'ı; bir **StringWriter** üzerinden de **WriteLine()**'ı çağırabilirsiniz.

**StringReader** için yapılandırıcı aşağıda gösterilmiştir:

```
StringReader(string str)
```

Burada **str**, okunacak karakter katarıdır.

**StringWriter** birkaç yapılandırıcı tanımlar. Bunlardan bizim burada kullanacağımız şudur:

```
StringWriter()
```

Bu yapılandırıcı, çıktısını bir karakter katarına yerleştirecek olan bir yazıcı oluşturur. Bu karakter katarı **StringWriter** tarafından otomatik olarak oluşturulur. Bu karakter katarının içeriğini **ToString()**'ı çağırarak elde edebilirsiniz.

İşte, **StringReader** ve **StringWriter** kullanan bir örnek:

```
// StringReader ve StringWriter'ı gösterir.

using System;
using System.IO;

class StrRdrDemo {
    public static void Main() {
        // Bir StringWriter oluştur
        StringWriter strwtr = new StringWriter();

        // StringWriter'a yaz.
        for(int i = 0; i < 10; i++)
            strwtr.WriteLine("This is i: " + i);

        // Bir StringReader oluştur

        StringReader strrdr = new StringReader(strwtr.ToString());

        // Şimdi, StringReader'dan oku.
        string str = strrdr.ReadLine();
        while(str != null) {
            str = strrdr.ReadLine();
            Console.WriteLine(str);
        }
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
This is i: 1
This is i: 2
This is i: 3
This is i: 4
This is i: 5
This is i: 6
This is i: 7
This is i: 8
This is i: 9
```

Program ilk önce **strwrt** adında bir **StringWriter** oluşturur ve **WriteLine()** kullanarak çıktıyu buna yazar. Sonra, **strwrt** içindeki karakter katarını kullanarak bir

`StringReader` oluşturur. Bu karakter katarı, `strwrt` üzerinden `ToString()` çağrılarak elde edilir. Son olarak, bu karakter katarının içeriği `ReadLine()` kullanılarak okunur.

## Nümerik Karakter Katarlarını Dahili Gösterimlerine Dönüştürmek

I/O konusunu terk etmeden önce nümerik karakter katarlarını okurken kullanışlı olan bir teknigi inceleyeceğiz. Bildiğiniz gibi, C#'ın `WriteLine()` metodu, `int` ve `double` gibi standart tiplerin nümerik değerleri de dahil olmak üzere, çeşitli tipten verilerin konsola çıktısını almak için kullanışlı yollardan biridir. Böylece, `WriteLine()` nümerik değerleri otomatik olarak insanların okuyabileceği şekilde dönüştürür. Ancak, C# bunun tersini sağlamaz: Nümerik değerleri simgeleyen karakter katarlarını okuyup, bunları dahili, ikili biçimde dönüştüren bir girdi metodu sağlamaz. Örneğin, “**100**” gibi bir karakter katarını okuyan ve bunu karşılık gelen ve bir `int` değişkende saklanabilecek ikili değere otomatik olarak dönüştüren bir girdi metodu mevcut değildir. Bunu gerçekleştirmek için tüm standart nümerik tipler için tanımlı olan bir metod kullanmanız gerekecektir: `Parse()`.

Konuya başlamadan önce önemli bir gerçeği bildirmek gereklidir: `int` ve `double` gibi C#'ın tüm standart tipleri aslında .NET Framework tarafından tanımlanan yapılar için birer *takma isimden (alias)* ibarettir. Gerçekte, C# tiplerinin .NET yapı tiplerinden ayırsanamaz olduğunu Microsoft açıkça bildirmektedir. Biri, diğerine verilen bir başka isimden ibarettir. C#'ın değer tipleri yapılar tarafından desteklendikleri için, değer tipleri kendileri için tanımlanmış üyelere sahiplerdir.

Nümerik tipler için, .NET yapı isimleri ve bunların C# anahtar kelimeleri cinsinden eşdeğerleri aşağıda gösterilmiştir:

.NET Yapı Adı	C# Adı
<code>Decimal</code>	<code>decimal</code>
<code>Double</code>	<code>double</code>
<code>Single</code>	<code>float</code>
<code>Int16</code>	<code>short</code>
<code>Int32</code>	<code>int</code>
<code>Int64</code>	<code>long</code>
<code>UInt16</code>	<code>ushort</code>
<code>UInt32</code>	<code>uint</code>
<code>UInt64</code>	<code>ulong</code>
<code>Byte</code>	<code>byte</code>
<code>SByte</code>	<code>sbyte</code>

Yapılar **System** isim uzayında tanımlanmıştır. Böylece, **Int32** için tam olarak belirtilmiş isim **System.Int32**'dir. Bu yapılar, değer tiplerini C#'ın nesne hiyerarşisine tam olarak entegre etmeye yardımcı olan geniş bir metot dizisi sunmaktadır. Bir yan fayda olarak, nümerik yapılar ayrıca bir nümerik karakter katarını, karşılık gelen ikili eşdeğeriye dönüştüren statik metotlar da tanımlamaktadırlar. Bu dönüşüm metotları aşağıda gösterilmiştir. Bunların her biri, söz konusu karakter katarına karşılık gelen bir değer döndürür.

<b>Yapı</b>	<b>Dönüşüm Metodu</b>
<b>Decimal</b>	<b>static decimal Parse(string str)</b>
<b>Double</b>	<b>static double Parse(string str)</b>
<b>Single</b>	<b>static float Parse(string str)</b>
<b>Int64</b>	<b>static long Parse(string str)</b>
<b>Int32</b>	<b>static int Parse(string str)</b>
<b>Int16</b>	<b>static short Parse(string str)</b>
<b>UInt64</b>	<b>static ulong Parse(string str)</b>
<b>UInt32</b>	<b>static uint Parse(string str)</b>
<b>UInt16</b>	<b>static ushort Parse(string str)</b>
<b>Byte</b>	<b>static byte Parse(string str)</b>
<b>SByte</b>	<b>static sbyte Parse(string str)</b>

Eğer **str**, metodu çağrıran tip tarafından tanımlanan şekliyle geçerli bir sayı içermiyorsa **Parse()** metodu bir **FormatException** fırlatacaktır. **str null** değerine sahipse bir **ArgumentNullException** fırlatılır; **str**'ın içindeki değer çağrıran tipi aşyorsa **OverflowException** fırlatılır.

Ayrıştırma (parsing) metotları, klavyeden ya da bir metin dosyasından bir karakter katarı olarak okunan bir nümerik değeri, gerçek dahili biçimine dönüştürmek için kolay bir yöntem sunmaktadır. Örneğin aşağıdaki program, kullanıcı tarafından girilen bir sayı listesinin ortalamasını hesaplar. Program öncelikle, kullanıcıya ortalaması hesaplanacak değerlerin adedini sorar. Sonra, **ReadLine()** kullanarak bu sayıyı okur ve söz konusu karakter katarını bir tamsayıya dönüştürmek için **Int32.Parse()**'ı kullanır. Daha sonra, karakter katarlarını karşılık gelen **double** eşdeğerlerine dönüştürmek için **Double.Parse()**'ı kullanarak değerlerin girdisini alır.

```
/* Bu program, kullanıcı tarafından girilen bir sayı listesinin
ortalamasını hesaplar. */

using System;
using System.IO;

class AvgNums {
    public static void Main() {
        string str;
```

```
int n;
double sum = 0.0;
double avg, t;

Console.WriteLine("How many numbers will you enter: ");
str = Console.ReadLine();
try {
    n = Int32.Parse(str);
}
catch(FormatException exc) {
    Console.WriteLine(exc.Message);
    n = 0;
}
catch(OverflowException exc) {
    Console.WriteLine(exc.Message);
    n = 0;
}

Console.WriteLine("Enter " + n + " values.");
for(int i = 0; i < n; i++) {
    Console.Write(": ");
    str = Console.ReadLine();
    try {
        t = Double.Parse(str);
    } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        t = 0.0;
    }
    catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        t = 0;
    }
    sum += t;
}
avg = sum / n;
Console.WriteLine("Average is " + avg);
}
```

İşte, programın örnek bir çalışması:

```
How many numbers will you enter: 5
Enter 5 values.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Average is 3.3
```

Son bir husus: Dönüşürmeye çalışığınız değerin tipine uygun bir ayırtırma metodu kullanmalısınız. Örneğin, kayan noktalı bir değer içeren bir karakter katarı üzerinde **Int32.Parse()**'ı kullanmaya çalışmamak istenilen sonucu vermeyecektir.

15

ONBEŞİNCİ BÖLÜM

---

## DELEGELER VE OLAYLAR

Bu bölümde C#'ın iki yeni özelliği incelenmektedir: Delegeler ve olaylar. Bir *delegate* (*delegate*), bir metodu paketleme (*encapsulation*) yollarından biridir. Bir *olay* (*event*) ise bir faaliyetin meydana geldiğini belirten bir bildirgedir. Delegeler ve olaylar birbiriyle bağlantılıdır, çünkü bir olay bir delege üzerine inşa edilir. Bunların her ikisi de birtakım programlama görevlerini C#'a uygulanabilecek şekilde genişletirler.

## Delegeler

Gelin, *delegate* terimini tanımlayarak başlayalım. Açıkça ifade etmek gerekirse, bir delege bir metoda referansta bulunabilen bir nesnedir. Böylece, bir delege oluşturduğunuzda bir metoda yönelik bir referans tulabilen bir nesne oluşturuyorsunuz, Üstelik, söz konusu metot bu referans aracılığıyla çağrılabılır. Yani, bir delege, referansta bulunduğu metodu çağırabilir.

Bir metoda yönelik referans fikri başlangıçta tuhaf görünebilir, çünkü genellikle biz nesnelere atıfta bulunan referansları düşünürüz; fakat gerçekte ikisi arasında küçük bir fark vardır. Elinizdeki kitabın daha önceki bölümlerinde de açıklandığı gibi, bir referans aslında bir bellek adresidir. Böylece, bir nesneye yönelik bir referans da aslında söz konusu nesnenin adresidir. Bir metot其实 bir nesne olmasa bile, metodun da bellekte fiziksel bir konumu vardır ve metodun başlangıç adresi, metot çağrıldığı zaman çağrılan adrestir. Bu adres bir delegeye atanabilir. Bir delege bir kez bir metoda referansta bulunduktan sonra, söz konusu metot bu delegenin aracılığıyla çağrılabılır.

**NOT**      Eğer C/C++ dillerine aşınaysanız, C#'taki bir delegenin C/C++'taki fonksiyon işaretçisine benzediğini bilmek işinize yarayacaktır.

Bir programın çalışması sırasında, delegenin referansta bulunduğu metodu değiştirmek suretiyle, farklı metotları çağrırmak için aynı delegenin kullanılabilceğini kavramak önemlidir. Bu nedenle, bir delege tarafından çağrılabilecek metot, derleme zamanında değil, çalışma zamanında belirlenir. Bu, delegelerin sağladığı başlıca avantajdır.

Bir delege, **delegate** anahtar kelimesi kullanılarak deklare edilir. Delege deklarasyonunun genel şekli aşağıda gösterilmiştir:

```
delegate dönüş-tipi isim(parametre-listesi);
```

Burada **dönüş-tipi**, söz konusu delegenin çağrıracağı metotlar tarafından döndürülen değerin tipidir. Delegenin ismi **isim** ile belirtilir. Delege aracılığıyla çağrılan metotların gerektirdiği parametreler **parametre-listesi**'nde belirtilir. Bir delege yalnızca, dönüş tipi ve parametre listesi (yani, imzası) deklarasyonunda belirtilen değerlerle eşlenen metotları çağrıabilir.

Bir delege, ya bir nesne ile ilişkilendirilmiş bir örnek metodu ya da bir sınıf ile ilişkilendirilmiş bir **static** metodu çağrılabılır. Önemli olan, yalnızca söz konusu metodun dönüş tipinin ve imzasının delegeninkilerle uyusmasıdır.

Delegelerin kullanımlarını anlamak amacıyla gelin, aşağıda gösterilen basit bir örnekle başlayalım:

```
// Basit bir delege ornegi.

using System;

// Bir delege deklare et.
delegate string strMod(string str);
class DelegateTest {
    // Bosluklari tire ile degistirir.
    static string replaceSpaces(string a) {
        Console.WriteLine("Replaces spaces with hyphens.");
        return a.Replace(' ', '-');
    }

    // Bosluklari yok et.
    static string removeSpaces(string a) {
        string temp = "";
        int a;

        Console.WriteLine("Removing spaces.");
        for(i = 0; i < a.Length; i++)
            if(a[i] != ' ') temp += a[i];

        return temp;
    }

    // Bir karakter katarini ters cevir.
    static string reverse(string a) {
        string temp = "";
        int i, j;

        Console.WriteLine("Reversing string.");
        for(j = 0; i = a.Length-1; i >= 0; i--, j++)
            temp += a[i];

        return temp;
    }

    public static void Main() {
        // Bir delege yapislandir.
        strMod strOp = new strMod(replaceSpaces);
        string str;

        // Delege aracılıgiyla metotlari cagir.
        str = strOp("This is a test.");
        Console.WriteLine("Resulting string: " + str);
        Console.WriteLine();

        strOp = new strMod(removeSpaces);
        str = strOp("This is a test.");
        Console.WriteLine("Resulting string: " + str);
        Console.WriteLine();
    }
}
```

```

        strOp = new strMod(reverse);
        str = strOp("This is a test.");
        Console.WriteLine("Resulting string: " + str);
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Replaces spaces with hyphens.
Resulting string: This-is-a-test.

```

```

Removing spaces.
Resulting string: Thisisatest.

```

```

Reversing string.
Resulting string: .tset a si sihT

```

Şimdi gelin bu programı yakından inceleyelim. Program, bir **string** parametresi alan ve bir **string** döndüren **strMod** adında bir delege deklare etmektedir. **DelegateTest** içinde her biri eşlenen bir imzaya sahip olan üç **static** metot deklare edilmektedir. Bu metotlar bir tür **string** değişikliği gerçekleştirmektedir. **replaceSpaces()**'in boşlukları tire ile değiştirmek için **Replace()** adında, **string** metotlarından birini kullandığına dikkat edin.

**Main()**'de **strOp** adında bir **strMod** referansı oluşturulur ve buna **replaceSpaces()**'a atıfta bulunan bir referans atanır. Aşağıdaki satırda özellikle dikkat edin:

```
strMod strOp = new strMod(replaceSpaces);
```

**replaceSpaces()**'in parametre olarak nasıl aktarıldığına dikkat edin. Metodun yalnızca ismi kullanılmaktadır; parametreleri belirtilmemiştir. Bu genelleştirilebilir. Bir delegeyi örneklerken yalnızca delegenin referansta bulunmasını istediğiniz metodun ismini belirtirsiniz. Ayrıca, metodun deklarasyonu, delegenin deklarasyonuyla eşlenmelidir. Eğer eşlenmeyorsa derleme sırasında bir hata ortaya çıkacaktır.

Sonra, aşağıda gösterildiği gibi **strOp** adındaki delege örneği aracılığıyla **replaceSpacea()** çağrırlar:

```
str = strOp("This is a test.");
```

**strOp**, **replaceSpaces()**'a referansta bulunduğu için burada çağrılan **replaceSpaces()**'dır.

Daha sonra **strOp**'a, **removeSpaces()**'e atıfta bulunan bir referans atanır ve **strOp** tekrar çağrırlar. Bu kez **removeSpaces()** çağrılmıştır.

Son olarak; **strOp**'a, **reverse()**'e atıfta bulunan bir referans atanır ve **strOp** çağrırlar. Bu, **reverse()**'in çağrılmamasıyla sonuçlanır.

Bu örnek, çok önemli bir özelliğe sahiptir: **strOp**'a yapılan çağrı, **strOp**'un çağrıldığı sırada **strOp**'un referansta bulunduğu metodun da çağrılmasıyla sonuçlanır. Böylece, hangi metodun çağrılacığı derleme zamanında değil, çalışma zamanında çözümlenir.

Yukarıdaki örnekle **static** metotlar kullanılmış olsa da, bir delege ayrıca örnek metotlara da referansta bulunabilir. Ancak bunu bir nesne referansı aracılığıyla gerçekleştirmelidir. Örneğin, yukarıdaki programın yeniden yazılmış bir versiyonu karakter katarı işlemlerinin **StringOps** adında bir sınıf içine yerleştirilmiş şekilde aşağıda yer almaktadır:

```
// Delegeler örnek metotlara da referansta bulunabilir.

using System;

// Bir delege deklare et.
delegate string strMod(string str);

class StringOps {
    // Boslukları tire ile değiştir.
    public string replaceSpaces(string a) {
        Console.WriteLine("Replaces spaces with hyphens.");
        return a.Replace(' ', '-');
    }

    // Boslukları yok et.
    public string removeSpaces(string a) {
        string temp = "";
        int i;

        Console.WriteLine("Removing spaces.");
        for(i = 0; i < a.Length; i++)
            if(a[i] != ' ') temp += a[i];

        return temp;
    }

    // Bir karakter katarını ters çevir.
    public string reverse[string a) {
        string temp = "";
        int i, j;

        Console.WriteLine("Reversing string.");
        for(j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];

        return temp;
    }
}

class DelegateTest {
    public static void Main() {
        StringOps so = new StringOps(); // StringOps'un bir
                                       ornegini olustur
```

```

// Bir delege yapılandır.
strMod strOp = new strMod(so.replaceSpaces);
string str;

// Delegeler aracılığıyla metodları çağır.
str = strOp("This is a test.");
Console.WriteLine("Resulting string: " + str);
Console.WriteLine();

strOp = new strMod(so.removeSpaces);
str = strOp("This is a test.");
Console.WriteLine("Resulting string: " + str);
Console.WriteLine();

strOp = new strMod(so.reverse);
str = strOp("This is a test.");
Console.WriteLine("Resulting string: " + str);
}
}

```

Bu program, ilk programla aynı çıktıyı üretir, fakat bu kez söz konusu delege, metodlara **StringOps**'ın bir örneği üzerinden referanssta bulunur.

## Çoklu Çağrı

Bir delegenin en heyecan verici özelliklerinden biri *çoklu çağrıyi* (*multicasting*) desteklemesidir. Basit bir ifadeyle çoklu çağrı, bir delege etkin kılındığında otomatik olarak çağrılmış metotlar için bir *çağrı listesi* ya da bir çağrı zinciri oluşturma becerisidir. Bu tür bir zinciri oluşturmak çok kolaydır. Yapmanız gereken yalnızca bir delege örneklemek, sonra da metodları zincire eklemek için **+=** operatörünü kullanmaktır. Bir metodu zincirden çıkarmak için **-=** kullanın. (Delegeleri eklemek ve çıkarmak için **+**, **-** ve **=** işaretlerini ayrı ayrı da kullanabilirsiniz, fakat **+=** ve **-=** kullanımı çok daha yaygındır.) Tek kısıtlama, çoklu çağrı uygulanmakta olan delegenin dönüş tipinin **void** olması gereklidir.

İşte bir çoklu çağrı örneği. Bu örnek, önceki örnekler üzerinde şu değişiklikler yapılarak elde edilmiştir: Karakter katarını manipüle eden metodun dönüş tipi **void** olarak değiştirilmiştir; değiştirilen karakter katarını, metodu çağrıran koda döndürmek için bir **ref** parametresi kullanılmıştır.

```

// çoklu çağrıyi gösterir.

using System;

// Bir delege deklare et.
delegate void strMod(ref string str);

class StringOps {
    // Boslukları tire ile değiştirir.
    static void replaceSpaces(ref string a) {
        Console.WriteLine("Replaces spaces with hyphens.");
    }
}

```

```
a = a.Replace(' ', '-');  
}  
  
// Bosluklari yok et.  
static void removeSpaces(ref string a) {  
    string temp = "";  
    int i;  
  
    Console.WriteLine("Removing spaces.");  
    for(i = 0; i < a.Length; i++)  
        if(a[i] != ' ') temp += a[i];  
  
    a = temp;  
}  
  
// Bir karakter katarini ters cevir.  
static void reverse(ref string a) {  
    string temp = "";  
    int i, j;  
  
    Console.WriteLine("Reversing string.");  
    for(j = 0, i = a.Length-1; i >= 0; i--, j++)  
        temp += a[i];  
  
    a = temp;  
}  
  
public static void Main() {  
    // Delegeleri yapislandir.  
    strMod strOp;  
    strMod replaceSp = new strMod(replaceSpaces);  
    strMod removeSp = new strMod(removeSpaces);  
    strMod reverseStr = new strMod(reverse);  
    string str = "This is a test";  
  
    // Coklu cagriyi hazirla.  
    strOp = replaceSp;  
    strOp += reverseStr;  
  
    // coklu cagriyi cagir.  
    strOp(ref str);  
    Console.WriteLine("Resulting string: " + str);  
    Console.WriteLine();  
  
    // replace'i kaldır ve remove'u ekle.  
    strOp -= replaceSp;  
    strOp += removeSp;  
  
    str = "This is a test."; // karakter katarini sifirla  
  
    // coklu cagriyi cagir.  
    strOp(ref str);  
    Console.WriteLine("Resulting string: " + str);  
    Console.WriteLine();  
}
```

```
}
```

Çıktı şu şeklidedir:

```
Replaces spaces with hyphens.  
Reversing string.  
Resulting string: tset-a-si-sihT
```

```
Reversing string.  
Removing spaces.  
Resulting string: .tsetasisihT
```

**Main()**'de dört delege örneği oluşturulmaktadır. Bunlardan biri olan **strOp**'a ilk değer verilmemiştir. Diğer üçü karakter katarı üzerinde değişiklik yapan belirli metodlara referansta bulunmaktadır. Sonra, **removeSpaces()**'i ve **reverse()**'i çağrıran bir çoklu çağrı oluşturulur. Bu, aşağıdaki satırlar yardımıyla gerçekleştirilir:

```
strOp = replaceSp;  
strOp += reverseStr;
```

Öncelikle, **strOp**'a **replaceSp**'ye atıfta bulunan bir referans atanmaktadır. Sonra, **+=** kullanılarak **reverseStr** eklenmektedir. **strOp** çağrıldığında her iki metod da çağrılr; bunun sonucu olarak, çıktıdan da görüleceği üzere boşluklar tire ile değiştirilir ve karakter katarı ters çevrilir.

Daha sonra, aşağıdaki satır kullanılarak **replaceSp** zincirden çıkarılır:

```
strOp -= replaceSp;
```

**removeSp** şu satır kullanılarak zincire eklenir:

```
strOp += removeSp;
```

Sonra, **strOp** yeniden çağrılr. Bu kez, boşluklar yok edilmiş ve karakter katarı da ters çevrilmiş durumdadır.

Delege zincirleri güçlü bir mekanizmadır, çünkü bir bütün olarak çalıştırılabilen bir takım metodlar tanımlanmanıza olanak tanırlar. Bu, bazı tip kodların yapısını büyütебilir. Ayrıca, delege zincirlerinin olaylar (events) açısından da özel bir değeri olduğunu yakında öğreneceksiniz.

## System.Delegate

Tüm delegeler **System.Delegate**'ten kapalı olarak türetilen sınıflardır. Normalde **System.Delegate**'in üyelerini doğrudan kullanmanız gerekmekz. Bu kitapta **System.Delegate** açık olarak kullanılmamaktadır. Ancak, **System.Delegate**'in üyeleri belirli özelleştirilmiş durumlarda kullanışlı olabilirler.

## Delegelere Neden Gerek Vardır?

Önceki örnekler, delegelerin ardından “nasıl” gerçeğini göstermiş olsalar da, aslında “neden” sorusunun yanıtını vermemişlerdir. Genel olarak delegeler başlıca iki nedenden ötürü kullanışlıdır. Öncelikle, bir sonraki bölümde de görüleceği gibi, delegeler olayları destekler. İkincisi, delegeler sayesinde programınız, derleme zamanında hangi metodun çalıştıracağını tam olarak bilmese de, çalışma zamanında söz konusu metodu çalıştırabilecektir. Bu beceri, bileşenlerin yerleştirilmesine olanak tanıyan bir çatı oluşturmak istediğinizde oldukça işe yarar. Örneğin, bir çizim programı hayal edin (standart Windows Paint donatısı gibi bir program). Bir delege kullanarak kullanıcının özel renk filtreleri ya da görüntü çözümleyicileri kullanmasına olanak tanıyabilirsiniz. Üstelik, kullanıcı bu filtrelerden ya da çözümleyicilerden oluşan bir sıra da oluşturabilir. Bu tür bir plan, delege kullanılarak kolaylıkla ele alınabilir.

## Olaylar

Delege temeli üzerine inşa edilen bir başka önemli C# özelliği ise *olaydır* (*event*). Bir olay aslında bir faaliyetin meydana geldiğini bildiren otomatik bir bildirgedir. Olaylar şu şekilde çalışır: Bir olayla ilgilenen bir nesne, söz konusu olay için bir olay yöneticisi (*event handler*) kaydeder. Olay meydana geldiğinde tüm olay yöneticileri çağrılır. Olay yöneticileri delegelerle simgelenir. Olaylar bir sınıfın üyeleridir ve **event** anahtar kelimesi kullanılarak deklare edilirler. Olayların en sık kullanılan biçimi aşağıda gösterilmiştir:

```
event olay-delegesi nesne; I
```

Burada **olay-delegesi**, olayı desteklemek için kullanılan delegenin adıdır; **nesne** ise oluşturulmakta olan spesifik olay nesnesinin adıdır.

Gelin şimdi çok basit bir örnekle başlayalım:

```
// Cok basit bir olay gosterimi.

using System;

// Bir olay icin bir delege deklare et.
delegate void MyEventHandler();

// Bir olay sinifi deklare et.
class MyEvent {
    public event MyEventHandler SomeEvent;

    // Olayi ateslemek icin bu cagrılır.
    public void OnSomeEvent() {
        if(SomeEvent != null)
            SomeEvent();
    }
}

class EventDemo {
```

```
// Bir olay yoneticisi.  
static void handler() {  
    Console.WriteLine("Event occurred");  
}  
  
public static void Main() {  
    MyEvent evt = new MyEvent();  
  
    // handler()'i olay listesine ekle.  
    evt.SomeEvent += new MyEventHandler(Handler);  
  
    // Olayi atesle.  
    evt.OnSomeEvent();  
}  
}
```

Bu program aşağıdaki çıktıyı ekranda gösterir:

```
Event occurred
```

Basit olmasına rağmen bu program, düzgün bir olay yönetici için gerekli olan tüm öğeleri içermektedir. Gelin şimdi programı dikkatle inceleyelim.

Program, olay yönetici için aşağıda gösterildiği gibi bir delege deklare ederek başlamaktadır:

```
delegate void MyEventHandler();
```

Tüm olaylar bir delege aracılığıyla etkili hale getirilirler. Böylece, olay delegesi olayın imzasını tanımlamaktadır. Bu örnekte olay parametreleri haricinde hiç bir parametreye izin verilmemiştir. Olaylara genellikle multicast uygulandığı için bir olay **void** döndürmelidir.

Sonra, **MyEvent** adında bir olay sınıfı oluşturulur. Bu sınıfın içinde **someEvent** adında bir olay nesnesi deklare edilir. Bunun için aşağıdaki satır kullanılır:

```
public event MyEventHandler SomeEvent;
```

Söz dizimine dikkat edin. Tüm olay tipleri bu şekilde deklare edilirler.

Ayrıca **OnSomeEvent()** metodu da **MyEvent**'in içinde deklare edilmektedir. **OnSomeEvent()** bir programın bir olaya işaret etmek (ya da "ateslemek") için çağrıracığı bir metottur. (Yani, olay meydana geldiğinde çağrılan metot budur.) Söz konusu metot, **SomeEvent** delegesi aracılığıyla, aşağıda gösterilen şekilde, bir olay yöneticiyi çağrırlar:

```
if(SomeEvent != null)  
    SomeEvent();
```

Bir yöneticinin sadece ve sadece **someEvent null** değere sahip olmadığından çağrılmışına dikkat edin. Programınızın diğer bölümleri olay bildirgelerini almak için bir olayı kaydetmeye ilgi göstirmelidir. Bu nedenle, herhangi bir olay yöneticiyi kaydedilmeden önce **OnSomeEvent()**'in çağrılabilmesi mümkündür. **null** değerine sahip bir nesnenin

çağırılmasını önlemek amacıyla olay delegesinin `null` değerine sahip olmadığını garanti etmek için olay delegesi test edilmelidir.

`EventDemo` içinde `handler()` adında bir olay yöneticisi oluşturulur. Bu örnekte olay yöneticisi yalnızca ekranda bir mesaj görüntüler, fakat diğer yöneticiler daha anlamlı faaliyetlerde bulunabilirler. `Main()`'de, aşağıda gösterildiği gibi, bir `MyEvent` nesnesi oluşturulur ve `handler()` bu olay için bir yönetici olarak kaydedilir:

```
My Event evt = new MyEvent();

// handler()'i olay listesine ekle.
evt.SoneEvent += new MyEventHandler(handler);
```

Dikkat ederseniz, yönetici `+=` operatörü kullanılarak eklenir. Olaylar yalnızca `+=` ve `-=` operatörlerini desteklerler. Bu örnekte `handler()`, `static` bir metottur; fakat olay yöneticileri ayrıca örnek metot da olabilirler.

Son olarak, söz konusu olay aşağıda gösterildiği gibi ateşlenmektedir:

```
// Olayı atesle.
evt.OnSomeEvent();
```

`OnSomeEvent()`'ı çağırmak kayıtlı olay yöneticilerinin tümünün çağrılmasına neden olur. Bu örnekte yalnızca tek bir kayıtlı yönetici mevcuttur; fakat bir sonraki bölümde açıklandığı gibi daha fazla da olabilirildi.

## Bir Çoklu Çağrı Olay Örneği

Tıpkı delegeler gibi olaylara da çoklu çağrı uygulanabilir. Bu sayede, bir olay bildirgesine birden fazla nesnenin yanıt vermesi mümkün olur. İşte bir olay çoklu çağrısı örneği:

```
// Bir olay coklu cagrisi gosterimi.

using System;

// Bir olay icin bir delege deklare et.
delegate void MyEventHandler();

// Bir olay sinifi deklare et.
class MyEvent {
    public event MyEventHandler SomeEvent;

    // Olayi ateslemek icin bu cagrilar.
    public void OnSomeEvent() {
        if(SomeEvent != null)
            SomeEvent();
    }
}

class X {
```

```
public void Xhandler() {
    Console.WriteLine("Event received by X object");
}

class Y {
    public void Yhandler() {
        Console.WriteLine("Event received by Y object");
    }
}

class EventDemo {
    static void handler() {
        Console.WriteLine("Event received by EventDemo");
    }
}

public static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();

    // Yoneticileri olay listesine ekle.
    evt.SomeEvent += new MyEventHandler(handler);
    evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
    evt.SomeEvent += new MyEventHandler(yOb.Yhandler);

    // Olayi atesle.
    evt.OnSomeEvent();
    Console.WriteLine();

    // Yoneticiyi ortadan kaldır.
    evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
    evt.OnSomeEvent();
}
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Event received by EventDemo
Event received by X object
Event received by Y object

Event received by EventDemo
Event received by Y object
```

Bu örnek **x** ve **y** adında iki ilave sınıf oluşturmaktadır. Bu sınıflar da ayrıca **MyEventHandler** ile uyumlu olay yöneticileri tanımlamaktadırlar. Böylece, bu yöneticiler olay zincirinin bir halkası haline gelebilirler. **x** ve **y**'nin içindeki yöneticilerin **static** olmadığına dikkat edin. Bunun anlamı şudur: Bunların her biri için nesneler oluşturulmalı ve her nesne örneğine bağlanan yönetici olay zincirine eklenmelidir. Örnek ve **static** yöneticiler arasındaki farklar bir sonraki bölümde incelenmektedir.

## Olay Yöneticileri Olarak static Metotlara Karşı Örnek Metotlar

Örnek rnetotlar ve **static** metotların her ikisi de olay yöneticileri olarak kullanılabilir olmalarına rağmen, önemli bir açıdan farklılık gösterirler. **static** bir metot olay yönetici olarak kullanıldığında, olay bildirgesi sınıfı (ve dolaylı olarak sınıfın tüm nesnelerine) uygulanır Bir örnek metot olay yönetici olarak kullanıldığında ise, olaylar spesifik nesne örneklerine gönderilir. Böylece, bir sınıfın olay bildirgesi almak isteyen her bir nesnesi ayrı ayrı kaydedilmelidir. Pratikte olay yöneticilerinin birçoğu örnek metotlardır; fakat bu durum, kuşkusuz söz konusu spesifik uygulamaya bağlı olarak değişebilir. Gelin şimdi bunların her biri için bir örneğe göz atalım.

Aşağıdaki program, olay yönetici olarak bir örnek metot tanımlayan **x** adında bir sınıf oluşturur. Bunun anlamı şudur: Olayları almak için her **x** nesnesi ayrı ayrı kaydedilmelidir. Bu gerçeği göstermek için program, bir olayı **x** tipinde üç nesneye çoklu çağrı uygular.

```
// Ornek olay yoneticileri kullanildiginda
// nesneler bildirgeleri ayrı ayrı alirlar. */

using System;

// Bir olay icin bir delege deklare et.
delegate void MyEventHandler();

// Bir olay sinifi deklare et.
class MyEvent {
    public event MyEventHandler SomeEvent;

    // Olayi ateslemek icin bu cagrilir.
    public void OnSomeEvent() {
        if(SomeEvent != null)
            SomeEvent();
    }
}

class X {
    int id;

    public X(int x) { id = x; }

    /* Bu, olay yoneticisi olarak kullanilacak olan
       bir ornek metottur. */
    public void Xhandler() {
        Console.WriteLine("Event received by object " + id);
    }
}

class EvantDemo {
    public static void Main() {
        MyEvent evt = new MyEvent();
        X o1 = new X(1);
```

```
X o2 = new X(2);
X o3 = new X(3);

evt.SomeEvent += new MyEventHandler(o1.Xhandler);
evt.SomeEvent += new MyEventHandler(o2.Xhandler);
evt.SomeEvent += new MyEventHandler(o3.Xhandler);

// Olayi atesle.
evt.OnSomeEvent();
}

}
```

Bu programın çıktısı aşağıdaki gibidir:

```
Event received by object 1
Event received by object 2
Event received by object 3
```

Çıktıdan görüldüğü gibi, her nesne ilgilendiği olaya kendi ilgisini ayrı ayrı kaydeder, ayrıca her nesne ayrı bir bildirge alır.

Alternatif olarak, bir **static** metot olay yöneticisi olarak kullanıldığında olaylar nesnelerden bağımsız olarak ele alınırlar. Aşağıdaki program bunu gösterir:

```
// Bir static metot olay yoneticisi olarak kullanildiginda bir
// sinif, bildirgeyi alir. */

using System;

// Bir olay icin bir delege deklare et.
delegate void MyEventHandler();

// Bir olay sinifi deklare et.
class MyEvent {
    public event MyEventHandler SomeEvent;

    // Olayi ateslemek icin bu cagrilir.
    public void OnSameEvent() {
        if(SomeEvent != null)
            SomeEvent();
    }
}

class X {
    /* Bu, olay yoneticisi olarak kullanilacak olan
       static bir metottur. */
    public static void Xhandler() {
        Console.WriteLine("Event received by class.");
    }
}

class EventDemo {
    public static void Main() {
        MyEvent evt = new MyEvent();
```

```

        evt.SomeEvent += new MyEventHandler(X.Xhandler);

        // Olayı atesle.
        evt.OnSomeEvent();
    }
}

```

Bu programın çıktısı aşağıdaki gibidir:

```
Event received by class.
```

Programda hiç **x** tipinde bir nesne oluşturulmadığına dikkat edin. Ancak; **handler()** **x**'in bir **static** metodu olduğu için **SomeEvent**'e ilişirilebilir ve **OnSomeEvent()** çağrılığında gerçekleştirilebilir.

## Olay Erişimcilerinin Kullanımı

**event** ifadesinin iki şekli mevcuttur. Önceki örneklerde kullanılan şekli ile olay yöneticisinin çağrı listesini otomatik olarak yöneten olaylar oluşturulmuştur. Olay yöneticilerini listeye eklemek ya da listeden çıkarmak da bu kapsamda yer alıyordu. Böylece, listenin yönetimiyle ilgili işlevselligin herhangi bir kısmını sizin kendi kendinize uygulamanıza gerek kalmıyordu. Bu tür olaylar ayrıntıları sizin adınıza kontrol altına aldıkları için açık farkla en yaygın olarak kullanılan türlerdir. Olay yönetici listesinin işlemlerini kendiniz sağlamanız, belki de bir tür özelleştirilmiş olay depolama mekanizması uygulamanız da her şeye rağmen mümkündür.

Olay yönetici listesini kontrol altına almak için *olay erişimcilerinin* (*event accessor*) kullanımına olanak tanıyan, **event** ifadesinin ikinci şeklini kullanacaksınız. Erişimciler, olay yöneticisinin uygulanmış biçimini üzerinde size kontrol sağlarlar.

Olay yönetici listesini kontrol altına almak için olay erişimcilerin kullanımına olanak tanıyan, **event** ifadesinin ikinci şeklini kullanacaksınız. Erişimciler, olay yönetici listesinin uygulanış biçimini üzerinde kontrol kurmanızı sağlarlar. **event** ifadesinin bu şekli aşağıda gösterilmiştir:

```

event olay-delegesi olay-ismi {
    add {
        // zincire bir olay eklemek için kod
    }

    remove {
        // zincirden bir olay çıkarmak için kod
    }
}

```

**event** ifadesinin bu şekli iki olay erişimcisi içermektedir: **add** ve **remove**. **add** erişimcisi, bir olay yönetici **+=** kullanılarak bir olay zincirine eklendiğinde çağrılır. **remove** erişimcisi ise, bir olay yönetici **--=** kullanılarak zincirden çıkarıldığında çağrılır.

**add** ya da **remove** çağrıldığında eklenilecek ya da çıkarılacak olay yöneticisi parametre olarak aktarılır. Diğer tipteki erişimcilerde olduğu gibi, bu parametre **value** olarak adlandırılır. **add** ve **remove**'u uygulayarak kendi isteklerinize uygun bir olay yöneticisi depolama planı tanımlayabilirsiniz. Örneğin, yöneticileri saklamak için bir dizi, yığın ya da kuyruk kullanabilirsiniz.

İşte, **event** ifadesinin erişimci şeklini kullanan bir örnek. Programda olay yöneticilerini tutmak için bir dizi kullanılmaktadır. Dizi yalnızca üç eleman uzunluğunda olduğu için, herhangi bir anda zincirde yalnızca üç olay yöneticisi tutulabilir.

```
/* Olay cagri listesini yonetmek için
istege gore bir yontem olusturmak. */

using System;

// Bir olay icin bir delege deklare et.
delegate void MyEventHandler();

// En fazla 3 olay tutabilen bir olay sinifi deklare et.
class MyEvent {
    MyEventHandler[] evnt = new MyEventHandler[3];

    public event MyEventHandler SomeEvent {
        // Listeye bir olay ekle.
        add {
            int i;

            for(i = 0; i < 3; i++)
                if(evnt[i] == null) {
                    evnt[i] = value;
                    break;
                }
            if(i == 3) Console.WriteLine("Event list full.");
        }

        // Listededen bir olay cikart.
        remove {
            int i;

            for(i = 0; i < 3; i++)
                if(evnt[i] == value) {
                    evnt[i] = null;
                    break;
                }
            if(i == 3)
                Console.WriteLine("Event handler not found.");
        }
    }

    // Olaylari ateslemek icin bu cagrilir.
    public void OnSomeEvent() {
        for(int i = 0; i < 3; i++)
            if(evnt[i] != null) evnt[i]();
    }
}
```

```
        }

    }

// MyEventHandler kullanan bazi siniflar olustur.
class W {
    public void Whandler() {
        Console.WriteLine("Event received by W object");
    }
}

class X {
    public void Xhandler() {
        Console.WriteLine("Event received by X object");
    }
}

class Y {
    public void Yhandler() {
        Console.WriteLine("Event received by Y object");
    }
}

class Z {
    public void Zhandler() {
        Console.WriteLine("Event received by Z object");
    }
}

class EventDemo {
    public static void Main() {
        MyEvent evt = new MyEvent();
        W wOb = new W();
        X xOb = new X();
        Y yOb = new Y();
        Z zOb = new Z();

        // Yoneticileri olay listesine ekle.
        Console.WriteLine("Adding events.");
        evt.SomeEvent += new MyEventHandler(wOb.Whandler);
        evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
        evt.SomeEvent += new MyEventHandler(yOb.Yhandler);

        // Bunu saklayamaz -- dolu.
        evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
        Console.WriteLine();
        // Olaylari atesle.
        evt.OnSomeEvent();
        Console.WriteLine();

        // Yoneticiyi ortadan kaldir.
        Console.WriteLine("Remove xOb.Xhandler.");
        evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
        evt.OnSomeEvent();

        Console.WriteLine();
    }
}
```

```

    // Tekrar cikarmaya calis.
    Console.WriteLine("Try to remove xOb.Xhandler again.");
    evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
    evt.OnSomeEvent();

    Console.WriteLine();

    // Simdi, Zhandler'i ekle.
    Console.WriteLine("Add zOb.Zhandler.");
    evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
    evt.OnSomeEvent();
}
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Adding events.
Event list full.

Event received by W object
Event received by X object
Event received by Y object

Remove xOb.Xhandler.
Event received by W object
Event received by Y object

Try to remove xOb.Xhandler again.
Event handler not found.
Event received by W object
Event received by Y object

Add zOb.Zhandler.
Event received by W object
Event received by Z object
Event received by Y object

```

Gelin şimdi bu programı yakından inceleyelim. İlk önce, **MyEventHandler** adında bir olay yöneticisi delegesi tanımlanır. Sonra; **MyEvent**, aşağıda gösterildiği gibi, olay yöneticilerinden oluşan üç elemanlı **evnt** adında bir dizi tanımlayarak başlar:

```
MyEventHandler[] evnt = new MyEventHandler[3];
```

Bu dizi, olay zincirine eklenen olay yöneticilerini saklamak için kullanılacaktır. **evnt**'in elemanlarına başlangıçta varsayılan değer olarak **null** atanır.

Sırada, erişimci tabanlı **event** ifadesi vardır. Bu ifade aşağıda gösterilmiştir:

```

public event MyEventHandler SomeEvent {
    // Listeye bir olay ekle.
    add {
        int i;

```

```

        for(i = 0; i < 3; i++)
            if(evnt[i] == null) {
                evnt[i] = value;
                break;
            }
        if(i == 3) Console.WriteLine("Event list full.");
    }

    // Listededen bir olay cikart.
    remove {
        int i;

        for(i = 0; i < 3; i++)
            if(evnt[i] == value) {
                evnt[i] = null;
                break;
            }
        if (i == 3) Console.WriteLine("Event handler not found.");
    }
}

```

Bir olay yöneticisi ekleneceği zaman **add** çağrılr ve yöneticiye atıfta bulunan bir referans (**value** içinde tutulan) **evnt**'in kullanılmayan ilk elemanına yerleştirilir. Eğer boş bir eleman yoksa, bir hata durumu rapor edilir. **evnt** yalnızca üç elemanlı olduğu için yalnızca üç olay yöneticisi saklanabilir. Bir olay yöneticisi çıkarılacağı zaman **remove** çağrılr ve **value** üzerinden aktarılan ve söz konusu yöneticiye atıfta bulunan referans **evnt** dizisi içinde aranır. Eğer bulunursa dizide karşılık gelen elemana **null** değeri atanır, böylece yönetici listeden çıkarılmış olur.

Bir olay ateşlendiğinde **OnSomeEvent()** çağrılr. **OnSomeEvent()**, olay yöneticilerinin her birini sırayla çağırarak **evnt** dizisi üzerinden tekrarlayarak ilerler.

Önceki örneklerde görüldüğü gibi, gerekli olduğunda, olay yöneticileri için istege uygun bir depolama mekanızması uygulamak nispeten kolaydır. Uygulamaların birçoğunda aslında **event**'in erişimci olmayan şeklinin sağladığı varsayılan depolama daha iyidir. **event**'in erişimci tabanlı şekli, belirli özelleştirilmiş durumlarda her şeye rağmen kullanılabilir. Örneğin elinizdeki bir programda olay yöneticilerinin zincire eklendikleri sıraya göre değil de kendi öncelik sıralarına göre çalıştırılması gerekiyorsa, bu durumda yöneticileri saklamak için bir öncelikli kuyruk kullanabilirsiniz.

## Olayların Çeşitli Özellikleri

Olaylar arayüzlerde belirtilebilir. Söz konusu olay, uygulamayı gerçekleştiren sınıflar tarafından sağlanmalıdır. Olaylar **abstract** olarak belirtilebilir. Olayı bir türetilmiş sınıf uygulamalıdır. Ancak, erişimci tabanlı olaylar **abstract** olamaz. Bir olay **sealed** olarak belirtilebilir. Bir olay sanal (**virtual**) olabilir; yani, türetilmiş bir sınıf içinde devre dışı bırakılabilir.

## .NET'te Olaylarla İlgili Esaslar

C# istediğiniz her tipten olayı yazmanıza olanak tanımaktadır. Bununla birlikte, bileşenlerin .NET Framework ile uyumluluğu için Microsoft'un bu amaca yönelik yerleştirdiği esasları takip etmeniz gerekecektir. Bu esasların özünde, olay yöneticilerinin iki parametreye sahip olması gerektiği yer almaktadır. İlk parametre, olay üreten nesneye bir referanstır. İkincisi ise, yönetici tarafından istenen diğer bilgileri içeren **EventArgs** tipinde bir parametredir. Böylece, .NET uyumlu olay yöneticileri genel olarak şu şekilde olacaktır:

```
void handler(object kaynak, EventArgs arg) {  
    // ...  
}
```

Genellikle **kaynak** parametresine metodu çağrıran kod tarafından **this** aktarılır. **EventArgs** parametresi ek bilgiler içerir ve gerekli değilse dikkate alınmayabilir.

**EventArgs** sınıfının kendisi bir yöneticiye ek veri aktarmak için kullanacağınız alanlar içermez. Bunun yerine **EventArgs**, gerekli alanları içeren bir sınıfı kendisinden türetebileceğiniz bir temel sınıf olarak kullanılır. Ancak, birçok yönetici ek veri gerektirmediği için, veri içermeyen bir nesneyi belirten **Empty** adındaki **static** alan **EventArgs** içinde yer almaz.

İşte, .NET uyumlu bir olay oluşturan bir örnek:

```
// .NET-uyumlu bir olay.  
  
using System;  
  
// EventArgs' dan bir sınıf turet.  
class MyEventArgs : EventArgs {  
    public int eventnum;  
}  
  
// Bir olay için bir delege deklare et.  
delegate void MyEventHandler(object source, MyEventArgs arg);  
  
// Bir olay sınıfı deklare et.  
class MyEvent {  
    static int count = 0;  
  
    public event MyEventHandler SomeEvent;  
  
    // Bu, SomeEvent'i atesler.  
    public void OnSomeEvent() {  
        MyEventArgs arg = new MyEventArgs();  
  
        if(SomeEvent != null) {  
            arg.eventnum = count++;  
            SomeEvent(this, arg);  
        }  
    }  
}
```

```

    }

    class X {
        public void handler(object source, MyEventArgs arg) {
            Console.WriteLine("Event " + arg.eventnum +
                " received by an X object.");
            Console.WriteLine("Source is " + source);
            Console.WriteLine();
        }
    }

    class Y {
        public void handler(object source, MyEventArgs arg) {
            Console.WriteLine("Event " + arg.eventnum +
                " received by a Y object.");
            Console.WriteLine("Source is " + source);
            Console.WriteLine();
        }
    }

    class EventDemo {
        public static void Main() {
            X ob1 = new X();
            Y ob2 = new Y();
            MyEvent evt = new MyEvent();

            // handler()'i olay listesine ekle.
            evt.SomeEvent += new MyEventHandler(ob1.handler);
            evt.SomeEvent += new MyEventHandler(ob2.handler);

            // Olayi atesle.
            evt.OnSomeEvent();
            evt.OnSomeEvent();
        }
    }
}

```

Çıktı, aşağıdadır:

```
Event 0 received by an X object.
Source is MyEvent
```

```
Event 0 received by a Y object.
Source is MyEvent
```

```
Event 1 received by an X object.
Source is MyEvent
```

```
Event 1 received by a Y object.
Source is MyEvent
```

Bu örnekte **MyEventArgs**, **EventArgs**'tan türetilmiştir. **MyEventArgs** kendisi yalnızca tek bir alan ekler; **eventnum**. Olay yöneticisi delegesi **MyEventHandler** şimdi .NET Framework tarafından istenen iki parametre almaktadır. Önceden açıklandığı gibi, ilk parametre olayın üreticisine bir referanştır. İkincisi ise, **EventArgs**'a bir referans ya da

**EventArgs**'dan türetilen bir sınıfıtır. Bu örnekte **MyEventArgs** tipinde bir nesneye referanstır.

## EventHandler Kullanımı

Bir çok olay için **EventArgs** parametresi kullanılmaz. Bu gibi durumlarda kodun oluşturulmasına kolaylık sağlama amacıyla .NET Framework, **EventHandler** adında standart bir delege tipi içerir. **EventHandler**, ek bir bilgi gerektirmeyen olay yöneticilerini deklare etmek için kullanılabilir. İşte, **EventHandler** kullanan bir örnek:

```
// Standart EventHandler delegatesini kullan.

using System;

// Bir olay sınıfı deklare et.
class MyEvent {
    public event EventHandler SomeEvent; // EventHandler
                                         // delegatesini kullanır

    // SomeEvent'i ateslemek için bu çağrılr.
    public void OnSomeEvent() {
        if(SomeEvent != null)
            SomeEvent(this, EventArgs.Empty);
    }
}

class EventDemo {
    static void handler(object source, EventArgs arg) {
        Console.WriteLine("Event occurred");
        Console.WriteLine("Source is " + source);
    }

    public static void Main() {
        MyEvent evt = new MyEvent();

        // handler()'i olay listesine ekle.
        evt.SomeEvent += new EventHandler(handler);

        // Olayı atesle.
        evt.OnSomeEvent();
    }
}
```

Bu örnekte, **EventArgs** parametresi kullanılmamıştır; yer belirleyici bir nesne olan **EventArgs.Empty** parametre olarak aktarılmıştır. Çıktı aşağıda gösterildiği gibidir:

```
Event occurred
Source is MyEvent
```

## Olayları Uygulamak: Bir Örnek Çalışma

Olaylar, Windows gibi mesaj tabanlı ortamlarda sıkça kullanılırlar. Bu tür bir ortamda bir program, bir mesaj alana kadar bekler ve sonra uygun bir davranışta bulunur. Bu tür bir mimari C# stili olay yönetimi için çok uygundur, çünkü çeşitli mesajlar için olay yöneticileri oluşturmak ve bir mesaj alındığında yalnızca bir yöneticiyi çağrırmak mümkündür. Örneğin, farenin sol tuşuna basılmasına karşılık gelen mesaj bir olaya bağlanabilir. Sol tuşa basılmasıyla, kayıtlı tüm yöneticilere haber verilir.

Bu yaklaşımı gösteren bir Windows programı geliştirmek bu bölümün kapsamının dışında kalmasına rağmen, bu tür bir yaklaşımın nasıl çalışacağı hakkında fikir vermek mümkündür. Aşağıdaki program klavye vuruşlarını işleyen bir olay yöneticisi oluşturmaktadır. Söz konusu olay **KeyPress** olarak adlandırılır ve ne zaman bir tuşa basılsa **OnKeyPress()** çağrılarak ilgili olay ateşlenir.

```
// Klavye vuruslariyla ilgili bir olay ornegi.

using System;

// Tusu tutan, istege uygun bir EventArgs sınıfı turet.
class KeyEventArgs : EventArgs {
    public char ch;
}

// Bir olay icin bir delege deklare et.
delegate void KeyHandler(object source, KeyEventArgs arg);

// Tus vuruslariyla ilgili bir olay sınıfı deklare et.
class KeyEvent {
    public event KeyHandler KeyPress;

    // Bir tusa basildiginda bu cagrilir.
    public void OnKeyPress(char key) {
        KeyEventArgs k = new KeyEventArgs();

        if(KeyPress != null) {
            k.ch = key;
            KeyPress(this, k);
        }
    }
}

// Tusa basildiginin haberini alan bir sınıf.
class ProcessKey {
    public void keyhandler(object source, KeyEventArgs arg) {
        Console.WriteLine("Received keystroke: " + arg.ch);
    }
}

// Tusa basildiginin haberini alan bir baska sınıf.
class CountKeys {
    public int count = 0;
```

```
public void keyhandler(object source, KeyEventArgs arg) {
    count++;
}
}

// KeyEvent'i göster.
class KeyEventDemo {
    public static void Main() {
        KeyEvent kevt = new KeyEvent();
        ProcessKey pk = new ProcessKey();
        CountKeys ck = new CountKeys();
        char ch;

        kevt.KeyPress += new KeyHandler(pk.keyhandler);
        kevt.Keypress += new KeyHandler(ck.keyhandler);

        Console.WriteLine("Enter some characters. " +
                           "Enter a period to stop.");
        do {
            ch = (char) Console.Read();
            kevt.OnKeyPress(ch);
        } while(ch != '.');
        Console.WriteLine(ck.count + " keys pressed.");
    }
}
```

İşte örnek bir çalışma:

```
Enter some characters. Enter a period to stop.
test.
Received keystroke: t
Received keystroke: e
Received keystroke: s
Received keystroke: t
Received keystroke: .
5 keys pressed.
```

Program, basılan bir tuşu bir olay yöneticisine aktarmak için kullanılan **KeyEventArgs** adında bir sınıf türeterek başlar. Sonra, **KeyHandler** adında bir delege, tuşa basma olayları için ilgili olay yöneticisini tanımlar. **KeyEvent** sınıfı, tuşa basma olayını sınıf içine paketlemektedir.

Program, basılan tuşları ele alan iki sınıf oluşturmaktadır: **ProcessKey** ve **CountKeys**. **ProcessKey** sınıfı, basılan tuşları görüntüleyen **keyhandler()** adında bir yönetici içermektedir. **CountKeys** ise basılan tuşların sayısını sürekli olarak takip etmektedir. **Main()**'de bir **KeyEvent** nesnesi oluşturulur. Sonra, **ProcessKey** ve **CountKey**'nın nesneleri tanımlanır ve bunların **keyhandler()** metodlarına yönelik referansları **kevt.KeyPress** çağrı listesine eklenir. Daha sonra, bir tuşa basıldığında **kevt.OnKeyPress()**'ı çagıran bir döngü başlatılır. Bu, kayıtlı olay yöneticilerinin durumdan haberdar edilmesini sağlar.

# **İSİM UZAYLARI, ÖN İŞLEMCI VE ASSEMBLY'LER**

Bu bölümde, bir programın organizasyonu ve erişilebilirliği üzerinde size daha fazla kontrol sağlayan üç C# özelliği ele alınmaktadır. Bunlar isim uzayları (namespaces), önişlemci ve assembly'lerdir.

## İsim Uzayları

İsim uzayı (namespace), C#'ta temel bir kavram olduğu için Bölüm 2'de kısaca bahsedilmiştir. Aslında, her C# programı şu veya bu şekilde bir isim uzayından yararlanır. Bundan önce isim uzaylarını ayrıntılı olarak incelemeye gerek görmedik, çünkü C#, programınız için otomatik olarak bir varsayılan isim uzayı sağlar. Böylece, baştaki böölülerdeki programlar yalnızca varsayılan isim uzayını kullanmışlardır. Gerçek dünyada ise birçok programın kendi isim uzaylarını geliştirmeleri ya da diğer isim uzaylarıyla etkileşmeleri gerekecektir. Bu bölümde bunlar ayrıntılı olarak incelelmektedir.

Bir *isim uzayı*, bir takım isimleri diğerlerinden ayrı tutmanın yollarından biri olan bir deklaratif alan tanımlar. Aslında, bir isim uzayında deklare edilen isimler bir başka isim uzayında deklare edilen aynı isimlerle karışmayacaktır. .NET Framework kütüphanesi (yani, C# kütüphanesi) tarafından kullanılan isim uzayı **System**'dır. Her programın başına yakın bir yere aşağıdaki satırı dahil etmenizin nedeni budur;

```
using System;
```

Bölüm 14'te gördüğünüz gibi I/O sınıfları, **System**'ın bir alt isim uzayı olan **System.IO** içinde tanımlanmıştır. C# kütüphanesinin diğer parçalarını tutan ve **System**'in altında yer alan daha birçok isim uzayı mevcuttur.

İsim uzayları önemlidir, çünkü son birkaç yıl içinde değişken, metot, özellik ve sınıf isimlerinde bir patlama olmuştur. Kütüphane rutinleri, üçüncü parti kodlar ve kendi kodlarınız da bu kapsamda yer almaktadır. İsim uzayı olmadan bu isimlerin tümü global isim uzayındaki boşluklar için yarışacaktı ve karışıklıklar ortaya çıkacaktı. Örneğin, eğer programınız **Finder** adında bir sınıf tanımlamış olsa; bu sınıf, sizin programınızı kullanan üçüncü parti kütüphane tarafından sağlanan **Finder** adındaki bir başka sınıf ile çakışırı. Neyse ki, isim uzayları bu tür problemleri önlemektedir, çünkü bir isim uzayı kendi içinde deklare edilen isimlerin erişilebilirliğini lokalize eder.

## İsim Uzayını Deklare Etmek

Bir isim uzayı **namespace** anahtar kelimesi kullanılarak deklare edilir. **namespace**'in genel olarak şekli şöyledir:

```
namespace isim {  
    // üyeleri  
}
```

Burada **isim**, isim uzayının ismidir. Bir isim uzayı içinde tanımlanan her şey, o isim uzayının kapsamı içindedir. Böylece, bir isim uzayı bir kapsam (scope) tanımlar. Bir isim uzayı

îçinde sınıf, yapı, delege, numaralandırma, arayüz ya da bir başka isim uzayı deklare edebilirsiniz.

İşte, **Counter** adında bir isim uzayı oluşturan bir **namespace** örneği. Program, geriye doğru sayan **CountDown** adında basit bir sayaç uygulamak için kullanılan ismi lokalize eder.

```
// Sayaclar icin bir isim uzayı deklare et.

namespace Counter {
    // Geriye doğru sayan basit bir sayaç.
    class ConntDown {
        int val;

        public CountDown(int n) {
            val = n;
        }

        public void reset{int n) {
            val = n;
        }

        public int count() {
            if(val > 0) return val--;
            else return 0;
        }
    }
}
```

Bu örnekte **CountDown** sınıfı **Counter** isim uzayı tarafından tanımlanan kapsam içinde deklare edilmektedir.

İşte **Counter** isim uzayıının kullanımını gösteren bir program:

```
Bir isim uzayı goster.

using System;

// Sayaclar icin bir isim uzayı deklare et.
namespace Counter {
    // Geriye doğru sayan basit bir sayaç.
    class CountDown {
        int val;

        public CountDown(int n) { val = n; }

        public void reset(int n) {
            val = n;
        }

        public int count() {
            if(val > 0) return val--;
            else return 0;
        }
    }
}
```

```

    }

class NSDemo {
    public static void Main() {
        Counter.CountDown cd1 = new Counter.CountDown(10);
        int i;

        do {
            i = cd1.count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        Counter.CountDown cd2 = new Counter.CountDown(20);

        do {
            i = cd2.count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        cd2.reset(4);
        do {
            i = cd2.count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();
    }
}

```

Programın çıktısı aşağıdaki gibidir:

```

10 9 8 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0

```

Bu programın yakından incelemeye değer bazı önemli yanları mevcuttur. Öncelikle, **CountDown**, **Counter** isim uzayında deklare edildiği için bir nesne oluşturulurken **CountDown**, aşağıdaki gibi **Counter** ile birlikte nitelenmelidir:

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

Ancak, **Counter** tipinde bir nesne bir kez oluşturulduktan sonra, artık bunu ya da bu nesnenin herhangi bir üyesini isim uzayı ile birlikte nitellemek gerekli değildir. Böylece, **cd1.count()** isim uzayı niteleyicisi olmadan doğrudan çağrılabılır; aşağıdaki satırda gösterildiği gibi:

```
i = cd1.count();
```

## İsim Uzayları İsim Karışıklıklarını Önler

Bir isim uzayı ile ilgili kilit öneme sahip husus, bir isim uzayı içinde deklare edilen isimlerin isim uzayı dışında deklare edilen aynı isimlerle karışmayacak olmalarıdır. Örneğin,

aşağıdaki programda **CountDown** adında bir başka sınıf daha oluşturulmaktadır; fakat bu sınıf **Counter2** adında bir isim uzayındadır.

```
// Isim uzaylari isim karisikliklarini onler.

using System;

// Sayaclar icin bir isim uzayi deklare et.
namespace Counter {
    // Seriye dogru sayan basit bir sayac.
    class CountDown {
        int val;

        public CountDown(int n) {
            val = n;
        }

        public void reset(int n) {
            val = n;
        }

        public int count() {
            if(val > 0) return val--;
            else return 0;
        }
    }
}

// Bir baska isim uzayi deklare et.
namespace Counter2 {
    /* Bu CountDown varsayılan isim uzayı içindedir ve
       Counter içindekiyle karışmaz. */
    class CountDown {
        public void count() {
            Console.WriteLine("This is count() in the " +
                "Counter2 namespace.");
        }
    }
}

class NSDemo {
    public static void Main() {
        // Bu, Counter isim uzayı içindeki CountDown'dur.
        Counter.CountDown cd1 = new Counter.CountDown(10);

        // Bu, varsayılan isim uzayı içindeki CountDown'dur.
        Counter2.Countdown cd2 = new Counter2.CountDown();

        int i;

        do {
            i = cd1.count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();
    }
}
```

```

        cd2.count();
    }
}

```

Çıktı aşağıda gösterilmiştir:

```

10 9 8 7 6 5 4 3 2 1 0
This is count() in the Counter2 namespace.

```

Çıktının da doğruladığı gibi, **Counter** içindeki **CountDown** sınıfı, **Counter2** isim uzayındaki **CountDown** sınıfından ayrıdır ve hiç isim karışıklığı ortaya çıkmaz. Bu örnek oldukça basittir. Buna rağmen, sizin yazdığınız sınıfları bir isim uzayı içine yerleştirmenin sizin kodunuz ve başkaları tarafından yazılan kod arasındaki isim karışıklıklarını önlemeye nasıl yardımcı olduğu bu örnektan kolaylıkla görülmektedir.

## using

Bölüm 2'de açıklandığı gibi, eğer programınız bir isim uzayının üyelerine sıkça referansta bulunuyorsa, bu üyelere referansta bulunmanız gereken her seferde isim uzayını belirtmek zorunda olmanız hızla usandırıcı bir hal alır. **using** direktifi bu problemi hafifletir. Bu kitabın başından sonuna C# **System** isim uzayını programlarınıza dahil etmek için **using**'i kullanmaktasınız, bu nedenle zaten buna aşinasınız. Tahmin edebileceğiniz gibi **using**, kendi oluşturduğunuz isim uzaylarını programlarınıza dahil etmek için de kullanılabilir.

**using** direktifinin iki şekli mevcuttur. İlkı aşağıda gösterilmiştir;

```
using isim;
```

Burada **isim**, erişmek istediğiniz isim uzayının adını belirtir. Bu, şimdije kadar görmüş olduğunuz **using** şeklidir. Belirtilen isim uzayı içinde tanımlanan üyelerin tümü programa dahil edilir (yani, mevcut isim uzayının bir parçası olurlar) ve niteleyici olmadan da kullanılabilirler. **using** direktifi, diğer deklarasyonlardan önce, dosyaların her birinin en başında belirtilmelidir.

Aşağıdaki program önceki bölümdeki sayıç örneğinin yeniden düzenlenmiş versiyonudur. Bu program, kendi oluşturduğunuz bir isim uzayını programınıza dahil etmek için **using**'i nasıl kullanabileceğinizi göstermektedir:

```

// Bir isim uzayı kullanımını gösterir.

using System;

// Counter'i programa dahil et.
using counter;

// Sayaclar için bir isim uzayı deklare et.
namespace Counter {
    // Geriye doğru sayan basit bir sayac.
    class CountDown {

```

```
int val;

public CountDown(int n) {
    val = n;
}

public void reset(int n) {
    val = n;
}

public int count() {
    if(val == 0) return val--;
    else return val;
}

class NSDemo {
    public static void Main() {
        // Şimdi, CountDown doğrudan kullanılabilir.
        CountDown cd1 = new CountDown(10);
        int i;

        do {
            i = cd1.count();
            Console.Write(i + " ");
        }
        while(i > 0);
        Console.WriteLine();

        CountDown cd2 = new CountDown(20);

        do {
            i = cd2.count();
            Console.Write(i + " ");
        }
        while(i > 0);
        Console.WriteLine();
        cd2.reset(4);
        do {
            i = cd2.count();
            Console.Write(i + " ");
        }
        while(i > 0);
        Console.WriteLine();
    }
}
```

Program, bir başka önemli hususu ortaya koymaktadır: Bir isim uzayını kullanmak bir başkasını devre dışı bırakmaz. Bir isim uzayını programınıza dahil ettiğinizde, bu isim uzayı yalnızca kendisindeki isimleri halihazırda kullanımında olan diğer isim uzaylarına ekler. Böylece, hem **System** hem de **Counter** programa dahil edilmişdir.

## using'in İkinci Kullanım Şekli

**using** direktifinin aşağıda gösterildiği gibi ikinci bir şekli daha mevcuttur:

```
using takma-isim = isim;
```

Burada **takma-isim**, isim ile belirtilen sınıf ya da isim uzayı için bir başka isim halini alır. Sayaç programı **Counter.CountDown** için **Count** adında bir takma isim (alias) oluşturulacak biçimde bir kez daha yeniden düzenlenmiştir.

```
// using icin takma isim kullanimini gosterir.

using System;

// Counter.CountDown icin bir takma isim olustur.
using Count = Counter.CountDown;

// Sayaclar icin bir isim uzayi deklare et.
namespace Counter {
    // Geriye dogru sayan basit bir sayac.
    class CountDown {
        int val;

        public CountDown(int n) {
            val = n;
        }

        public void reset(int n) {
            val = n;
        }

        public int count() {
            if(val > 0) return val--;
            else return 0;
        }
    }
}

class NSDemo {
    public static void Main() {
        /* Burada Count, Counter.CountDown'un
           ismi olarak kullaniliyor. */
        Count cd1 = new Count(10);
        int i;

        do {
            i = cd1.count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        Count cd2 = new Count(20);

        do {
```

```

        i = cd2.count();
        Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
        i = cd2.count();
        Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
}
}

```

**Count** bir kez **Counter.CountDown** için bir isim olarak belirtildikten sonra artık başka isim uzayı niteleyicileri kullanmadan nesneleri deklare etmek için kullanılabilir. Örneğin, programda aşağıdaki satır bir **CountDown** nesnesi oluşturur.

```
Count cd1 = new Count(10);
```

## İsim Uzayları Eklenebilir

Aynı isimde birden fazla isim uzayı deklarasyonu mevcut olabilir. Bu durum, bir isim uzayının birkaç dosya arasında bölünmesine, hatta aynı dosya içinde ayrılmasına imkan verir. Örneğin, aşağıdaki programda iki tane **Counter** isim uzayı tanımlanmaktadır. Bunlardan biri **CountDown** sınıfını içermektedir. Diğer ise **CountUp** sınıfını içermektedir. Program derlendiğinde her iki **Counter** isim uzayının içerikleri birbirine eklenir.

```

// Isim uzaylari eklenebilir.

using System;

// Counter'i programa dahil et.
using Counter;

// Iste Counter isim uzayindan biri.
namespace Counter {
    // Geriye dogru sayan basit bir sayac.
    class CountDown {
        int val;

        public CountDown(int n) {
            val = n;
        }

        public void reset(int n) {
            val = n;
        }

        public int count() {
            if(val > 0) return val-- ;
            else return 0;
        }
    }
}
```

```
        }

    // Iste bir baska Counter isim uzayi.
    namespace Counter {
        // Ileriye dogru sayan basit bir sayac.
        class CountUp {
            int val;
            int target;

            public int Target {
                get{
                    return target;
                }
            }

            public CountUp(int n) {
                target = n;
                val = 0;
            }

            public void reset(int n) {
                target = n;
                val = 0;
            }

            public int count() {
                if(val < target) return val++;
                else return target;
            }
        }
    }

    class NSDemo {
        public static void Main() {
            CountDown cd = new CountDown(10);
            CountUp cu = new CountUp(8);
            int i;

            do {
                i = cd.count();
                Console.Write(i + " ");
            } while(i > 0);
            Console.WriteLine();

            do {
                i = cu.count();
                Console.Write(i + " ");
            } while(i < cu.Target);
        }
    }
}
```

Bu program aşağıdaki çıktıyi üretir:

10 9 8 7 6 5 4 3 2 1 0

0 1 2 3 4 5 6 7 8

Bir başka şeye dikkat edin: Şu ifade

```
using Counter;
```

**Counter** isim uzayının bütün içeriğini programa dahil etmektedir. Böylece, hem **CountDown** hem de **CountUp** isim uzayı niteleyicileri olmaksızın doğrudan referansta bulunulabilirler. **Counter** isim uzayının iki parçaya ayrılmış olması önemli değildir.

## İsim Uzayları Kümelenebilir

Bir isim uzayı bir diğerinin içine yerleştirilebilir. Şu programı ele alın:

```
// Isim uzaylari kumelenebilir.

using System;

namespace NS1 {
    class ClassA {
        public ClassA() {
            Console.WriteLine("constructing ClassA");
        }
    }

    namespace NS2 { // kumenmis bir isim uzayi
        class ClassB {
            public ClassB() {
                Console.WriteLine("constructing ClassB");
            }
        }
    }
}

class NestedNSDemo {
    public static void Main() {
        NS1.ClassA a = new NS1.ClassA();

        // NS2.ClassB b = new NS2.ClassB(); //Hata!!! NS2 gorunurde degil
        NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // bu doğrudur
    }
}
```

Bu program aşağıdaki çıktıyı üretir:

```
constructing ClassA
constructing ClassB
```

Programda **NS2** isim uzayı **NS1** içine kümelenmiştir. Böylece, **ClassB**'ye referansta bulunmak için bunu hem **NS1** hem de **NS2** isim uzayları ile birlikte nitelemelisiniz. **NS2** tek başına yeterli değildir. Görüldüğü gibi, isim uzayları nokta ile birbirinden ayrılmışlardır.

Kümelenmiş bir isim uzayını, her isim uzayını nokta ile birbirinden ayırarak tek bir isim uzayı ifadesi içinde belirtebilirsiniz. Örneğin, şu ifade

```
namespace OuterNS {  
    namespace InnerNS {  
        // ...  
    }  
}
```

şu şekilde de belirtilebilir:

```
namespace OuterNS.InnerNS {  
    // ...  
}
```

## Varsayılan İsim Uzayı

Programınız için bir isim uzayı deklare etmezseniz varsayılan isim uzayı kullanılır. Önceki bölümlerde yer alan programlarda **namespace** kullanmanız gereklidir. Bu kitaptaki kısa, örnek programlarda varsayılan isim uzayı kullanışlı olmakla birlikle gerçek dünyaya ait uygulamaların birçoğunda kod, bir isim uzayı içinde yer alacaktır. Kodunuzu bir isim uzayı içine paketlemenin başlıca nedeni isim karışıklıklarını önlemektir. İsim uzayları, programlarınızı organize etmeniz ve günümüzün karmaşık, ağıla donatılmış ortamında varlıklarını sürdürmelerini sağlamanız için size yardımcı olan araçlardan bir diğeridir.

## Önişlemci

C#'ta programınızın kaynak dosyasının derleyici tarafından derlenme biçimini etkileyen birkaç önişlemci direktifi (*preprocessor directive*) tanımlanmaktadır. Bu direktifler, içinde bulundukları kaynak dosyanın metnini program nesne koduna çevrilmeden önce etkilerler. Önişlemci direktiflerinin büyük bölümü C++'tan alınmıştır. Aslında, C# önişlemcisi C++ tarafından tanımlanana çok benzer. Önişlemci komutları geleneksel olarak önişlemci olarak adlandırılan ayrı bir derleme evresinde ele alınır. *Önişlemci direktifi* terimi bu gerçekten gelmektedir. Günümüzün modern derleyici teknolojisi direktifleri ele almak için artık ayrı bir önişlemci evresi gerektirmez; ancak, isim bakı kalmıştır.

C#'ta aşağıdaki önişlemci direktifleri tanımlıdır:

```
#define      #elif      #else      #endif  
#endregion   #error     #if       #line  
#region     #undef     #warning
```

Tüm önişlemci direktifleri **#** simgesi ile başlar. İlaveten, her önişlemci direktifi kendi satırında yer almmalıdır.

Açıkçası, C#'ın modern, nesne yönelimli mimarisi yanında önişlemci direktiflerine daha eski dillerdekine kıyasla pek fazla gerek yoktur. Bununla birlikte, özellikle koşullu

derlemelerde olmak üzere önişlemci direktifleri zaman zaman önemli olabilmektedirler. Bu bölümde her direktif sırayla incelenmektedir.

## #define

**#define** direktifi, *sembol* olarak adlandırılan bir karakter sekansı tanımlar. Bir sembolün varlığı ya da eksikliği **#if** ya da **#elif** ile belirlenir ve derlemeyi kontrol etmek amacıyla kullanılır. **#define** genel olarak şu şekildedir:

```
#define sembol
```

Bu ifadede noktalı virgül kullanılmadığını dikkat edin. **#define** ve *sembol* arasında herhangi sayıda boşluk olabilir; fakat, *sembol* bir kez başladıkten sonra yalnızca yeni satır karakteri ile sona erer. Örneğin, **EXPERIMENTAL** sembolünü tanımlamak için şu direktifi kullanın:

```
#define EXPERIMENTAL
```

### NOT

C/C++'ta metinleri birbiri yerine yerleştirmek için **#define** kullanabilirsiniz. Örneğin, bir değer için bir isim tanımlamak ya da fonksiyon benzeri makrolar oluşturmak gibi. C#, **#define**'in bu tür kullanımlarını desteklemez. C#'ta **#define** yalnızca bir sembol tanımlamak için kullanılır.

## #if ve #endif

**#if** ve **#endif** direktifleri, bir veya daha fazla sembol içeren bir deyimin **true** değerine sahip olup olmamasına bağlı olarak bir kod sekansının koşullu derlenmesini mümkün kılarlar. Bir sembol eğer tanımlanmışsa **true** değerine sahiptir. Aksi halde sembolün değeri yanlıştır. Bu nedenle, eğer bir sembol **#define** direktifi ile tanımlanmışsa bu sembolün değeri doğru olacaktır.

**#if** genel olarak şu şekildedir:

```
#if sembol-deyimi
    ifade sekansi
#endif
```

**#if**'i takip eden deyim doğru değerindeyse **#if** ile **#endif** arasında kalan kod derlenir. Aksi halde, arada kalan kod atlanır. **#endif** direktifi **#if** bloğunun sonunu işaret eder.

Sembol deyimi yalnızca bir sembol isminden ibaret olacak kadar basit olabilir. Bir sembol deyiminde şu operatörleri de kullanabilirsiniz: **!**, **==**, **!=**, **&&** ve **||**. Parantezlere de ayrıca izin verilir.

İşte **#if**, **#endif** ve **#define** kullanan bir örnek:

```
// #if, #endif ve #define kullanımını gösterir.

#define EXPERIMENTAL
```

```
using System;

class Test {
    public static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("Compiled for experimental version.");
        #endif

        Console.WriteLine("This is in all versions.");
    }
}
```

Bu program aşağıdaki çıktıyı görüntüler:

```
Compiled for experimental version.
This is in all versions.
```

Programda **EXPERIMENTAL** simbolü tanımlanmaktadır. Böylece, **#if** ile karşılaşınca simbol deyiminin değeri doğru olur ve ilk **WriteLine()** ifadesi derlenir.

**EXPERIMENTAL** tanımını kaldırıp programı yeniden derlerseniz, ilk **WriteLine()** ifadesi derlenmeyecektir, çünkü **#if, false** değerine sahip olacaktır. İkinci **WriteLine()** ifadesi her durumda derlenir, çünkü bu ifade **#if** bloğunun bir parçası değildir.

Önceden açıklandığı gibi, **#if** içinde bir simbol deyimi kullanabilirsiniz. Örneğin:

```
// Bir simbol deyimi kullanır.

#define EXPERIMENTAL
#define TRIAL

using System;

class Test {
    public static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("compiled for experimental version.");
        #endif

        #if EXPERIMENTAL && TRIAL
            Console.Error.WriteLine("Testing experimental
                                    trial version.");
        #endif

        Console.WriteLine("This is in all versions.");
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Compiled for experimental version.
```

---

Testing experimental trial version.  
This is in all versions.

Bu örnekte iki simbol tamlanmaktadır: **EXPERIMENTAL** ve **TRIAL**. İkinci **WriteLine()** ifadesi sadece bu sembollerin her ikisi de tanımlanırsa derlenmektedir.

## #else ve #elif

**#else** direktifi C# dilinin bir parçası olan **else**'e çok benzer şekilde çalışmaktadır: **#if** başarısız olursa ona bir alternatif oluşturur. Önceki örnek aşağıda gösterildiği gibi genişletilebilir:

```
// #else kullanımını gösterir.

#define EXPERIMENTAL

using System;

class Test {
    public static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("Compiled for experimental version.");
        #else
            Console.WriteLine("Compiled for release.");
        #endif

        #if EXPERIMENTAL && TRIAL
            Console.Error.WriteLine("Testing experimental
                                    trial version.");
        #else
            Console.Error.WriteLine("Not experimental
                                    trial version.");
        #endif

        Console.WriteLine("This is in all versions.");
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Compiled for experimental version.
Not experimental trial version.
Tnis is in all versions.
```

**TRIAL** artık tanımlı olmadığı için ikinci koşul kod sekansının **#else** bölümü kullanılmaktadır.

Dikkat ederseniz, **#else** hem **#if** bloğunun sonunu hem de **#else** bloğunun başını işaret etmektedir. Bu gereklidir, çünkü herhangi bir **#if** ile ilişkilendirilmiş yalnızca tek bir **#endif** olabilir.

**#elif** direktifi “else if” anlamına gelir ve birden fazla derleme seçeneği için bir **if-else-if** zinciri kurar. **#elif**'in ardından bir simbol deyimi gelir. Deyim doğru değerine sahipse, karşılık gelen kod bloğu derlenir ve diğer **#elif** deyimlerinden hiçbir test edilmez. Aksi halde, seri içindeki bir sonraki blok kontrol edilir. **#elif** genel olarak şu şekildedir:

```
#if simbol-deyimi
    ifade sekansi
#elif simbol-deyimi
    ifade sekansi
#elif simbol-deyimi
    ifade sekansi
#elif simbol-deyimi
    ifade sekansi
#elif simbol-deyimi
    .
.
.
#endif
```

Örneğin:

```
// #elif kullanımını gösterir.

#define RELEASE

using System;

class Test {
    public static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("Compiled for experimental version.");
        #elif RELEASE
            Console.WriteLine("Compiled for release.");
        #else
            Console.WriteLine("Compiled for internal testing.");
        #endif

        #if TRIAL && !RELEASE
            Console.WriteLine("Trial version.");
        #endif

        Console.WriteLine("This is in all versions.");
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Compiled for release.
This is in all versions.
```

## #undef

**#undef** direktifi, kendisini takip eden sembolün önceden tanımlanmış tanımını ortadan kaldırır. Yani, bir sembolü “tanımsızlaştırır” (“*undefined*”). **#undef** direktifi genel olarak şu şekildedir:

```
#undef sembol
```

İşte bir örnek:

```
#define SMALL
```

```
#if SMALL
    ...
#undef SMALL
    // bu noktada SMALL tanımsızdır.
```

**#undef** direktifinden sonra **SMALL** artık tanımlı değildir.

**#undef** esasen, sembollerin yalnızca kendilerine gerek duyan kod bölümüyle sınırlandırılmalarına olanak tanımak için kullanılır.

## #error

**#error** direktifi derleyiciyi derlemeyi bırakması için zorlar. Bu direktif hata ayıklama işlemi için kullanılır. **#error** direktifi genel olarak şu şekildedir:

```
#error hata-mesajı
```

**#error** direktifi ile karşılaşınca hata mesajı ekranda görüntülenir. Örneğin, derleyici aşağıdaki satırı rastlayınca derleme işlemi durdurulur ve “This is test error!” hata mesajı ekranda görüntülenir.

```
#error This is a test error!
```

## #warning

**#warning** direktifi **#error**'la aynıdır. Tek fark, **#warning** ile bir hata yerine bir uyarı üretilmesidir. Böylece, derleme işlemi durdurulmaz. **#warning** direktifi genel olarak şu şekildedir:

```
#warning uyarı-mesajı
```

## #line

**#line** direktifi, bu direktifi içeren dosya için satır numarasını ve dosya ismini ayarlar. Satır numarası ve dosya ismi derleme sırasında hataların ya da uyarıların çıktısı için kullanılır. **#line** genel olarak şu şekildedir:

```
#line sayı "dosyaadı"
```

Burada **sayı** herhangi pozitif tamsayıdır. Bu ifadeyle yeni satır numarası artık **sayı** olur. **dosyaadı** isteğe bağlıdır ve geçerli herhangi bir dosya tanımlayıcısı, **dosyaadı** olarak kullanılabilir; bu ifadeyle **dosyaadı** da artık yeni dosya adı olacaktır. **#line**'ın başlıca kullanım alanları hata ayıklama işlemi ve özel uygulamalarıdır.

Satır numaralama işlemini orijinal haline döndürmek için **default** kelimesini aşağıda gösterildiği gibi belirtin:

```
#line default
```

## #region ve #endregion

**#region** ve **#endregion** direktifleri, Visual Studio IDE içinde ana hat görünümü kullanılırken genişletecek ya da daraltılacak bir alan tanımlamanıza olanak tanır. Genel olarak nasıl kullanılacakları aşağıda gösterilmiştir:

```
#region alan-adi
    // kod sekansı
#endregion
```

Burada **alan-adi**, söz konusu alanın ismidir.

## Assembly'ler ve internal Erişim Niteleyicisi

C#'ın önemli bir parçası assembly'dir. Assembly, bir program için tüm kurulum ve versiyon bilgilerini içeren bir dosyadır (ya da dosyalardır). Assembly'ler .NET ortamı için temel niteliğindedirler. Microsoft'tan alıntı yaparsak, “Assembly'ler, .NET Framework'ün temel taşlarıdır.” Assembly'ler; güvenli bileşen etkileşimini, diller arası uyumluluğu ve yeni versiyonların geliştirilmesini destekleyen mekanizmalardır. Bir assembly aynı zamanda bir kapsam (scope) da tanımlar.

Bir assembly dört bölümden meydana gelir. Bunlardan ilki, assembly *manifestosudur* (manifest). Manifesto, assembly'nin kendisiyle ilgili bilgi içerir. Bu veriler arasında assembly'nin ismi, versiyon numarası, tip eşleme bilgileri ve kültürel ayarlamalar gibi bilgiler yer almaktadır. İkinci bölüm, program tarafından kullanılan veri tipleriyle ilgili bilgileri içeren *tip metadata'sıdır*. Diğer avantajlarının yanı sıra tip metadata'sı diller arası uyuma yardımcı olmaktadır. Assembly'nin üçüncü bölümü Microsoft Intermediate Language (MSIL) biçiminde saklanan program kodudur. Assernbly'nin dördüncü bileşeni ise program tarafından kullanılan kaynaklardır.

Neyse ki, C#'ı kullanırken assembly'ler sizin çok az bir gayretinizle ya da ek bir çaba harcamanıza gerek kalmadan otomatik olarak üretilirler. Bunun nedeni şudur: Bir C# programı derlediğinizde oluşturulan **exe** dosyası, aslında diğer tür bilgilerin yanı sıra programınızın çalıştırılabilir kodunu içeren bir assembly'dir. Böylece, bir C# programını derlediğinizde bir assembly otomatik olarak üretilir.

Assembly'lerle ilgili daha birçok özellik ve konu mevcuttur; ancak bunlar, elinizdeki kitabın kapsamı dışındadır. (Assembly'ler .NET geliştirme sürecinin önemli bir parçasıdır, fakat C# dilinin teknik bir özelliği değildir.) Bununla birlikte, C#'ın assembly'lerle doğrudan bağlantılı bir parçası mevcuttur: **internal** erişim niteleyicisi. Bu, aşağıda incelenmektedir.

## internal Erişim Niteleyicisi

Bu kitabın başından beri kullanmakta olduğunuz **public**, **private** ve **protected** erişim niteleyicilerine ek olarak C#'ta ayrıca **internal** da tanımlıdır. **internal** niteleyicisi, bir üyenin bir assembly içindeki tüm dosyalar tarafından bilindiğini ancak assembly dışında tanınmadığını bildirir. Böylece, daha basit bir ifadeyle, **internal** olarak işaretlenmiş bir üye program boyunca bilinir, fakat bunun haricinde bilinmez. **internal** erişim niteleyicisi özellikle yazılım bileşenleri oluştururken işe yarar.

**internal** niteleyicisi sınıflara, sınıf üyelerine, yapılara ve yapı üyelerine uygulanabilir. **internal** niteleyicisi arayüz ve numaralandırma deklarasyonlarına da uygulanabilir.

**protected internal** erişim niteleyicisi çifti oluşturmak amacıyla **protected**'ı **internal** ile birlikte kullanabilirsiniz. **protected internal** erişim düzeyi yalnızca sınıf üyelerine verilebilir. **protected internal** erişim izni verilerek deklare edilen bir üye kendi assembly'si içinden ya da türetilmiş tiplerden erişilebilir.

İşte, **internal** kullanılan bir örnek:

```
// internal kullanır.

using System;

class InternalTest {
    internal int x;
}

class InternalDemo {
    public static void Main() {
        InternalTest ob = new InternalTest();

        ob.x = 10; // erisebilir - aynı dosyadalar
        Console.WriteLine("Here is ob.x: " + ob.x);
    }
}
```

**InternalTest** içinde **x** alanı **internal** olarak deklare edilmektedir. Bunun anlamı şudur: **x** program içinden erişilebilir. **x**'in **InternalDemo** içindeki kullanımı da bunu göstermektedir. Fakat, **x** program dışından erişime açık değildir.

# **ÇALIŞMA ZAMANI TİP TANIMLAMASI, YANSIMA VE NİTELİKLER**

Bu bölümde birbiriyle bağlantılı ve güçlü üç C# özelliği ele alınmaktadır: Çalışma zamanı tip tanımlaması, yansıtma ve nitelikler. *Çalışma zamanı tip tanımlaması (RTTI - Runtime Type ID)* bir programın çalışması sırasında bir tipin tanınmasına imkan veren bir mekanizmadır. *Yansıtma (reflection)*, bir tip hakkında bilgi edinmenizi mümkün kılan bir özelliktir. Bu bilgiyi kullanarak programın çalışması sırasında nesneler yapılandırıp, bu nesneleri kullanabilirsiniz. Bu özellik çok güçlündür, çünkü bir programa programın çalışması sırasında dinamik olarak işlevsellik ekleme imkanı verir. *Nitelik (attribute)* bir C# programının herhangi bir elemanın bir özelliğini tarif eder. Örneğin, diğer öğelerin yanı sıra sınıflar, metodlar ve alanlar için nitelikler belirtebilirsiniz. Nitelikler programın çalışması sırasında sorgulanabilir ve nitelik bilgisi elde edilebilir. Nitelikler hem çalışma zamanı tip tanımlamasını, hem de yansımayı kullanırlar.

## Çalışma Zamanı Tip Tanımlaması

Çalışma zamanı tip tanımlaması (RTTI - Runtime Type Identification), bir nesnenin tipinin programın çalışması sırasında belirlenmesine imkan verir, RTTI'ın kullanılabilirliğinin birçok nedeni vardır. Örneğin, bir temel sınıf referansının ne tür bir nesneye referansta bulunmakta, olduğunu bu sayede tam olarak keşfedebilirsiniz. RTTI'ın bir başka kullanımı, geçersiz tip atamalarıyla ilgili kural dışı bir duruma sebebiyet vermemek için, bir tip atamasının başarılı olup olmayacağına önceden test etmektir. Çalışma zamanı tip tanımlaması ayrıca, yansımayı temel bir bileşenidir.

C#, çalışma zamanı tip tanımlamasını destekleyen üç anahtar kelime içermektedir: **is**, **as** ve **typeof**. Bunların her biri sırayla incelenmektedir.

### Bir Tipi **is** İle Test Etmek

**is** operatörünü kullanarak bir nesnenin belirli bir tipte olup olmadığını belirleyebilirsiniz. **is** operatörünün genel olarak kullanımı aşağıda gösterilmiştir:

```
deyim is tip
```

Burada **deyim**, tipi **tip** ile karşılaştırılmakta olan bir deyimdir. Eğer **deyim**'in tipi **tip** ile aynı ya da uyumlu ise, bu işlemin sonucu **true** değerine sahiptir. Aksi halde, sonuç **false** olur. Böylece, eğer sonuç **true** değerindeyse, **deyim** tip ataması yoluyla **tip**'e dönüştürülebilir.

İşte, **is** kullanılan bir örnek:

```
// is kullanımı gösterir.

using System;

class A {}
class B : A {}

class UseIs {
```

```

public static void Main() {
    A a = new A();
    B b = new B();

    if(a is A) Console.WriteLine("a is an A");
    if(b is A)
        Console.WriteLine("b is an A because
                           it is derived from A");
    if(a is B)
        Console.WriteLine("This won't display -- a not
                           derived from B");

    if(b is B) Console.WriteLine("B is a B");
    if(a is object) Console.WriteLine("a is an Object");
}
}

```

Çıktı aşağıda gösterilmiştir:

```

a is an A
b is an A because it is derived from A
B is a B
a is an Object

```

**is** deyimlerinin birçoğu kendiliğinden anlaşılır durumdadır, ama bunlardan ikisi biraz açıklama gerektirebilir. Öncelikle, şu ifadeye dikkat edin:

```

if(b is a)
    Console.WriteLine("b is an A because it is derived from A");

```

**if** ifadesi başarılı olur, çünkü **b**, **B** tipinde bir referanstır. **B** ise **A** tipinden türetilmiştir. Böylece **b**, **A** ile uyumludur. Ancak, bunun tersi doğru değildir. Şu ifade gerçekleştirildiinde,

```

if(a is B)
    Console.WriteLine("This won't display -- a not derived from B");

```

**if** başarılı olmaz, çünkü **a**, **A** tipindedir; fakat **A**, **B**'den türetilmemiştir. Bu nedenle, her ikisi uyumlu değildir.

## as Kullanımı

Kimi zaman programın çalışması sırasında bir tip ataması denemek isteyeceksiniz. Öyle ki, tip ataması başarısız olursa bir kural dışı durum ortaya çıkmamasını isteyeceksiniz. Bunu gerçekleştirmek için **as** operatörünü kullanın. Bu operatör genel olarak şı sekilde kullanılır:

```
deyim as tip
```

Burada **deyim**, **tip**'e dönüştürülmemekte olan deyimdir. Tip ataması başarılı olursa, **tip**'e atıfta bulunan bir referans döndürülür. Aksi halde, bir **null** referans döndürülür.

**as** operatörü bazı durumlarda **is**'e standart bir alternatif oluşturur. Örneğin, geçersiz bir tip atamasının meydana gelmesini önlemek için **is** kullanan aşağıdaki programı ele alın:

```
// Gecersiz bir tip atamasini onlemek icin is kullanir.

using System;

class A {}
class B : A {}

class CheckCast {
    public static void Main() {
        A a = new A();
        B b = new B();

        /* a'nin tip atamasi yoluyla B'ye donusturulup
           donusturulemeyecegini anlamak icin kontrol et. */
        if(a is B) // eger oyleyse, tip atamasini uygula
            b = (B) a;
        else // degilse, tip atamasini atla
            b = null;

        if(b == null)
            Console.WriteLine("Cast b = (B) a is NOT allowed.");
        else
            Console.WriteLine("Cast b = (B) a is allowed");
    }
}
```

Bu program aşağıdaki çıktıyi ekranda görüntüler:

```
Cast b = (B) a is NOT allowed.
```

Cıktıdan da görüleceği gibi; **a**, **B** tipinde olmadığı için **a**'nın tip ataması yoluyla **B**'ye dönüştürülmesi geçerli değildir; **if** ifadesi tarafından bu durum engellenir. Ancak, bu yöntem iki aşama gerektirir. Birincisi, tip atamasının geçerliliği doğrulanmalıdır. İkincisi, tip ataması gerçeklenmelidir. Bu aşamalar, aşağıdaki programda gösterildiği gibi, **as** kullanılarak tek bir adımda toplanabilir:

```
// as kullanimini gösterir.

using System;

class A {}
class B : A {}

class CheckCast {
    public static void Main() {
        A a = new A();
        B b = new B();

        b = a as B; // tip atamasini gerceklestir, eger mumkunse

        if(b == null)
            Console.WriteLine("Cast b = (B) a is NOT allowed.");
        else
            Console.WriteLine("Cast b = (B) a is allowed");
```

```

    }
}

```

Daha önceki çıktı ile aynı olan çıktı aşağıdaki gibidir:

```
Cast b = (B) a is NOT allowed.
```

Programın bu versiyonunda **as** ifadesi, tip atamasının geçerliliğini kontrol eder; eğer geçerliyse, tek bir ifade içinde tip atamasını gerçekleştirir.

## typeof Kullanımı

**as** ve **is** operatörleri kendilerine göre kullanışlı olmalarına rağmen yalnızca iki tipin uyumluluğunu test ederler. Genellikle, bir tip hakkında bilgi elde etmeniz gerekecektir. Bunun için C# **typeof** operatörünü sağlamaktadır. **typeof**, belirli bir tip için bir **System.Type** nesnesi edinir. Bu nesneyi kullanarak söz konusu tipin özelliklerini belirleyebilirsiniz.

**typeof** operatörünün genel olarak kullanımı şu şekildedir:

```
typeof(tip)
```

Burada **tip**, elde edilmekte olan tiptir. Döndürülen **Type** nesnesi, **tip** ile ilişkilendirilmiş bilgiyi paketler.

Belirli bir tip için bir kez bir **Type** nesnesi elde ettikten sonra artık **Type** tarafından tanımlanan çeşitli özellik, alan ve metotları kullanarak söz konusu tip hakkında bilgi elde edebilirsiniz. **Type**, birçok üyesi olan büyük bir sınıfır ve bu konuya ilgili açıklamalar yansımamanın incelendiği bir sonraki bölüme kadar ertelenmiştir. Yine de, **Type**'ı kısaca tanıtmak için aşağıdaki program, onun üç özelliğini kullanmaktadır: **FullName**, **IsClass** ve **IsAbstract**. Söz konusu tipin tam ismini elde etmek için **FullName** kullanın. Eğer söz konusu tip bir sınıf ise **IsClass** doğru değerini döndürür. **IsAbstract** ise eğer bir sınıf özet ise doğru değeri döndürür.

```

// typeof kullanımını gösterir.

using System;
using System.IO;

class UseTypeof {
    public static void Main() {
        Type t = typeof(StreamReader);

        Console.WriteLine(t.FullName);

        if(t.IsClass) Console.WriteLine("Is a class.");
        if(t.IsAbstract) Console.WriteLine("Is abstract.");
        else Console.WriteLine("Is concrete.");
    }
}

```

Bu program aşağıdaki çıktıyı üretir:

```
System.IO.StreamReader
Is a class.
Is concrete.
```

Bu program, **StreamReader**'ı tarif eden bir **Type** nesnesi elde eder. Program daha sonra söz konusu tipin tam ismini ekranda görüntüler ve bunun bir sınıf olup olmadığını ve özet olup olmadığını belirler.

## Yansıma

Bu bölümün başında bahsedildiği gibi *yansıma* (*reflection*), bir tip hakkında bilgi edinmenizi mümkün kıyan bir C# özelliğidir. *Yansıma* terimi söz konusu sürecin işleyiş şeklinden gelmektedir: Bir **Type** nesnesi simgelediği tipin aynadaki görüntüsü gibidir. Bilgi elde etmek için **Type** nesnesine sorular sorarsınız ve Type nesnesi de tiple ile ilişkili bilgileri size geri döndürür (yansıtır). Yansıma güçlü bir mekanizmadır, çünkü yansıma, tiplerin yalnızca programın çalışması sırasında bilinen becerilerini öğrenmenize ve kullanmanıza olanak tanır.

Yansımayı destekleyen sınıfların birçoğu **System.Reflection** isim uzayında yer alan .NET Reflection API'sının bir parçasıdır. Bu nedenle yansıma kullanılan programlarda aşağıdaki satırı normal olarak programınıza dahil edeceksiniz:

```
using System.Reflection;
```

## Yansımanın Çekirdeği: System.Type

**System.Type** yansımı alt sisteminin çekirdeğinde yer alır, çünkü **System.Type** ile tip paketlenmektedir. **System.Type**, programın çalışması sırasında bir tip hakkında bilgi elde etmek için kullanacağınız özellikleri ve metodları içerir. **Type**, **System.Reflection.MemberInfo** adında bir özet sınıfından türetilmiştir.

**MemberInfo** aşağıdaki özet, salt okunur özellikleri tanımlamaktadır:

<b>Type DeclaringType</b>	Üyenin deklare edildiği sınıf veya arayüzün tipi.
<b>MemberTypes MemberType</b>	Üyenin tipi.
<b>string Name</b>	Tipin ismi.
<b>Type ReflectedType</b>	Yansıtılmakta olan nesnenin tipi.

**MemberType**'ın tipinin **MemberTypes** olduğuna dikkat edin. **MemberTypes**, çeşitli üye tipleri için değerler tanımlayan bir numaralandırmadır. **MemberTypes** içinde yer alan değerlerin bir kısmı şunlardır:

```
MemberTypes.Constructor
MemberTypes.Method
MemberTypes.Field
MemberTypes.Event
MemberTypes.Property
```

Böylece, bir üyenin tipi **MemberType** kontrol edilerek belirlenebilir. Örneğin, **MemberType** eğer **MemberTypes.Method**'a eşitse söz konusu üye bir metottur.

**MethodInfo** iki özet metot içermektedir: **GetCustomAttributes()** ve **IsDefined()**. Bunların her ikisi de niteliklerle bağlantılıdır.

**MethodInfo** tarafından tanımlanan metod ve özelliklere **Type** da kendinden daha pek çok ilave etmektedir. Örneğin, işte **Type** tarafından tanımlanmış sıkça kullanılan metodlardan bir kaçı:

Metot	Amaç
<b>ConstructorInfo[] GetConstructors()</b>	Belirtilen tip için yapılandırıcıların bir listesini elde eder.
<b>EventInfo[] GetEvents()</b>	Belirtilen tip için olayların listesini elde eder.
<b>FieldInfo[] GetFields()</b>	Belirtilen tip için alanların listesini elde eder.
<b>MemberInfo[] GetMembers()</b>	Belirtilen tip için üyelerin listesini elde eder.
<b>MethodInfo[] GetMethods()</b>	Belirtilen tip için metodların listesini elde eder.
<b> PropertyInfo[] GetProperties()</b>	Belirtilen tip için özelliklerin listesini elde eder.

İşte, **Type** tarafından tanımlanmış sıkça kullanılan salt okunur özelliklerden birkaçı:

Özellik	Amaç
<b>Assembly Assembly</b>	Belirtilen tip için bir assembly elde eder.
<b>TypeAttributes Attributes</b>	Belirtilen tip için nitelikleri elde eder.
<b>Type BaseType</b>	Belirtilen tip için en yakın temel tipi elde eder.
<b>string FullName</b>	Belirtilen tipin tam ismini elde eder.
<b>bool IsAbstract</b>	Belirtilen tip özet ise <b>true</b> 'dur.
<b>bool isArray</b>	Belirtilen tip bir dizi ise <b>true</b> 'dur.
<b>bool IsClass</b>	Belirtilen tip bir sınıf ise <b>true</b> 'dur.
<b>bool IsEnum</b>	Belirtilen tip bir numaralandırma ise <b>true</b> 'dur.
<b>string Namespace</b>	Belirtilen tip için isim uzayını elde eder.

## Yansımmanın Kullanımı

**Type**'ta tanımlı metodları ve özellikleri kullanarak programın çalışması sırasında bir tip hakkında ayrıntılı bilgi edinmek mümkündür. Bu son derece güçlü bir özelliktir, çünkü bir tip hakkında bilgileri bir kez elde ettikten sonra, artık söz konusu tipin yapılandırıcılarını etkin kılabilir, metodlarını çağırabilir ve özelliklerini kullanabilirsiniz. Böylece, derleme sırasında elverişli olmayan kodu yansıtma sayesinde kullanmanız mümkün olur.

Reflection API oldukça büyüktür; konunun tamamını burada ele almak mümkün değildir. (Yansımıyı tam olarak ele almak kolaylıkla bu kitabın tamamını doldurabilir!) Ancak, Reflection API mantıksal olarak tasarlandığı için bunun bir bölümünün nasıl kullanıldığını bir kez anladıkten sonra, gerisi ardından gelir. Bu düşüneden hareketle, aşağıdaki kısımlarda büyük öneme sahip dört yansımaya teknigi göslerilmektedir: Metotlar hakkında bilgi edinmek, metotları çağrırmak, nesneleri kurmak ve assembly'lerden tipleri yüklemek.

## Metotlar Hakkında Bilgi Edinmek

Elinizde bir kez bir **Type** nesnesi olduktan sonra söz konusu tip tarafından desteklenen metotların listesini **GetMethods()**'ı kullanarak elde edebilirsiniz. Bunun bir şekli aşağıda gösterilmiştir:

```
MethodInfo[] GetMethods()
```

Bu ifade **MethodInfo** nesnelerinden oluşan bir dizi döndürür. **MethodInfo** nesneleri, çağrıran tipin desteklediği metotları tarif eder. **MethodInfo**, **System.Reflection** isim uzayı içindedir.

**MethodInfo**, **MethodBase** özet sınıfından türetilmiştir. **MethodBase** ise **MemberInfo**'dan kalıtım yoluyla elde edilir. Böylece, bu üç sınıfın tümü tarafından tanımlanan özellikler ve metotlar sizin kullanımınıza hazırlıdır. Örneğin, bir metodun ismini elde etmek için **Name** özelliğini kullanın. Bu aşamada özellikle enteresan olan iki üye **ReturnType** ve **GetParameters()**'dır.

Bir metodun dönüş tipi **Type** tipinde bir nesne olan **ReturnType** özelliğinde mevcuttur.

**GetParameters()** metodu, bir metotla ilişkili olan parametrelerin listesini döndürür. Genel olarak bu metot şu şekildedir:

```
ParameterInfo[] GetParameters()
```

Parametre bilgileri **ParameterInfo** nesnesinde tutulur. **ParameterInfo**, söz konusu parametreyi tarif eden çok sayıda özellik ve metod tanımlanmaktadır. Özellikle değeri olan iki özellik **Name** ve **ParameterType**'dır. **Name**, söz konusu parametrenin adını içeren bir karakter katarıdır. **ParameterType** ise parametrenin tipini tarif eder. Parametrenin tipi bir **Type** nesnesi içinde paketlenir.

Aşağıda, **MyClass** adında bir sınıf tarafından desteklenen metotları elde etmek amacıyla yansımaya kullanan bir program yer almaktadır. Program, her metod için dönüş tipini, metodun adını; metotların her birinin sahip olabileceği parametrelerin isimlerini ve tiplerini ekranda görüntüler.

```
// Yansıma kullanarak metotları analız etmek.

using System;
using System.Reflection;
```

```
class MyClass {
    int x;
    int y;

    public MyClass(int i, int j) {
        x = i;
        y = j;
    }

    public int sum() {
        return x + y;
    }

    public bool isBetween(int i) {
        if(x < i && i < y) return true;
        else return false;
    }

    public void set(int a, int b) {
        x = a;
        y = b;
    }

    public void set(double a, double b) {
        x = (int) a;
        y = (int) b;
    }

    public void show() {
        Console.WriteLine(" x: {0}, y: {1}", x, y);
    }
}

class ReflectDemo {
    public static void Main() {
        Type t = typeof(MyClass); // MyClass'i simgeleyen bir
                                  // Type nesnesi al

        Console.WriteLine("Analyzing methods in " + t.Name);
        Console.WriteLine();

        Console.WriteLine("Methods supported: ");
        MethodInfo[] mi = t.GetMethods();

        // MyClass tarafindan desteklenen metotlari goster.
        foreach(MethodInfo m in mi) {
            // Donus tipini ve ismini goster.
            Console.Write(" " + m.ReturnType.Name +
                         " " + m.Name + "(");

            // Parametreleri goster.
            ParameterInfo[] pi = m.GetParameters();

            for(int i = 0; i < pi.Length; i++) {
                Console.Write(pi[i].ParameterType.Name +
```

```
        " " + pi[i].Name);
    if(i + 1 < pi.Length) Console.WriteLine(", ");
}

Console.WriteLine(")");
Console.WriteLine();
}
}
}
```

Çıktı aşağıda gösterilmiştir:

```
Analyzing methods in MyClass
Methods supported:
    Int32 GetHashCode()

    Boolean Equals(Object obj)

    String ToString()

    Int32 sum()

    Boolean isBetween(Int32 i)

    Void set(Int32 a, Int32 b)

    Void set(Double a, Double b)

    Void show()

    Type GetType()
```

**MyClass** tarafından tanımlanan metodlara ek olarak **object** tarafından tanımlanan metodların da ekranda gösterildiğine dikkat edin. Bunun nedeni, C#'ta tüm tiplerin kalıtım yoluyla **object**'ten türetilmesidir. Ayrıca, .NET yapı isimlerinin tip isimleri olarak kullanıldığına da dikkat edin. **set()**'in iki kez gösterildiğini gözleyin. Bu durum **set()** aşırı yükleniği için söz konusu olmuştur. **set()**'in bir versiyonu **int** argümanlar; diğerisi ise **double** argümanlar almaktadır.

Gelin şimdi bu programa yakından bakalım. Öncelikle, **MyClass**'ın bir **public** yapılandırıcı ve aşırı yüklenmiş **set()** metodu da dahil olmak üzere birkaç tane de **public** metod tanımladığını dikkat edin.

**Main()**'in içinde aşağıdaki kod satırı kullanılarak **MyClass**'ı simgeleyen bir **Type** nesnesi elde edilmektedir:

```
Type t = typeof(MyClass); // MyClass'i simgeleyen bir
                           Type nesnesi al
```

Program, daha sonra **t** ve Reflection API'ı kullanarak **MyClass** tarafından desteklenen metotlarla ilgili bilgileri ekranda görüntülemektedir. İlk önce, aşağıdaki ifade ile metodların listesi elde edilir:

```
MethodInfo[] mi = t.GetMethods();
```

Sonra, **mi** üzerinden tekrarlayan bir **foreach** döngüsü kurulur. Döngünün her tekrarında aşağıdaki kod tarafından her metod için dönüş tipi ve parametreler ekranda görüntülenir:

```
// Donus tipini ve ismini goster.  
Console.WriteLine(" " + m.ReturnType.Name + " " + m.Name + "());  
  
// Parametreleri goster.  
ParameterInfo[] pi = m.GetParameters();  
  
for(int i = 0; i < pi.Length; i++) {  
    Console.WriteLine(pi[i].ParameterType.Name + " " + pi[i].Name);  
    if(i + 1 < pi.Length) Console.Write(", ");  
}
```

Bu kod sekansı içinde, metodların her biriyle ilişkili parametreler **GetParameters()** çağrılarak elde edilir ve **pi** dizisinde saklanır. Sonra **pi** dizisi üzerinde bir **for** döngüsü her parametrenin tipini ve ismini görüntüleyerek tekrarlar. Buradaki en önemli nokta, bu bilgilerin **Myclass** hakkında ön bilgiye bel bağlamadan programın çalışması sırasında dinamik olarak elde edilmesidir.

## GetMethod()’ın İkinci Kullanım Şekli

**GetMethod()**’ın ikinci kullanım şekli, elde edilen metodları süzen çeşitli işaretler kullanmanıza olanak tanır. Genel olarak şu şekilde kullanılır:

```
MethodInfo[] GetMethods(BindingFlags işaretler)
```

Bu versiyon ile yalnızca sizin belirtmiş olduğunuz kriterlerle eşlenen metodlar elde edilir. **BindingFlags** bir numaralandırmadır. En sık kullanılan **BindingFlags** değerlerinden birkaçı aşağıda gösterilmiştir:

Değer	Anlamı
<b>DeclaredOnly</b>	Yalnızca belirtilen sınıf tarafından tanımlanan metodları elde eder. Kalıtım yoluyla elde edilen metodlar dahil edilmez.
<b>Instance</b>	Örnek metodları elde eder.
<b>NonPublic</b>	<b>public</b> olmayan metodları elde eder.
<b>Public</b>	<b>public</b> metodları elde eder.
<b>Static</b>	<b>static</b> metodları elde eder.

İki veya daha fazla işaretre VEYA uygulayabilirsiniz. Aslında, en azından **Public** veya **NonPublic** ile birlikte ya **Instance**'ı ya da **Static**'ı dahil etmelisiniz. Bunun başarılmaması durumunda metodların hiç biri elde edilmeyecektir.

**GetMethods()**'ın **BindingFlags** şeklinde başlıca kullanımlarından biri, türetilmiş metodları ayrıca elde etmeden bir sınıf tarafından tanımlanan metodların listesini elde etmektir. Bu, özellikle **object** tarafından tanımlanan metodların elde edilmesini önlemek için kullanışlıdır. Örneğin, **GetMethods()**'a yapılan şu çağrıyı önceki programa yerleştirmeyi deneyin:

```
/* Simdi, yalnızca MyClass tarafından deklare edilen metodlar
   elde edilecektir. */
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                                BindingFlags.Instance |
                                BindingFlags.Public);
```

Bu değişikliği yaptıktan sonra program aşağıdaki çıktıyı üretir:

```
Analyzing methods in MyClass
```

```
Methods supported:
  Int32 sum()

  Boolean isBetween(Int32 i)

  Void set(Int32 a, Int32 b)

  Void set(Double a, Double b)

  Void show()
```

Gördüğünüz gibi, yalnızca **MyClass** tarafından açıkça tanımlanan metodlar ekranda görüntülenmektedir.

## Metotları Yansıma Kullanarak Çağırma

Bir tipin hangi metodları desteklediğini bildikten sonra bu metodların birini ya da daha fazlasını çağrılabilsiniz. Bunun için **MethodInfo** içinde bulunan **Invoke()** metodunu kullanacaksınız. Bu metot aşağıda gösterilmiştir:

```
object Invoke(object nesne, object[ ] argümanlar)
```

Burada **nesne**, metodun üzerinde çağrıldığı nesneye bir referanstır. **static** metodlar için **nesne null** değerinde olmalıdır. Metoda aktarılması gereken herhangi argüman **argümanlar** dizisinde belirtilir. Hiç argümana gerek yoksa **argümanlar** dizisi **null** olmalıdır. Ayrıca, **argümanlar** dizisi tam olarak argüman sayısı kadar eleman içermelidir. Bu nedenle, eğer iki argüman gerekiyorsa **argümanlar** dizisi iki eleman uzunluğunda olmalıdır. Örneğin üç ya da dört eleman uzunluğunda olamaz.

Bir metodu çağrılmak için **GetMethods()** çağrılarak elde edilmiş olan bir **MethodInfo** örneği üzerinden **Invoke()**'u çağrılmak yeterlidir. Aşağıdaki program bu prosedürü göstermektedir:

```
// Yansima kullanarak metodları çağırma.

using System;
using System.Reflection;

class MyClass {
    int x;
    int y;

    public MyClass(int i, int j) {
        x = i;
        y = j;
    }

    public int sum() {
        return x + y;
    }

    public bool isBetween(int i) {
        if((x < i) && (i < y)) return true;
        else return false;
    }

    public void set(int a, int b) {
        Console.Write("Inside set(int, int). ");
        x = a;
        y = b;
        show();
    }

    // set'i asırı yükle.
    public void set(double a, double b) {
        Console.Write("Inside set(double, double). ");
        x = (int) a;
        y = (int) b;
        show();
    }

    public void show() {
        Console.WriteLine("Values are x: {0}, y: {1}", x, y);
    }
}

class InvokeMethDemo {
    public static void Main() {
        Type t = typeof(MyClass);
        MyClass reflectOb = new MyClass(10, 20);
        int val;

        Console.WriteLine("Invoking methods in " + t.Name);
```

```
Console.WriteLine();
MethodInfo[] mi = t.GetMethods();

// Metotların her birini çağır.
foreach(MethodInfo m in mi) {
    // Parametreleri al.
    ParameterInfo[] pi = m.GetParameters();

    if(m.Name.CompareTo("set") == 0 &&
       pi[0].ParameterType == typeof(int)) {
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
    }

    else if(m.Name.CompareTo("set") == 0 &&
            pi[0].ParameterType == typeof(double)) {
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
    }

    else if(m.Name.CompareTo("sum") == 0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("sum is " + val);
    }

    else if(m.Name.CompareTo("isBetween") == 0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
            Console.WriteLine("14 is between x and y");
    }

    else if(m.Name.CompareTo("show") == 0) {
        m.Invoke(reflectOb, null);
    }
}
}
```

Cıktı aşağıda gösterilmistir:

```
Invoking methods in MyClass  
  
sum is 30  
14 is between x and y  
Inside set(int, int). Values are x: 9, y: 18  
Inside set(double, double). Values are x: 1, y: 23  
Values are x: 1, y: 23
```

Metotların nasıl çağrıldığına yakından bakın. Öncelikle, metotların listesi elde edilmektedir. Sonra, **foreach** döngüsü içinde parametre bilgileri alınır. Daha sonra, bir dizi **if/else** ifadesi kullanılarak metotların her biri uygun tip ve argüman sayısı ile birlikte

çalıştırılır. Aşırı yüklenmiş **set()** metodunun aşağıdaki kod kullanılarak çalıştırılmış biçimine özellikle dikkat edin:

```
if(m.Name.CompareTo("set") == 0 &&
   pi[0].ParameterType == typeof(int)) {
    object[] args = new object[2];
    args[0] = 9;
    args[1] = 18;
    m.Invoke(reflectOb, args);
}
else if(m.Name.CompareTo("set") == 0 &&
   pi[0].ParameterType == typeof(double)) {
    object[] args = new object[2];
    args[0] = 1.12;
    args[1] = 23.4;
    m.Invoke(reflectOb, args);
}
```

Eğer metodun ismi ayarlanmışa, söz konusu metodun hangi versiyonunun mevcut olduğunu belirlemek için ilk parametrenin tipi test edilir. Eğer söz konusu olan **set(int, int)** ise bu durumda **int** argümanlar **args**'a yüklenir ve **set()** çağrılır. Aksi halde, **double** argümanlar kullanılır.

## Type'ın Yapılandırıcılarını Elde Etmek

Önceki örnekte **MyClass** üzerinden metotları çağrırmak için yansımak avantaj sağlamamıştır, çünkü **MyClass** tipinde bir nesne zaten açıkça oluşturulmuştur. Yalnızca bu nesnenin metodlarını çağrırmak normal olarak daha kolay olacaktır. Ancak, yansımının gücü, bir nesne, programın çalışması sırasında dinamik olarak oluşturulduğunda ortaya çıkmaya başlar. Bunun için öncelikle yapılandırıcıların listesini edinmeniz gerekecektir. Sonra, yapılandırıldan birini çağırarak söz konusu tipin bir örneğini oluşturacaksınız. Bu mekanizma, herhangi bir tipteki nesneyi, bir deklarasyon ifadesi içinde isimlendirmeden örneklemenize imkan verir.

Bir tip için yapılandırıcıları elde etmek amacıyla bir **Type** nesnesi üzerinde **GetConstructors()**'ı çağırın. En sık kullanılan biçimlerden biri aşağıda gösterilmiştir:

```
ConstructorInfo[] GetConstructors()
```

Bu ifade, yapılandırıcıları tarif eden **ConstructorInfo** nesnelerinin bir dizisini döndürür.

**ConstructorInfo**, **MethodBase** özet sınıfından türetilmiştir. **MethodBase**, kalıtım yoluyla **MemberInfo**'dan gelmektedir. **ConstructorInfo** ayrıca kendisine özgü birkaç üye daha tanımlar. Biz bu üyelerden, bir yapılandırıcıyla ilişkili parametrelerin listesini döndüren **GetParameters()** ile ilgilenmekteyiz. Bu metot, tipki daha önce anlatılan, **MethodInfo** tarafından tanımlı **GetParameters()** gibi çalışmaktadır.

Uygun bir yapılandırıcı bulduktan sonra **ConstructorInfo** tarafından tanımlanmış **Invoke()** metodu çağrılarak bir nesne oluşturulur. **Invoke()** aşağıda gösterilmiştir:

```
object Invoke(object[] argümanlar)
```

Metoda aktarılması gereken argümanlar, **argümanlar** dizisinde belirtilir. Eğer hiç argüman gerekmiyorsa **argümanlar** dizisi **null** olmalıdır. Ayrıca, **argümanlar** dizisi tam olarak argüman sayısı kadar eleman içermelidir. **Invoke()**, söz konusu nesneye bir referans döndürür.

Aşağıdaki programda, **Myclass**'ın bir örneğini oluşturmak için yansımaya kullanılmaktadır:

```
// Yansima kullanarak bir nesne olustur.

using System;
using System.Reflection;

class MyClass {
    int x;
    int y;

    public MyClass(int i) {
        Console.WriteLine("Constructing MyClass(int, int). ");
        x = y = i;
    }

    public MyClass(int i, int j) {
        Console.WriteLine("Constructing MyClass(int, int). ");
        x = i;
        y = j;
        show();
    }

    public int sum() {
        return x + y;
    }

    public bool isBetween(int i) {
        if((x < i) && (i < y)) return true;
        else return false;
    }

    public void set(int a, int b) {
        Console.Write("Inside set(int, int). ");
        x = a;
        y = b;
        show();
    }

    // set'i asiri yukle.
    public void set(double a, double b) {
        Console.Write("Inside set(double, double). ");
        x = (int) a;
        y = (int) b;
    }
}
```

```
        show();
    }

    public void show() {
        Console.WriteLine("Values are x: {0}, y: {1}", x, y);
    }
}

class InvokeConsDemo {
    public static void Main() {
        Type t = typeof(MyClass);
        int val;

        // Yapilandirici bilgilerini al.
        ConstructorInfo[] ci = t.GetConstructors();

        Console.WriteLine("Available constructors: ");
        foreach(ConstructorInfo c in ci) {
            // Donus tipini ve ismini goster.
            Console.Write(" " + t.Name + "(");

            // Parametreleri goster.
            ParameterInfo[] pi = c.GetParameters();

            for(int i = 0; i < pi.Length; i++) {
                Console.Write(pi[i].ParameterType.Name +
                    " " + pi[i].Name);
                if(i + 1 < pi.Length) Console.Write(", ");
            }

            Console.WriteLine(")");
        }

        Console.WriteLine();
        // Eslenen yapilandiriciyi bul.
        int x;

        for(x = 0; x < ci.Length; x++) {
            ParameterInfo[] pi = ci[x].GetParameters();
            if(pi.Length == 2) break;
        }

        if(x == ci.Length) {
            Console.WriteLine("No matching constructor found.");
            return;
        }
        else
            Console.WriteLine("Two-parameter constructor found.\n");

        // Nesneyi yapilandir.
        object[] consargs = new object[2];
        consargs[0] = 10;
        consargs[1] = 20;
        object reflectOb = ci[x].Invoke(consargs);
```

```
Console.WriteLine("\nInvoking methods on reflectOb.");
Console.WriteLine();
MethodInfo[] mi = t.GetMethods();

// Metotların her birini çağır.
foreach(MethodInfo m in mi) {
    // Parametrelerini al.
    ParameterInfo[] pi = m.GetParameters();

    if(m.Name.CompareTo("set") == 0 &&
       pi[0].ParameterType == typeof(int)) {
        // Bu, set(int, int).
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
    }
    else if(m.Name.CompareTo("set") == 0 &&
            pi[0].ParameterType == typeof(double)) {
        // Bu, set(double, double).
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
    }
    else if(m.Name.CompareTo("sum") == 0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("sum is " + val);
    }
    else if(m.Name.CompareTo("isBetween") == 0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
            Console.WriteLine("14 is between x and y");
    }
    else if(m.Name.CompareTo("show") == 0) {
        m.Invoke(reflectOb, null);
    }
}
```

Cıktı aşağıda gösterilmiştir:

Available constructors:

```
MyClass(Int32 i)
MyClass(Int32 i, Int32 j)
```

Two-parameter constructor found.

```
Constructing MyClass(int, int).  
Values are x: 10, y: 20
```

Invoking methods on reflectOb.

```

sum is 30
14 is between x and y
Inside set(int, int). Values are x: 9, y: 18
Inside set(double, double). Values are x: 1, y: 23
Values are x: 1, y: 23

```

Gelin şimdi, bir **MyClass** nesnesi yapılandırmak için yansımamanın nasıl kullanıldığına göz atalım. Öncelikle, aşağıdaki ifade kullanılarak açık yapılandırıcıların bir listesi elde edilir:

```
ConstructorInfo[] ci = t.GetConstructors();
```

Bir sonraki işlemde, gösterim amacıyla yapılandırıcılar ekranda görüntülenir. Sonra, aşağıdaki kod kullanılarak iki argüman alan bir yapılandırıcı listede aranır:

```

for(x = 0; x < ci.Length; x++) {
    ParameterInfo[] pi = ci[x].GetParameters();
    if(pi.Length == 2) break;
}

```

Eğer yapılandırıcı bulunursa (bu örnekte olduğu gibi), aşağıdaki kod sekansı tarafından bir nesne örneklenir:

```

// Nesneyi yapılandır.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);

```

**Invoke()** çağrısından sonra **reflectOb**, **MyClass** tipinde bir nesneye referansta bulunacaktır.

Önemli bir hususun altını çizmek gereklidir. Bu örnekte, işleri karmaşıklaştırmamak için iki argümanlı tek yapılandırıcının iki **int** argüman alan fonksiyon olduğu varsayılmıştır. Gerçek dünyaya ait bir uygulamada, her argümanın parametre tipi kontrol edilerek bu durumun doğrulanması gerekecektir.

## Assembly'lerden Tip Elde Etmek

Önceki örnekte, tek bir öğe haricinde, **MyClass** ile ilgili her şey yansımıma kullanılarak keşfedilmiştir. Keşfedilmemiş olan **MyClass** tipinin kendisidir. Yani, önceki örneklerde **MyClass** ile ilgili bilgiler dinamik olarak elde edilmiş olsa da, bu örnekler yine de **MyClass** tip isminin önceden bilindiği ve **MyClass** tip isminin, üzerinde tüm yansımıma metodlarının doğrudan ya da dolaylı olarak işlem gördüğü bir **Type** nesnesi elde etmek amacıyla bir **typeof** ifadesinde kullanıldığı gerçeğine dayanıyorlardı. Bu durum belki birkaç koşul için işe yarıyor olsa da yansımamanın tam gücü, diğer assembly'lerin içerikleri analiz edilerek bir programın kullanımına hazır olan tipler dinamik olarak belirlendiğinde ortaya çıkmaktadır.

Bölüm 16'da öğrendiğiniz gibi, bir assembly içeriği sınıflar, yapılar vs hakkında tip bilgilerini kendisiyle birlikte taşıır. Reflection API bir assembly'yi yüklemenize, bu assembly

ile ilgili bilgileri keşfetmenize ve bu assembly'nin açık olarak kullanılabilir tiplerinin örneklerini oluşturmanıza olanak tanır. Bu mekanizmayı kullanarak bir program kendi ortamını, mevcut olabilecek işlevsellikten yararlanarak - bu işlevselligi derleme sırasında açıkça tanımlamak zorunda kalmadan - araştırabilir. Bu son derece güçlü ve heyecan verici bir kavramdır. Örneğin, bir sistemdeki mevcut olan tüm tipleri gösteren bir "tip tarayıcısı" gibi davranışan bir program hayal edebilirsiniz. Bir başka uygulama, sistem tarafından desteklenen çeşitli tiplerden oluşan bir programı görsel olarak "bağlamanıza" imkan veren bir tasarım aracı olabilir. Bir tip hakkında tüm bilgiler keşfedilebilir olduğu için, yansımının uygulanabileceği yöntemlere doğal bir kısıtlama söz konusu değildir.

Bir assembly hakkında bilgi edinmek için öncelikle bir **Assembly** nesnesi oluşturacaksınız. **Assembly** sınıfı bir **public** yapılandırıcı tanımlamaz. Bunun yerine, bir **Assembly** nesnesi metodlarından biri çağrılarak elde edilir. Bizim kullanacağımız **LoadForm()**'dur. Bizim kullanacağımız şekil aşağıda gösterilmiştir:

```
static Assembly LoadFrom(string dosyaadi)
```

Burada ***dosyaadi***, assembly'nin adını belirtir.

Bir kez bir **Assembly** nesnesi elde ettikten sonra bu nesne üzerinde **GetTypes()**'ı çağrıarak nesnenin tanımladığı tipleri keşfedebilirsiniz. **GetTypes()** genel olarak şu şekildedir:

```
Type[] GetTypes()
```

Bu ifade, assembly'nin içinde bulunan tiplerin bir dizisini döndürür.

Bir assembly içindeki tiplerin ortaya çıkarılmasını göstermek için iki dosyaya gerek duyacaksınız. İlk dosya, ikinci dosya tarafından keşfedilecek birtakım sınıflar içerir. Programlamaya aşağıdaki kodu içeren **MyClasses.cs** adında bir dosya oluşturarak başlayın.

```
// Uc sınıf içeren bir dosya. Bu dosyaya MyClasses.cs ismini verin.

using System;

class MyClass {
    int x;
    int y;

    public MyClass(int i) {
        Console.WriteLine("Constructing MyClass(int). ");
        x = y = i;
        show();
    }

    public MyClass(int i, int j) {
        Console.WriteLine("Constructing MyClass(int, int). ");
        x = i;
        y = j;
        show();
    }
}
```

```

public int sum() {
    return x + y;
}

public bool isBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
}

public void set(int a, int b) {
    Console.Write("Inside set(int, int). ");
    x = a;
    y = b;
    show();
}

// set'i asiri yükle.
public void set(double a, double b) {
    Console.Write("Inside set(double, double). ");
    x = (int) a;
    y = (int) b;
    show();
}

public void show() {
    Console.WriteLine("Values are x: {0}, y: {1}", x, y);
}
}

class AnotherClass {
    string remark;

    public AnotherClass(string str) {
        remark = str;
    }

    public void show() {
        Console.WriteLine(remark);
    }
}

class Demo {
    public static void Main() {
        Console.WriteLine("This is a placeholder.");
    }
}

```

Bu dosya, önceki örneklerde kullanmakta olduğumuz **MyClass** sınıfını içermektedir. Ayrıca **AnotherClass** adında ikinci bir sınıf ve **Demo** adında da üçüncü bir sınıf eklemektedir. Böylece bu programdan elde edilen assembly, üç sınıf içerecektir. Sonra, bu dosyayı **MyClasses.exe** dosyası üretilecek şekilde derleyin. Bu, işlem sırasında sorgulanacak olan assembly'dir.

**MyClasses.exe** hakkında bilgileri ortaya çıkaracak program aşağıda gösterilmiştir. Bu programı şu aşamada derleyicinizin editörüne girin.

```
/* Yansima kullanarak bir assembly yapılandırın, tipleri
belirleyin ve bir nesne oluşturun. */

using System;
using System.Reflection;

class ReflectAssemblyDemo {
    public static void Main() {
        int val;

        // MyClasses.exe assembly'sini yükleyin.
        Assembly asm = Assembly.LoadFrom("MyClasses.exe");

        // MyClasses.exe'nin hangi tipleri içerdigini saptayın.
        Type[] alltypes = asm.GetTypes();
        foreach(Type temp in alltypes)
            Console.WriteLine("Found: " + temp.Name);

        Console.WriteLine();

        // İlk tipi kullanın, bu kez bu tip MyClass'dır.
        Type t = alltypes[0]; // bulunan ilk sınıfı kullan
        Console.WriteLine("Using: " + t.Name);

        // Yapılandırıcı bilgisini edin.
        ConstructorInfo[] ci = t.GetConstructors();

        Console.WriteLine("Available constructors: ");
        foreach(ConstructorInfo c in ci) {
            // Dönüş tipini ve ismini göster.
            Console.Write("    " + t.Name + "(");

            // Parametreleri göster.
            ParameterInfo[] pi = c.GetParameters();

            for(int i = 0; i < pi.Length; i++) {
                Console.Write(pi[i].ParameterType.Name +
                             " " + pi[i].Name);
                if(i + 1 < pi.Length) Console.Write(", ");
            }

            Console.WriteLine(")");
        }
        Console.WriteLine();

        // Eslenen yapılandırıcıyı bul.
        int x;

        for(x = 0; x < ci.Length; x++) {
            ParameterInfo[] pi = ci[x].GetParameters();
            if(pi.Length == 2) break;
        }
    }
}
```

```
if(x == ci.Length) {
    Console.WriteLine("No matching constructor found.");
    return;
}
else
    Console.WriteLine("Two-parameter constructor found.\n");

// Nesneyi yapılandır.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);

Console.WriteLine("\nInvoking methods on reflectOb.");
Console.WriteLine();
MethodInfo[] mi = t.GetMethods();

// Metotların her birini çağır.
foreach(MethodInfo m in mi) {
    // Parametreleri al.
    ParameterInfo[] pi = m.GetParameters();

    if(m.Name.CompareTo("set") == 0 &&
       pi[0].ParameterType == typeof(int)) {
        // Bu, set(int, int).
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
    }
    else if(m.Name.CompareTo("set") == 0 &&
            pi[0].ParameterType == typeof(double)) {
        // Bu, set(double, double).
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
    }
    else if (m.Name.CompareTo("sum") == 0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("sum is " + val);
    }
    else if(m.Name.CompareTo("isBetween") == 0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
            Console.WriteLine("14 is between x and y");
    }
    else if(m.Name.CompareTo("show") == 0) {
        m.Invoke(reflectOb, null);
    }
}
}
```

Programın çıktısı aşağıda gösterilmiştir:

```

Found: MyClass
Found: AnotherClass
Found: Demo

Using: MyClass
Available constructors:
    MyClass(Int32 i)
    MyClass(Int32 i, Int32 j)

Two-parameter constructor found.

Constructing MyClass(int, int).
Values are x: 10, y: 20

Invoking methods on reflectOb.

sum is 30
14 is between x and y
Inside set(int, int). Values are x: 9, y: 10
Inside set(double, double). Values are x: 1, y: 23
Values are x: 1, y: 23

```

Cıktıdan da görüldüğü gibi, **MyClasses.exe** içinde yer alan sınıfların üçü de bulunmuştur. Bunlardan ilki - bu örnekte **MyClass** - sonra bir nesneyi örneklemek ve metotları gerçeklemek için kullanılmıştır. Bütün bu işlemler **MyClasses.exe** içeriğinin ön bilgisine gerek kalmadan gerçekleştirilmiştir.

**MyClasses.exe** içindeki tipler, **Main()**'in başlangıcına yakın bir yerlerde yer alan aşağıdaki kod sekansı kullanılarak ortaya çıkarılmıştır:

```

// MyClasses.exe assembly'sini yükleyin.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");

// MyClasses.exe'nin hangi tipleri içerdigini saptayın.
Type[] alltypes = asm.GetTypes();
foreach(Type temp in alltypes)
    Console.WriteLine("Found: " + temp.Name);

```

Bir assembly'yi dinamik olarak yüklemeniz ve sorgulamanız gereken her durumda bu tür bir sekans kullanabilirsiniz.

Bununla bağlantılı bir konu şudur: Bir assembly'nin bir **exe** dosyası olması gerekmez. Assembly'ler ayrıca **dll** uzantısını kullanan “dinamik bağlantı kütüphanesi” (DLL - Dynamic Link Library) dosyalarının içinde de yer alabilirler. Örneğin, **MyClasses.cs** programını şu komut satırını kullanarak derlemek durumunda kalmış olsaydınız,

```
csc /t:library MyClasses.cs
```

çıktı dosyası **MyClasses.dll** olurdu. Kodu bir DLL içine yerleştirmenin avantajı, bu durumda **Main()** metoduna gerek olmamasıdır. Tüm **exe** dosyaları **Main()** gibi bir başlangıç

noktası gerektirirler. Demo sınıfının, yer tutucusu işlevini gören bir **Main()** metodu içermesinin sebebi de budur. DLL'ler başlangıç noktası gerektirmezler. Eğer **MyClass**'ı bir DLL'e çevirmeye çalışırsanız, **LoadFrom()**'a yapılan çağrıyı aşağıda gösterildiği gibi değiştirmeniz gerekecektir.

```
Assembly asm = Assembly.LoadFrom("MyClasses.dll");
```

## Tiplerin Ortaya Çıkarılmasını Tam Olarak Otomatize Etmek

Yansıma konusunu terk etmeden önce son bir örnek öğretici olacaktır. Önceki program **Myclass**'ı program içinde açıkça belirtmeye gerek kalmadan tam olarak kullanabiliyor olmasına rağmen, yine de **MyClass** içeriğiyle ilgili bir ön bilgiye dayanıyordu. Örneğin, program **MyClass**'ın metodlarının isimlerini biliyordu; söz gelişî **set** ve **sum** gibi. Ancak, yansıma kullanarak hakkında hiç ön bilgiye sahip olmadığınız bir tipten yararlanmanız mümkündür. Bunun için, bir nesnenin kurulumu ve metod çağrılarının üretilmesi için gerekli tüm bilgiyi ortaya çıkarmalısınız. Bu tür bir yaklaşım, ömeğin bir görsel tasarım aracı için kullanışlı olacaktır, çünkü bu sayede sistem üzerinde mevcut tiplerden yararlanılabilir.

Bir tipin dinamik olarak nasıl keşfedilebileceğini görmek için aşağıdaki örneği ele alın. Bu örnekte, **MyClasses.exe** assembly'si yüklenmekte, **MyClass** nesnesi yapılandırılmakta ve daha sonra **MyClass** tarafından tanımlanan tüm metodlar çağrılmaktadır. Bunlar yapılrken herhangi bir ön bilgi varsayımda bulunulmamaktadır:

```
// On bilgi varsayımda bulunmadan MyClass'in kullanımı.

using System;
using System.Reflection;

class ReflectAssemblyDemo {
    public static void Main() {
        int val;
        Assembly asm = Assembly.LoadFrom("MyClasses.exe");

        Type[] alltypes = asm.GetTypes();

        Type t = alltypes[0]; // bulunan ilk sınıfı kullan
        Console.WriteLine("Using: " + t.Name);

        ConstructorInfo[] ci = t.GetConstructors();

        // Bulunan ilk yapılandıriciyi kullan.
        ParameterInfo[] cpi = ci[0].GetParameters();
        object reflectOb;

        if(cpi.Length > 0) {
            object[] consargs = new object[cpi.Length];
```

```
//argumanlara baslangic degeri ata
for(int n = 0; n < cpi.Length; n++)
    consargs[n] = 10 + n * 20;

// nesneyi yapislandir
reflectOb = ci[0].Invoke(consargs);
} else
    reflectOb = ci[0].Invoke(null);

Console.WriteLine("\nInvoking methods on reflectOb.");
Console.WriteLine();

// kalitimla alinan metodlari dikkate alma.
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                                BindingFlags.Instance |
                                BindingFlags.Public);

// Metotlarin her birini cagir.
foreach(MethodInfo m in mi) {
    Console.WriteLine("Calling {0} ", m.Name);

    // Parametreleri al.
    ParameterInfo[] pi = m.GetParameters();

    // Metodlari calistir.
    switch(pi.Length) {
        case 0: // argumansiz
            if(m.ReturnType == typeof(int)) {
                val = (int) m.Invoke(reflectOb, null);
                Console.WriteLine("Result is " + val);
            }
            else if(m.ReturnType == typeof(void)) {
                m.Invoke(reflectOb, null);
            }
            break;
        case 1: // bir argumanli
            if(pi[0].ParameterType == typeof(int)) {
                object[] args = new object[1];
                args[0] = 14;
                if((bool) m.Invoke(reflectOb, args))
                    Console.WriteLine("14 is between x and y");
                else
                    Console.WriteLine("14 is not between x and y");
            }
            break;
        case 2: // iki argumanli
            if((pi[0].ParameterType == typeof(int)) &&
               (pi[1].ParameterType == typeof(int))) {
                object[] args = new object[2];
                args[0] = 9;
                args[1] = 18;
                m.Invoke(reflectOb, args);
            }
            else if((pi[0].ParameterType == typeof(double)) &&
                   (pi[1].ParameterType == typeof(double))) {
```

```
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
    }
    break;
}
Console.WriteLine();
}
}
}
```

Programın çıktısı şöyledir:

```
Using: MyClass
Constructing MyClass(int).
Values are x: 10, y: 10
```

```
Invoking methods on reflectOb.
```

```
Calling sum
Result is 20
```

```
Calling isBetween
14 is not between x and y
```

```
Calling set
Inside set(int, int). Values are x: 9, y: 18
```

```
Calling set
Inside set(double, double). Values are x: 1, y: 23
```

```
Calling show
Values are x: 1, y: 23
```

Programın çalışması gayet açıkta, ama iki noktaya degeinmeye yarar vardır. Birincisi şudur: Dikkat ederseniz sadece **MyClass** tarafından açıkça bildirilen metodlar alınmış ve kullanılmışlardır. Bu, **GetMethod()**'ın **BindingFlags**'ı kullanılarak sağlanmıştır. Burada amaç, **object**'ten kalıtım yoluyla alınan metodların çağrılmasını engellemektir. İkinci nokta ise şudur: Parametre sayısının ve her metodun dönüş tipinin dinamik olarak elde edilme şekline dikkat edin. Parametre sayısı bir **switch** ifadesi tarafından belirlenmektedir. Her **case** içinde parametre tipi (ya da tipleri) ve dönüş tipleri kontrol edilmektedir. Daha sonra metod çağrısını bu bilgiye dayalı olarak yapılandırılmaktadır.

## Nitelikler

C#, bir programa deklaratif bilgileri nitelik biçiminde eklemenize izin verir. Bir nitelik; bir sınıf, yapı, metod ve benzeriyle ilgili ek bilgileri tanımlar. Örneğin, bir sınıfın ekrana getireceği düğme (button) tipini belirleyen bir nitelik tanımlayabilirsiniz. Nitelikler, uygulandıkları ögeden önce ve köşeli parantez içinde belirtirler. Bu nedenle, bir nitelik bir sınıfın üyesi değildir ve ilişkili olduğu öğe hakkında ek bilgi sağlar.

## Niteliklerle İlgili Temel Bilgiler

Bir nitelik, `System.Attribute`'tan kalıtım yoluyla türetilen bir sınıf tarafından desteklenir. Dolayısıyla, tüm nitelik sınıfları, `Attribute`'ın alt sınıfları olmak durumundadır. `Attribute`'ın kayda değer ölçüde işlevsellik tanımlamasına rağmen bu işlevsellik, niteliklerle çalışırken her zaman gerekli değildir. Geleneksel olarak nitelik sınıflarında `Attribute` soneki kullanılır. Örneğin, `ErrorAttribute`, bir hatayı tanımlayan bir nitelik sınıfının ismi olarak tercih edilir.

Bir nitelik sınıfı deklare edilirken, daha önce `AttributeUsage` adlı bir nitelik yazılır. Sistemde tanımlı olan bu nitelik, deklare edilen niteliğin uygulanabileceği öğe tiplerini belirler.

### Nitelik Oluşturmak

Bir nitelik sınıfında niteliği destekleyen üyeleri sizin tanımlamanız gereklidir. Genellikle nitelik sınıfları oldukça basittir, sadece az sayıda alan ya da özellik içerirler. Örneğin, bir nitelik, ilişkili olduğu öğeyi tarif eden bir açıklama tanımlayabilir. Böyle bir niteliğin biçimini aşağıdaki gibi olabilir:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // remark ozelliginin altında yer alir

    public RemarkAttribute(string comment) {
        pri_remark = comment;
    }

    public string remark {
        get {
            return pri_remark;
        }
    }
}
```

Şimdi bu sınıfı satır satır ele alalım.

Bu niteliğin adı `RemarkAttribute`'dır. Niteliğin deklarasyonundan önce `AttributeUsage` adlı nitelik yer almıştır. `AttributeUsage`, `RemarkAttribute`'ın tüm öğe tiplerine uygulanabilir olduğunu belirlemektedir. `AttributeUsage`'ı kullanarak bir niteliğin ilişirileceği öğeler listesini kısaltmak mümkündür ve bu niteliğin neler yapabileceği bu bölümde ele alınacaktır.

Daha sonra `RemarkAttribute` deklare edilmiştir ve `Attribute`'tan kalıtım yoluyla türetilmiştir. `RemarkAttribute`'nın içinde `private` bir alan olan `pri_remark` yer almaktadır ve bu alan bir adet `public`, salt okunur özelliği yani `remark`'ı desteklemektedir. Bir karakter katarını argüman olarak alan ve bunu `remark`'a atayan açık bir yapılandırıcı da yer almaktadır.

Bu noktada artık başka bir adıma gerek kalmamıştır ve **RemarkAttribute** kullanıma hazırlıdır.

## Bir Niteliği İliştirmek

Bir nitelik sınıfını tanımladıktan sonra artık o niteliği bir öğeye ilişirebilirsiniz. Bir nitelik, ilişirildiği ögeden önce gelir ve yapılandırıcısı köşeli parantez içine alınarak belirlenir. Örneğin, **RemarkAttribute**'ın bir sınıfla ilişkilendirilmesi şu şekilde yapılabilir:

```
[RemarkAttribute("This class uses an attribute.")]  
class UseAttrib {  
    // ...  
}
```

Bu kod ite "This class uses an attribute" açıklamasını içeren bir **RemarkAttribute** kurulmaktadır. Daha sonra nitelik **useAttrib** ile ilişkilendirilmektedir.

Bir niteliği ilişirirken, **Attribute** soneğini kullanmak aslında gerekli değildir. Örneğin, yukarıdaki sınıf şu şekilde de bildirilebilir:

```
[Remark("This class uses an attribute.")]  
class UseAttrib {  
    // ...  
}
```

Burada isim olarak sadece **Remark** kullanılmıştır. Bu şekilde kısa kullanım da doğrudur ama nitelikleri ilişirirken tam isimlerini kullanmak daha güvenlidir. Çünkü böylece karışık belirsizlik ihtimalinden kaçınılmış olur.

## Bir Nesnenin Niteliklerini Almak

Bir nitelik bir öğeye ilişirildikten sonra, programın diğer bölümleri o niteliği okuyabilir. Bir niteliği okumak için genelde iki yöntemden biri kullanılır. İlk yöntem, **MemberInfo** tarafından tanımlanan ve **Type** tarafından kalıtım ile alınan **GetCustomAttributes()**'dır. Bu metot, bir nesneye ilişirilmiş olan tüm niteliklerin listesini okur ve şu genel biçimde sahiptir:

```
Object[ ] GetCustomAttributes(bool temelSınıflarıAra)
```

Burada **temelSınıflarıAra true** ise, kalıtım zinciri üzerindeki tüm temel sınıfların nitelikleri dahil edilecektir. Aksi halde sadece belirtilen tip tararından tanımlanan nitelikler bulunacaktır.

İkinci metot ise **Attribute** tarafından tanımlanan **GetCustomAttribute()**'dır. Bu metodun biçimlerinden biri aşağıda gösterilmiştir:

```
static Attribute GetCustomAttribute(MemberInfo mi, Type niteliktipi)
```

Burada **mi** niteliği, okunan öğeyi tanımlayan bir **MemberInfo** nesnesidir. İstenen nitelik **niteliktipi** tarafından belirtilmektedir. Bu metodu, okumak istediğiniz niteliğin adını

bildiğiniz durumlarda kullanırsınız - bu çoğu zaman geçerli olan bir durumdur. Örneğin **RemarkAttribute**'a bir referans almak için şu sekansı kullanırsınız:

```
// RemarkAttribute'u oku.
Type tRemAtt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)
    Attribute.GetCustomAttribute(t, tRemAtt);
```

Bir niteliğe ait bir referans elde ettikten sonra artık o niteliğin üyelerine erişebilirsiniz. Dolayısıyla, bir niteliğin ilişirildiği bir ögeyi kullanan bir program, o nitelikle ilintili bilgilere erişebilir. Örneğin, aşağıdaki ifade **remark** alanını gösterir:

```
Console.WriteLine(ra.remark);
```

Aşağıdaki program tüm parçaları bir araya getirmekte ve **RemarkAttribute**'ın kullanımını göstermektedir:

```
// Basit bir nitelik ornegi.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // remark ozelliginin altında yer alır

    public RemarkAttribute(string comment) {
        pri_remark = comment;
    }

    public string remark {
        get {
            return pri_remark;
        }
    }
}

[RemarkAttribute("This class uses an attribute.")]
class UseAttrib {
    // ...
}

class AttribDemo {
    public static void Main() {
        Type t = typeof(UseAttrib);

        Console.Write("Attributes in " + t.Name + ": ");

        object[] attrs = t.GetCustomAttributes(false);
        foreach(object o in attrs) {
            Console.WriteLine(o);
        }

        Console.Write("Remark: ");
    }
}
```

```

    // RemarkAttribute'i oku.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
        Attribute.GetCustomAttribute(t, tRemAtt);

    Console.WriteLine(ra.remark);
}
}

```

Programın çıktısı şöyledir:

```

Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.

```

## Konumsal ve İsimlendirilmiş Parametreler

Önceki örnekte, **RemarkAttribute**'a başlangıç değeri atamak için tanımlayıcı karakter katarı, yapılandırıcıya normal yapılandırıcı söz dizimi ile aktarılmıştı. Bu durumda **RemarkAttribute()**'un **comment** parametresine *konumsal parametre (positional parameter)* denir. Bu terim, argümanın parametreye konumu itibariyle bağlanmasıyla ilgilidir. C#'ta tüm metotlar ve yapılandırıcılar bu şekilde çalışır. Ancak, bir nitelik için isimlendirilmiş parametreler (*named parameter*) de oluşturabilirsiniz ve bunlara, isimlerini kullanarak başlangıç değerleri atayabilirsiniz.

İsimlendirilmiş bir parametre **public** bir alan ya da özellik tarafından desteklenir ve bu alan ya da özelliğin salt okunur olmaması gereklidir. Böyle herhangi bir alan ya da özellik, otomatik olarak, isimlendirilmiş bir parametre gibi kullanılabilir. İsimlendirilmiş bir parametreye değer atanması, bir öğe için nitelik belirtilirken niteliğin yapılandırıcısı içinde yer alan bir atama ifadesi ile yapılır. İsimlendirilmiş parametreleri de içine alan bir nitelik spesifikasyonu genel şekil ile aşağıda gösterilmiştir:

```
[attrib(konumsal-parametre-listesi, isimlendirilmiş-param1 = değer,
isimlendirilmiş-param2 = değer, ...)]
```

Konumsal parametreler (eğer mevcut iseler) önce gelirler. Daha sonra her bir isimlendirilmiş parametreye bir değer atanır. İsimlendirilmiş parametrelerin sırası önemli değildir.

İsimlendirilmiş parametrelere değer atanması gereklidir. Bu durumda varsayılan değerleri kullanılır.

İsimlendirilmiş bir parametrenin nasıl kullanılacağını anlamanın en iyi yolu, bir örnek üzerinden gitmektedir. **RemarkAttribute**'ın **supplement** adlı yeni bir alanı olan ve ek bir açıklamayı tutabilen bir versiyonu aşağıda yer almaktadır:

```

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // remark ozelliginin altında yer alir
}

```

```

public string supplement; // bu isimlendirilmiş bir parametredir

public RemarkAttribute(string comment) {
    pri_remark = comment;
    supplement = "None";
}

public string remark {
    get {
        return pri_remark;
    }
}
}

```

Gördüğünüz gibi, yapılandırıcı tarafından **supplement**'a başlangıç değeri olarak "None" karakter katarı atanmaktadır. Yapılandırıcıyı başka bir şekilde kullanarak farklı bir başlangıç değeri atamak mümkün değildir. Ancak, **supplement** şu şekilde isimlendirilmiş bir parametre olarak kullanılabilir:

```

[RemarkAttribute("This class uses an attribute.",
    supplement = "This is additional info.")]
class UseAttrib {
    // ...
}

```

**RemarkAttribute**'ın yapılandırıcısının nasıl çağrıldığına çok dikkat edin. İlk olarak, daha önce de olduğu gibi, konumsal argüman belirtilmiştir. Ardından bir virgül ve onun peşinden de isimlendirilmiş parametre olan **supplement** gelmiş ve bu parametreye bir değer atanmıştır. Son olarak kapanış parantezi ) yapılandırıcıya yapılan çağrıya son vermiştir. Bir başka ifade ile, isimlendirilmiş parametreye başlangıç değeri, yapılandırıcıya yapılan çağrı içinde atanmıştır. Bu söz dizimini genelleştirmek mümkündür. Konumsal parametreler, göründükleri sırada belirtilmelidirler. Isimlendirilmiş parametreler ise isimlerine değer atanarak belirtilirler.

Aşağıdaki program **supplement** alanını göstermektedir:

```

// isimlendirilmiş bir nitelik parametresi kullanır.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // remark ozelliginin altında yer alır
    public string supplement; // bu isimlendirilmiş bir parametredir

    public RemarkAttribute(string comment) {
        pri_remark = comment;
        supplement = "None";
    }

    public string remark {

```

```

        get {
            return pri_remark;
        }
    }

[RemarkAttribute("This class uses an attribute.",
                 supplement = "This is additional info.")]
class UseAttrib {
    // ...
}

class NamedParamDemo {
    public static void Main() {
        Type t = typeof(UseAttrib);

        Console.WriteLine("Attributes in " + t.Name + ": ");

        object[] attribs = t.GetCustomAttributes(false);
        foreach(object o in attribs) {
            Console.WriteLine(o);
        }

        // RemarkAttribute'u oku.
        Type tRemAtt = typeof(RemarkAttribute);
        RemarkAttribute ra = (RemarkAttribute)
            Attribute.GetCustomAttribute(t, tRemAtt);

        Console.WriteLine("Remark: ");
        Console.WriteLine(ra.remark);

        Console.WriteLine("Supplement: ");
        Console.WriteLine(ra.Supplement);
    }
}

```

Programın çıktısı şöyledir:

```

Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.
Supplement: This is additional info.

```

Daha önce açıklandığı gibi, bir özellik aynı zamanda isimlendirilmiş bir parametre olarak da kullanılabilir. Örneğin burada **priority** adlı bir int parametresi **RemarkAttribute**'a eklenmektedir:

```

// Bir ozelligin, isimlendirilmis bir parametre olarak kullanimi.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
    string pri_remark; // remark ozelliginin altında yer alır
}

```

```
int pri_priority; // priority ozelliginin altinda yer alir
public string supplement; // bu isimlendirilmis bir parametredir
public RemarkAttribute(string comment) {
    pri_remark = comment;
    supplement = "None";
}

public string remark {
    get {
        return pri_remark;
    }
}

// Bir ozelligin isimlendirilmis bir parametre olarak kullanimi.
public int priority {
    get {
        return pri_priority;
    }
    set {
        pri_priority = value;
    }
}
}

[RemarkAttribute("This class uses an attribute.",
    supplement = "This is additional info.",
    priority = 10)]
class UseAttrib {
    // ...
}

class NamedParamDemo {
    public static void Main() {
        Type t = typeof(UseAttrib);

        Console.WriteLine("Attributes in " + t.Name + ": ");

        object[] attrs = t.GetCustomAttributes(false);
        foreach(object o in attrs) {
            Console.WriteLine(o);
        }

        // RemarkAttribute'u oku.
        Type tRemAtt = typeof(RemarkAttribute);
        RemarkAttribute ra = (RemarkAttribute)
            Attribute.GetCustomAttribute(t, tRemAtt);

        Console.WriteLine("Remark: ");
        Console.WriteLine(ra.remark);

        Console.WriteLine("Supplement: ");
        Console.WriteLine(ra.supplement);
        Console.WriteLine("Priority: " + ra.priority);
```

```

    }
}

```

Programın çıktısı şöyledir:

```

Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.
Supplement: This is additional info.
Priority: 10

```

Programın ilginç bir yönü vardır. Dikkat ederseniz, **UseAttrib**'den önce belirtilen nitelik aşağıda gösterilmiştir:

```
[RemarkAttribute("This class uses an attribute.",
    supplement = "Tnis is additional info.",
    priority = 10)]
```

İsimlendirilmiş nitelikler **supplement** ve **priority** özel bir şekilde sıralanmamıştır. Bu atamaların sırası, nitelikte herhangi bir değişikliğe yol açmadan değiştirilebilir.

## Standart Niteliklerin Kullanımı

C#'ta üç standart nitelik tanımlanmıştır: **AttributeUsage**, **Conditional** ve **Obsolete**. Bu bölümde söz konusu nitelikleri incelemektedir.

### AttributeUsage

Daha önce de debynildiği gibi, **AttributeUsage** niteliği, bir niteliğin hangi ögelere uygulanabildiğini belirler. **AttributeUsage**, **System.AttributeUsageAttribute** sınıfının başka bir adıdır. **AttributeUsage**'nın yapılandırıcısı aşağıdaki gibidir:

```
AttributeUsage(AttributeTargets öge)
```

Burada **öge**, niteliğin üzerinde kullanılabileceği öge ya da ögeleri belirler.

**AttributeTargets**, aşağıdaki değerleri tanımlayan bir numaralandırmadır:

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
Interface	Method	Module	Parameter
Property	ReturnValue	Struct	

Bu değerlerden iki ya da daha fazlasına VEYA uygulanabilir. Örneğin, sadece alan ve özelliklere uygulanabilecek bir nitelik belirtmek için şunu kullanın:

```
AttributeTargets.Field | AttributeTargets.Property
```

**AttributeUsage** iki isimlendirilmiş parametreyi desteklemektedir. Bunlardan ilki bir **bool** değeri olan **AllowMultiple**'dır. İkincisi ise yine bir **bool** değeri olan **Inherited**'dır. Eğer bu değer **true** ise nitelik, türetilmiş sınıflar tarafından kalıtım yoluyla alınır. Aksi halde,

kalıtım yoluyla alınmaz. Hem **AllowMultiple** hem de **Inherited**'in varsayılan değerleri **false** olur.

## Conditional Niteliği

**Conditional** belki de C#'ın en ilginç standart niteliğidir ve *koşullu metodlar (conditional method)* oluşturmanızı sağlar. Koşullu bir metot, yalnızca spesifik bir sembol **#define** ile tanımlandığı zaman çağrılr. Aksi halde metot pas geçilir. Dolayısıyla, koşullu bir metot, **#if** ile yapılan koşullu derlemeye bir alternatif sunmaktadır.

**Conditional**, **System.Diagnostics.ConditionalAttribute**'ın başka bir adıdır. **Conditional** niteliğini kullanmak için **System.Diagnostics** isim uzayını kullanmalısınız. Bir örnekle başlayalım:

```
// Conditional niteliginini gosterir.

#define TRIAL

using System;
using System.Diagnostics;

class Test {

    [Conditional("TRIAL")]
    void trial() {
        Console.WriteLine("Trial version, not for distribution.");
    }

    [Conditional("RELEASE")]
    void release() {
        Console.WriteLine("Final release version.");
    }

    public static void Main() {
        Test t = new Test();

        t.trial(); // yalnızca TRIAL tanımlıysa çağrılr
        t.release(); // yalnızca RELEASE tanımlıysa çağrılr
    }
}
```

Programın çıktısı şu şekildedir:

```
Trial version, not for distribution.
```

Neden bu çıktıının üretildigini anlamak için programa daha yakından bakalım. Öncelikle, programın **TRIAL** simgesini tanımladığına dikkat edin. Ardından **trial()** ve **release()** metodlarının nasıl kodlandığını gözleyin. Her ikisinden önce **Conditional** niteliği yer almaktadır ve bu nitelik aşağıdaki genel biçimde sahiptir:

```
[Conditional "sembol"]
```

Burada **sembol**, metodun çalıştırılıp çalıştırılmayacağını belirleyen semboldür. Bu nitelik yalnızca metotlar üzerinde kullanılabilir. Eğer sembol tanımlanmışsa, metot çağrıldığında çalışacaktır. Eğer sembol tanımlanmamışsa, metot çalışmayaçaktır.

**Main()**'in içinde hem **trial()** hem de **release()** çağrılmaktadır. Ancak sadece **TRIAL** tanımlanmıştır. Bu nedenle **trial()** çalışmaktadır. **release()**'e yapılan çağrı göz ardı edilmektedir. **RELEASE** tanımlanırsa, o zaman **release()** da çağrılacaktır. **TRIAL**'ın tanımı kaldırılırsa, **trial()** çağrılmayacaktır.

Koşullu metotlarla ilgili bazı kısıtlamalar vardır. Bu metotların **void** döndürmeleri gereklidir. Bir arayüzün değil bir sınıfın üyesi olmalıdır. Önlerine **override** anahtar kelimesi gelemez.

## Obsolete Niteliği

**System.ObsoleteAttribute**'ın kısa biçimi olan **Obsolete** niteliği, bir program ögesini eski olarak işaretlemenize olanak sağlar. Bu niteliğin genel biçimini aşağıdaki gibidir:

```
[Obsolete("mesaj")]
```

Burada, ilgili program üyesi derlenirken **mesaj** ekrana gelmektedir. İşte, kısa bir örnek

```
// Obsolete niteliginin gosterir.

using System;

class Test {

    [Obsolete("Use myMeth2, instead.")]
    static int myMeth(int a, int b) {
        return a / b;
    }

    // myMeth'in gelistirilmis versiyonu.
    static int myMeth2(int a, int b) {
        return b == 0 ? 0 : a / b;
    }

    public static void Main() {
        // bunun icin uyarı mesaji verilir
        Console.WriteLine("4 / 3 is " + Test.myMeth(4, 3));

        // burada uyarı yapılmaz
        Console.WriteLine("4 / 3 is " + Test.myMeth2(4, 3));
    }
}
```

Program derlenirken **Main()**'de **myMeth()**'e yapılan çağrı ile karşılaşıldığında bir uyarı mesajı verilerek kullanıcının bu metot yerine **myMeth2()**'yi kullanması istenecektir.

**Obsolete**'in ikinci bir şekli şöyledir:

[Obsolete("mesaj", hata) ]

Burada **hata**, bir Boolean değerdir. Eğer **true** ise, eski ögenin kullanımı uyarı vermek yerine derleme hatasına neden olacaktır. Aradaki fark elbette, hatalı bir programın derlenerek çalıştırılabilir hale getirilmesine imkan vermemesidir.

# **EMNİYETSİZ KOD, İŞARETÇİLER VE ÇEŞİTLİ KONULAR**

Bu bölümde, ismi genellikle kullanıcılar arasında şaşkınlıkla karşılanan bir C# özelliği işlenmektedir: Emniyetsiz kod. Emniyetsiz kod (*unsafe code*) genellikle işaretçilerin kullanımını içerir. Emniyetsiz kod ile işaretçiler birlikte, normalde C++ ile ilişkilendirebileceğiniz uygulamaları oluşturmak amacıyla C#'ın kullanılmasını mümkün kılmaktadırlar: Yüksek performansa sahip sistem kodu. Üstelik, emniyetsiz kod ve işaretçilerin C#'a dahil edilmesi, Java'da olmayan becerileri C#'a vermektedir.

Bu bölümün sonunda, önceki bölümlerde işlenmeyen birkaç anahtar kelime ele alınmaktadır.

## Emniyetsiz Kod

C#, “emniyetsiz” kod olarak adlandırılan kodları yazmanıza olanak tanır. Bu ifade şok edici gibi görünüyor olsa da aslında değildir. Emniyetsiz kod, kötü yazılmış kod demek değildir; doğrudan Common Language Runtime’ın (CLR) yönetiminde çalıştırılmayan kod demektir. Bölüm 1’de açıkladığı gibi, C# normal olarak kontrol altında kod oluşturmak için kullanılmaktadır. Yine de, doğrudan CLR’ın kontrolü altında çalışmayan kod yazmak da mümkündür. Bu kontrol altında olmayan kod, kontrol altında olan kodun tabi olduğu denetim ve kısıtlamalara tabi değildir; bu nedenle, buna “emniyetsiz” denir, çünkü bu tür bir kodun bir tür zararlı faaliyette bulunmayacağıını doğrulamak mümkün değildir. Böylece, *emniyetsiz* terimi, söz konusu kodun yapısı gereği kusurlu olduğu anlamına gelmez. Bu yalnızca, kontrol altındaki kod çerçevesinde, söz konusu kodun denetime tabi olmayan faaliyetlerde bulunması mümkün demektir.

Emniyetsiz kodun problemlere neden olabileceği biliniyorsa, kikinin bu tür bir kodu neden isteyebileceğini sorabilirsiniz. Bu sorunun yanıtı şudur: Kontrol altındaki kod *işaretçi* (*pointer*) kullanımını önler. C ya da C++’a aşınaysanız, işaretçilerin diğer nesnelerin adreslerini tutan değişkenler olduklarını bilirsiniz. Yani, işaretçiler bir parça C#'taki referanslar gibidir. İkisi arasındaki temel fark, bir işaretçinin bellekte herhangi bir yere işaret edebileceğidir; bir referans ise daima kendi tipinde bir nesneye işaret eder. Bir işaretçi bellekte herhangi bir yere işaret edebileceği için, işaretçileri hatalı kullanmak mümkündür. Ayrıca, işaretçilerin kullanımı sırasında kodlama hatası yapmak da kolaydır. Kontrol altında kod oluşturulurken C#'ın işaretçileri desteklememesinin nedeni budur. Halbuki işaretçiler, bazı programlama türleri (söz gelişi, sistem düzeyinde yardımcı programlar) için hem kullanıcıları hem de gereklidir. C# işaretçileri oluşturmanıza ve kullanmanıza kesinlikle olanak tanır. Tüm işaretçi işlemleri güvensiz olarak işaretlenmelidir, çünkü bunlar kontrol altındaki kod kapsamı dışında gerçekleşmekte dirler.

C#'ta işaretçilerin deklarasyonu ve kullanımı C/C++’taki ile paralellik gösterir - C/C++’ta işaretçilerin nasıl kullanıldığını biliyorsanız, bunları C#'ta kullanabilirsiniz. Ancak, C#'ın dayandığı asıl konu kontrol altındaki kodların oluşturulmasıdır. C#'ın kontrol altında olmayan kodları destekleme becerisi, C#'ın özel bir dizi probleme uygulanmasına imkan vermektedir. Bu, normal C# programlama için geçerli değildir. Aslında, kontrol altında olmayan kodu derlemek için /unsafe derleyici seçeneğini kullanmalısınız.

İşaretçiler emniyetsiz kodun çekirdeğinde yer aldıkları için bunlarla başlayacağız.

## İşaretçilerin Esasları

İşaretçiler, diğer değişkenlerin adreslerini tutan değişkenlerdir. Örneğin, eğer *x*, *y*'nin adresini içeriyorsa; *x*, *y*'ye "işaret ediyor" denilir. Bir işaretçi bir kez bir değişkene işaret ettikten sonra, artık bu değişkenin değeri işaretçi aracılığıyla elde edilebilir ya da değiştirilebilir. İşaretçiler aracılığıyla gerçekleştirilen işlemler genellikle *dolaylılık* (*indirection*) olarak adlandırılır.

## İşaretçi Deklare Etmek

İşaretçi değişkenleri şu şekilde deklare edilmelidir. Bir işaretçi değişkeninin genel biçimini söyleyelim:

```
tip* değişken-ismi;
```

Burada **tip**, değişkenin temel tipidir. **tip**, referans tipi olmamalıdır. Bu nedenle, bir sınıf nesnesine işaret eden bir işaretçi deklare edemezsınız. İşaretçinin temel tipi ayrıca işaretçinin *kast edilen* (referent) *tipi* olarak da kullanılır. \* simgesinin yerine dikkat edin. Bu işaret, tip isminin peşinden gelmektedir. **değişken-ismi**, işaretçi değişkeninin ismidir.

İşte bir örnek. Bir **int**'e işaret eden **ip** adında bir işaretçi deklare etmek için şu deklarasyonu kullanın:

```
int* ip;
```

Bir **float** işaretçi için şunu kullanın:

```
float* fp;
```

Genel olarak, bir deklarasyon ifadesinde bir tip isminin ardından \* simgesini kullanmak bir işaretçi tipi oluşturur.

Bir işaretçinin işaret edeceği verinin tipi, işaretçinin temel tipi ile belirlenir. Böylece, önceki örnekte **ip**, bir **int**'e işaret etmek için; **fp** ise bir **float**'a işaret etmek için kullanılabilir. Bununla birlikte, bir işaretçinin başka bir yere işaret etmesini gerçekten önleyeceğin hiçbir şey olmadığını anlayın. İşaretçilerin potansiyel olarak güvenli olmamalarının sebebi budur.

Eğer C/C++ biliyorsanız, C# ve C/C++'ın işaretçileri deklare etme biçimleri arasındaki önemli bir farkın farkında olmanız gereklidir. C/C++'ta bir işaretçi tipi deklare edince \*, deklarasyondaki değişkenler listesi üzerinde dağılamaz. Yani, C/C++'ta şu ifade

```
int* p, q;
```

**p** adında bir **int** işaretçi ve **q** adında bir **int** deklare eder. Bu ifade, aşağıdaki iki deklarasyona denktir:

```
int* p;
int q;
```

Ancak, C#'ta \*, dağılma özelliğine sahiptir ve aynı deklarasyon iki işaretçi değişkeni oluşturur:

```
int* p, q;
```

Yani, C#'ta bu ifade aşağıdaki iki deklarasyonla aynıdır:

```
int* p;
int* q;
```

Bu, C/C++ kodunu C#'a taşırken aklınızda tutmanız gereken önemli bir faptır.

## \* ve & İşaretçi Operatörleri

İşaretçilerle birlikte iki operatör kullanılır: \* ve &. &, operandının bellek adresini döndüren tekli bir operatördür. (Tekli operatörlerin yalnızca tek operand gerektirdiklerini hatırlayın.) Örneğin,

```
int* ip;
int num = 10;

ip = &num;
```

**num** değişkeninin adresini **ip**'ye yerleştirir. Bu adres, değişkenin bilgisayarın dahili belleğindeki konumudur. Bunun **num**'ın *değeri* ile bir *ilişkisi yoktur*. Yani, **ip** 10 değerini (**num**'ın ilk değeri) *icermez*. **num**'ın saklandığı adresi içerir. & işlemi, önünde yer aldığı değişkenin “adresini” doldurmek olarak hatırlanabilir. Bu yüzden, yukarıdaki atama ifadesi “**ip**, **num**'ın adresini alır” olarak kelimeleme dökülebilir.

\* ise ikinci operatördür ve bu, & operatörünün tamamlayıcısıdır. \*, operandı tarafından belirtilen adresteki değişkenin değerine referansta bulunan tekli bir operatördür. Yani, bir işaretçi tarafından işaret edilen bir değişkenin değerine referansta bulunur. Aynı önekle devam edersek, eğer **ip**, **num**'ın bellek adresini içeriyorsa,

```
int val = *ip;
```

**val**'a 10 değerini yerlestirecektir. **10**, **ip** tarafından işaret edilen **num**'ın değeridir. \* işlemi “adresindeki” olarak hatırlanabilir. Bu önekte, yukarıdaki ifade “**val**, **ip** adresindeki değeri alır” şeklinde okunabilir.

\*, bir atama ifadesinin sol tarafında da kullanılabilir. Bu tür bir kullanım, işaretçi tarafından işaret edilen değeri ayarlar. Örneğin:

```
*ip = 100;
```

**ip** tarafından işaret edilen değişkene - bu önekte bu **num**'dır - 100 değerini atar. Yani, bu ifade “**ip** adresine 100 değerini koy” şeklinde okunabilir.

## unsafe Kullanımı

İşaretçileri kullanan herhangi bir kod, **unsafe** anahtar kelimesi kullanılarak emniyetsiz olarak işaretlenmelidir. Tek bir ifadeyi ya da bütün bir metodu **unsafe** olarak işaretleyebilirsiniz. Örneğin, **unsafe** olarak işaretlenmiş **Main()** içinde işaretçileri kullanan bir program aşağıda yer almaktadır:

```
// Isaretcileri ve unsafe kullanimini goster.

using System;

class UnsafeCode {
    // Main'i unsafe olarak isaretle.
    unsafe public static void Main() {
        int count = 99;
        int* p; // bir int isaretci olustur

        p = &count; // count'un adresini p'ye koy

        Console.WriteLine("Initial value of count is " + *p);

        *p = 10; // p uzerinden count'a 10 degerini ata

        Console.WriteLine("New value of count is " + *p);
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Initial value of count is 99
New value of count is 10
```

## fixed Kullanımı

İşaretçilerle çalışırken **fixed** niteleyicisi sıkça kullanılır. **fixed**, kontrol altındaki bir değişkenin anlamsız veri toplayıcısı (garbage collector) tarafından taşınmasını önler. Bir işaretçi, örneğin, bir sınıf nesnesi içindeki bir alana referansta bulunurken bu gereklidir. İşaretçi, anlamsız veri toplayıcısının faaliyetlerinden haberdar olmadığı için, söz konusu nesne taşınırsa işaretçi yanlış nesneye işaret ediyor olacaktır. **fixed**'in kullanımı genel olarak şu şekildedir:

```
fixed (tip* p = &değişken) {
    // sabit nesne kullanır
}
```

Burada **p**, kendisine bir değişkenin adresi atanmakta olan bir işaretcidir. Kod bloğu gerçeklenene kadar nesne bellekteki mevcut konumunda kalacaktır. Bir **fixed**'ı ifadenin hedefi olarak tek bir ifadede de kullanabilirsiniz. **fixed** anahtar kelimesi ayrıca emniyetsiz kod kapsamında da kullanılabilir. Virgülle ayrılmış bir liste kullanarak aynı anda birden fazla **fixed** işaretçi deklare edebilirsiniz.

İşte **fixed** kullanılan bir program örneği:

```
// fixed kullanımını gösterir.

using System;

class Test {
    public int num;
    public Test(int i) { num = i; }
}

class FixedCode {
    // Main'i unsafe olarak işaretle.
    unsafe public static void Main() {
        Test o = new Test(19);

        fixed (int* p = &o.num) { // o.num'ın adresini p'ye
            // yerlestirmek için fixed kullan
            Console.WriteLine("Initial value of o.num is " + *p);

            *p = 10; // p üzerinden count'a atama yap
            Console.WriteLine("New value of o.num is " + *p);
        }
    }
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Initial value of o.num is 19
New value of o.num is 10
```

Burada **fixed**, **o**'nun taşınmasına engel olur. **p**, **o.num**'a işaret ettiği için, **o** taşınmışsa **p** geçersiz bir konuma işaret ediyor olacaktır.

## Yapı Üyelerine Bir İşaretçi Aracılığıyla Erişmek

Bir yapı, referans tipleri içermediği sürece, bir işaretçi söz konusu yapı tipinde bir nesneye işaret edebilir. Bir yapının üyesine bir işaretçi aracılığıyla erişirken nokta (.) operatörü yerine ok (->) operatörünü kullanmalısınız. Örneğin, aşağıdaki yapının mevcut olduğunu varsayarsak,

```
struct MyStruct {
    public int x;
    public int y;
    public int sum() { return x + y; }
}
```

bu yapının üyelerine bir işaretçi aracılıyla şu şekilde erişebilirsiniz:

```
MyStruct o = new MyStruct();
MyStruct* p; // bir işaretçi deklare et

p = &o;
p->x = 10;
```

```
p->y = 20;  
Console.WriteLine("Sum is " + p->sum());
```

## İşaretçi Aritmetiği

İşaretçiler üzerinde kullanılabilecek yalnızca dört aritmetik operatörü mevcuttur: `++`, `--`, `+` ve `-`. İşaretçi aritmetığının nasıl uygulandığını anlamak için bir örnekle başlayacağız. `p1`, mevcut değeri `2000` olan bir `int` işaretçisi olsun (yani, `p1` `2000` adresini içersin). Şu deyimden sonra

```
pi++;
```

`p1`'in içeriği `2001` değil `2004` olacaktır! Bunun nedeni, `p1`'in her artırmada bir sonraki `int`'e işaret edecek olmasıdır. C#'ta tamsayılar **4** byte uzunluğunda olduğu için `p1`'i bir kez artırmak değerine **4** ekler. Bunun tersi de eksiltmeler için geçerlidir. Her eksiltme `p1`'in değerini **4** azaltır. Örneğin,

```
pi--;
```

`p1`'in bir önceki değerinin `2000` olduğunu varsayırsak, bu ifade `p1`'in `1996` değerine sahip olmasına neden olacaktır.

Yukarıdaki örneği genelleştirecek olursak, bir işaretçinin her artırmásında söz konusu işaretçi, kendi temel tipindeki bir sonraki elemanın bellek konumuna işaret edecektir. İşaretçinin her eksiltilişinde ise kendi temel tipindeki bir önceki elemanın konumuna işaret edecektir.

İşaretçi aritmetiği yalnızca artırma ve eksiltme işlemleriyle sınırlı değildir. İşaretçilere ayrıca tamsayı ekleyebilir ya da işaretçilerden tamsayı çıkarabilirsiniz. Aşağıdaki deyim,

```
p1 = p1 + 9;
```

`p1`'in halihazırda işaret ettiği elemandan sonraki (kendi tipindeki) 9. elemanı göstermesini sağlar.

İşaretçileri birbiriyle toplayamıyor olmanızı rağmen, bir işaretçiyi bir diğerinden (her ikisinin de aynı temel tipte olması şartıyla) çıkarabilirsiniz. Kalan, her iki işaretçi arasında kalan temel tipteki elemanların sayısıdır.

Bir işaretçinin bir tamsayı ile toplanması, bir işaretçiden bir tamsayı çıkartılması ya da iki işaretçinin birbirinden çıkartılması işlemlerinin haricinde işaretçiler üzerinde başka hiç bir aritmetik işlem gerçekleştirilemez. Örneğin, işaretçilere `float` ya da `double` değerler ekleyemezsiniz veya işaretçilerden bu değerleri çıkartamazsınız.

İşaretçi aritmetığının etkilerini görmek için bir sonraki kısa programı çalıştırın. Program, bir tamsayı işaretçisi (`ip`) ile bir kayan nokta işaretçisinin (`fp`) gösterdiği asıl fiziksel adresleri

yazdırır. Döngünün her tekrarında her iki işaretçinin kendi temel tiplerine bağlı olarak nasıl değişiklerini gözleyin.

```
// Isaretci aritmetiginin etkilerini gosterir.

using System;

class PtrArithDemo {
    unsafe public static void Main() {
        int x;
        int i;
        double d;

        int* ip = &i;
        double* fp = &d;

        Console.WriteLine("int      double\n");

        for(x = 0; x < 10; x++) {
            Console.WriteLine((uint) (ip) + " " + (uint) (fp));
            ip++;
            fp++;
        }
    }
}
```

Örnek çıktı aşağıda gösterilmiştir. Sizin çıktınız farklı olabilir, ancak aralıklar aynı olacaktır.

```
int      double
1243464 1243468
1243468 1243476
1243472 1243484
1243476 1243492
1243480 1243500
1243484 1243508
1243488 1243516
1243492 1243524
1243496 1343532
1243500 1243540
```

Cıktıdan da görüldüğü gibi işaretçi aritmetiği, işaretçinin temel tipine bağlı olarak gerçekleştirilir. Bir **int** 4 byte ve bir **double** 8 byte olduğu için adresler, bu değerlerin katları olarak değişir.

## İşaretçilerin Karşılaştırılması

İşaretçiler, **==**, **<** ve **>** gibi ilişkisel operatörler kullanılarak karşılaştırılabilir. Ancak işaretçilerin karşılaştırılmasıyla elde edilen sonucun anlamlı olması için, genellikle söz konusu iki işaretçinin kendi aralarında bir ilişki olması gereklidir. Örneğin, **p1** ve **p2** eğer birbirinden ayrı ve ilgisiz iki değişkene işaret ediyorlarsa, **p1** ve **p2** arasındaki herhangi bir karşılaştırma

genellikle anlamsız olacaktır. Ancak, eğer **p1** ve **p2**, söz gelişî aynı dizinin elemanları gibi birbirîyle bağınlı değişkenlere işaret ediyorlarsa, anlamlı bir şekilde karşılaştırılabilirler.

İşte, bir dizinin ortasındaki elemanı bulmak için işaretçilerin karşılaştırılmasını kullanan bir örnek:

```
// İşaretçilerin karşılastırılmasını gösterir.

using System;

class PtrCompDemo {
    unsafe public static void Main() {

        int[] nums = new int[11];
        int x;

        // ortadakini bul
        fixed (int* start = &nums[0]) {
            fixed(int* end = &nums[nums.Length-1]) {
                for(x = 0; start + x <= end - x; x++);
            }
        }
        Console.WriteLine("Middle element is " + x);
    }
}
```

Çıktı şöyledir:

```
Middle element is 6
```

Bu program, **start**'ı başlangıçta dizinin ilk elemanını, **end**'ı de son elemanını gösterecek şekilde ayarlayarak dizinin ortasındaki elemanı bulur. Sonra, işaretçi aritmetiği kullanılarak **start** işaretçisi artırılır ve **start**, **end**'e eşit ya da **end**'den küçük olana kadar **end** işaretçisi azaltılır.

Bir diğer husus da şudur: **start** ve **end** işaretçileri bir **fixed** ifadesi içinde oluşturulmalıdır, çünkü bu işaretçiler, kendisi bir referans tipi olan bir dizinin elemanlarına işaret ederler. C#’ta dizilerin nesneler olarak uygulandıklarını ve anlamsız veri toplayıcısı (garbage collector) tarafından taşınabileceğini hatırlayın.

## İşaretçiler ve Diziler

C#’ta işaretçiler ve diziler birbirîyle bağınlılıdır. Örneğin, indekssiz bir dizi ismi, dizinin başlangıcına bir işaretçi üretir. Aşağıdaki programı ele alın:

```
/* Indekssiz bir dizi ismi, dizinin baslangicina bir isaretci
uretir. */

using System;

class PtrArray {
```

```

unsafe public static void Main() {
    int[] nums = new int[10];

    fixed(int* p = &nums[0], p2 = nums) {
        if(p == p2)
            Console.WriteLine("p and p2 point to same address.");
    }
}

```

Çıktı aşağıda gösterilmiştir:

p and p2 point to same address.

Çıktıdan da görüldüğü gibi, aşağıdaki deyim

&nums[0]

şununla aynıdır:

nums

İkinci kullanım daha kısa olduğu için, bir dizinin başlangıcını gösteren bir işaretçiye gerek olduğunda programcıların birçoğu bunu kullanırlar.

## İşaretçiyi İndekslemek

Bir işaretçi bir diziye referansta bulununca söz konusu işaretçi sanki bir diziymiş gibi indekslenebilir. Bu söz dizimi, bazı durumlarda çok daha kullanışlı olabilen bir işaretçi aritmetığıne alternatif sağlar. İşte bir örnek:

```

// Bir isaretcisiyi sanki bir diziyimis gibi indeksler.

using System;

class PtrIndexDemo {
    unsafe public static void Main() {
        int[] nums = new int[10];

        // isaretcisiyi indeksle
        Console.WriteLine("Index pointer like array.");
        fixed (int* p = nums) {
            for(int i = 0; i < 10; i++)
                p[i] = i; // isaretcisiyi tipki dizi gibi indeksle

            for(int i = 0; i < 10; i++)
                Console.WriteLine("p[{0}]: {1} ", i, p[i]);
        }

        // isaretcisi aritmetigi kullan.
        Console.WriteLine("\nUse pointer arithmetic.");
        fixed (int* p = nums) {
            for(int i=0; i < 10; i++)
                *(p+i) = i; // isaretcisi aritmetigi kullan.
        }
    }
}

```

```

        for(int i = 0; i < 10; i++)
            Console.WriteLine("*(p+{0}): {1} ", i, *(p+i));
    }
}
}

```

Çıktı aşağıda gösterilmiştir:

Index pointer like array.

```

p[0]: 0
p[1]: 1
p[2]: 2
p[3]: 3
p[4]: 4
p[5]: 5
p[6]: 6
p[7]: 7
p[8]: 8
p[9]: 9

```

Use pointer arithmetic.

```

*(p+0): 0
*(p+1): 1
*(p+2): 2
*(p+3): 3
*(p+4): 4
*(p+5): 5
*(p+6): 6
*(p+7): 7
*(p+8): 8
*(p+9): 9

```

Programda da gösterildiği gibi, aşağıdaki gibi genel biçimde olan bir işaretçi deyimi

`* (ptr + i)`

şu tür bir dizi indeksleme söz dizimi kullanılarak yeniden yazılabilir:

`ptr[i]`

Bir işaretçi indekslemekle ilgili anlaşılması gereken iki önemli husus mevcuttur: Öncelikle, dizi sınırları kontrol edilmez. Böylece, işaretçinin referansta bulunduğu dizinin sonundan ileride bulunan bir elemana erişmek mümkündür. İkincisi, işaretçilerin `Length` özelliği yoktur. Bu nedenle, işaretçi kullanıldığından dizinin ne kadar uzunlukta olduğunu bilmenin bir yolu yoktur.

## İşaretçiler ve Karakter Katarları

Karakter katarları C#'ta nesneler olarak uygulansalar da, bir karakter katarı içindeki karakterlere bir işaretçi aracılığıyla da erişmek mümkündür. Bunun için karakter katarının

başını gösteren bir işaretçiyi aşağıdaki gibi bir **fixed** ifade kullanarak bir **char\*** işaretçi olarak atayacaksınız:

```
fixed(char* p = str) { // ...
```

**fixed** ifade çalıştırıldıktan sonra **p**, karakter katarını oluşturan karakter dizisinin başına işaret edecektir. Bu dizi *null sonlandırılmalıdır*; yani, dizinin sonunda **0** vardır. Dizinin sonunu test etmek için bu gerçeği kullanabilirsiniz. Karakter katarları C/C++'ta da null sonlandırmalı karakter dizileri şeklinde uygulanırlar. Böylece, bir **string**'e işaret eden bir **char\*** işaretçisi edinmek, karakter katarları üzerinde C/C++'takine çok benzer şekilde işlem yapmanıza olanak tanır.

İşte, bir karakter katarına bir **char\*** işaretçisi aracılığıyla erişimi gösteren bir program:

```
// Bir karakter katarının basını gösteren bir işaretçi edinmek
// için fixed kullanır.

using System;

class FixedString {
    unsafe public static void Main() {
        string str = "this is a test";

        // p, str'in basına işaret etsin
        fixed(char* p = str) {

            // str'in içeriğini p üzerinden görüntüle.
            for(int i = 0; p[i] != 0; i++)
                Console.Write(p[i]);
        }

        Console.WriteLine();
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
this is a test
```

## Çoklu Dolaylılık

Hedef değere işaret eden işaretçiye işaret eden bir başka işaretçiye sahip olabilirsiniz. Bu durum, *çoklu dolaylılık* (*multiple indirection*) ya da *işaretçilere işaret eden işaretçiler* olarak adlandırılır. İşaretçilere işaret eden işaretçiler karışık görünebilir. Şekil 18.1, çoklu dolaylılık kavramını netleştirmeye yardımcı olmaktadır. Gördüğünüz gibi, normal bir işaretçinin değeri, denilen değeri içeren değişkenin adresidir. Bir işaretçiye işaret eden başka bir işaretçi durumunda, ilk işaretçi ikinci işaretçinin adresini içerir. İkinci işaretçi ise, istenilen değeri içeren değişkene işaret eder.

Çoklu dolaylılık istenilen seviyeye kadar uzatılabilir; fakat bir işaretçiye işaret eden ikinci bir işaretçiden daha fazlasına çok ender olarak gerek duyulur. Aslında, aşırı dolaylılığı takip etmesi zordur ve bu, kavramsal hatalara açıktır.

Başka bir işaretçiye işaret eden bir işaretçi niteliğindeki bir değişken şu şekilde deklare edilmelidir. Bunu, tip isminden sonra ilave bir **\*** yerleştirerek gerçekleştirirsınız. Örneğin, aşağıdaki deklarasyon derleyiciye **q**'nun **int** tipinde bir işaretçiye işaret eden bir işaretçi olduğunu bildirmektedir:

```
int** q;
```

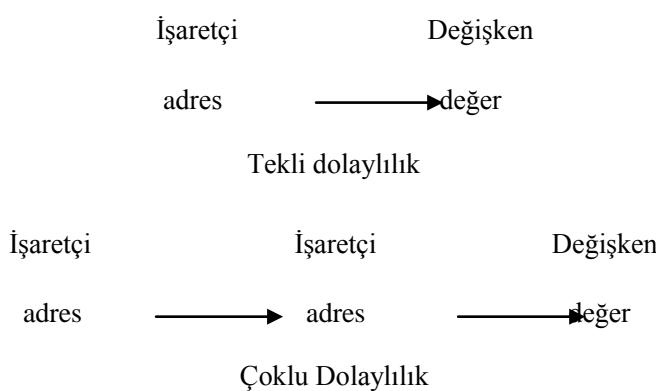
**q**'nun bir tamsayıya işaret eden bir işaretçi olmadığını kavramalısınız. Bunun yerine **q**, bir **int** işaretçiye işaret eden bir işaretçidir,

Bir işaretçiye işaret eden başka bir işaretçi tarafından, işaret edilen hedef değere erişmek için **\*** operatörünü aşağıdaki örnekteki gibi iki kez uygulamalısınız:

```
using System;
class MultipleIndirect {
    unsafe public static void Main() {
        int x; // bir int değeri tutar
        int* p; // bir int işaretcisi tutar
        int** q; // bir int işaretcisine işaret eden bir işaretci
                  // tutar
        x = 10;
        p = &x; // x'in adresini p'ye yerlestir
        q = &p; // p'nin adresini q'ya yerlestir
        Console.WriteLine(**q); // x'in değerini göster
    }
}
```

Cıktı, **x**'in değeri olan **10**'dur. Programda **p**, bir **int**'e işaret eden bir işaretçi olarak; **q** ise bir **int** işaretçiye işaret eden bir işaretçi olarak deklare edilmiştir.

Son bir husus: Çoklu dolaylılığı, söz gelişi bağlı listeler gibi işaretçi kullanan yüksek düzeyli veri yapılarıyla karıştırmamalısınız. Bunlar temel olarak farklı iki kavramdır.



**ŞEKİL 18.1:** *Tekli ve çoklu dolaylılık*

## İşaretçi Dizileri

İşaretçiler de diğer veri tipleri gibi diziye yerleştirilebilirler. 3 elemanlı bir **int** işaretçi dizisinin deklarasyonu şu şekildedir:

```
int * [] ptrs = new int * [3];
```

**var** adında bir **int** değişkeninin adresini işaretçi dizisinin üçüncü elemanına atamak için şöyle yazın:

```
ptrs[2] = &var;
```

**var**'ın değerini bulmak için şöyle yazın:

```
*ptrs[2]
```

## Ceşitli Anahtar Kelimeler

Kısım 1'i noktalamadan önce, başka hiç bir yerde bahsedilmemiş olan C#'ta tanımlı kalan birkaç anahtar kelime kısaca ele alınacaktır.

### sizeof

Arada sırada, C#'ın değer tiplerinden birinin büyüğünü byte cinsinden bilmek sizin için yararlı olabilir. Bu bilgiyi elde etmek için **sizeof** operatörünü kullanın. Bu operatör genel olarak şu şekildedir:

```
sizeof(tip)
```

Burada **tip**, büyüğünü elde etmekte olan tiptir. **sizeof** operatörü yalnızca emniyetsiz kod içinde kullanılabilir. Yani, **sizeof** operatörü esasen özel durumlarda, özellikle kontrol altında olan ve olmayan kodların bir harmanı ile çalışırken kullanılmak üzere tasarlanmıştır.

### lock

**lock** anahtar kelimesi, *çoklu kanallı* (*multiple threads*) programlarla çalışırken kullanılır. C#'ta bir program, iki ya da daha fazla *çalışma kanalı* (*threads of execution*) içerebilir. Böyle bir durum söz konusu olunca, programın parçaları çok görevli (*multitasked*) olur. Yani, programın parçaları birbirinden bağımsız olarak ve kavramsal olarak ifade ederse, eşzamanlı olarak çalışır. Bu durum özel bir problem tipinin ortaya çıkma olasılığını artırır: Bir kerede yalnızca tek bir kanal tarafından kullanılabilen bir kaynak, iki kanal tarafından kullanılmaya çalışılırsa ne olur? Bu problemin çözümü için, bir kerede sadece ve sadece tek bir kanal tarafından çalıştırılacak olan bir *kritik kod bölümü* (*critical code section*) tanımlayabilirsiniz. Bu, **lock** ile gerçekleştirilir. Genel olarak **lock**'un kullanımı şöyledir:

```
lock(nesne) {
    // kritik bölüm
}
```

Burada **nesne**, kilidi (lock) yakalamaya çalışan nesnedir. Eğer kanallardan biri önceden kritik bölüme girmişse, ikinci kanal ilk kanal çalmana kadar bekleyecektir. Kilit kabul edilince, kritik bölüm çalıştırılabilir. (Ek ayrıntılar için Bölüm 21'e bakın.)

## readonly

Bir sınıf içindeki bir alanı **readonly** olarak deklare ederek salt okunur bir alan oluşturabilirsiniz. Bir **readonly** alana ilk değer atanabilir, fakat bu değer daha sonra değiştirilemez. Böylece, **readonly** alanlar, program boyunca kullanılan sabitleri, söz gelişî dizi boyutlarını tanımlamak için iyi bir yoldur. Statik ve statik olmayan **readonly** alanlar tanımlamak mümkündür.

İşte, **readonly** bir alan oluşturan bir örnek:

```
// readonly kullanımını gösterir.

using System;

class MyClass {
    public static readonly int SIZE = 10;
}

class DemoReadOnly {
    public static void Main() {
        int[] nums = new int[MyClass.SIZE];

        for(int i = 0; i < MyClass.SIZE; i++)
            nums[i] = i;

        foreach(int i in nums)
            Console.Write(i + " ");

        // MyClass.SIZE = 100; // Hata!!! Degistirilemez
    }
}
```

Burada **MyClass.SIZE**'a ilk değer olarak **10** atanmaktadır. Bundan sonra, bu değer kullanılabilir ama değiştirilemez. Bunu kanıtlamak için son satırda açıklama simgelerini kaldırmayı ve sonra programı yeniden derlemeyi deneyin. Bir hatanın ortaya çıktığını göreceksiniz.

## stackalloc

**stackalloc** kullanarak yiğindan bellek alanı ayırabilirisiniz, **stackalloc** yalnızca yerel değişkenlere ilk değer atanırken kullanılabilir. Genel olarak kullanımı şu şekildedir:

```
tip *p = stackalloc tip[büyüklük]
```

Burada **p**, **tip** tipinde nesnelerin **büyüklük** adedini tutabilecek kadar büyük olan belleğin adresini alan bir işaretçidir. **stackalloc**, emniyetsiz kod kapsamında kullanılmalıdır.

Normal olarak, nesneler için gerekli bellek alanı, *katmandan* (heap) ayrıılır. Katman, boş bellek alanıdır. Bellek alanını yığından ayırmak bir istisnadır. Yığından bellek alanı ayrılan değişkenler, anlamsız verilerin toplanması işlemine tabi tutulmazlar. Bunun yerine, bu değişkenler, deklare edildikleri blok çalıştığı sürece varlıklarını sürdürürler. Blok terk edilince, ilgili bellek alanı serbest bırakılır. **stackalloc** kullanmanın bir avantajı olarak, anlamsız veri toplayıcısı tarafından taşınan belleği dert etmenize gerek yoktur.

İşte, **stackalloc** kullanan bir örnek:

```
// stackalloc kullanımını gösterir.
using System;

class UseStackAlloc {
    unsafe public static void Main() {
        int* ptrs = stackalloc int[3];

        ptrs[0] = 1;
        ptrs[1] = 2;
        ptrs[2] = 3;

        for(int i = 0; i < 3; i++)
            Console.WriteLine(ptrs[i]);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
1
2
3
```

## using İfadesi

Önceden ele alınan **using direktifine** ek olarak **using'in using ifadesi** olarak adlandırılan ikinci bir şekli daha mevcuttur. **using**'in kullanımı genel olarak şu şekildedir:

```
using (nesne) {
    // nesne'yi kullan
}

using (tip nesne = ilk-değer) {
    // nesne'yi kullan
}
```

Burada **nesne**, **using** bloğu içinde kullanılmakta olan nesnedir. İlk biçimde, söz konusu nesne **using** ifadesinin dışında deklere edilmektedir. İkinci biçimde, nesne **using** ifadesinin içinde deklare edilmektedir. Blok sona erdiğinde, **nesne** üzerinde **Dispose()** metodu (**System.IDisposable** arayüzü tarafından tanımlanmıştır) çağrılacaktır. **using** ifadesi yalnızca **System.IDisposable** arayüzüne gerçekleyen nesnelere uygulanabilir.

İşte, **using** ifadesinin her iki biçimini de kullanan bir örnek:

```

// using ifadesinin kullanımını gösterir.
using System;
using System.IO;

class UsingDemo {
    public static void Main() {
        StreamReader sr = new StreamReader("test.txt");

        // using ifadesi içinde nesne kullan.
        using(sr) {
            Console.WriteLine(sr.ReadLine());
            sr.Close();
        }
        // using ifadesi içinde StreamReader oluştur.
        using(StreamReader sr2 = new StreamReader("test.txt")) {
            Console.WriteLine(sr2.ReadLine());
            sr2.Close();
        }
    }
}

```

**StreamReader** sınıfı **IDisposable** arayüzüünü (**TextReader** temel sınıfı aracılığıyla) gerçekleştirmektedir. Böylece, **StreamReader** bir **using** ifadesinde kullanılabilir. (**IDisposable** ile ilgili açıklamalar için Bölüm 24'e bakın.)

## const ve volatile

**const** nitelikisi, değiştirilemeyen alanları ya da yerel değişkenleri deklare etmek için kullanılır. Bu değişkenlere ilk değerleri değişkenler deklare edilirken verilmelidir. Böylece, bir **const** değişken aslında bir sabittir. Örneğin,

```
const int i = 10;
```

bu ifade, **i** adında **10** değerine sahip bir **const** değişken oluşturur.

**volatile** nitelikisi, bir alanın değerinin program tarafından açıkça belirtilmeyen yöntemlerle değiştirilebileceğini derleyiciye bildirir. Örneğin, mevcut sistem zamanını tutan bir alan, işletim sistemi tarafından otomatik olarak değiştirilebilir. Bu durumda, söz konusu alanın içeriği açık bir atama ifadesi olmadan değiştirilir. Bir alanın haricen değiştirilmesi, önemli olabilir. Bunun nedeni şudur: Bir alan, bir atama ifadesinin sol tarafında yer almıyorsa söz konusu alanın içeriğinin değişmeden kalacağı varsayıma dayanarak, C# derleyicisinin belirli deyimleri otomatik olarak optimize etmesine izin verilmiştir. Ancak, mevcut kodun dışında kalan etmenler, söz gelisi ikinci bir kanal söz konusu alanın değerini değiştirirse, bu varsayımda yanlış olur. **volatile** kullanarak, bu alana ne zaman erişilse bu alanın değerinin elde etmesi gerektiğini derleyiciye bildiriyorsunuz.



# C# KÜTÜPHANESİNI KEŞFETMEK

Kısım II’de C# kütüphanesi incelenmektedir. Kısmı I’de açıklandığı gibi, C# tarafından kullanılan sınıf kütüphanesi .NET Framework kütüphanesidir. Bu nedenle, bu bölümde işlenen konular yalnızca C#’a uygulanmaz, aynı zamanda .NET Framework’e de bir bütün olarak uygulanır.

C# kütüphanesi isim uzayları şeklinde düzenlenmiştir. Kütüphanenin bir bölümünü kullanmak için normal olarak karşılık gelen isim uzayını `using` direktifini de dahil ederek programınızın içine aktarmalısınız. Elbette söz konusu ögenin ismini isim uzayıyla birlikte tam olarak nitelenmiş biçimde de kullanabilirsiniz, fakat bütün isim uzayını programa aktarmak genellikle çok daha kolaydır.

C# kütüphanesi büyktür ve bunun her parçasını incelemek elinizdeki kitabın kapsamının dışında kalmaktadır. (Konunun komple tanımı, bütün bir kitabı kolaylıkla doldururdu!) Bunun yerine, Kısmı II’de kütüphanenin çekirdek elemanları incelenmektedir. Bu elemanlar `System` isim uzayında bulunurlar. Ayrıca ele alınan konular, koleksiyon sınıfları, çok kanallılık (multithreading) ve ağ oluşturmada (networking).

I/O sınıfları Bölüm 14’te ele alınmıştır.

**19**

**ON DOKUZUNCU BÖLÜM**

---

# **SYSTEM İSİM UZAYI**

Bu bölümde **System** isim uzayı incelenmektedir. **System**, C# kütüphanesinde yer alan en üst düzeydeki isim uzayıdır. **System**, bir C# programı tarafından en yaygın olarak kullanılan ya da diğer bakımlardan .NET Framework'e bütünsel zannedilen sınıf, yapı, arayüz, delege ve numaralandırmaları doğrudan içerir. Bu nedenle **System**, C# kütüphanesinin çekirdeğini oluşturur.

**System** ayrıca **System.Net** gibi belirli alt sistemleri destekleyen birçok kümelenmiş isim uzayı da içerir. Bu alt sistemlerden birkaçı elinizdeki kitabın ileriki sayfalarında anlatılmıştır. Bu bölüm yalnızca **System**'in kendi üyelerine ayrılmıştır.

## System'in Üyeleri

Çok sayıdaki kural dışı durum sınıfına ek olarak **System**, aşağıdaki sınıfları da içerir:

Activator	AppDomain
AppDomainSetup	Array
AssemblyLoadEventArgs	Attribute
AttributeUsageAttribute	BitConverter
Buffer	CharEnumerator
CLSCompliantAttribute	Console
ContextBoundObject	ContextStaticAttribute
Convert	DBNull
Delegate	Enum
Environment	EventArgs
Exception	FlagsAttribute
GC	LoaderOptimizationAttribute
LocalDataStoreSlot	MarshalByRefObject
Math	MTAThreadAttribute
MulticastDelegate	NonSerializedAttribute
Object	ObsoleteAttribute
OperatingSystem	ParamArrayAttribute
Random	ResolveEventArgs
SerializableAttribute	STAThreadAttribute
String	ThreadStaticAttribute
TimeZone	Type
UnhandledExceptionEventArgs	Uri
UriBuilder	ValueType
Version	WeakReference

**System**'da aşağıdaki yapılar tanımlıdır:

ArgIterator	Boolean
Byte	Char
DateTime	Decimal
Double	Guid
Int16	Int32
Int64	IntPtr
RuntimeArgumentHandle	RuntimeFieldHandle
RuntimeMethodHandle	RuntimeTypeHandle
SByte	Single
TimeSpan	TypedReference
UInt16	UInt32

<b>UInt64</b>	<b>IntPtr</b>
<b>Void</b>	

**System**'da aşağıdaki arayüzler tanımlıdır:

<b>IAppDomainSetup</b>	<b>IAsyncResult</b>
<b>ICloneable</b>	<b>IComparable</b>
<b>IConvertible</b>	<b>ICustomFormatter</b>
<b>IDisposable</b>	<b>IFormatProvider</b>
<b>IFormattable</b>	<b>IServiceProvider</b>

**System**'da aşağıdaki delegeler tanımlıdır:

<b>AssemblyLoadEventHandler</b>	<b>AsyncCallback</b>
<b>CrossAppDomainDelegate</b>	<b>EventHandler</b>
<b>ResolveEventHandler</b>	<b>UnhandledExceptionEventHandler</b>

**System**'da bu numaralandırmalar tanımlıdır:

<b>AttributeTargets</b>	<b>DayOfWeek</b>
<b>Environment.SpecialFolder</b>	<b>LoaderOptimization</b>
<b>PlatformID</b>	<b>TypeCode</b>
<b>UriHostNameType</b>	<b>UriPartial</b>

Yukarıdaki tabloda görüldüğü gibi, **System** oldukça büyükür. Tek bir bölümde **System**'ın tüm bileşenlerini ayrıntılı olarak incelemek mümkün değildir. Üstelik, **System**'ın bazı üyeleri genel olarak .NET Framework'e uygulanabilir olsa da C# programcılar tarafından çoğunlukla kullanılmazlar. Ayrıca, **System**'ın birkaç sınıfı, söz gelişî **Type**, **Exception** ve **Attribute**, Kısım I'de ya da bu kitabın çeşitli yerlerinde anlatılmıştır. Son olarak, C# **string** tipini tanımlayan **System.String** büyük ve önemli bir konu olduğu için Bölüm 20'de biçimlendirme ile birlikle işlenmektedir. Bu nedenlerden ötürü, bu bölümde yalnızca C# programcılar tarafından en yaygın olarak kullanılan ve başka hiç bir yerde işlenmeyen üyeler incelenmektedir.

## Math Sınıfı

**Math**'te standart matematik işlemlerinden birkaççı tanımlanmaktadır. Söz gelişî, karekök, sinüs, kosinüs ve logaritmalar. **Math**'te tanımlı metodlar Tablo 19.I'de gösterilmiştir. Açıların tümü radyan cinsindendir. **Math**'te tanımlı tüm metodların **static** olduğuna dikkat edin. Bu nedenle, bir **Math** nesnesi kurmaya gerek yoktur; üstelik, açık **Math** yapılandırıcıları da sağlanmaz.

**Math**'te ayrıca aşağıdaki iki alan da tanımlanmıştır:

```
public const double E
public const double PI
```

**E**, genellikle *e* olarak bahsedilen doğal logaritma tabanının değeridir. **PI** ise pi sayısının değeridir.

`Math` sınıfı mühürlenmiştir; `sealed` niteliğine sahiptir. Yani, kalıtım yoluyla başka bir sınıf'a aktarılamaz.

**TABLO 19.1: Math Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public static double Abs(double v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static float Abs(float v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static int Abs(int v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static short Abs(short v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static long Abs(long v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static sbyte Abs(sbyte v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static double Acos(double v)</code>	$v$ 'nin mutlak değerini döndürür.
<code>public static double Asin(double v)</code>	$v$ 'nin ters kosinüs değerini döndürür. $v$ 'nin değeri $-1$ ile $1$ arasında olmalıdır.
<code>public static double Atan(double v)</code>	$v$ 'nin ters sinüs değerini döndürür. $v$ 'nin değeri $-1$ ile $1$ arasında olmalıdır.
<code>public static double Atan2(double y, double x)</code>	$v/x$ 'in ters tanjant değerini döndürür.
<code>public static double Ceiling(double v)</code>	$v$ 'den küçük olmayan (kayan noktalı değer olarak simgelenen) en küçük tamsayıyı döndürür. Örneğin, $1.02$ verildiğinde <code>Ceiling()</code> $2.0$ döndürür. $-1.02$ verildiğinde <code>Ceiling()</code> $-1$ döndürür.
<code>public static double Cos(double v)</code>	$v$ 'nin kosinüsünü döndürür.
<code>public static double Cosh(double v)</code>	$v$ 'nin hiperbolik kosinüsünü döndürür.
<code>public static double Exp(double v)</code>	Doğal logaritma tabanı $e$ 'nin $v$ 'nci kuvvetini döndürür.
<code>public static double Floor(double v)</code>	$v$ 'den büyük olmayan (kayan noktalı değer olarak simgelenen) en büyük tamsayıyı döndürür. Örneğin, $1.02$ verildiğinde <code>Floor()</code> $1.0$ döndürür. $-1.02$ verildiğinde <code>Floor()</code> $-2$ döndürür.
<code>public static double IEEERemainder(double dividend, double divisor)</code>	<code>dividend / divisor</code> işleminin kalanını döndürür.
<code>public static double Log(double v)</code>	$v$ için doğal logaritmayı döndürür.
<code>public static double Log(double v, double base)</code>	Taban olarak <code>base</code> 'i kullanarak $v$ için doğal logaritmayı döndürür.

<code>public static double Log10(double v)</code>	Taban olarak 10'u kullanarak <code>v</code> için doğal logaritmayı döndürür.
<code>public static double Max(double v1, double v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static float Max(float v1, float v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static decimal Max(decimal v1, decimal v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static int Max(int v1, int v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static short Max(short v1, short v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static long Max(long v1, long v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static uint Max(uint v1, uint v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static ushort Max(ushort v1, ushort v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static ulong Max(ulong v1, ulong v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static byte Max(byte v1, byte v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static sbyte Max(sbyte v1, sbyte v2)</code>	<code>v1</code> ve <code>v2</code> 'nin büyüğünü döndürür.
<code>public static double Min(double v1, double v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static float Min(float v1, float v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static decimal Min(decimal v1, decimal v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static int Min(int v1, int v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static short Min(short v1, short v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static long Min(long v1, long v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static uint Min(uint v1, uint v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static ushort Min(ushort v1, ushort v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static ulong Min(ulong v1, ulong v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.
<code>public static byte Min(byte v1, byte v2)</code>	<code>v1</code> ve <code>v2</code> 'nin küçüğünü döndürür.

<code>public static sbyte Min(sbyte v1, sbyte v2)</code>	$v1$ ve $v2$ 'nin küçüğünü döndürür.
<code>public static double Pow(double base, double exp)</code>	$base$ in $exp$ kuvvetini döndürür ( $base^{exp}$ ).
<code>public static double Round(double v)</code>	En yakın tamsayıya yuvarlanmış $v$ değerini döndürür.
<code>public static decimal Round(decimal v)</code>	En yakın tamsayıya yuvarlanmış $v$ değerini döndürür.
<code>public static double Round(double v, int frac)</code>	$frac$ ile belirtilen ondalık basamak sayısına yuvarlanmış $v$ değerini döndürür.
<code>public static decimal Round(decimal v, int frac)</code>	$frac$ ile belirtilen ondalık basamak sayısına yuvarlanmış $v$ değerini döndürür.
<code>public static int Sign(double v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(float v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(decimal v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(int v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(short v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(long v)</code>	$v$ sıfırdan küçükse <b>-1</b> , $v$ sıfır ise <b>0</b> ve $v$ sıfırdan büyükse <b>1</b> döndürür.
<code>public static int Sign(sbyte v)</code>	$v$ 'nin sinüsünü döndürür.
<code>public static double Sin(double v)</code>	$v$ 'nin hiperbolik sinüsünü döndürür.
<code>public static double Sinh(double v)</code>	$v$ 'nin karekökünü döndürür.
<code>public static double Sqrt(double v)</code>	$v$ 'nin tanjantını döndürür.
<code>public static double Tan(double v)</code>	$v$ 'nin hiperbolik tanjantını döndürür.
<code>public static double Tanh(double v)</code>	

İşte, Pisagor teoremini uygulamak için `Sqrt()` kullanan bir örnek. Bu program, bir dik üçgenin dik kenarlarının uzunluklarını verildiğinde hipotenüsün uzunluğunu hesaplar.

```
// Pisagor teoremini uygular.

using System;

class Pythagorean {
    public static void Main() {
        double s1;
        double s2;
        double hypot;
        string str;
```

```

Console.WriteLine("Enter length of first side: ");
str = Console.ReadLine();
s1 = Double.Parse(str);

Console.WriteLine("Enter length of second side: ");
str = Console.ReadLine();
s2 = Double.Parse(str);

hypot = Math.Sqrt(s1 * s1 + s2 * s2);

Console.WriteLine("Hypotenuse is " + hypot);
}
}

```

Programın örnek çalışması aşağıda gösterilmiştir:

```

Enter length of first side: 3
Enter length of second side: 4
Hypotenuse is 5

```

Bir sonraki örnek, yıllık faiz oranı ve faizin uygulanacağı yıl sayısı verildiğinde, gelecekte istenilen değere ulaşması için başlangıç yatırımının ne kadar olması gerektiğini hesaplamak amacıyla **Pow()** metodunu kullanmaktadır. Başlangıç yatırımını hesaplayan formül aşağıda gösterilmiştir

$$\text{Başlangıçtaki Yatırım} = \text{Gelecekteki Değer} / (1 + \text{FaizOranı})^{\text{YılSayısı}}$$

**Pow()**, **double** tipinde argümanlar gerektirdiği için faiz oranı ve yıl sayısı **double** değerler içinde tutulmaktadır. Gelecekteki değer ve başlangıçtaki yatırım için **decimal** tipi kullanılmaktadır.

```

/* Yillik faiz orani ve faizin uygulanacagi yil sayisi
verildiginde, bilinen bir gelecek degere ulasmak icin
gerekli baslangic yatirimini hesaplar. */

using System;

class InitialInvestment {
    public static void Main() {
        decimal InitInvest; // baslangictaki yatirim
        decimal FutVal; // gelecekteki deger

        double NumYears; // yil sayisi
        double IntRate; // ondalik degerde, yillik getiri orani

        string str;

        Console.Write("Enter future value: ");
        str = Console.ReadLine();
        try {
            FutVal = Decimal.Parse(str);
        } catch(FormatException exc) {
            Console.WriteLine(exc.Message);
        }
    }
}

```

```

        return;
    }

    Console.WriteLine("Enter interest rate (such as 0.085): ");
    str = Console.ReadLine();
    try {
        IntRate = Double.Parse(str);
    } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        return;
    }

    Console.WriteLine("Enter number of years: ");
    str = Console.ReadLine();
    try {
        NumYears = Double.Parse(str);
    } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        return;
    }

    InitInvest = FutVal / (decimal) Math.Pow(IntRate + 1.0,
                                         NumYears);

    Console.WriteLine("Initial investment required: {0:C}",
                      InitInvest);
}
}

```

Programın örnek çalışması aşağıda gösterilmiştir:

```

Enter future value: 10000
Enter interest rate (such as 0.085): 0.07
Enter number of years: 10
Initial investment required: $5,083.49

```

## Değer Tipindeki Yapılar

Değer tipindeki yapılar Bölüm 14'te tanıtılmıştı. Bunlar, insanlar tarafından okunabilen nümerik değerleri tutan karakter katarlarını karşılık gelen ikili değerlere dönüştürmek için kullanılmışlardır. Burada ayrıntılı olarak incelenmektedirler.

Değer tipindeki yapılar, C#'ın değer tiplerinin altında yatan yapılardır. Bu yapılar tarafından tanımlanan üyeleri kullanarak, değer tipleriyle bağlantılı işlemleri gerçekleştirebilirsiniz.

.NET yapı adları ve bunların C#'taki anahtar kelime olarak eşdeğerleri aşağıda gösterilmiştir:

<b>.NET Yapı Adı</b>	<b>C# Adı</b>
<b>Boolean</b>	<b>bool</b>

<b>Char</b>	<b>char</b>
<b>Decimal</b>	<b>decimal</b>
<b>Double</b>	<b>double</b>
<b>Single</b>	<b>float</b>
<b>Int16</b>	<b>short</b>
<b>Int32</b>	<b>int</b>
<b>Int64</b>	<b>long</b>
<b>UInt16</b>	<b>ushort</b>
<b>UInt32</b>	<b>uint</b>
<b>UInt64</b>	<b>ulong</b>
<b>Byte</b>	<b>byte</b>
<b>SByte</b>	<b>sbyte</b>

Bu yapıların her biri aşağıdaki konulurda incelenmektedir.

**NOT** Değer tipindeki yakılar tarafından tanımlanan bazı metodlar **IFormatProvider** ya da **NumberStyles** tipinde bir parametre alırlar. **IFormatProvider** bu bölümün ileriki sayfalarında kısaca incelenecktir. **NumberStyles** ise **System.Globalization** isim uzayında bulunan bir numaralandırmadır. Biçimlendirme konusu Bölüm 20'de ele alınmaktadır.

## Tamsayı Tipinde Yapılar

Tamsayı tipinde yapılar şunlardır:

<b>Byte</b>	<b>SByte</b>	<b>Int16</b>	<b>UInt16</b>
<b>Int32</b>	<b>UInt32</b>	<b>Int64</b>	<b>UInt64</b>

Bu yapıların her biri aynı metodları içerirler. Bu metodlar Tablo 19.2'de gösterilmiştir. Bir yapıdan diğerine fark eden tek şey **Parse()**'ın dönüş tipidir. Her yapı için **Parse()**, yapı tarafından simgelenen tipin değerini döndürür. Örneğin, **Int32** için **Parse()** bir **int** değer döndürür. **UInt16** için **Parse()** bir **ushort** değer döndürür. **Parse()**'ı tanıtan bir örnek için Bölüm 14'e bakın.

Tablo 19.2'de gösterilen metodlara ek olarak tamsayı tipindeki yapılar aşağıdaki **const** alanları da içerirler:

<b>.MaxValue</b>
<b>.MinValue</b>

Her yapı için bu alanlar, söz konusu tamsayı tipinin tutabileceği en büyük ve en küçük değeri içerirler.

Tamsayı tipindeki tüm yapılar şu arayüzleri uygularlar: **IComparable**, **IConvertible** ve **IFormattable**.

**TABLO 19.2: Tamsayı Tipindeki Yapılar Tarafından Desteklenen Metotlar**

Metot	Anlamı
<code>public int CompareTo(object v)</code>	Metodu çağırılan nesnenin nümerik değerini <code>v</code> 'nin değeri ile karşılaştırır. Değerler eşitse sıfır döndürür, Metodu çağırılan nesne daha küçük bir değere sahipse negatif bir değer döndürür. Metodu çağırılan nesne daha büyük bir değere sahipse pozitif bir değer döndürür.
<code>public override bool Equals(object v)</code>	Metodu çağırılan nesnenin değeri <code>v</code> 'nin değerine eşitse <code>true</code> döndürür.
<code>public override int GetHashCode()</code>	Metodu çağırılan nesnenin “hash” kodunu döndürür.
<code>public TypeCode GetTypeCode()</code>	Eşdeğer değer tipleri için <code>TypeCode</code> numaralandırmasını döndürür. Örneğin, <code>Int32</code> için tip kodu <code>TypeCode.Int32</code> 'dir.
<code>public static retType Parse(string str)</code>	<code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, yapı tipi tarafından tanımlanan şekliyle bir nümerik değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static retType Parse(string str, IFormatProvider fmtpvdr)</code>	<code>fmtpvdr</code> ile sağlanan kültürel bilgiyi kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, yapı tipi tarafından tanımlanan şekliyle bir nümerik değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static retType Parse(string str, NumberStyles styles)</code>	<code>styles</code> ile sağlanan stil bilgisini kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, yapı tipi tarafından tanımlanan şekliyle bir nümerik değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static retType Parse(string str, NumberStyles styles, IFormatProvider fmtpvdr)</code>	<code>styles</code> ile sağlanan stil bilgisini ve <code>fmtpvdr</code> ile sağlanan kültürel bilgiyi kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, yapı tipi tarafından tanımlanan şekliyle bir nümerik değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public override string ToString()</code>	Metodu çağırılan nesnenin değerinin karakter katarı olarak gösterimini döndürür.

<code>public string ToString(string format)</code>	Metodu çağırın nesnenin değerinin karakter katarı olarak gösterimini, <code>format</code> içinde aktarılan biçim karakter katarı tarafından belirtilen şekliyle döndürür.
<code>public string ToString(IFormatProvider fmtvpdr)</code>	<code>fmtvpdr</code> içinde belirtilen kültürel bilgiyi kullanarak, metodu çağırın nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public string ToString(string format, IFormatProvider fmtvpdr)</code>	<code>fmtvpdr</code> içinde belirtilen kültürel bilgiyi ve <code>format</code> ile belirtilen biçim kullanarak, metodu çağırın nesnenin değerinin karakter katarı olarak gösterimini döndürür.

## Kayan Nokta Tipinde Yapılar

Kayan nokta tipinde iki yapı mevcuttur: `Double` ve `Single`. `Single`, `float`'ı simgeler. Bu yapının metodları Tablo 19.3'te ve alanları da Tablo 19.4'te gösterilmiştir. `Double`, `double`'ı simgeler. Bu yapının metodları Tablo 19.5'te ve alanları Tablo 19.6'da gösterilmiştir, Tamsayı tipindeki yapılarda olduğu gibi, `Parse()`'a ya da `ToString()`'e çağrıda bulunurken kültürel bilgileri ve biçim bilgilerini belirtebilirsiniz.

Kayan nokta tipindeki yapılar şu arayüzleri uygularlar: `IComparable`, `IConvertible` ve `IFormattable`.

**TABLO 19.3: Single Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public int CompareTo(object v)</code>	Metodu çağırın nesnenin nümerik değerini <code>v</code> 'nin değeri ile karşılaştırır. Değerler eşitse sıfır döndürür, Metodu çağırın nesne daha küçük bir değere sahipse negatif bir değer döndürür. Metodu çağırın nesne daha büyük bir değere sahipse pozitif bir değer döndürür.
<code>public override bool Equals(object v)</code>	Metodu çağırın nesnenin değeri <code>v</code> 'nin değerine eşitse <code>true</code> döndürür.
<code>public override int GetHashCode()</code>	Metodu çağırın nesnenin "hash" kodunu döndürür.
<code>public TypeCode GetTypeCode()</code>	<code>Single</code> için bir <code>TypeCode</code> numaralandırma değeri döndürür ki bu, <code>TypeCode.Single</code> 'dır.
<code>public static bool IsInfinity(float v)</code>	<code>v</code> , sonsuz (pozitif ya da negatif) simgeliyorsa <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsNaN(float v)</code>	<code>v</code> , bir sayı değilse, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.

<code>public static bool IsPositiveInfinity(float v)</code>	<code>v</code> , pozitif sonsuzu simgeliyorsa, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsNegativeInfinity(float v)</code>	<code>v</code> , negatif sonsuzu simgeliyorsa, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static float Parse(string str)</code>	<code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>float</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static float Parse(string str, IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> ile sağlanan kültürel bilgiyi kullanarak, <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>float</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static float Parse(string str, NumberStyles styles)</code>	<code>styles</code> ile sağlanan stil bilgisini kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>float</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static float Parse(string str, NumberStyles styles, IFormatProvider fmpvdr)</code>	<code>styles</code> ile sağlanan stil bilgisini ve <code>fmpvdr</code> ile sağlanan kültürel bilgiyi kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>float</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public override string ToString()</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public string ToString(string format)</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini, <code>format</code> içinde aktarılan biçim karakter katarı tarafından belirtilen şekilde döndürür.
<code>public string ToString(IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> içinde belirtilen kültürel bilgiyi kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public string ToString(string format, IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> içinde belirtilen kültürel bilgiyi ve <code>format</code> ile belirtilen biçim kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.

TABLO 19.4: Single Tarafından Tanımlanan Alanlar

**Alan**

```
public const float Epsilon
public const float MaxValue
public const float MinValue
```

**Anlamı**

Sıfırdan farklı en küçük pozitif değer.  
 Bir `float`'ın tutabileceği en büyük değer.  
 Bir `float`'ın tutabileceği en küçük değer.

<code>public const float NaN</code>	Sayı olmayan bir değer.
<code>public const float NegativeInfinity</code>	Negatif sonsuzu simgeleyen bir değer.
<code>public const float PositiveInfinity</code>	Pozitif sonsuzu simgeleyen bir değer.

TABLO 19.5: Double Tarafından Tanımlanan Metotlar

Metot	Anlamı
<code>public int CompareTo(object v)</code>	Metodu çağrıran nesnenin nümerik değerini <code>v</code> 'nin değeri ile karşılaştırır. Değerler eşitse sıfır döndürür, Metodu çağrıran nesne daha küçük bir değere sahipse negatif bir değer döndürür. Metodu çağrıran nesne daha büyük bir değere sahipse pozitif bir değer döndürür.
<code>public override bool Equals(object v)</code>	Metodu çağrıran nesnenin değeri <code>v</code> 'nin değerine eşitse <code>true</code> döndürür.
<code>public override int GetHashCode()</code>	Metodu çağrıran nesnenin "hash" kodunu döndürür.
<code>public TypeCode GetTypeCode()</code>	<code>Double</code> için bir <code>TypeCode</code> numaralandırma değeri döndürür ki bu, <code>TypeCode.Double</code> 'dır.
<code>public static bool IsInfinity(double v)</code>	<code>v</code> , sonsuz (pozitif ya da negatif) simgeliyorsa <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsNaN(double v)</code>	<code>v</code> , bir sayı değilse, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsPositiveInfinity(double v)</code>	<code>v</code> , pozitif sonsuzu simgeliyorsa, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsNegativeInfinity(double v)</code>	<code>v</code> , negatif sonsuzu simgeliyorsa, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static double Parse(string str)</code>	<code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>double</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static double Parse(string str, IFormatProvider fmtpvdr)</code>	<code>fmtpvdr</code> ile sağlanan kültürel bilgiyi kullanarak, <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>double</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static double Parse(string str, NumberStyles styles)</code>	<code>styles</code> ile sağlanan stil bilgisini kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>double</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.

<code>public static double Parse(string str, NumberStyles styles, IFormatProvider fmpvdr)</code>	<code>styles</code> ile sağlanan stil bilgisini ve <code>fmpvdr</code> ile sağlanan kültürel bilgiyi kullanarak <code>str</code> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <code>double</code> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public override string ToString()</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public string ToString(string format)</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini, <code>format</code> içinde aktarılan biçim karakter katarı tarafından belirtilen şekliyle döndürür.
<code>public string ToString(IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> içinde belirtilen kültürel bilgiyi kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public string ToString(string format, IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> içinde belirtilen kültürel bilgiyi ve <code>format</code> ile belirtilen biçimini kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.

**TABLO 19.6: Double Tarafından Tanımlanan Alanlar**

Alan	Anlamı
<code>public const double Epsilon</code>	Sıfırdan farklı en küçük pozitif değer.
<code>public const double MaxValue</code>	Bir <code>double</code> 'ın tutabileceği en büyük değer.
<code>public const double MinValue</code>	Bir <code>double</code> 'ın tutabileceği en küçük değer.
<code>public const double NaN</code>	Sayı olmayan bir değer.
<code>public const double NegativeInfinity</code>	Negatif sonsuz simgeleyen bir değer.
<code>public const double PositiveInfinity</code>	Pozitif sonsuz simgeleyen bir değer.

## Decimal

`Decimal` yapısı, tamsayı ve kayan noktalı akrabalarına kıyasla biraz daha karmaşıktır. Bu yapı, birçok yapılandırıcı, alan, metot ve `decimal`'i C# tarafından desteklenen diğer nümerik tiplere entegre etmeye yardımcı olan operatörler içerir. Örneğin, metodların birkaçı `decimal` ve diğer nümerik tipler arasında dönüşüm sağlarlar.

`Decimal`, sekiz tane `public` yapılandırıcı sunmaktadır. Aşağıdaki altı tanesi en sık kullanılanlardır:

```
public Decimal(int v)
public Decimal(uint v)
public Decimal(long v)
public Decimal(ulong v)
```

```
public Decimal(float v)
public Decimal(double v)
```

Bunların her biri belirtilen değerden bir **Decimal** oluşturur.

Bir **Decimal**'i, aşağıdaki yapılandırıcının kullanımıyla **Decimal**'i oluşturan parçaları belirterek de oluşturabilirsiniz:

```
public Decimal(int alt, int orta, int üst, bool işaretBayrağı,
               byte ölçekFaktörü)
```

Bir **decimal** değer üç parçadan oluşur. İlk parça, **96** bit uzunluğunda bir tamsayıdır. İlkinci parça, işaret bayrağıdır. Üçüncü parça ise ölçekleme faktörüdür. **96** bit tamsayı, **alt**, **orta** ve **üst** aracılığıyla **32** bit parçalar halinde aktarılır. İşaret, **İşaretBayrağı** aracılığıyla aktarılır. **İşaretBayrağı**, pozitif bir sayı için **false**; negatif bir sayı için **true** değerindedir. Ölçekleme faktörü **ölçekFaktörü** içinde aktarılır. Bu değer **0** ile **28** arasında olmalıdır. Bu faktör, sayının ondalık bileşenini ortaya çıkarmak için sayıyı bölen **10**'nun kuvvetini (yani, **10<sup>ölçekFaktörü</sup>**) belirtir.

Her bileşeni ayrı ayrı aktarmak yerine, bir **Decimal**'i oluşturan parçaları, aşağıda gösterilen yapılandırıcıyı kullanarak, bir tamsayı dizisinde belirtebilirsiniz:

```
public Decimal(int[] parts)
```

**parts** içindeki ilk üç **int**, **96** bit tamsayı değeri içerir. **parts[3]**'deki **31**'nci bit işaret bayrağını (pozitif sayılar için **0**, negatif sayılar için **1**) içerir ve **16** ila **23**'ncü bitler de ölçekleme faktörünü içerirler.

**Decimal** şu arayüzler'i uygular: **IComparable**, **IConvertible** ve **IFomattable**.

İşte, bir **decimal** değeri elle oluşturan bir örnek:

```
// Elle ondalik bir sayı olustur.

using System;

class CreateDec {
    public static void Main() {
        decimal d = new decimal(12345, 0, 0, false, 2);

        Console.WriteLine(d);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
123.45
```

Bu örnekte, **96** bit tamsayının değeri **12345**'tir. Bunun işaretti pozitiftir ye iki ondalık kısım içerir.

**Decimal** tarafından tanımlanan metodlar Tablo 19.7'de gösterilmiştir. **Decimal** tarafından tanımlanan alanlar ise Tablo 19.8'de gösterilmiştir. **Decimal**'de ayrıca **decimal** değerlerin diğer nümerik tiplerle birlikte deyimlerde kullanılmasına olanak tanıyan çok sayıda operatör ve dönüşümler de tanımlanmaktadır. Deyimlerde ve atamalarda **decimal** kullanımını düzenleyen kurallar Bölüm 3'te anlatılmıştır.

**TABLO 19.7: Decimal Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public static decimal Add(decimal v1, decimal v2)</code>	$v'$ nin mutlak değerini döndürür.
<code>public static int CompareTo(decimal v1, decimal v2)</code>	$v1$ 'in nümerik değerini $v2$ 'ninkiyle karşılaştırır. Değerler eşitse sıfır döndürür. $v1$ , $v2$ 'den küçükse negatif bir değer döndürür. $v1$ , $v2$ 'den büyükse pozitif bir değer döndürür.
<code>public int CompareTo(object v)</code>	Metodu çağrıran nesnenin değerini $v'$ ninkiyle karşılaştırır. Değerler eşitse sıfır döndürür. Metodu çağrıran nesnenin değeri daha küçükse negatif bir sayı döndürür. Metodu çağrıran nesnenin değeri daha büyükse pozitif bir sayı döndürür.
<code>public static decimal Divide(decimal v1, decimal v2)</code>	$v1 / v2$ sonucunu döndürür.
<code>public override bool Equals(object v)</code>	Metodu çağrıran nesnenin değeri $v'$ nin değerine eşitse <b>true</b> döndürür.
<code>public override bool Equals(decimal v1, decimal v2)</code>	$v1, v2$ 'ye eşitse <b>true</b> döndürür.
<code>public static decimal Floor(decimal v)</code>	$v'$ den büyük olmayan (ondalık değer olarak simgelenen) en büyük tamsayıyı döndürür. Örneğin, <b>1.02</b> verildiğinde <b>Floor()</b> <b>1.0</b> döndürür. <b>-1.02</b> verildiğinde <b>Floor()</b> <b>-2</b> döndürür.
<code>public static decimal FromOACurrency(long v)</code>	$v'$ nin içindeki OLE Otomasyon değerini karşılık gelen <b>decimal</b> eşdeğerine dönüştürür ve sonucu döndürür.
<code>public static int[] GetBits(decimal v)</code>	$v'$ nin ikili gösterimini döndürür ve bu değeri bir <b>int</b> dizisi içinde döndürür. Bu dizinin nasıl organize edildiği metin içinde anlatılmıştır.
<code>public override int GetHashCode()</code>	Metodu çağrıran nesne için "hash" kodu döndürür.
<code>public TypeCode GetTypeCode()</code>	Decimal için bir <b>TypeCode</b> numaralandırma değeri döndürür ki bu, <b>TypeCode.Decimal</b> 'dır.

<code>public static decimal Multiply(decimal v1 * v2 sonucunu döndürür. v1, decimal v2)</code>	
<code>public static decimal Negate (decimal v)</code>	- <b>v</b> değerini döndürür.
<code>public static decimal Parse(string str)</code>	<b>str</b> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <b>decimal</b> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static decimal Parse(string str, IFormatProvider fmtpvdr)</code>	<b>fmtpvdr</b> ile sağlanan kültürel bilgiyi kullanarak, <b>str</b> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <b>decimal</b> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static decimal Parse(string str, NumberStyles styles, IFormatProvider fmtpvdr)</code>	<b>styles</b> ile sağlanan stil bilgisini ve <b>fmtpvdr</b> ile sağlanan kültürel bilgiyi kullanarak <b>str</b> içindeki nümerik karakter katarının ikili eşdeğerini döndürür. Eğer karakter katarı, bir <b>decimal</b> değeri temsil etmiyorsa, bir kural dışı durum fırlatılır.
<code>public static decimal Remainder(decimal v1, decimal v2)</code>	<b>v1 / v2</b> tamsayı bölme işleminin kalanını döndürür.
<code>public static decimal Round(decimal v, int decPlaces)</code>	<b>decPlaces</b> ile belirtilen ondalık basamak sayısına yuvarlanmış <b>v</b> değerini döndürür. <b>decPlaces</b> , 0 ile 28 arasında olmalıdır.
<code>public static decimal Subtract(decimal v1, decimal v2)</code>	<b>v1 - v2</b> sonucunu döndürür.
<code>public static byte ToByte(decimal v)</code>	<b>v</b> 'nin <b>byte</b> eşdeğerini döndürür. Ondalık kısım kesilir. <b>v</b> , <b>byte</b> 'ın menzili (tanımlı olduğu aralık) içinde değilse bir <b>OverflowException</b> meydana gelir.
<code>public static double ToDouble (decimal v)</code>	<b>v</b> 'nin <b>double</b> eşdeğerini döndürür. Duyarlılık (precision) kaybı meydana gelebilir, çünkü <b>double</b> , <b>decimal</b> 'den daha az anlamlı (significant) basamağa sahiptir.
<code>public static shortToInt16(decimal v)</code>	<b>v</b> 'nin <b>short</b> eşdeğerini döndürür. Ondalık kısım kesilir. <b>v</b> , <b>short</b> 'un menzili (tanımlı olduğu aralık) içinde değilse bir <b>OverflowException</b> meydana gelir.
<code>public static intToInt32(decimal v)</code>	<b>v</b> 'nin <b>int</b> eşdeğerini döndürür. Ondalık kısım kesilir. <b>v</b> , <b>int</b> 'in menzili (tanımlı olduğu aralık) içinde değilse bir <b>OverflowException</b> meydana gelir.

<code>public static longToInt64(decimal v)</code>	<code>v</code> 'nin <code>long</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>long</code> 'un menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public static longToOACurrency(decimal v)</code>	<code>v</code> 'yi karşılık gelen OLE Otomasyon değerine dönüştürür ve sonucu döndürür.
<code>public static sbyteToSByte(decimal v)</code>	<code>v</code> 'nin <code>sbyte</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>sbyte</code> 'in menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public static floatToSingle(decimal v)</code>	<code>v</code> 'nin <code>float</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>float</code> 'un menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public override stringToString()</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public stringToString(string format)</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini, <code>format</code> içinde aktarılan biçim karakter katarı tarafından belirtilen şekilde döndürür.
<code>public stringToString(IFormatProvider fmtvpdr)</code>	<code>fmtvpdr</code> içinde belirtilen kültürel bilgiyi kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public stringToString(string format, IFormatProvider fmtvpdr)</code>	<code>fmtvpdr</code> içinde belirtilen kültürel bilgiyi ve <code>format</code> ile belirtilen biçim kullanarak, metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>public static ushortToUInt16(decimal v)</code>	<code>v</code> 'nin <code>ushort</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>ushort</code> 'un menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public static uintToUInt32(decimal v)</code>	<code>v</code> 'nin <code>uint</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>uint</code> 'in menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public static ulongToUInt64(decimal v)</code>	<code>v</code> 'nin <code>ulong</code> eşdeğerini döndürür. Ondalık kısım kesilir. <code>v</code> , <code>ulong</code> 'un menzili (tanımlı olduğu aralık) içinde değilse bir <code>OverflowException</code> meydana gelir.
<code>public static decimalTruncate(decimal v)</code>	<code>v</code> 'nin tamsayı kısmını döndürür. Böylece, ondalık basamaklar kesilmiş olur.

TABLO 19.8: Decimal Tarafından Desteklenen Alanlar

Alan	Anlamı
<code>public static readonly decimal.MaxValue</code>	Bir <code>decimal</code> 'in tutabileceği en büyük değer.
<code>public static readonly decimal.MinusOne</code>	<code>-1</code> 'in <code>decimal</code> olarak gösterimi
<code>public static readonly decimal.MinValue</code>	Bir <code>decimal</code> 'in tutabileceği en küçük değer.
<code>public static readonly decimal.One</code>	<code>1</code> 'in <code>decimal</code> olarak gösterimi
<code>public static readonly decimal.Zero</code>	<code>0</code> 'ın <code>decimal</code> olarak gösterimi

## Char

Değer tipinde yapılardan günlük bazda belki de en kullanışlı olanı `Char`'dır, çünkü bu yapı, karakterleri işleyip kategorize etmenize imkan veren çok sayıda metod sunmaktadır. Örneğin, `ToUpper()`'ı çağırarak bir küçük harfi büyük harfe çevirebilirsiniz. `IsDigit()`'ı çağırarak bir karakterin rakam olup olmadığını belirleyebilirsiniz.

`Char` tarafından tanımlanan metodlar Tablo 19.9'da gösterilmiştir. `Char`'da aşağıdaki alanlar tanımlıdır:

```
public const char.MaxValue
public const char.MinValue
```

Bunlar, bir `char` değişkenin tutabileceği en büyük ve en küçük değerleri simgelerler. `Char` şu arayüzleri uygular: `IComparable` ve `IConvertible`.

TABLO 19.9: Char Tarafından Tanımlanan Metotlar

Metot	Anlamı
<code>public int CompareTo(object v)</code>	Metodu çağrıran nesnenin değerini <code>v</code> 'ninkiyle karşılaştırır. Değerler eşitse sıfır döndürür, Metodu çağrıran nesnenin değeri daha küçükse negatif bir sayı döndürür. Metodu çağrıran nesnenin değeri daha büyükse pozitif bir sayı döndürür.
<code>public override bool Equals(object v)</code>	Metodu çağrıran nesnenin değeri <code>v</code> 'nin değerine eşitse <code>true</code> döndürür.
<code>public override int GetHashCode()</code>	Metodu çağrıran nesne için "hash" kodu döndürür.
<code>public static double GetNumericValue(char ch)</code>	<code>ch</code> bir rakam ise <code>ch</code> 'in nümerik değerini döndürür. Aksi halde, <code>-1</code> döndürür.
<code>public static double GetNumericValue(string str, int idx)</code>	<code>str[idx]</code> bir rakam ise bu karakterin nümerik değerini döndürür. Aksi halde, <code>-1</code> döndürür.

```

public TypeCode GetTypeCode()

public static UnicodeCategory
GetUnicodeCategory(char ch)

public static UnicodeCategory
GetUnicodeCategory(string str,
int idx)

public static bool IsControl
(char ch)

public static bool IsControl(string
str, int idx)

public static bool IsDigit(char ch)

public static bool IsDigit(string
str, int idx)

public static bool IsLetter(char ch)

public static bool IsLetter(string
str, int idx)

public static bool
IsLetterOrDigit(char ch)

public static bool IsLetterOrDigit
(string str, int idx)

public static bool IsLower(char ch)

public static bool IsLower(string
str, int idx)

public static bool IsNumber(char ch)

public static bool IsNumber(string
str, int idx)

public static bool
IsPunctuation(char ch)

public static bool IsPunctuation
(string str, int idx)

```

**Char** için bir **TypeCode** numaralandırma değeri döndürür. Bu değer, **TypeCode.Char**'dır.

**ch** için bir **UnicodeCategory** numaralandırma değeri döndürür. **UnicodeCategory**, Unicode karakterleri kategorize eden **System.Globalization** tarafından tanımlanan bir numaralandırmadır.

**str[idx]** için bir **UnicodeCategory** numaralandırma değeri döndürür. **UnicodeCategory**, Unicode karakterleri kategorize eden **System.Globalization** tarafından tanımlanan bir numaralandırmadır.

**ch** bir kontrol karakteri ise **true** döndürür. Aksi halde **false** döndürür.

**str[idx]** bir kontrol karakteri ise **true** döndürür. Aksi halde **false** döndürür.

**ch** bir rakam ise **true** döndürür. Aksi halde **false** döndürür.

**str[idx]** bir rakam ise **true** döndürür. Aksi halde, **false** döndürür.

**ch** alfabetin bir harfi ise **true** döndürür. Aksi halde, **false** döndürür.

**str[idx]** alfabetin bir harfi ise **true** döndürür. Aksi halde, **false** döndürür.

**ch** ya alfabetin bir harfi ya da bir rakam ise **true** döndürür. Aksi halde, **false** döndürür.

**str[idx]** ya alfabetin bir harfi ya da bir rakam ise **true** döndürür. Aksi halde, **false** döndürür.

**ch** ya alfabetin küçük harflerinden biri ise **true** döndürür. Aksi halde, **false** döndürür.

**str[idx]** alfabetin küçük harflerinden biri ise **true** döndürür. Aksi halde, **false** döndürür.

**ch** onaltılık bir rakam ise **true** döndürür. Onaltılık rakam **0** ile **9** ya da **A** ile **F** arasındadır. Aksi halde, **false** döndürür.

**str[idx]** onaltılık bir rakam ise **true** döndürür. Onaltılık rakam **0** ile **9** ya da **A** ile **F** arasındadır. Aksi halde, **false** döndürür.

**ch** bir noktalama işaretü ise **true** döndürür. Aksi halde **false** döndürür.

**str[idx]** bir noktalama işaretü ise **true** döndürür. Aksi halde **false** döndürür.

<code>public static bool IsSeparator (char ch)</code>	<code>ch</code> bir ayırcı karakter ise, örneğin boşluk gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsSeparator(string str, int idx)</code>	<code>str[idx]</code> bir ayırcı karakter ise, örneğin boşluk gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsSurrogate (char ch)</code>	<code>ch</code> bir Unicode yedek karakteri ise <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsSurrogate(string str, int idx)</code>	<code>str[idx]</code> bir Unicode yedek karakteri ise <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsSymbol(char ch)</code>	<code>ch</code> bir simgesel karakter ise, örneğin para birimi simgesi gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsSymbol(string str, int idx)</code>	<code>str[idx]</code> bir simgesel karakter ise, örneğin para birimi simgesi gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsUpper(char ch)</code>	<code>ch</code> büyük harf ise <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.
<code>public static bool IsUpper(string str, int idx)</code>	<code>str[idx]</code> büyük harf ise <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.
<code>public static bool IsWhiteSpace(char ch)</code>	<code>ch</code> bir boşluk ise, örneğin boşluk karakteri ya da sekme gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static bool IsWhiteSpace(string str, int idx)</code>	<code>str[idx]</code> bir boşluk ise, örneğin boşluk karakteri ya da sekme gibi, <code>true</code> döndürür. Aksi halde <code>false</code> döndürür.
<code>public static char Parse(string str)</code>	<code>str</code> içindeki karakterin <code>char</code> eşdeğerini döndürür. Eğer <code>str</code> birden fazla karakter içeriyorsa, bir <code>FormatException</code> fırlatılır.
<code>public static char ToLower(char ch)</code>	<code>ch</code> büyük harf ise <code>ch</code> 'in küçük harf eşdeğerini döndürür. Aksi halde, <code>ch</code> değişiklikle uğramadan döndürülür.
<code>public static char ToLower(char ch, CultureInfo c)</code>	<code>ch</code> büyük harf ise <code>ch</code> 'in küçük harf eşdeğerini döndürür. Aksi halde, <code>ch</code> değişiklikle uğramadan döndürülür. Dönüşüm, belirtilen kültürel bilgiye bağlı olarak ele alınır. <code>CultureInfo</code> , <code>System.Globalization</code> içinde tanımlı bir sınıfır.
<code>public static char ToUpper(char ch)</code>	<code>ch</code> küçük harf ise <code>ch</code> 'in büyük harf eşdeğerini döndürür. Aksi halde, <code>ch</code> değişiklikle uğramadan döndürülür.
<code>public static char ToUpper(char ch, CultureInfo c)</code>	<code>ch</code> küçük harf ise <code>ch</code> 'in büyük harf eşdeğerini döndürür. Aksi halde, <code>ch</code> değişiklikle uğramadan döndürülür. Dönüşüm, belirtilen kültürel bilgiye bağlı olarak ele alınır. <code>CultureInfo</code> , <code>System.Globalization</code> içinde tanımlı bir sınıfır.

<code>public override string ToString()</code>	Metodu çağrıran <code>Char</code> 'ın değerinin karakter katarı olarak gösterimini döndürür.
<code>public static string ToString (char ch)</code>	<code>ch</code> 'in karakter katarı olarak gösterimini döndürür.
<code>public string ToString(IFormatProvider fmpvdr)</code>	Belirtilen kültürel bilgiyi kullanarak metodu çağrıran <code>Char</code> 'ın değerinin karakter katarı olarak gösterimini döndürür.

İşte, `Char` tarafından tanımlanan metodlardan birkaçını gösteren bir program:

```
// Birkac Char metodunu gösterir.

using System;

class CharDemo {
    public static void Main() {
        string str = "This is a test. $23";
        int i;

        for(i = 0; i < str.Length; i++) {
            Console.Write(str[i] + " is");
            if(Char.IsDigit(str[i]))
                Console.Write(" digit");
            if(Char.IsLetter(str[i]))
                Console.Write(" letter");
            if(Char.IsLower(str[i]))
                Console.Write(" lowercase");
            if(Char.IsUpper(str[i]))
                Console.write(" uppercase");
            if(Char.IsSymbol(str[i]))
                Console.Write(" symbol");
            if(Char.IsSeparator(str[i]))
                Console.Write(" separator");
            if(Char.IsWhiteSpace(str[i]))
                Console.Write(" whitespace");
            if(Char.IsPunctuation(str[i]))
                Console.Write(" punctuation");

            Console.WriteLine();
        }

        Console.WriteLine("Original: " + str);

        // Buyuk harfe cevir.
        string newstr = "";

        for(i = 0; i < str.Length; i++)
            newstr += Char.ToUpper(str[i]);

        Console.WriteLine("Uppercased: " + newstr);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
T is letter uppercase
h is letter lowercase
i is letter lowercase
s is letter lowercase
    is separator whitespace
i is letter lowercase
s is letter lowercase
    is separator whitespace
a is letter lowercase
    is separator whitespace
t is letter lowercase
e is letter lowercase
s is letter lowercase
t is letter lowercase
    is punctuation
    is separator whitespace
$ is symbol
2 is digit
3 is digit
Original: This is s test. $23
Uppercased: THIS IS A TEST. S23
```

## Boolean Yapısı

**Boolean** yapısı, **bool** veri tipini destekler. **Boolean** tarafından tanımlanan metodlar Tablo 19.10'da gösterilmiştir. **Boolean**'da aşağıdaki alanlar tanımlıdır:

```
public static readonly string FalseString
public static readonly string TrueString
```

Bunlar, **true** ve **false**'un insanların okuyabileceği biçimlerini içerirler. Örneğin, **WriteLine()** çağrıları kullanarak **FalseString**'in çıktısını alırsanız, ekranda “False” karakter katarı görüntülenir.

**Boolean**'da şu arayüzler tanımlıdır: **IComparable** ve **IConvertible**.

**TABLO 19.10: Boolean Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<b>public int CompareTo(object v)</b>	Metodu çağrıran nesnenin değerini <b>v</b> 'ninkiyle karşılaştırır. Değerler eşitse sıfır döndürür, Metodu çağrıran nesne <b>false</b> ve <b>v true</b> ise negatif bir sayı döndürür. Metodu çağrıran nesne <b>true</b> ve <b>v false</b> ise pozitif bir sayı döndürür.
<b>public override bool Equals(object v)</b>	Metodu çağrıran nesnenin değeri <b>v</b> 'nin değerine eşitse <b>true</b> döndürür.

<code>public override int GetHashCode()</code>	Metodu çağrıran nesne için “hash” kodu döndürür.
<code>public TypeCode GetTypeCode()</code>	<code>TypeCode</code> için bir <code>TypeCode</code> numaralandırma değeri döndürür. Bu değer, <code>TypeCode.Boolean</code> ’dır.
<code>public static bool Parse(string str)</code>	<code>str</code> içindeki karakter katarının <code>bool</code> eşdeğerini döndürür. Eğer karakter katarı ne “ <code>True</code> ” ne de “ <code>False</code> ” ise kural dışı bir durum fırlatılır. Büyük-küçük harf farklılığı dikkate alınmaz.
<code>public override string ToString()</code>	Metodu çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.
<code>string ToString(IFormatProvider fmpvdr)</code>	<code>fmpvdr</code> içinde belirtilen kültürel bilgiyi kullanarak, çağrıran nesnenin değerinin karakter katarı olarak gösterimini döndürür.

## Array Sınıfı

`System` içinde yer alan çok kullanışlı sınıflardan biri `Array`’dir. `Array`, C#’taki tüm diziler için temel bir sınıfır. Bu nedenle, `Array`’in metodları, standart tipteki dizilere ya da sizin oluşturduğunuz tipteki dizilere uygulanabilir. `Array`’de Tablo 19.11’de gösterilen özellikler ve Tablo 19.12’de gösterilen metodlar tanımlıdır.

`Array`’de şu arayüzler tanımlıdır: `ICloneable`, `ICollection`, `IEnumerable` ve `IList`. `ICollection`, `IEnumerable` ve `IList`, `System.Collections` isim uzayında tanımlanmışlardır ve Bölüm 22’de anlatılmaktadır.

Metotların birkaçı `IComparer` tipinde bir parametre kullanır. Bu arayüz, `System.Collections` içinde tanımlıdır. İki nesnenin değerini karşılaştırın `Compare()` adında bir metot bu arayüzde tanımlanmıştır. `Compare()` aşağıda gösterilmiştir:

```
int Compare(object v1, object v2)
```

`v1`, `v2`’den büyükse metot, sıfırdan büyük bir değer döndürür; `v1`, `v2`’den küçükse sıfırdan küçük bir değer döndürür; iki değer eşitse sıfır döndürülür.

Sıradaki birkaç konuda, sıkça kullanılan dizi işlemlerinden birkaçı gösterilmektedir.

## Dizileri Sıralamak ve Aramak

En çok kullanılan dizi işlemlerinden biri sıralamadır. Bu nedenle `Array`, sıralama metodlarından oluşan zengin bir bütünü destekler. `Sort()` metodunu kullanarak bütün bir diziyi, dizi içinde bir aralığı ya da uygun anahtar/değer çiftlerini içeren bir dizi çiftini sıralayabilirsiniz. Bir diziyi bir kez sıralandıktan sonra `BinarySearch()` kullanarak artık diziyi verimli ve hızlı biçimde arayabilirsiniz. Bir `int` dizisini sıralayarak `Sort()` ve `BinarySearch()` metodlarını gösteren program aşağıda yer almaktadır:

```
// Bir diziyi sırala ve dizi içinde bir değeri ara.

using System;

class SortDemo {
    public static void Main() {
        int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0 };

        // Orijinal sırayı göster.
        Console.WriteLine("Original order: ");
        foreach(int i in nums)
            Console.Write(i + " ");
        Console.WriteLine();

        // Diziyi sırala.
        Array.Sort(nums);

        // Sıralanmış sırayı göster.
        Console.WriteLine("Sorted order: ");
        foreach(int i in nums)
            Console.Write(i + " ");
        Console.WriteLine();

        // 14'u ara.
        int idx = Array.BinarySearch(nums, 14);

        Console.WriteLine("Index of 14 is " + idx);
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Original order: 5 4 6 3 14 9 8 17 1 24 -1 0
Sorted order: -1 0 1 3 4 5 6 8 9 14 17 24
Index of 14 is 9
```

Yukarıdaki örnekte dizinin temel tipi **int**'tir. **int**, bir değer tipidir. **Array** tarafından tanımlı tüm metotlar otomatik olarak standart veri tiplerinin tümünün kullanımına hazırlıdır. Ancak, nesne referansları içeren diziler için bu durum geçerli olmayabilir. Nesne referansları içeren dizileri sıralamak ya da aramak için söz konusu nesnelerin sınıf tipi **IComparable** arayüzüne uygulamalıdır. Eğer söz konusu sınıf **IComparable**'ı uygulamazsa, diziyi sıralama ya da arama girişiminde bulunurken programın çalışması sırasında kural dışı bir durum meydana gelir. Neyse ki, **IComparable** kolaylıkla uygulanabilir, çünkü yalnızca şu metodu içermektedir:

```
int CompareTo(object v)
```

Bu metot, metodu çağırılan nesneyi **v** değeri ile karşılaştırır. Metodu çağırılan nesne **v**'den büyükse metot, sıfırdan büyük bir değer döndürür. İki nesne eşitse metot, sıfır döndürür. Çağrıda bulunan nesne **v**'den küçükse metot, sıfırdan küçük bir değer döndürür. Kullanıcı

tarafından tanımlanan sınıf nesnelerinden oluşan bir diziyi sıralamayı ve aramayı gösteren bir örnek aşağıda yer almaktadır:

```
// Bir nesne dizisini sıralamak ve aramak.

using System;

class MyClass : IComparable {
    public int i;
    public MyClass(int x) { i = x; }

    // IComparable'i uygula.
    public int CompareTo(object v) {
        return i - ((MyClass)v).i;
    }
}

class SortDemo {
    public static void Main() {
        MyClass[] nums = new MyClass[5];

        nums[0] = new MyClass(5);
        nums[1] = new MyClass(2);
        nums[2] = new MyClass(3);
        nums[3] = new MyClass(4);
        nums[4] = new MyClass(1);

        // Orijinal sırayı göster.
        Console.WriteLine("Original order: ");
        foreach(MyClass o in nums)
            Console.Write(o.i + " ");
        Console.WriteLine();

        // Diziyi sırala.
        Array.Sort(nums);

        // Sıralanmış diziyi göster.
        Console.WriteLine("Sorted order: ");
        foreach(MyClass o in nums)
            Console.Write(o.i + " ");
        Console.WriteLine();

        // MyClass(2)'yi ara.
        MyClass x = new MyClass(2);
        int idx = Array.BinarySearch(nums, x);

        Console.WriteLine("Index of MyClass(2) is " + idx);
    }
}
```

Cıktı aşağıdaki gibidir:

```
Original order: 5 2 3 4 1
Sorted order: 1 2 3 4 5
Index of MyClass(2) is 1
```

## Diziyi Ters Çevirmek

Kimi zaman bir dizinin içeriğini ters çevirmek kullanışlı olur. Örneğin, artan sırada sıralanmış bir diziyi azalan sırada sıralı olacak şekilde değiştirmek isteyebilirsiniz. Bir diziyi ters çevirmek kolaydır: Yalnızca **Reverse()**'ı çağırın. Aşağıdaki program bu işlemi göstermektedir:

```
// Bir diziyi ters çevir.  
  
using System;  
  
class ReverseDemo {  
    public static void Main() {  
        int[] nums = { 1, 2, 3, 4, 5 };  
  
        // Orijinal sırayı göster.  
        Console.WriteLine("Original order: ");  
        foreach(int i in nums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
  
        // Bütün diziyi ters çevir.  
        Array.Reverse (nums);  
  
        // Tersten sıralı diziyi göster.  
        Console.WriteLine("Reversed order: ");  
        foreach(int i in nums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
  
        // Bir aralığı tersine çevir.  
        Array.Reverse(nums, 1, 3);  
  
        // Tersten sıralı diziyi göster.  
        Console.WriteLine("Range reversed: ");  
        foreach(int i in nums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
    }  
}
```

Cıktı aşağıda gösterilmiştir:

```
Original order: 1 2 3 4 5  
Reversed order: 5 4 3 2 1  
Range reversed: 5 2 3 4 1
```

## Diziyi Kopyalamak

Bir dizinin tümünü ya da bir kısmını bir başka diziye kopyalamak yaygın dizi işlemlerinden bir başkasıdır. Bir diziyi kopyalamak için **Copy()**'yi kullanın. **Copy()**'nin hangi

versiyonunu kullandığınıza bağlı olarak **Copy()**, öğeleri hedef dizinin başına ya da ortasına yerleştirir. **Copy()** aşağıdaki programda gösterilmiştir:

```
// Bir diziyi kopyala.

using System;

class CopyDemo {
    public static void Main() {
        int[] source = { 1, 2, 3, 4, 5 };
        int[] target = { 11, 12, 13, 14, 15 };
        int[] source2 = { -1, -2, -3, -4, -5 };

        // Kaynagi goster.
        Console.Write("source: ");
        foreach(int i in source)
            Console.Write(i + " ");
        Console.WriteLine();

        // Orijinal hedefi goster.
        Console.Write("Original contents of target: ");
        foreach(int i in target)
            Console.Write(i + " ");
        Console.WriteLine();

        // Butun diziyi kopyala.
        Array.Copy(source, target, source.Length);

        // Kopyayi goster.
        Console.Write("target after copy:      ");
        foreach(int i in target)
            Console.Write(i + " ");
        Console.WriteLine();

        // Hedefin ortasina kopyala.
        Array.Copy(source2, 2, target, 3, 2);

        // Kopyayi goster.
        Console.Write("target after copy:      ");
        foreach(int i in target)
            Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

Çıktı aşağıdaki gibidir:

```
source: 1 2 3 4 5
Original contents of target: 11 12 13 14 15
target after copy:      1 2 3 4 5
target after copy:      1 2 3 -3 -4
```

TABLO 19.11: Array Tarafından Tanımlı Özellikler

Özellik	Anlamı
<code>public virtual bool IsFixedSize { get; }</code>	Dizi sabit büyüklükteyse <code>true</code> değerinde, dizi dinamik ise <code>false</code> değerinde salt okunur bir özellik.
<code>public virtual bool IsReadOnly { get; }</code>	Array nesnesi salt okunur ise <code>true</code> değerinde; değilse <code>false</code> değerinde salt okunur bir özellik.
<code>public virtual bool IsSynchronized { get; }</code>	Dizi çok kullanılmış bir ortamda kullanılmak için emniyetli ise <code>true</code> değerinde; aksi halde <code>false</code> değerinde salt okunur bir özellik.
<code>public int Length { get; }</code>	Dizinin eleman sayısını içeren salt okunur bir özellik.
<code>public int Rank { get; }</code>	Dizinin boyut sayısını içeren salt okunur bir özellik.
<code>public virtual object SyncRoot { get; }</code>	Diziye erişimi senkronize etmek amacıyla kullanılması gereken nesneyi içeren salt okunur bir özellik.

TABLO 19.12: Array Tarafından Tanımlı Metotlar

Metot	Anlamı
<code>public static int BinarySearch(Array a, object v)</code>	<code>a</code> ile belirtilen dizi içinde <code>v</code> ile belirtilen değeri arar. İlk eşleşmenin indeksini döndürür. <code>v</code> bulunamazsa negatif bir değer döndürür. Dizi sıralı ve tek boyutlu olmalıdır.
<code>public static int BinarySearch(Array a, object v, IComparer comp)</code>	<code>comp</code> ile belirtilen karşılaştırma metodunu kullanarak <code>a</code> ile belirtilen dizi içinde <code>v</code> ile belirtilen değeri arar. İlk eşleşmenin indeksini döndürür. <code>v</code> bulunamazsa negatif bir değer döndürür. Dizi sıralı ve tek boyutlu olmalıdır.
<code>public static int BinarySearch(Array a, int start, int count, object v)</code>	<code>a</code> ile belirtilen dizinin bir bölümünde <code>v</code> ile belirtilen değeri arar. Arama <code>start</code> ile belirtilen indeksten başlar ve <code>count</code> sayıda eleman ile sınırlıdır. İlk eşleşmenin indeksi döndürülür. <code>v</code> bulunamazsa negatif bir değer döndürülür. Dizi sıralı ve tek boyutlu olmalıdır.
<code>public static int BinarySearch(Array a, int start, int count, object v, IComparer comp)</code>	<code>comp</code> ile belirtilen karşılaştırma metodunu kullanarak <code>a</code> ile belirtilen dizinin bir bölümünde <code>v</code> ile belirtilen değeri arar. Arama <code>start</code> ile belirtilen indeksten başlar ve <code>count</code> sayıda eleman ile sınırlıdır. İlk eşleşmenin indeksi döndürülür. <code>v</code> bulunamazsa negatif bir değer döndürülür. Dizi sıralı ve tek boyutlu olmalıdır.

```
public static void Clear(Array a,
int start, int count)
```

**a**'nın belirtilen elemanlarına sıfır değeri verir. Sıfırlanacak elemanlar **start** ile belirtilen indeksten başlar ve **count** sayıda eleman için geçerlidir.

```
public virtual object Clone()
```

Çağrıda bulunan dizinin kopyasını döndürür. Kopya dizi, orijinal dizideki elemanlara referansta bulunur. Buna "sıg kopyalama" denir. Böylece, elemanlar üzerindeki değişiklikler her iki diziyi de etkiler, çünkü her iki dizi de aynı elemanları kullanmaktadır.

```
public static void Copy(Array source,
Array dest, int count)
```

Her dizinin başından başlayarak **source**'dan **dest**'e **count** sayıda eleman kopyalar. Her iki dizi de aynı referans tipinde olunca **Copy()** "sıg kopyalama" yapar; yani, her iki dizi aynı elemanlara referansta bulunacaktır.

```
public static void Copy(Array source,
int srcStart, Array dest, int
destStart, int count)
```

**source[srcStart]**'dan **dest[destStart]**'a **count** sayıda eleman kopyalar. Her iki dizi de aynı referans tipinde olunca **Copy()** "sıg kopyalama" yapar; yani, her iki dizi aynı elemanlara referansta bulunacaktır.

```
public virtual void CopyTo(Array
dest, int start)
```

Çağrıda bulunan dizinin elemanlarını, **dest[start]**'tan başlayarak **dest**'e kopyalar.

```
public static Array
CreateInstance(Type t, int size)
```

**t** tipinde **size** eleman içeren tek boyutlu bir diziye bir referans döndürür.

```
public static Array
CreateInstance(Type t,
int size1, int size2)
```

**size1 x size2** boyutunda iki boyutlu bir diziye bir referans döndürür. Elemanların her biri **t** tipindedir.

```
public static Array
CreateInstance(Type t, int size1,
int size2, int size3)
```

**size1 x size2 x size3** boyutunda üç boyutlu bir diziye bir referans döndürür. Elemanların her biri **t** tipindedir.

```
public static Array
CreateInstance(Type t, int[] sizes)
```

**sizes** ile belirtilen boyutlara sahip çok boyutlu bir diziye bir referans döndürür. Elemanların her biri **t** tipindedir.

```
public static Array
CreateInstance(Type t, int[] sizes,
int[] startIndexes)
```

**sizes** ile belirtilen boyutlara sahip çok boyutlu bir diziye bir referans döndürür. Elemanların her biri **t** tipindedir. Her boyutun başlangıç indeksi **startIndexes** içinde belirtilmiştir. Böylece, sıfırdan farklı bir indeksten başlayan diziler oluşturmak mümkündür.

```
public override bool Equals(object v)
```

Çağrıda bulunan nesnenin değeri **v**'nin değerine eşitse **true** döndürür.

```

public virtual IEnumarator
GetEnumarator()

public int GetLength(int dim)

public int GetLowerBound(int dim)

public override int GetHashCode()

public TypeCode GetTypeCode()

public int GetUpperBound(int dim)

public object GetValue(int idx)

public object GetValue(int idx1,
int idx2)

public object GetValue(int idx1,
int idx2, int idx3)

public object GetValue(int[] idxs)

public static int IndexOf(Array a,
object v)

public static int IndexOf(Array a,
object v, int start)

```

Dizi için bir numaralandırıcı (enumerator) nesne döndürür. Numaralandırıcı, dizi üzerinden dönerek tekrar tekrar geçmenizi mümkün kılar. Numaralandırıcılar Bölüm 22'de (Koleksiyonlar) anlatılmıştır.

Belirtilen boyutun uzunluğunu döndürür. Söz konusu boyut sıfır tabanlıdır; yani ilk boyutun uzunluğunu almak için metoda **0** aktarın. İkinci boyutun uzunluğunu elde etmek için **1** aktarın.

Belirtilen boyutun ilk indeksini döndürür. Bu değer genellikle sıfırdır. Söz konusu boyut sıfır tabanlıdır; yani ilk boyutun uzunluğunu almak için metoda **0** aktarın. İkinci boyutun uzunluğunu elde etmek için **1** aktarın.

Metodu çağrıran nesne için “hash” kodu döndürür.

**Array** için bir **TypeCode** numaralandırma değeri döndürür. Bu değer, **TypeCode.Array**'dir.

Belirtilen boyutun son indeksini döndürür. **dim** parametresi sıfır tabanlıdır; yani ilk boyutun son indeksini almak için metoda **0** aktarın. İkinci boyutun son indeksini elde etmek için **1** aktarın.

Çağrıda bulunan dizinin **idx** indeksindeki elemanın değerini döndürür. Dizi tek boyutlu olmalıdır.

Çağrıda bulunan dizinin **[idx1, idx2]** indeksindeki elemanın değerini döndürür. Dizi iki boyutlu olmalıdır.

Çağrıda bulunan dizinin **[idx1, idx2, idx3]** indeksindeki elemanın değerini döndürür. Dizi üç boyutlu olmalıdır.

Çağrıda bulunan dizi içinde, belirtilen indekslerdeki elemanların değerlerini döndürür. Dizi, **idxs**'in sahip olduğu eleman sayısı kadar boyuta sahip olmalıdır.

Tek boyutlu **a** dizisinde, **v** ile belirtilen degere sahip ilk elemanın indeksini döndürür. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

Tek boyutlu **a** dizisinde, **v** ile belirtilen degere sahip ilk elemanın indeksini döndürür. Arama **a[start]**'tan başlar. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

```

public static int IndexOf(Array a,
object v, int start, int count)

public void Initialize()

public static int LastIndexOf(Array
a, object v)

public static int LastIndexOf(Array
a, object v, int start)

public static int LastIndexOf(Array
a, object v, int start, int count)

public static void Reverse(Array a)

public static void Reverse(Array a,
int start, int count)

public void SetValue(object v, int
idx)

public void SetValue(object v, int
idx1, int idx2)

public void SetValue(object v, int
idx1, int idx2, int idx3)

public void SetValue(object v,
int[] idxs)

public static void Sort(Array a)

```

Tek boyutlu **a** dizisinde, **v** ile belirtilen değere sahip ilk elemanın indeksini döndürür. Arama **a[start]**'tan başlar ve count eleman için geçerlidir. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

Çağrıda bulunan dizinin her elemanına, elemanın yapılandırıcısını çağırarak ilk değer atar. Bu metot yalnızca değer tipindeki dizilerde kullanılabilir.

Tek boyutlu **a** dizisinde, **v** ile belirtilen değere sahip ilk elemanın indeksini döndürür. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

Tek boyutlu **a** dizisinde, **v** ile belirtilen değere sahip ilk elemanın indeksini döndürür. Arama **a[start]**'tan başlayıp **a[0]**'da sona erecek şekilde ters sırada ilerler. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

Tek boyutlu **a** dizisinde, **v** ile belirtilen değere sahip ilk elemanın indeksini döndürür. Arama **a[start]**'tan başlayıp **count** eleman için geçerli olacak şekilde ters sırada ilerler. Değer bulunamazsa **-1** döndürür. (Eğer dizinin alt sınırı sıfırdan farklıysa; başarısızlık değeri, alt sınır **-1**'dir.)

**a**'nın elemanlarını ters çevirir.

**a** içindeki bir aralığın elemanlarını ters çevirir. Ters çevrilecek aralık **a[start]**'tan başlar ve **count** eleman için geçerlidir.

Çağrıda bulunan dizinin **idx** indeksindeki elemana **v** değerini verir. Dizi tek boyutlu olmalıdır.

Çağrıda bulunan dizinin **[idx1, idx2]** indeksindeki elemana **v** değerini verir. Dizi iki boyutlu olmalıdır.

Çağrıda bulunan dizinin **[idx1, idx2,** **idx3]** indeksindeki elemana **v** değerini verir. Dizi üç boyutlu olmalıdır.

Çağrıda bulunan dizi içinde, belirtilen indekslerdeki elemana **v** değerini verir. Dizi, **idxs**'nin sahip olduğu eleman sayısına kadar boyuta sahip olmalıdır.

**a** dizisini artan sırada sıralar. Dizi tek boyutlu olmalıdır.

```
public static void Sort(Array a,
IComparer comp)
```

```
public static void Sort(Array k,
Array v)
```

```
public static void Sort(Array k,
Array v, IComparer comp)
```

```
public static void Sort(Array a, int
start, int count)
```

```
public static void Sort(Array a, int
start, int count, IComparer comp)
```

```
public static void Sort(Array k,
Array v, int start, int count)
```

```
public static void Sort(Array k,
Array v, int start, int count,
IComparer comp)
```

**comp** ile belirtilen karşılaştırma metodunu kullanarak **a** dizisini artan sırada sıralar. Dizi tek boyutlu olmalıdır.

Bir çift tek boyutlu diziyi artan sırada sıralar. **k** dizisi sıralama anahtarlarını içerir. **v** dizisi bu anahtarlarla bağlantılı değerleri içerir. Böylece, iki dizi anahtar/değer çiftlerini içerirler. Sıralamadan sonra her iki dizi de artan anahtar sırasındadır.

**comp** ile belirtilen karşılaştırma metodunu kullanarak bir çift tek boyutlu diziyi artan sırada sıralar. **k** dizisi sıralama anahtarlarını içerir. **v** dizisi bu anahtarlarla bağlantılı değerleri içerir. Böylece, iki dizi anahtar/değer çiftlerini içerirler. Sıralamadan sonra her iki dizi de artan anahtar sırasındadır.

**a** içindeki bir aralığın elemanlarını artan sırada sıralar. Aralık **a[start]**'tan başlar ve **count** eleman uzunluğundadır. Dizi tek boyutlu olmalıdır.

**comp** ile belirtilen karşılaştırma metodunu kullanarak **a** içindeki bir aralığın elemanlarını artan sırada sıralar. Aralık **a[start]**'tan başlar ve **count** eleman uzunluğundadır. Dizi tek boyutlu olmalıdır.

Bir çift tek boyutlu dizi içinde bir aralığı artan sırada sıralar. Her iki dizide de sıralanacak aralık **start** ile aktarılan indeksten başlar ve **count** eleman uzunluğundadır. **k** dizisi sıralama anahtarlarını içerir. **v** dizisi bu anahtarlarla bağlantılı değerleri içerir. Böylece, iki dizi anahtar/değer çiftlerini içerirler. Sıralamadan sonra her iki dizi de artan anahtar sırasındadır.

**comp** ile belirtilen karşılaştırma metodunu kullanarak bir çift tek boyutlu dizi içinde bir aralığı artan sırada sıralar. Her iki dizide de sıralanacak aralık **start** ile aktarılan indeksten başlar ve **count** eleman uzunluğundadır. **k** dizisi sıralama anahtarlarını içerir. **v** dizisi bu anahtarlarla bağlantılı değerleri içerir. Böylece, iki dizi anahtar/değer çiftlerini içerirler. Sıralamadan sonra her iki dizi de artan anahtar sırasındadır.

## BitConverter

Programlama yaparken sık sık standart veri tiplerini bir byte dizisine dönüştürmemiz gereklidir. Söz gelişi, bir donanım cihazı bir tamsayı değer gerektiriyor olabilir; fakat bu değer bir kerede bir byte şeklinde gönderilmelidir. Tersi durum da sık sık meydana gelir. Bazen veriler, standart tiplerden birine dönüştürülmesi gereken sıralı bir byte sekansı olarak alınacaktır.

Örneğin, bir cihaz, bir byte akışı şeklinde gönderilen tamsayı çıktılar üretebilir. Ne tür bir dönüşüm ihtiyaç duyarsanız duyun C# bu problemlerin çözümü için **BitConverter** sınıfını sağlamaktadır.

**BitConverter** Tablo 19.13'te gösterilen metotları içerir. Ayrıca aşağıdaki alan da bu sınıf içinde tanımlıdır:

```
public static readonly bool IsLittleEndian
```

Mevcut ortam bir sözcüğü en az anlamlı byte ilk sırada, en çok anlamlı byte ikinci sırada yer alacak şekilde saklıyorsa bu alan **true** değerine sahiptir. Buna "küçük endian" biçimini denir. Eğer mevcut ortam en çok anlamlı byte ilk sırada, en az anlamlı byte ikinci sırada yer alacak şekilde saklarsa **isLittleEndian false**'tur. Buna "büyük endian" biçimini denir. Intel Pentium tabanlı makineler "küçük endian" biçimini kullanırlar.

**BitConverter** mühürlenmiştir (**sealed**), yani kalıtım yoluyla aktarılamaz.

**TABLO 19.13: BitConverter Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public static long DoubleToInt64Bits(double v)</code>	v'yi bir <b>long</b> tamsayiya dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(bool v)</code>	v'yi 1 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(char v)</code>	v'yi 2 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(double v)</code>	v'yi 8 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(float v)</code>	v'yi 4 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(int v)</code>	v'yi 4 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(long v)</code>	v'yi 8 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(short v)</code>	v'yi 2 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(uint v)</code>	v'yi 4 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(ulong v)</code>	v'yi 8 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static byte[ ] GetBytes(ushort v)</code>	v'yi 2 byte uzunlığında bir diziye dönüştürür ve sonucu döndürür.
<code>public static double Int64Bits.ToDouble(long v)</code>	v'yi <b>double</b> tipinde bir kayan noktalı değere dönüştürür ve sonucu döndürür.

<code>public static bool ToBoolean(byte[ ] a, int idx)</code>	<code>a[idx]</code> 'teki byte'ı karşılık gelen <code>bool</code> eşdeğeri dönüştürür ve sonucu döndürür. Sıfırdan farklı bir değer <code>true</code> 'ya dönüştürülür; sıfır, <code>false</code> 'a dönüştürülür.
<code>public static char ToChar(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan iki byte'ı karşılık gelen <code>char</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static double ToDouble(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan sekiz byte'ı karşılık gelen <code>double</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static shortToInt16(byte[] a, int start)</code>	<code>a[start]</code> 'tan başlayan iki byte'ı karşılık gelen <code>short</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static intToInt32(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan dört byte'ı karşılık gelen <code>int</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static longToInt64(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan sekiz byte'ı karşılık gelen <code>long</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static floatToSingle(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan dört byte'ı karşılık gelen <code>float</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static stringToString(byte[ ] a)</code>	<code>a</code> 'nın içindeki byte'ları bir karakter katarına dönüştürür. Karakter katarı, byte'larla ilintili onaltılık değerleri tire ile ayrılmış olarak içerir.
<code>public static stringToString(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayarak <code>a</code> 'nın içindeki byte'ları bir karakter katarına dönüştürür. Karakter katarı, byte'larla ilintili onaltılık değerleri tire ile ayrılmış olarak içerir.
<code>public static stringToString(byte[ ] a, int start, int count)</code>	<code>a[start]</code> 'tan başlayarak <code>a</code> 'nın içindeki <code>count</code> sayıda byte'ı bir karakter katarına dönüştürür. Karakter katarı, byte'larla ilintili onaltılık değerleri tire ile ayrılmış olarak içerir.
<code>public static ushortToUInt16(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan iki byte'ı karşılık gelen <code>ushort</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static uintToUInt32(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan dört byte'ı karşılık gelen <code>uint</code> eşdeğeri dönüştürür ve sonucu döndürür.
<code>public static ulongToUInt64(byte[ ] a, int start)</code>	<code>a[start]</code> 'tan başlayan sekiz byte'ı karşılık gelen <code>ulong</code> eşdeğeri dönüştürür ve sonucu döndürür.

## Random ile Rasgele Sayılar Üretmek

Sözde-rasgele (pseudorandom) sayılar sekansı üretmek için `Random` sınıfını kullanacaksınız. Rasgele sayı sekansları simülasyonlar ve modelleme de dahil olmak üzere çok çeşitli durumlarda işe yararlar. Sekansın başlama noktası bir *tohum* (*seed*) değer ile belirlenir. Bu değer otomatik olarak `Random` tarafından sağlanabilir ya da açıkça belirtilebilir.

`Random`'da aşağıdaki iki yapılandırıcı tanımlıdır:

```
public Random()
public Random(int tohum)
```

İlk versiyonda, tohum değerini hesaplamak için sistem saatini kullanan bir Random nesnesi oluşturulur. İkinci versiyon, tohum değeri olarak `tohum`'u kullanır.

`Random` tarafından tanımlanan metodlar Tablo 19.14'te gösterilmiştir.

**TABLO 19.14: Random Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public virtual int Next()</code>	0 ve <code>Int32.MaxValue-1</code> (bu sayılar dahil) arasındaki bir sonraki rasgele tamsayıyı döndürür.
<code>public virtual int Next(int upperBound)</code>	0 ve <code>upperBound-1</code> (bu sayılar dahil) arasındaki bir sonraki rasgele tamsayıyı döndürür.
<code>public virtual int Next(int lowerBound, int upperBound)</code>	<code>lowerBound</code> ve <code>upperBound-1</code> (bu sayılar dahil) arasındaki bir sonraki rasgele tamsayıyı döndürür.
<code>public virtual void NextBytes(byte[] buf)</code>	<code>buf</code> 'i bir seri rasgele tamsayı ile doldurur. Dizi içindeki her byte 0 ile <code>Byte.MaxValue-1</code> (bu sayılar dahil) arasında olmalıdır.
<code>public virtual int NextDouble()</code>	0 . 0'dan büyük veya 0 . 0'a eşit ve 1 . 0'den küçük bir kayan noktalı değer ile simgelenen seriden bir sonraki rasgele değeri döndürür.
<code>public virtual double Sample()</code>	0 . 0'dan büyük veya 0 . 0'a eşit ve 1 . 0'den küçük bir kayan noktalı değer ile simgelenen seriden bir sonraki rasgele değeri döndürür. Çarpık ya da özelleştirilmiş bir dağılım oluşturmak için bu metodu türetilmiş bir sınıf içinde devre dışı bırakın.

İşte, kompüterize edilmiş bir çift zar oluşturarak `Random`'u tanıtan bir program:

```
// Otomatize edilmiş bir çift zar.

using System;

class RandDice {
    public static void Main() {
        Random ran = new Random();

        Console.WriteLine(ran.Next(1, 7) + " ");
        Console.WriteLine(ran.Next(1, 7));
    }
}
```

Programın çalışmasından üç örnek işte şu şekildedir:

```
5 2
4 4
1 6
```

Bu program öncelikle bir **Random** nesnesi oluşturarak çalışmaktadır. Program sonra **1** ile **6** arasında iki rasgele sayı talep etmektedir.

## Bellek Yönetimi ve GC Sınıfı

**GC** sınıfı C#'ın anlamsız verileri toplama (garbage-collection) becerisini kendi bünyesinde paketlemektedir. **GC** tarafından tanımlanan metodlar Tablo 19.15'te gösterilmiştir. Aşağıda gösterilen salt okunur özellik de bu sınıf içinde tanımlıdır:

```
public static int MaxGeneration { get; }
```

**MaxGeneration** tahsis edilen en eski bellek parçasının üretim numarasını içerir. Ne zaman bir tahsis söz konusu olsa (söz geliş, **new** kullanımı ile) tahsis edilen belleğe üretim numarası olarak sıfır verilir. Önceden ayrılan birimlerin üretim numaraları artırılır. Böylece, **MaxGeneration** ayrılan en eski bellek biriminin numarasını gösterir. Üretim numaraları, anlamsız veri toplama işleminin verimliliğini artırmaya yardımcı olmaktadır.

Birçok uygulamada **GC**'nin becerilerinin hiçbirini kullanmayacaksınız. Yine de, özelleştirilmiş durumlarda, bu beceriler çok kullanışlı olabilir. Örneğin, anlamsız verilerin toplanması işleminin istediğiniz bir anda meydana gelmesini zorlamak için **Collect()**'i kullanmak isteyebilirsiniz. Normal olarak, anlamsız verilerin toplanması işlemi, programınız tarafından belirtilmeyen anlarda meydana gelir. Anlamsız verilerin toplanması işlemi zaman alıcı olduğundan, zaman açısından kritik bazı görevler sırasında bu işlemin gerçekleşmesini istemeyebilirsiniz. Ya da, boş geçen zamanı değerlendirmek için anlamsız verilerin toplanması işlemini ve diğer “toparlama” işlerini uygulamak isteyebilirsiniz.

GC mühürlenmiştir (**sealed**), yani kalıtım yoluyla aktarılamaz.

TABLO 19.15: GC Tarafından Tanımlanan Metotlar

Metot	Anlamı
<code>public static void Collect()</code>	Anlamsız verilerin toplanması (garbage collection) işlemini başlatır.
<code>public static void Collect(int MaxGen)</code>	Üretim numaraları <b>0</b> ile <b>MaxGen</b> arasında olan bellek için anlamsız verilerin toplanması işlemini başlatır.
<code>public static int GetGeneration(object o)</code>	<i>o</i> ile referansta bulunulan bellek için üretim numarasını döndürür.
<code>public static int GetGeneration(WeakReference o)</code>	<i>o</i> ile belirtilen zayıf referans tarafından atıfta bulunulan bellek için üretim numarasını döndürür. Zayıf referans, nesnenin anlamsız veri olarak toplanmasını önlemez.
<code>public static long GetTotalMemory(bool collect)</code>	Halihazırda tahsis edilmiş olan toplam byte sayısını döndürür. <b>collect true</b> ise öncelikle anlamsız verilerin toplanması işlemi gerçekleştirilir..
<code>public static void KeepAlive(object o)</code>	<i>o</i> 'ya bir referans oluşturur; böylece, anlamsız veri olarak toplanmasını önlemiştir.
<code>public static void ReRegisterForFinalize(object o)</code>	<i>o</i> için sonuçlandırıcının (yani, yok edicinin) çağrılmasına neden olur. Bu metot, <b>SuppressFinalize()</b> ’ın etkilerini etkisiz kılar.
<code>public static void SuppressFinalize(object o)</code>	<i>o</i> için sonuçlandırıcının (yani, yok edicinin) çağrılmasını önerir.
<code>public static void WaitForPendingFinalizers()</code>	Çağrıda bulunan program kanalının çalışmasını askıda bulunan tüm sonuçlandırıcılar (yani, yok ediciler) çağrırlana kadar durdurur.

## Object

**Object**, C# nesne tipinin altında yatan sınıfıdır. **Object**’in üyeleri Bölüm 11’de ele alınmıştır; fakat **Object**’in C#’ta sahip olduğu merkezi rolden dolayı **Object**’in metodları, size kolaylık sağlaması açısından, Tablo 19.16’da tekrarlanmıştır. **Object**’te, aşağıda gösterilen, bir adet yapılandırıcı tanımlıdır:

`Object()`

Bu yapılandırıcı boş bir nesne oluşturur.

**TABLO 19.16: Object Tarafından Tanımlanan Metotlar**

Metot	Anlamı
<code>public virtual bool Equals(object ob)</code>	Çağrıda bulunan nesne <code>object</code> 'in referansta bulunduğu nesne ile aynı ise <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.
<code>public static bool Equals(object ob1, object ob2)</code>	<code>ob1, ob2</code> ile aynı ise <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.
<code>protected Finalize()</code>	Anlamsız veri toplama işleminden önce kapatma(shutdown) faaliyetlerini gerçekleştirir. C#'ta <code>Finalize()</code> bir yok edici aracılığıyla erişilir.
<code>public virtual int GetHashCode()</code>	Çağrıda bulunan nesne ile ilintili "hash" kodu döndürür.
<code>public Type GetType()</code>	Bir nesnenin tipini programın çalışması sırasında elde eder.
<code>protected object MemberwiseClone()</code>	Nesnenin "sıg kopyasını" çıkarır. Bu tür kopyalamada üyeleri kopyalanır ama üyeleri tarafından referansta bulunan nesneler kopyalanmaz.
<code>public static bool ReferenceEquals(object ob1, object ob2)</code>	<code>ob1</code> ve <code>ob2</code> aynı nesneye referansta bulunuyorsa <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.
<code>public virtual string ToString()</code>	Nesneyi tarif eden bir karakter katarı döndürür.

## IComparable Arayüzü

Birçok sınıf `IComparable` arayüzüni uygulamak zorunda kalacaktır, çünkü `IComparable`, C# kütüphanesinde tanımlı çeşitli metotlar aracılığıyla bir nesnenin bir diğer ile karşılaştırılmasını mümkün kılar. `IComparable`, yalnızca aşağıdaki metodu içerdiği için kolaylıkla uygulanır:

```
int CompareTo(object v)
```

Bu metot, metodu çağrıran nesneyi `v` değeri ile karşılaştırır. Çağrıda bulunan nesne `v`'den büyükse sıfırdan büyük bir değer döndürülür. Eğer iki nesne eşitse sıfır döndürülür. Çağrıda bulunan nesne `v`'den küçük ise sıfırdan küçük bir değer döndürülür.

## IConvertible Arayüzü

`IConvertible` arayüzü, değer tipindeki yapıların tümü tarafından uygulanır. `IConvertible` çeşitli tip dönüşümlerini belirtir. Normal olarak, sizin oluşturduğunuz sınıfların bu arayüzü uygulamaları gerekmeyecektir.

## ICloneable Arayüzü

**ICloneable** arayüzünü uygulayarak bir nesnenin kopyasının çıkarılmasını mümkün kıalarsınız. **ICloneable**'da yalnızca tek bir metot tanımlıdır: **Clone()**. Bu metot aşağıda gösterilmiştir:

```
object Clone()
```

Bu metot, metodu çağrıran nesnenin bir kopyasını çıkartır. **Clone()**'u uygulama şekliniz kopyanın ne şekilde yapılacağını belirler. Genel olarak iki tür kopya mevcuttur: Derin ve sıç. Derin kopyalama yapılırken kopya ve orijinal nesne birbirinden tamamen bağımsızdır. Böylece, eğer orijinal nesne **o** adında başka bir nesneye bir referans içeriyorsa **o**'nun da bir kopyası çıkarılacaktır. Sıç kopyalamada ise üyeleri kopyalanır, fakat üyelerin referansta bulunduğu nesneler kopyalanmaz. Eğer bir nesne **o** adında başka bir nesneye referansta bulunuyorsa, sıç kopyalamadan sonra hem kopya hem de orijinal nesne aynı **o** nesnesine referansta bulunacaklardır ve **o** üzerinde gerçekleştirilen her türlü değişiklik hem kopyayı hem orijinali etkileyecektir. Genellikle derin kopyalama gerçekleştirmek istediğinizde **Clone()**'u uygulayacaksınız. Sıç kopyalamalar, **Object** içinde tanımlı olan **MemberwiseClone()** kullanılarak yapılabilir.

İşte **ICloneable**'ı gösteren bir örnek. Bu program, **X** adında bir sınıfı bir referans içeren **Test** adında bir sınıf oluşturur. **Test**, derin kopya oluşturmak için **Clone()**'u kullanır.

```
// ICloneable kullanımını gösterir.

using System;

class X {
    public int a;

    public X(int x) { a = x; }
}

class Test : ICloneable {
    public X o;
    public int b;

    public Test(int x, int y) {
        o = new X(x);
        b = y;
    }

    public void show(string name) {
        Console.WriteLine(name + " values are ");
        Console.WriteLine("o.a: {0}, b: {1}", o.a, b);
    }

    // Cagrida bulunan nesnenin derin kopyasını cikart.
    public object Clone() {
        Test temp = new Test(o.a, b);
        return temp;
    }
}
```

```

        return temp;
    }
}

class CloneDemo {
    public static void Main() {
        Test ob1 = new Test(10, 20);
        ob1.show("ob1");

        Console.WriteLine("Make ob2 a clone of ob1.");
        Test ob2 = (Test) ob1.Clone();
        ob2.show("ob2");

        Console.WriteLine("Changing ob1.o.a to 99 and ob1.b to 88.");
        ob1.o.a = 99;
        ob1.b = 88;

        ob1.show("ob1");
        ob2.show("ob2");
    }
}

```

Cıktı aşağıda gösterilmiştir:

```

ob1 values are o.a: 10, b: 20
Make ob2 a clone of ob1.
ob2 values are o.a: 10, b: 20
Changing ob1.o.a to 99 and ob1.b to 88.
ob1 values are o.a: 99, b: 88
ob2 values are o.a: 10, b: 20

```

Cıktıdan görüldüğü gibi, **ob2** **ob1**'in klonudur, fakat **ob1** ve **ob2** tamamen ayrı nesnelerdir. Birini değiştirmek diğerini etkilemez. Bu durum, kopya için yeni bir **x** nesnesi ayırarak ve buna orijinal nesne içindeki **x**'in değerini vererek gerçekleştirilir.

Sıg kopyalamayı uygulamak için **Clone()**'un, **Object** tarafından tanımlanan **MemberwiseClone()**'u çağırmasını sağlamamanız yeterlidir. Örneğin, yukarıdaki programdaki **Clone()**'u şu şekilde değiştirmeyi deneyin:

```

// Cagiran nesnenin bir sig kopyasini olustur.
public object Clone() {
    Test temp = (Test) MemberwiseClone();
    return temp;
}

```

Bu değişikliği yaptıktan sonra programın çıktısı aşağıdaki gibi görünecektir:

```

ob1 values are o.a: 10, b: 20
Make ob2 a clone of ob1.
ob2 values are o.a: 10, b: 20
Changing ob1.o.a to 99 and ob1.b to 88.
ob1 values are o.a: 99, b: 88
ob2 values are o.a: 99, b: 20

```

Dikkat ederseniz, **ob1** içindeki **o** ile **ob2** içindeki **o** aynı **x** nesnesine referansta bulunmaktadırlar. Elbette her birinin **b** adındaki **int** alanı her şeye rağmen ayrıdır, çünkü değer tipleri referanslar aracılığıyla erişilemezler.

## IFormatProvider ve IFormattable

**IFormatProvider** arayüzü **GetFormat()** adında tek bir metot tanımlamaktadır. **GetFormat()**, verilerin insanların okuyabileceği bir karakter katarına dönüştürülerek biçimlendirilmesini kontrol eden bir nesne döndürür. **GetFormat()** genel olarak şu şekilde kullanılır:

```
object GetFormat(Type fmt)
```

Burada **fmt**, elde edilecek biçim nesnesini belirtir. Biçimlendirme Bölüm 20'de anlatılmaktadır.

**IFormattable** arayüzü, çıktıının insanların okuyabileceği şekilde biçimlendirilmesini destekler. **IFormattable**'da şu metot tanımlıdır:

```
string ToString(string fmt, IFormatProvider fmtpvdr)
```

Burada **fmt**, biçimlendirme komutlarını belirtir; **fmtpvdr** ise biçim sağlayıcıyı belirtir. Biçimlendirme ayrıntılı olarak Bölüm 20'de anlatılmaktadır.

20

YİRMİNCİ BÖLÜM

---

## KARAKTER KATARLARI VE BİÇİMLENDİRME

Bu bölümde **String** sınıfı incelenmektedir. Bütün programcıların bildiği gibi, karakter katarlarını düzenleme işlemi hemen hemen her programın bir parçasıdır. Bu nedenle **String** sınıfı, karakter katarlarının kurulumunu ve işlenmesini ayrıntılı olarak kontrol etmenizi sağlayan birtakım çok kapsamlı metodlar, özellikler ve alanlar tanımlar. Verileri insanların okuyabildiği bir şekilde biçimlendirmek, karakter katarlarını düzenleme işlemiyle yakından ilgilidir. Biçimlendirme alt sistemini kullanarak C#'ın nümerik tiplerini, tarih ve saatı ve numaralandırmaları biçimlendirebilirsiniz.

## C#'ta Karakter Katarları

C#'ın karakter katarlarını düzenleme işleminin bir özeti Bölüm 7'de sunulmuştu; bahsedilen konular burada tekrarlanmamıştır. Ancak, **String** sınıfını incelemeden önce karakter katarlarının nasıl uygulandığını gözden geçirmekte fayda vardır.

Tüm bilgisayar dillerinde bir *karakter katarı* bir dizi karakter demektir, fakat bu karakter sekansının tam olarak ne şekilde uygulandığı dilden dile değişiklik gösterir. C++ gibi bazı bilgisayar dillerinde karakter katarları, karakter dizileridir. Ancak C#'ta böyle bir durum söz konusu değildir. Bunun yerine, C#'ın karakter katarları, standart **string** veri tipinde nesnelerdir. Bu nedenle, **string** bir referans tipidir. Üstelik, **string** C#'ın standart .NET **string** tipi **System.String** için verdiği bir isimdir. Böylece, bir C# karakter katarı **String** tarafından tanımlı tüm metot, özellik, alan ve operatörlere erişim hakkına sahiptir.

Bir karakter katarı bir kez oluşturulduktan sonra, karakter katarını oluşturan karakter sekansı değiştirilemez. Bu kısıtlama C#'ın, karakter katarlarını daha verimli uygulamasına imkan verir. Bu kısıtlama belki de ciddi bir dezavantaj gibi görülmektedir, fakat değildir. Mevcut bir karakter katarı üzerinde bazı değişiklikler yapılmış bir karakter katarına ihtiyacınız olduğunda, yalnızca istenilen değişiklikleri içeren yeni bir karakter katarı oluşturmanız ve eğer orijinale artık ihtiyacınız yoksa onu göz ardı etmeniz yeterlidir. Kullanılmayan karakter katarı nesneleri otomatik olarak anlamsız veri kapsamında toplandığı için göz ardı edilen karakter katarlarına ne olduğunu dert etmenize gerek yoktur. Ancak bir konunun netlik kazanması gereklidir: **string** referans değişkenleri elbette referansta bulundukları nesneleri değiştirebilirler. Değiştirilmeyen yalnızca, spesifik bir **string** nesnesi oluşturulduktan sonra söz konusu nesnenin karakter sekansıdır.

Değiştirilebilen bir karakter katarı oluşturmak için C#, **StringBuilder** adında bir sınıf sunmaktadır. **StringBuilder**, **System.Text** isim uzayındadır. Ancak, birçok amaç için **StringBuilder** yerine **string**'i kullanmak isteyeceksiniz.

## String Sınıfı

**String**, **System** isim uzayında tanımlıdır. **String**; **IComparable**, **ICloneable**, **IConvertible** ve **IEnumerable** arayüzlerini uygular. **string** bir **sealed** sınıfı; yani, kalıtım yoluyla aktarılamaz. **String**, C# için karakter katarlarını düzenleyici bir işlevsellik

sağlar. **String**, C#'ın standart **string** tipinin altında yatar ve .NET Framework'ün bir parçasıdır. Sonraki birkaç başlık altında, **String** sınıfı ayrıntılı olarak incelenmektedir.

## String Yapılandırıcıları

**String** sınıfında bir karakter katarını çeşitli yollardan kurmanıza imkan veren birkaç yapılandırıcı tanımlıdır. Bir karakter dizisinden bir karakter katarı oluşturmak için aşağıdaki yapılandırıcılardan birini kullanın:

```
public String(char[] karakterler)
public String(char[] karakterler, int start, int toplam)
```

İlkinde, **karakterler** içindeki karakterleri içeren bir karakter katarı yapılandırılır. İkinci-sinde ise **start** ile belirtilen indeksten başlayarak karakterler içinden **toplam** sayıda karakter kullanılır.

Aşağıdaki yapılandırıcıyı kullanarak belirli bir karakteri birkaç kez tekrarlayarak içeren bir karakter katarı oluşturabilirsiniz:

```
public String(char karakter, int toplam)
```

Burada **karakter**, **toplam** sayıda tekrarlanacak olan karakteri belirtir.

Bir karakter dizisine işaret eden bir işaretçi verildiğinde, aşağıdaki yapılandırıcılardan birini kullanarak bir karakter katarı oluşturabilirsiniz:

```
unsafe public String(char* karktr)
unsafe public String(char* karktr, int start, int toplam)
```

İlk ifadede **karktr** tarafından işaret edilen karakterleri içeren bir karakter katarı kurulur. **karktr**'in null sonlandırmalı bir diziye işaret ettiği varsayılr. Bu dizi kendi bütünlüğü içinde kullanılır. İkinci ifadede **start** ile belirtilen indeksten başlayarak **karktr** tarafından işaret edilen diziden **toplam** sayıda karakter kullanılır.

Bir byte dizisine işaret eden bir işaretçi verildiğinde aşağıdaki yapıandrıcılardan birini kullanarak bir karakter katarı yapılandırabilirsiniz:

```
unsafe public String(sbyte* karktr)
unsafe public String(sbyte* karktr, int start, int toplam)
unsafe public String(sbyte* karktr, int start,
                    int toplam, Encoding en)
```

İlk ifade ile **karktr** tarafından işaret edilen karakterleri içeren bir karakter katarı kurulur. **karktr**'in null sonlandırmalı bir diziye işaret ettiği varsayılr. Bu dizi kendi bütünlüğü içinde kullanılır. İkinci ifade **start** ile belirtilen indeksten başlayarak **karktr** tarafından işaret edilen diziden **toplam** sayıda karakter kullanır. Üçüncü ifade byte'ların nasıl kodlanacağını belirtmenize olanak tanır.

Varsayılan kodlama **ASCIIEncoding**'dir. **Encoding** sınıfı **System.Text** isim uzayı içindedir.

Bir karakter katarı literalı otomatik olarak bir karakter katarı nesnesi oluşturur. Bu nedenle, bir string nesnesi genellikle kendisine bir karakter katarı literalı atanarak ilk kullanıma hazırlanır. Bunun bir örneği aşağıda gösterilmiştir:

```
string str = "a new string";
```

## String Alanı, İndeksleyici ve Özellik

**String** sınıfında, aşağıda gösterildiği gibi, tek bir alan tanımlıdır:

```
public static readonly string Empty
```

**Empty**, boş bir karakter katarı belirtir. Boş karakter katarı, hiç karakter içermeyen bir **string**'dir. Bu, **null String** referansından farklıdır. **null String** referansı hiç bir nesneye atıfta bulunmaz.

**String** için yalnızca bir adet salt okunur indeksleyici mevcuttur. Söz konusu indeksleyici aşağıda gösterilmiştir:

```
public char this[int idx] { get; }
```

Bu indeksleyici, belirtilen indeksteki karakteri elde etmeye imkan verir. Típkı diziler gibi, karakter katarları için de indeksleme sıfırdan başlar. **String** nesneleri değiştirilemez oldukları için **String** sınıfının tek bir salt okunur indeksleyiciyi desteklemesi mantıklıdır.

Ayrıca, yalnızca tek salt okunur özellik mevcuttur:

```
public int Length { get; }
```

**Length**, karakter katarı içindeki karakter sayısını döndürür.

## String Operatörleri

**String** sınıfı iki operatörü aşırı yükler: **==** ve **!=**. İki karakter katarının eşitliğini test etmek için **==** operatörünü kullanın. Normal olarak **==** operatörü nesne referanslarına uygulandığında her iki referansın da aynı nesneye atıfta bulunup bulunmadığını belirler. **==**, iki string referansına uygulandığında ise karakter katarlarının içerikleri eşitlik için karşılaştırılır. Aynı durum **!=** operatörü için de geçerlidir: **String** nesneleri karşılaştırıldığında karakter katarlarının içerikleri karşılaştırılır. Ancak, diğer ilişkisel operatörler, soz geliş **<**, **>** ya da **=**, referansları típkı diğer nesne tiplerini karşılaştırır gibi karşılaştırırlar. Bir karakter katarının diğerinden daha büyük ya da küçük olduğunu belirlemek için **String** sınıfı içinde tanımlı olan **Compare()** metodunu kullanın.

## String Metotları

**String** sınıf içinde çok sayıda metot tanımlıdır. Üstelik, metotların birçoğu iki ya da daha fazla aşırı yüklenmiş şeke sahiptir. Bu nedenle, bu metotların tümünü listelemek ne pratiktir ne de kullanışlıdır. Bunun yerine, en çok kullanılan metotlardan birkaçı, kullanımlarını gösteren örneklerle birlikte sunulacaktır.

### Karakter Katarlarını Karşılaştırmak

Karakter katur işlemleriinden belki de en sık kullanılanı bir karakter katarını bir diğer ile karşılaştırmaktır. Öneminden ötürü **String**, çok sayıda karşılaştırma metodu sağlar. Bu metotlar Tablo 20.1'de gösterilmiştir. Bu metotlar içinde en çok yönlü olan **Compare()**'dır. **Compare()**, iki karakter katarını bütünüyle ya da parçalar halinde karşılaştırabilir. Büyükküçük harf ayrimı yaparak karşılaştırma yapabilir ya da harfin büyüğünü dikkate almayabilir. Genel olarak karakter katurı karşılaştırmalarında bir karakter katarının diğerinden daha büyük, küçük ya da ikisinin eşit olup olmadığını belirlemek için sözlük sırası kullanılır. Karşılaştırmayı etkileyen kültürel bilgileri de ayrıca belirtebilirsiniz.

Aşağıdaki programda **Compare()**'ın birkaç versiyonu gösterilmektedir:

```
// Karakter katarlarını karşılastırır.

using System;

class CompareDemo {
    public static void Main() {
        string str1 = "one";
        string str2 = "one";
        string str3 = "ONE";
        string str4 = "two";
        string str5 = "one, too";

        if(String.Compare(str1, str2) == 0)
            Console.WriteLine(str1 + " and " + str2 + " are equal.");
        else
            Console.WriteLine(str1 + " and " + str2 +
                " are not equal.");

        if(String.Compare(str1, str3) == 0)
            Console.WriteLine(str1 + " and " + str3 + " are equal.");
        else
            Console.WriteLine(str1 + " and " + str3 +
                " are not equal.");

        if(String.Compare(str1, str3, true) == 0)
            Console.WriteLine(str1 + " and " + str3 +
                " are equal ignoring case.");
        else
            Console.WriteLine(str1 + " and " + str3 +
                " are not equal ignoring case.");
    }
}
```

```

if(String.Compare(str1, str5) == 0)
    Console.WriteLine(str1 + " and " + str5 + " are equal.");
else
    Console.WriteLine(str1 + " and " + str5 +
                      " are not equal.");

if(String.Compare(str1, 0, str5, 0, 3) == 0)
    Console.WriteLine("First part of " + str1 + " and " +
                      str5 + " are equal.");
else
    Console.WriteLine("First part of " + str1 + " and " +
                      str5 + " are not equal.");

int result = String.Compare(str1, str4);
if(result < 0)
    Console.WriteLine(str1 + " is less than " + str4);
else if(result > 0)
    Console.WriteLine(str1 + " is greater than " + str4);
else
    Console.WriteLine(str1 + " equals " + str4);
}
}

```

Çıktı aşağıdaki gibidir:

```

one and one are equal.
one and ONE are not equal.
one and ONE are equal ignoring case.
one and one, too are not equal.
First part of one and one, too are equal.
one is less than two

```

**TABLO 20.1: String Karşılaştırma Metotları**

### Metot

```
public static int Compare(string
    str1, string str2)
```

```
public static int Compare(string
    str1, string str2, bool ignoreCase)
```

### Anlamı

**str1** tarafından referansta bulunan karakter katarını **str2** ile karşılaştırır. **str1, str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1, str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür.

**str1** tarafından referansta bulunan karakter katarını **str2** ile karşılaştırır. **str1, str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1, str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür. Eğer **ignoreCase true** ise karşılaştırma büyük-küçük harf ayrimını dikkate almaz. Aksi halde, harflerin büyülüklüğü önemlidir.

```
public static int Compare(string
    str1, string str2, bool ignoreCase,
    CultureInfo ci)
```

**str1** tarafından referansta bulunan karakter katarını **str2** ile karşılaştırır. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür. Eğer **ignoreCase true** ise karşılaştırma büyük-küçük harf ayrimini dikkate almaz. Aksi halde, harflerin büyülüklüğü önemlidir. **CultureInfo** sınıfı **System.Globalization** isim uzayında tanımlıdır.

```
public static int Compare(string
    str1, int start1, string str2, int
    start2, int count)
```

**str1** ve **str2** tarafından referansta bulunan karakter katarlarının parçalarını karşılaştırır. Karşılaştırma **str1[start1]** ile **str2[start2]**'tan başlar ve **count** sayıda karakter için geçerlidir. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür.

```
public static int Compare(string
    str1, int start1, string str2, int
    start2, int count, bool ignoreCase)
```

**str1** ve **str2** tarafından referansta bulunan karakter katarlarının parçalarını karşılaştırır. Karşılaştırma **str1[start1]** ile **str2[start2]**'tan başlar ve **count** sayıda karakter için geçerlidir. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür. Eğer **ignoreCase true** ise karşılaştırma büyük-küçük harf ayrimini dikkate almaz. Aksi halde, harflerin büyülüklüğü önemlidir.

```
public static int Compare(string
    str1, int start1, string str2, int
    start2, int count, bool ignoreCase,
    CultureInfo ci)
```

**str1** ve **str2** tarafından referansta bulunan karakter katarlarının parçalarını karşılaştırır. Karşılaştırma **str1[start1]** ile **str2[start2]**'tan başlar ve **count** sayıda karakter için geçerlidir. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür. Eğer **ignoreCase true** ise karşılaştırma büyük-küçük harf ayrimini dikkate almaz. Aksi halde, harflerin büyülüklüğü önemlidir. **CultureInfo** sınıfı **System.Globalization** isim uzayında tanımlıdır.

```

public static int
CompareOrdinal(string str1, string
str2)

public static int
CompareOrdinal(string str1, int
start1, string str2, int start2, int
count)

public int CompareTo(object str)

public int CompareTo(string str)

```

Kültür, bölge ya da dilden bağımsız olarak **str1** tarafından referansta bulunulan karakter katarını **str2** ile karşılaştırır. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür.

Kültür, bölge ya da dilden bağımsız olarak **str1** ve **str2** tarafından referansta bulunulan karakter katarlarının parçalarını karşılaştırır. Karşılaştırma **str1[start1]** ile **str2[start2]**'tan başlar ve **count** sayıda karakter için geçerlidir. **str1**, **str2**'den büyükse sıfırdan büyük bir değer döndürür; **str1**, **str2**'den küçükse sıfırdan küçük bir değer döndürür; **str1** ile **str2** eşitse sıfır döndürür.

Çağrıda bulunan karakter katarı ile **str'**laştırır. Çağrıda bulunan karakter katarı **str'**dan büyükse sıfırdan büyük bir değer döndürülür; **str'**dan küçükse sıfırdan küçük bir değer döndürülür; her ikisi de eşitse sıfır döndürülür.

Çağrıda bulunan karakter katarı ile **str'**laştırır. Çağrıda bulunan karakter katarı **str'**dan büyükse sıfırdan büyük bir değer döndürülür; **str'**dan küçükse sıfırdan küçük bir değer döndürülür; her ikisi de eşitse sıfır döndürülür.

## Karakter Katarlarını Bitistirmek

İki ya da daha fazla karakter katarını bitistirmek için iki yöntem mevcuttur. Birincisi, Bölüm 7'de gösterildiği gibi + operatörünü kullanabilirsiniz. İkincisi, **String** tarafından tanımlanan çeşitli ekleme metodlarından birini kullanabilirsiniz. Birçok durumda + kullanmak en kolay yaklaşım olmasına rağmen ekleme metodları size ilave seçenekler sunarlar.

Bitistirme işlemini gerçekleştiren metot **Concat()** olarak adlandırılır. Bu metodun en basit şekli aşağıda gösterilmiştir:

```
public static string Concat(string str1, string str2)
```

Bu metot, **str1**'in sonuna **str2** eklendikten sonra oluşan yeni karakter katarı döndürür. **Concat()**'ın bir başka şekli, aşağıda gösterildiği gibi, üç karakter katarını bitistirir:

```
public static string Concat(string str1, string str2, string str3)
```

Bu versiyonda **str1**, **str2** ve **str3**'ün peş peşe eklenmesiyle elde edilen karakter katarı döndürülür. Açıkçası, bu işlemleri gerçekleştirmek için **Concat()** metodunu kullanmak yerine + operatörünü kullanmak daha kolaydır.

**Concat()**'ın bir diğer kullanışlı versiyonu keyfi sayıda karakter katarını peş peşe ekler:

```
public static string Concat(params string[] strs)
```

Burada **strs**, bitişitirilen, değişen sayıda argümana referansta bulunmaktadır. Ekleme ile elde edilen sonuç döndürülür. Aşağıdaki program **Concat()**'ın bu versiyonunu göstermektedir:

```
// Concat()'i gösterir.

using System;

class ConcatDemo {
    public static void Main() {

        string result = String.Concat("This ", "is ", "a ",
                                      "test ", "of ", "the ",
                                      "String ", "class.");

        Console.WriteLine("result: " + result);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
result: This is a test of the String class.
```

**Concat()** metodunun bazı versiyonları **string** referansları yerine **object** referanslarını alırlar. Bunlar, birlikte çağrıldıkları nesnelerin karakter katarı olarak gösterimlerini elde ederler ve bu karakter katarlarının bitişitirilmiş şeklini içeren bir karakter katarı döndürürler. **Concat()**'ın bu versiyonları aşağıda gösterilmiştir:

```
public static string Concat(object v1, object v2)
public static string Concat(object v1, object v2, object v3)
public static string Concat(params object[] v)
```

İlk ifade, **v1**'in karakter katarı gösteriminin peşine **v2**'nin karakter katarı gösteriminin eklenmiş şeklini içeren bir karakter katarı döndürür. İkinci ifade; **v1**, **v2** ve **v3**'ün karakter katarı gösterimlerinin bitişitirilmiş şeklini içeren bir karakter katarı döndürür. Üçüncü ifade, **v** içinde aktarılan argümanların karakter katarı gösterimlerinin bitişitirilmiş şeklini içeren bir karakter katarı döndürür. Bu metodların ne kadar kullanışlı olabileceklerini görmek için aşağıdaki programı ele alın:

```
// Concat()'i gösterir.

using System;

class ConcatDemo {
    public static void Main() {

        string result = String.Concat("hi ", 10, " ",
                                      20.0, " ",
                                      "world");
        Console.WriteLine(result);
    }
}
```

```

        false, " ",
        23.45M);

    Console.WriteLine("result: " + result);
}
}

```

Cıktı aşağıda gösterilmiştir:

```
result: hi 10 20 False 23.45
```

Bu örnekte **Concat()** çeşitli tiplerdeki verilerin karakter katarı gösterimlerini peş peşe eklemektedir. Her argüman için bir karakter katarı gösterimi elde etmek amacıyla söz konusu argümanla ilişkili **ToString()** metodu çağrılır. Böylece, 10 değeri için **Int32.ToString()** çağrılır. **Concat()** daha sonra bu karakter katarlarını peş peşe ekler ve sonu döndürür. **Concat()**'ın bu şekli çok kullanışlıdır, çünkü sizi ekleme öncesinde karakter katarı gösterimlerini elle elde etmek zorunda kalmaktadır.

## Karakter Katarı İçinde Arama Yapmak

**String**, bir karakter katarı içinde arama yapmanıza olanak tanıyan iki metod kümesi sunmaktadır. Bir alt karakter katarı ya da bir karakter arayabilirsiniz. İkisinden birine ilk ya da son kez rastlanılan konumu da arayabilirsiniz. Karakter ya da alt karakter katarına ilk rastlanılan konumu aramak için **IndexOf()** metodunu kullanın. Bunun iki kullanım şekli işte söylenir:

```

public int IndexOf(char karktr)
public int IndexOf(String str)

```

İlk ifade, çağrıda bulunan karakter katarı içinde **karktr** karakterine ilk rastlanılan konumdaki indeksi döndürür. İkincisinde ise, **str** karakter katarına ilk rastlanılan konumdaki indeksi döndürülür. Aranılan öğe bulunmazsa her iki ifade de **-1** döndürür.

Bir karaktere ya da alt karakter katarına son kez rastlanılan konumu aramak için **LastIndexOf()** metodunu kullanın. Bunun iki kullanım şekli işte söylenir:

```

public int LastIndexOf(char karktr)
public int LastIndexOf(string str)

```

İlk ifade, çağrıda bulunan karakter katarının içinde **karktr** karakterine son kez rastlanılan konumdaki indeksi döndürür. İkinci ifade, **str** karakter katarına son kez rastlanılan konumdaki indeksi döndürür. Aranılan öğe bulunmazsa her ikisi de **-1** döndürür.

**String**, iki enteresan ilave arama metodu sunar: **IndexOfAny()** ve **LastIndexOfAny()**. Bu metodlar, bir karakter kümesinin herhangi bir ögesiyle eşlenen ilk ya da son karakteri ararlar. Bunların en basit şekilleri söylenir:

```

public int IndexOfAny(char[] a)
public int LastIndexOfAny(char[] a)

```

**IndexOfAny()**, *a*'nın içindeki herhangi bir karakterin metodu çağrıran karakter katarı içinde ilk rastlanıldığı konumdaki indeksini döndürür. **LastIndexOfAny()**, *a*'nın içindeki herhangi bir karakterin metodu çağrıran karakter katarı içinde son rastlanıldığı konumdaki indeksini döndürür. Hiç eşleşme bulunamazsa her iki metot da **-1** döndürür.

Karakter katarlarıyla çalışırken bir karakter katarının belirli bir alt karakter katarıyla başlayıp başlamadığını ya da sona erip ermediğini bilmek genellikle yararlıdır. Bu görevleri gerçekleştirmek için aşağıda gösterilen **StartsWith()** ve **EndsWith()** metodlarını kullanın:

```
public bool StartsWith(string str)
public bool EndsWith(string str)
```

Metodu çağrıran karakter katarı, *str* ürerinden aktarılan karakter katarı ile başlıyorsa **StartsWith()** **true** döndürür. Çağrıda bulunan karakter katarı, *str* ürerinden aktarılan karakter katarı ile sona eriyorsa **EndsWith()** **true** döndürür. Başarısızlık durumunda her ikisi de **false** döndürür,

Aşağıda, karakter katarı arama yöntemlerinden birkaçını gösteren bir program görülmeye:

```
// Karakter katarlarını ara.

using System;

class StringSearchDemo {
    public static void Main() {
        string str = "C# has powerful string handling.";
        int idx;

        Console.WriteLine("str: " + str);

        idx = str.IndexOf('h');
        Console.WriteLine("Index of first 'h': " + idx);

        idx = str.LastIndexOf('h');
        Console.WriteLine("Index of last 'h': " + idx);

        idx = str.IndexOf("ing");
        Console.WriteLine("Index of first \"ing\": " + idx);

        idx = str.LastIndexOf("ing");
        Console.WriteLine("Index of last \"ing\": " + idx);

        char[] chrs = { 'a', 'b', 'c' };
        idx = str.IndexOfAny(chrs);
        Console.WriteLine("Index of first 'a', 'b', or 'c': " + idx);

        if(str.StartsWith("C# has"))
            Console.WriteLine("str begins with \"C# has\"");

        if(str.EndsWith("ling."))
            Console.WriteLine("str ends with \"ling.\\"");
    }
}
```

```
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
str: C# has powerful string handling.
Index of first 'h': 3
Index of last 'h': 23
Index of first "ing": 19
Index of last "ing": 28
Index of first 'a', 'b', or 'c': 4
str begins with "C# has"
str ends with "ling."
```

Arama metodlarının birkaç, aramaya belirli bir indeksten başlamanıza ya da arama yapıla-  
cak bir aralık belirtmenize olanak tanıyan ek biçimlere de sahiptirler. **String** arama metodları-  
nın tüm versiyonları Tablo 20.2'de gösterilmiştir.

**TABLO 20.2: String Tarafından Sunulan Arama Metotları**

Metot	Anlamı
<code>public bool EndWith(string str)</code>	Çağrıda bulunan nesne, <b>str</b> ürerinden aktarılan karakter katarı ile sona eriyorsa <b>true</b> döndürür. Aksi halde, <b>false</b> döndürür.
<code>public int IndexOf(char ch)</code>	Çağrıda bulunan karakter katarı içinde <b>ch</b> 'in bulunduğu ilk konumun indeksini döndürür, <b>ch</b> bulunamazsa <b>-1</b> döndürür.
<code>public int IndexOf(string str)</code>	Çağrıda bulunan karakter katarı içinde <b>str</b> 'in bulunduğu ilk konumun indeksini döndürür. <b>str</b> bulunamazsa <b>-1</b> döndürür.
<code>public int IndexOf(char ch, int start)</code>	Çağrıda bulunan karakter katarı içinde <b>ch</b> 'in bulunduğu ilk konumun indeksini döndürür. Arama <b>start</b> ile belirtilen indeksten başlar. <b>ch</b> bulunamazsa <b>-1</b> döndürür.
<code>public int IndexOf(string str, int start)</code>	Çağrıda bulunan karakter katarı içinde <b>str</b> 'in bulunduğu ilk konumun indeksini döndürür. Arama <b>start</b> ile belirtilen indeksten başlar. <b>str</b> bulunamazsa <b>-1</b> döndürür.
<code>public int IndexOf(char ch, int start, int count)</code>	Çağrıda bulunan, karakter katarı içinde <b>ch</b> 'in bulunduğu ilk konumun indeksini döndürür. Arama <b>start</b> ile belirtilen indeksten başlar ve <b>count</b> eleman için sürdürülür. <b>ch</b> bulunamazsa <b>-1</b> döndürür.

```
public int IndexOf(string str, int
start, int count)
```

Çağrıda bulunan karakter katarı içinde **str**'in bulunduğu ilk konumun indeksini döndürür. Arama **start** ile belirtilen indeksten başlar ve **count** eleman için sürdürülür. **str** bulunamazsa **-1** döndürür.

```
public int IndexOfAny(char[] a)
```

**a** içindeki herhangi bir karakterin, metodu çağrıran karakter katarı içinde ilk rastlanılan konumdaki indeksini döndürür. Hiç eşleşme bulunamazsa **-1** döndürülür.

```
public int IndexOfAny(char[] a, int
start)
```

**a** içindeki herhangi bir karakterin, metodu çağrıran karakter katarı içinde ilk rastlanılan konumdaki indeksini döndürür. Arama **start** ile belirtilen indeksten başlar. Hiç eşleşme bulunamasa **-1** döndürülür.

```
public int IndexOfAny(char[] a, int
start, int count)
```

**a** içindeki herhangi bir karakterin, metodu çağrıran karakter katarı içinde ilk rastlanılan konumdaki indeksini döndürür. Arama **start** ile belirtilen indeksten başlar ve **count** eleman için geçerlidir. Hiç eşleşme bulunamazsa **-1** döndürülür.

```
public int LastIndexOf(char ch)
```

Çağrıda bulunan karakter katarı içinde **ch**'in bulunduğu son konumun indeksini döndürür. **ch** bulunamazsa **-1** döndürür.

```
public int LastIndexOf(string str)
```

Çağrıda bulunan karakter katarı içinde **str**'in bulunduğu son konumun indeksini döndürür. **str** bulunamazsa **-1** döndürür.

```
public int LastIndexOf(char ch, int
start)
```

Çağrıda bulunan karakter katarının bir bölümü içinde **ch**'in bulunduğu son konumun indeksini döndürür. Arama, **start** ile belirtilen indeksten başlayıp sıfırda sona erecek şekilde ters sırada gerçekleştirilir. **ch** bulunamaisa **-1** döndürülür.

```
public int LastIndexOf(string str, int
start)
```

Çağrıda bulunan karakter katarının bir bölümü içinde **str**'in bulunduğu son konumun indeksini döndürür. Arama, **start** ile belirtilen indeksten başlayıp sıfırda sona erecek şekilde ters sırada gerçekleştirilir. **str** bulunamazsa **-1** döndürülür.

```
public int LastIndexOf(char ch, int
start, int count)
```

Çağrıda bulunan karakter katarı içinde **ch**'in bulunduğu son konumun indeksini döndürür. Arama, **start** ile belirtilen indeksten başlayıp, **count** sayıda eleman üzerinde ters sırada gerçekleştirilir. **ch** bulunamazsa **-1** döndürülür.

<code>public int LastIndexOf(string str, int start, int count)</code>	Çağrıda bulunan karakter katarı içinde <code>str</code> 'ın bulunduğu son konumun indeksini döndürür. Arama, <code>start</code> ile belirtilen indeksten başlayıp, <code>count</code> sayıda eleman üzerinde ters sırada gerçekleştirilir. <code>str</code> bulunamazsa <code>-1</code> döndürülür.
<code>public int LastIndexOfAny(char[] a)</code>	<code>a</code> içindeki herhangi bir karakterin, metodu çağırılan karakter katarı içinde son rastlanılan konumdaki indeksini döndürür. Hiç eşleşme bulunamazsa <code>-1</code> döndürülür,
<code>public int LastIndexOfAny(char[] a, int start)</code>	<code>a</code> içindeki herhangi bir karakterin, metodu çağırılan karakter katarı içinde son rastlanılan konumdaki indeksini döndürür. Arama, <code>start</code> ile belirtilen indeksten başlayıp sıfırda sona erecek şekilde ters sırada gerçekleştirilir. Hiç eşleşme bulunamazsa <code>-1</code> döndürülür.
<code>public int LastIndexOfAny(char[] a, int start, int count)</code>	<code>a</code> içindeki herhangi bir karakterin, metodu çağırılan karakter katarı içinde son rastlanılan konumdaki indeksini döndürür. Arama, <code>start</code> ile belirtilen indeksten başlar ve <code>count</code> sayıda eleman üzerinde ters sırada gerçekleştirilir. Hiç eşleşme bulunamazsa <code>-1</code> döndürülür.
<code>public bool StartsWith(string str)</code>	Çağrıda bulunan karakter katarı <code>str</code> ürerinden aktarılan karakter katarı ile başlıyorsa <code>true</code> döndürür. Aksi halde, <code>false</code> döndürür.

## Karakter Katarlarını Ayırmak ve Birleştirmek

İki temel karakter katarı işlemi ayırma ve birleştirmedir. *Ayırma*, bir karakter katarını kendisini oluşturan parçalara ayırır. *Birleştirme* ise birtakım parçalardan bir karakter katarı kurar. Bir karakter katarını ayırmak için `String`'de `Split()` tanımlıdır. Birtakım parçaları birleştirmek için `String`, `Join()`'i sağlar.

`Split()`'in genel olarak kullanımı şu şekildedir:

```
public string[] split(params char[] seps)
public string[] split(params char[] seps, int toplam)
```

İlk ifade, metodu çağırılan karakter katarını parçalara ayırır ve bu alt karakter katarlarını içeren bir dizi döndürür. Alt karakter katarlarını ayıran karakterler `seps` içinde aktarılır. Eğer `seps null` ise, bu durumda ayırcı olarak boşluk (whitespace) kullanılır. İkinci ifadede, `toplam`'dan daha fazla alt karakter katarı döndürilmeyecektir.

`Join()` metodunun iki kullanım şekli aşağıda gösterilmiştir:

```
public static string Join(string sep, string[] strs)
public static string Join(string sep, string[] strs,
                        int start, int toplam)
```

İlk ifader **strs** içindeki karakter katarlarının peş peşe eklenmiş şeklini içeren bir karakter katarı döndürür. İkinci ifade, **strs[start]**'tan başlayarak **strs** içindeki **toplam** sayıda karakter katarının peş peşe eklenmesi ile elde edilen karakter katarını döndürür. Her iki versiyonda her karakter katarı bir sonrakinden **sep** ile belirtilen karakter katarı ile ayrılır.

Aşağıdaki program **Split()** ve **Join()**'i göstermektedir:

```
// Split and join strings.

using System;

class SplitAndJoinDemo {
    public static void Main() {
        string str = "One if by land, two if by sea.";
        char[] seps = { ' ', '.', ',' };

        // Karakter katarini iki parçaya ayır.
        String[] parts = str.Split(seps);
        Console.WriteLine("Pieces from split: ");

        for(int i = 0; i < parts.Length; i++)
            Console.WriteLine(parts[i]);

        // Simdi, parçaları birleştir.
        string whole = String.Join(" | ", parts);
        Console.WriteLine("Result of join: ");
        Console.WriteLine(whole);
    }
}
```

Cıktı aşağıdadır:

```
Pieces from split:
One
if
by
land

two
if
by
sea

Result of join:
One | if | by | land | | two | if | by | sea |
```

Bir karakter katarını ayırmak, karakter katarı düzenlemeleriyle ilgili önemli bir prosedürdür, çünkü bu yöntem bir karakter katarını oluşturan simgelerin (token) tek tek elde edilmesi için sıkça kullanılmaktadır. Örneğin, bir veri tabanı programı, “100’den büyük tüm bütçeleri göster” şeklindeki bir sorgulamayı “100” ve “göster” şeklinde parçalara ayırmak için **Split()** kullanabilir. Bu işlem sırasında ayırıcılar ortadan kaldırılır. Böylece, “göster” yerine “göster” (başta ve sonda boşluk olmadan) elde edilir. Aşağıdaki program bu kavramı

göstermektedir. Program,  $10 + 5$  gibi ikili matematiksel işlemler içeren karakter katarlarını simgelere ayırır. Sonra matematiksel işlemi gerçekleştirir ve sonucu ekranda görüntüler.

```
// Karakter katarlarını simgelere ayır.

using System;

class TokenizeDemo {
    public static void Main() {
        string[] input = {
            "100 + 19",
            "100 / 3.3",
            "-3 * 9",
            "100 - 87"
        };
        char[] seps = {' '};

        for(int i = 0; i < input.Length; i++) {
            // karakter katarını parçalara ayır
            string[] parts = input[i].Split(seps);
            Console.Write("Command: ");

            for(int j = 0; j < parts.Length; j++)
                Console.Write(parts[i] + " ");

            Console.Write(", Result: ");
            double n = Double.Parse(parts[0]);
            double n2 = Double.Parse(parts[2]);

            switch(parts[1]) {
                case "+":
                    Console.WriteLine(n + n2);
                    break;
                case "-":
                    Console.WriteLine(n - n2);
                    break;
                case "*":
                    Console.WriteLine(n * n2);
                    break;
                case "/":
                    Console.WriteLine(n / n2);
                    break;
            }
        }
    }
}
```

Çıktı şöyledir:

```
Command: 100 + 19 , Result: 119
Command: 100 / 3.3 , Result: 30.3030303030303
Command: -3 * 9 , Result: -27
Command: 100 - 87 , Result: 13
```

## Karakter Katarlarını Doldurmak ve Budamak

Kimi zaman bir karakter katarının başındaki ve sonundaki boşlukları ortadan kaldırmak isteyeceksiniz. *Budama* (trimming) olarak adlandırılan bu tür bir işlem genellikle komut işlemciler için gerekli otur. Örneğin, bir veri tabanı “print” sözcüğünü tanıyor olabilir. Ancak, kullanıcı bu komutu başında ya da sonunda bir ya da daha fazla boşluk olacak şekilde girebilir. Karakter katarının veri tabanı tarafından tanınabilmesinden önce bu tür boşlukların her biri ortadan kaldırılmalıdır. Karşıt olarak, kimi zaman da bir karakter katarının belirli bir minimal uzunluğa erişmesi için karakter katarını boşluk ile doldurmak isteyeceksiniz. Örneğin, biçimlendirilmiş çıktı hazırlıyorsanız, hizayı korumak için her satırın belirli bir uzunlukta olmasını garanti etmeniz gerekebilir. Neyse ki, C# bu tür işlemleri kolaylaştırın metotlar içermektedir.

Bir karakter katarını budamak için aşağıdaki **Trim()** metotlarından birini kullanın:

```
public string Trim()  
public string Trim(params char[] karktrler)
```

İlk ifade, metodu çağrıran karakter katarının başında ya da sonundaki boşlukları (whitespace) ortadan kaldırır. İkinci ifade, **karktrler** tarafından belirtilen karakterleri karakter katarının başından ya da sonundan yok eder. Her iki durumda da sonuçta elde edilen karakter katarı döndürülür.

Bir karakter katarının ya sol tarafından ya da sağ tarafından karakterler eklemek suretiyle karakter katarını doldurabilirsiniz. Bir karakter katarını sol taraftan doldurmak için aşağıda gösterilen metotlardan birini kullanın:

```
public string PadLeft(int uzunluk)  
public string PadLeft(int uzunluk, char karakter)
```

İlk ifade, metodu çağrıran karakter katarının toplam uzunluğu **uzunluk**'a eşit olana kadar karakter katarına sol taraftan gerekli sayıda boşluk ekler. İkinci ifade, metodu çağrıran karakter katarının toplam uzunluğu **uzunluk**'a eşit olması için **karakter** tarafından belirtilen karakteri gerekli oldukça söz konusu karakter katarına ekler. Her iki durumda da sonuçta elde edilen karakter katarı döndürülür.

Bir karakter katarını sağdan doldurmak için aşağıdaki metotlardan birini kullanın:

```
public string PadRight(int uzunluk)  
public string PadRight(int uzunluk, char karakter)
```

İlk ifade, metodu çağrıran karakter katarının toplam uzunluğu **uzunluk**'a eşit olana kadar karakter katarına sağ taraftan gerekli sayıda boşluk ekler. İkinci ifade, metodu çağrıran karakter katarının toplam uzunluğu **uzunluk**'a eşit olması için **karakter** tarafından belirtilen karakteri gerekli oldukça söz konusu karakter katarına ekler. Her iki durumda da sonuçta elde edilen karakter katarı döndürülür.

Aşağıdaki program budama ve doldurma işlemlerini göstermektedir:

```
// Budama ve doldurma.

using System;

class TrimPadDemo {
    public static void Main() {
        string str = "test";

        Console.WriteLine("Original string: " + str);

        // Sol taraftan bosluk doldur.
        str = str.PadLeft(10);
        Console.WriteLine("|" + str + "|");

        // Sag taraftan bosluk doldur.
        str = str.PadRight(20);
        Console.WriteLine("|" + str + "|");

        // Bosluklari doldur.
        str = str.Trim();
        Console.WriteLine("|" + str + "|");

        // Sol taraftan # doldur.
        str = str.PadLeft(10, '#');
        Console.WriteLine("|" + str + "|");

        // Sag taraftan # doldur.
        str = str.PadRight(20, '#');
        Console.WriteLine("|" + str + "|");

        // #'leri buda.
        str = str.Trim("#");
        Console.WriteLine("|" + str + "|");
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Original string: test
|      test|
|      test      |
|test|
|#####test|
|#####test#####
|test|
```

## Ekleme, Çıkartma ve Değiştirme

Aşağıda gösterilen `Insert()` metodunu kullanarak bir karakter katarını diğerine ekleyebilirsiniz:

```
public string Insert(int start, string str)
```

Burada **str**, **start** ile belirtilen indekste metodu çağırın karakter katarına eklenir. Elde edilen karakter katarı döndürülür.

**Remove()** kullanarak bir karakter katarının bir bölümünü katardan çıkarabilirsiniz. **Remove()** şu şekildedir:

```
public string Remove(int start, int toplam)
```

Çıkarılacak karakterlerin sayısı **toplam** ile belirtilir. Çıkartmanın başlayacağı indeks **start** ile belirtilir. Elde edilen sonuç döndürülür.

**Replace()** kullanarak bir karakter katarının bir parçasını değiştirebilirsiniz. **Replace()**'in aşağıdaki kullanımı mevcuttur:

```
public string Replace(char ch1, char ch2)
public string Replace(string str1, string str2)
```

İlk ifade, metodu çağırın karakter katarı içinde **ch1**'in geçtiği yerlerin tümünde **ch1**'i **ch2** ile değiştirir. İkinci ifade ise, çağrıda bulunan karakter katarı içinde **str1**'in geçtiği yerlerin tümünde **str1**'i **str2** ile değiştirir. Her iki durumda da sonuçta elde edilen karakter katarı döndürülür.

İşte, **Insert()**, **Remove()** ve **Replace()**'i gösteren bir örnek:

```
// Ekler, degistirir ve cikartir

using System;

class InsRepRevDemo {
    public static void Main() {
        string str = "This test";

        Console.WriteLine("Original string: " + str);

        // Ekle
        str = str.Insert(5, "is a ");
        Console.WriteLine(str);

        // Karakter katarini degistir
        str = str.Replace("is", "was");
        Console.WriteLine(str);

        // Karakterleri degistir
        str = str.Replace('a', 'X');
        Console.WriteLine(str);

        // Cikart
        str = str.Remove(4, 5);
        Console.WriteLine(str);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Original string: This test
This is a test
Thwas was a test
ThwXs wXs X test
ThwX X test
```

## Harf Kipini Değiştirmek

**String**, bir karakter katarı içindeki harflerin kiplerini değiştirmenizi mümkün kıلان iki kullanışlı metot sunar. Bu metotlar **ToUpper()** ve **ToLower()** olarak adlandırılırlar ve aşağıda gösterilmiştirlerdir:

```
public string ToLower()
public string ToUpper()
```

**ToLower()**, metodu çağrıran karakter katarının içindeki tüm harfleri küçük harf olarak değiştirir. **ToUpper()** ise metodu çağrıran karakter katarının içindeki tüm harfleri büyük harf olarak değiştirir. Sonuçta elde edilen karakter katarı döndürülür. Bu metotların ayrıca kültürel ayarlamaları belirtmenize imkan veren versiyonları da mevcuttur.

## Substring () Metodunun Kullanımı

**Substring()** metodunu kullanarak bir karakter katarının bir parçasını elde edebilirsiniz. **Substring()** metodunun iki kullanımı mevcuttur:

```
public string Substring(int idx)
public string Substring(int idx, int toplam)
```

İlk ifadede alt karakter katarı **idx** ile belirtilen indeksten başlar ve çağrıda bulunan karakter katarının sonuna kadar sürer. İkinci ifadede, alt karakter katarı **idx** ile belirtilen indeksten başlar ve **toplam** sayıda karakter ile sınırlıdır. Her iki durumda da söz konusu alt karakter katarı döndürülür.

Aşağıdaki program **Substring()** metodunu göstermektedir:

```
// Substring()'i kullanır.

using System;

class SubstringDemo {
    public static void Main() {
        string str = "ABCDEFGHIJKLMNPQRSTUVWXYZ";

        Console.WriteLine("str: " + str);

        Console.Write("str.Substring(15): ");
        string substr = str.Substring(15);
        Console.WriteLine(substr);
```

```
        Console.WriteLine("str.Substring(0, 15): ");
        substr = str.Substring(0, 15);
        Console.WriteLine(substr);
    }
}
```

Aşağıdaki çıktı üretilir:

```
Str: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Str.Substring(15): PQRSTUVWXYZ
str.Substring(0, 15): ABCDEFGHIJKLMNOP
```

## Biçimlendirme

**int** ya da **double** gibi standart bir tipin insanların okuyabildiği formda olmasına gerek duyulunca bir karakter katarı gösterimi oluşturulmalıdır. C# bu tür bir gösterim için otomatik olarak varsayılan bir biçim sağlıyor olsa da kendi seçtiğiniz bir biçim belirtmeniz de ayrıca mümkündür. Örneğin, Kısım I'de gördüğünüz gibi dolar ve cent biçimini kullanarak nümerik verilerin çıktısını almak mümkündür. **Console.WriteLine()**, **String.Format()** ve nümerik yapı tipleri için tanımlanmış **ToString()** metodu da dahil olmak üzere birkaç sayıda metod verileri biçimlendirir. Bu metodların üçü de biçimlendirme için aynı yaklaşımı kullanırlar; bunlardan birinin verileri nasıl biçimlendirdiğini bir kez öğrendikten sonra bunu diğerlerine de uygulayabilirsiniz.

### Biçimlendirmeye Genel Bakış

Biçimlendirme iki bileşen tarafından yönetilir: *biçim belirleyicileri (format specifiers)* ve *biçim sağlayıcıları (format providers)*. Bir değerin karakter katarı olarak gösteriminin alacağı biçim bir biçim belirleyicisinin kullanımı ile kontrol edilir. Böylece, verinin insanlar tarafından okunabilir biçiminin nasıl görüneceğini dikte eden biçim belirleyicisidir. Örneğin, bilimsel notasyon kullanarak bir nümerik değerin çıktısını almak için E biçim belirleyicisini kullanacaksınız.

Bir çok durumda bir değerin doğru biçim, programın çalıştırıldığı kültür ve dilden etkilenecektir. Söz geliş, Birleşik Devletlerde para dolar ile simgelenir. Avrupa'da ise, euro ile temsil edilir. Kültürel ve dilsel farklılıkların kontrol altına almak için C# biçim sağlayıcılarını kullanmaktadır. Biçim sağlayıcısı, biçim belirleyicisinin nasıl yorumlanacağını tanımlar. Biçim sağlayıcısı **IFormatProvider** arayüzü uygulanarak oluşturulur. Biçim sağlayıcıları, standart nümerik tipler için ve .NET Framework içindeki birçok diğer tip için önceden tanımlıdır. Genel olarak, bir biçim sağlayıcısı belirtmek için kaygı duymak zorunda kalmadan da verilerinizi biçimlendirebilirsiniz. Biçim sağlayıcılarına bu kitapta daha fazla yer ayrılmamıştır.

Verileri biçimlendirmek için biçimlendirmeyi destekleyen bir metoda çağrıda bulunurken biçim belirleyicisini de dahil edersiniz. Biçim belirleyicilerinin kullanımı Bölüm 3'te tanıtılmıştı, fakat burada yeniden gözden geçirmeye değer. Aşağıdaki anlatımda **WriteLine()**

kullanılmıştır, ama aynı temel yaklaşım biçimlendirmeyi destekleyen diğer metotlara da uygulanabilir.

`WriteLine()` kullanarak verileri biçimlendirmek için `WriteLine()`'ın aşağıda gösterilen versiyonunu kullanın:

```
WriteLine("biçimlendirme karakter katarı", arg0, arg1, ... argN);
```

Bu versiyonda `WriteLine()`'a aktarılan argümanlar + yerine virgül ile ayrılır. *biçimlendirme karakter katarı* iki öğe içerir: Olduğu gibi görüntülenen düzgün, baskı karakterleri ve biçim komutları.

Biçim komutları genel olarak şu şekildedir:

```
{argno, genişlik: fmt}
```

Burada `argno`, görüntülenecek argüman sayısını (sıfırdan başlayarak) belirtir. Alanın minimum genişliği `genişlik` ile belirtilir. Biçim belirleyicisi ise `fmt` ile simgelenir. `genişlik` ve `fmt`'nin her ikisi de isteğe bağlıdır. Böylece, en basit şekilde bir biçim komutu yalnızca hangi argümanın görüntüleneceğine işaret eder. Söz gelişi, `{0} arg0`'a işaret eder, `{1} arg1`'e işaret eder vs.

Programın çalışması sırasında, biçim karakter katarı içinde bir biçim komutu ile karşılaşınca karşılık gelen (`argno` ile belirtilen) argüman yerine konulur ve ekranda görüntülenir. Böylece, eşlenen verilerin nerede görüntüleneceğini belirleyen biçim karakter katarı içindeki biçim belirleyicisinin konumudur,

Eğer `fmt` mevcut ise, bu durumda veriler, belirtilen biçim kullanılarak görüntülenir. Aksi halde, varsayılan biçim kullanılır. Eğer `genişlik` mevcut ise, minimum alan genişliğine ulaşıldığını garanti etmek için çıktı boşluk karakteri ile doldurulur. `genişlik` pozitif ise çıktı sağa hizalı gösterilir. `genişlik` negatif ise, çıktı sola hizalıdır.

Bu bölümün kalanında biçimlendirme ve biçim belirleyicileri ayrıntılı olarak incelenmektedir.

## Nümerik Biçim Belirleyicileri

Nümerik veriler için birkaç biçim belirleyicisi tanımlanmıştır. Bunlar Tablo 20.3'te gösterilmişlerdir. Her biçim belirleyicisi, isteğe bağlı olarak duyarlık (precision) belirleyicisi de içerebilir. Örneğin, bir değerin iki ondalık basamağı olan sabit noktalı bir değer olarak simgelenmesi gerektiğini belirtmek için F2 kullanın.

Önceden açıklandığı gibi, belirli biçim belirleyicilerinin tam etkisi kültürel ayarlamalara bağlıdır. Örneğin, para birimi belirleyicisi olan C, bir değeri otomatik olarak seçili olan kültürün para biriminde görüntüler. Birçok kullanıcı için varsayılan kültürel bilgi, kullanıcıların kendi yerel bilgileri ve dili ile eşleşir. Böylece, programın çalıştırıldığı kültürel çerçeveyi dert etmeye gerek kalmadan aynı biçim belirleyicisi kullanılabilir.

Nümerik biçim belirleyicilerinden birkaçını gösteren bir program aşağıda yer almaktadır:

```
// Cesitli bicim belirleyicilerini gosterir.

using System;

class FormatDemo {
    public static void Main() {
        double v = 17688.65849;
        double v2 = 0.15;
        int x = 21;

        Console.WriteLine("{0:F2}", v);
        Console.WriteLine("{0:N5}", v);
        Console.WriteLine("{0:e}", v);
        Console.WriteLine("{0:r}", v);
        Console.WriteLine("{0:p}", v2);
        Console.WriteLine("{0:X}", x);
        Console.WriteLine("{0:D12}", x);
        Console.WriteLine("{0:C}", 189.99);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
17688.66
17,688.65849
1.768866e+004
17688.65849
15.00 %
15
00000000021
$189.99
```

Biçimlerin birkaçında yer alan duyarlık belirleyicisinin etkisine dikkat edin.

**TABLO 20.3: Biçim Belirleyicileri**

Belirleyici	Biçim	Duyarlık Belirleyicisinin Anlamı
C	Para birimi(yani, parasal değer).	Ondalık basamakların sayısını belirtir.
c	C ile aynı.	

<b>D</b>	Tamsayı nümerik veri. (Yalnızca tamsayılarla kullanılır.)	En az basamak sayısı. Gerekirse, sonucu doldurarak tamamlamak için baş tarafta sıfırlar kullanabilir.
<b>d</b>	<b>D</b> ile aynı.	
<b>E</b>	Bilimsel notasyon (Büyük harf <b>E</b> kullanılır).	Ondalık basamakların sayısını belirtir. Varsayılan değer altıdır.
<b>e</b>	Bilimsel notasyon (Küçük harf <b>e</b> kullanılır).	Ondalık basamakların sayısını belirtir. Varsayılan değer altıdır.
<b>F</b>	Sabit noktalı notasyon.	Ondalık basamakların sayısını belirtir.
<b>f</b>	<b>F</b> ile aynı.	
<b>G</b>	<b>E</b> ya da <b>F</b> biçiminden kısa olan kullanılır.	<b>E</b> ve <b>F</b> 'ye bakın.
<b>g</b>	<b>e</b> ya da <b>f</b> biçiminden kısa olan kullanılır.	<b>e</b> ve <b>f</b> 'ye bakın.
<b>N</b>	Sabit noktalı notasyon. Ayırıcı olarak virgül kullanılır.	Ondalık basamakların sayısını belirtir.
<b>n</b>	<b>N</b> ile aynı.	
<b>P</b>	Yüzde	Ondalık basamakların sayısını belirtir.
<b>p</b>	<b>P</b> ile aynı.	
<b>R ya da r</b>	<b>Parse()</b> kullanılarak eşdeğer dahili şeklinde geri ayrıstırılan nümerik değer. (Buna “round-trip” biçim denir.)	Kullanılmaz.
<b>X</b>	Onaltılık ( <b>A</b> ile <b>F</b> arasındaki - <b>A</b> ile <b>F</b> dahil büyük harfler kullanılır).	En az basamak sayısı. Gerekirse, sonucu doldurarak tamamlamak için baş tarafta sıfırlar kullanabilir.
<b>x</b>	Onaltılık ( <b>a</b> ile <b>f</b> arasındaki - <b>a</b> ile <b>f</b> dahil küçük harfler kullanılır).	En az basamak sayısı. Gerekirse, sonucu doldurarak tamamlamak için baş tarafta sıfırlar kullanabilir.

## Verileri Biçimlendirmek için **String.Format()** ve **ToString()** Kullanımı

**WriteLine()** içine biçim komutlarını gömmek çıktıyi biçimlendirmek için kullanışlı bir yöntem olmasına rağmen, kimi zaman biçimlendirilmiş çıktıyi içeren bir karakter katarı oluşturmak isleyeceksiniz, fakat bu karakter katarını hemen ekranda görüntülemek istemeyeceksiniz. Bu şekilde davranış, verileri daha sonra biçimlendirmenize imkan vererek, çıktıyı daha sonra tercih ettiğiniz cihazdan almanızı olanağınız tanır. Bu tür bir yaklaşım özellikle konsol tabanlı I/O'nun ender olarak kullanıldığı Windows gibi bir GUI ortamında kullanışlıdır.

Genel olarak bir değerin biçimlendirilmiş karakter katarı ile gösterimi iki yoldan elde edilir. Bir yol **String.Format()**'ı kullanmaktadır. Diğer ise, standart nümerik tiplere özgü

`ToString()` metoduna bir biçim belirleyicisi aktarmaktır. Her iki yaklaşım da burada incelenmektedir.

## Değerleri Biçimlendirmek İçin `String.Format()` Kullanımı

`String` tarafından tanımlı `Format()` metodlarından birini çağırarak biçimlendirilmiş bir değer elde edebilirsiniz. Bunlar Tablo 20.4'te gösterilmiştir. `Format()`, `WriteLine()`'a çok benzer şekilde çalışır. Tek fark, `Format()`'ın biçimlendirilmiş çıktıyı konsola göndermek yerine bunu karakter katarı olarak döndürmesidir.

**TABLO 20.4: Format Metotları**

Metot	Tanımı
<code>public static string Format(string str, object v)</code>	<code>str</code> içindeki ilk biçim komutuna göre <code>v</code> 'yi biçimlendirir. Biçimlendirilmiş verinin biçim komutunun yerine yerleştirildiği <code>str</code> 'ın bir kopyasını döndürür.
<code>public static string Format(string str, object v1, object v2)</code>	<code>str</code> içindeki ilk biçim komutuna göre <code>v1</code> 'i ve ikinci biçim komutuna göre <code>v2</code> 'yi biçimlendirir. Biçimlendirilmiş verinin biçim komutunun yerine yerleştirildiği <code>str</code> 'ın bir kopyasını döndürür.
<code>public static string Format(string str, object v1, object v2, object v3)</code>	<code>v1</code> , <code>v2</code> ve <code>v3</code> 'ü <code>str</code> içinde karşılık gelen biçim komutlarına göre biçimlendirir. Biçimlendirilmiş verinin biçim komutlarının her biri yerine yerleştirildiği <code>str</code> 'ın bir kopyasını döndürür.
<code>public static string Format(string str, params object[] v)</code>	<code>v</code> üzerinden aktarılan değerleri <code>str</code> içindeki biçim komutlarına göre biçimlendirir. Biçimlendirilmiş verinin biçim komutlarının her biri yerine yerleştirildiği <code>str</code> 'ın bir kopyasını döndürür.
<code>public static string Format(IFormatProvider fmtpvdr, string str, params object[] v)</code>	<code>v</code> üzerinden aktarılan değerleri <code>fmtpvdr</code> tarafından belirtilen biçim sağlayıcısını kullanarak <code>str</code> içindeki biçim komutlarına göre biçimlendirir. Biçimlendirilmiş verinin biçim komutlarının her biri yerine yerleştirildiği <code>str</code> 'ın bir kopyasını döndürür.

Biçimleri gösteren önceki programın `String.Format()` kullanılarak yeniden yazılmış şekli aşağıdadır. Program, önceki versiyon ile aynı çıktıyı üretmektedir.

```
// Bir değeri biçimlendirmek için String.Format() kullanır.

using System;
```

```

class FormatDemo {
    public static void Main() {
        double v = 17688.65849;
        double v2 = 0.15;
        int x = 21;

        string str = String.Format("{0:F2}", v);
        Console.WriteLine(str);

        str = String.Format("{0:N5}", v);
        Console.WriteLine(str);

        str = String.Format("{0:e}", v);
        Console.WriteLine(str);

        str = String.Format("{0:r}", v);
        Console.WriteLine(str);

        str = String.Format("{0:p}", v2);
        Console.WriteLine(str);

        str = String.Format("{0:X}", x);
        Console.WriteLine(str);

        str = String.Format("{0:D12}", x);
        Console.WriteLine(str);

        str = String.Format("{0:C}", 189.99);
        Console.WriteLine(str);
    }
}

```

Tıpkı **WriteLine()** gibi **String.Format()** da sıradan metinleri biçim belirleyicileriyle birlikte kullanmanıza imkan verir. Ayrıca, birden fazla biçim belirleyicisi ve değer de kullanabilirsiniz. Örneğin, **1**'den **10**'a kadar sayıların aşamalı olarak toplamlarını ve çarpımlarını görüntüleyen aşağıdaki programı ele alın:

```

// Format()'a daha yakından bir bakış.

using System;

class FormatDemo2 {
    public static void Main() {
        int i;
        int sum = 0;
        int prod = 1;
        string str;

        /* 1 ile 10 arasındaki sayıların aşamalı olarak
           toplamlarını ve çarpımlarını ekranda gösterir. */
        for(i = 1; i <= 10; i++) {
            sum += i;
            prod *= i;

```

```
        str = String.Format("Sum:{0,3:D} Product:{1,8:D}",  
                           sum, prod);  
        Console.WriteLine(str);  
    }  
}
```

Cıktı aşağıda gösterilmiştir:

Sum:	1	Product:	1
Sum:	3	Product:	2
Sum:	6	Product:	6
Sum:	10	Product:	24
Sum:	15	Product:	120
Sum:	21	Product:	720
Sum:	28	Product:	5040
Sum:	36	Product:	40320
Sum:	45	Product:	362880
Sum:	55	Product:	3628800

Programdaki şu ifadeye özellikle dikkat edin:

```
str = String.Format("Sum:{0,3:D} Product:{1,8:D}",  
    sum, prod);
```

`Format()`'a yapılan bu çağrı, `sum` ve `prod` için birer adet olmak üzere iki biçim belirleyicisi içermektedir. Argüman numaralarının `WriteLine()` ile kullanırken belirtildikleri şekilde belirtildiklerine dikkat edin. Ayrıca, “`Sum:`” gibi sıradan metinlerin de dahil edildiğini dikkate alın. Bu metin, çıktı karakter katarı aracılığıyla aktarılır ve çıktı karakter katarının bir parçası halini alır.

## Verileri Biçimlendirmek İçin ToString() Kullanımı

**Int32** ya da **Double** gibi standart nümerik yapı tiplerinin tümü için, söz konusu değerin biçimlendirilmiş karakter katarı gösterimini elde etmek amacıyla **ToString()** kullanabilirsiniz. Bunun için **ToString()**'in şu versiyonunu kullanacaksınız:

```
public string ToString (string fmt)
```

Bu ifade, metodu çağrıran nesnenin karakter katarı gösterimini, **fmt** üzerinden aktarılan biçim belirleyicisi ile belirtilen şekliyle döndürür. Örneğin, aşağıdaki program **188.99** değerinin para birimi olarak gösterimini **C** biçim belirleyicisini kullanarak oluşturur:

```
string str = 189.99.ToString("C");
```

Biçim belirleyicisinin **ToString()**'e doğrudan aktarılış şekline dikkat edin. **WriteLine()** ya da **Format()** tarafından kullanılan gömülü biçim komutlarının bir argüman numarası ve alan genişliği bileşeni sağlamasından farklı olarak **ToString()** yalnızca biçim belirleyicisinin kendisini gerektirir.

Bİçimlendirilmiş karakter katarları elde etmek amacıyla **ToString()** kullanan önceki biçim programının yeniden yazılmış versiyonu aşağıdadır. Bu program da önceki versiyonlarla aynı çıktıyı üretir.

```
// Degerleri bicimlendirmek icin ToString() kullanir.

using System;

class ToStringDemo {
    public static void Main() {
        double v = 17688.65849;
        double v2 = 0.15;
        int x = 21;

        string str = v.ToString("F2");
        Console.WriteLine(str);

        str = v.ToString("N5");
        Console.WriteLine(str);

        str = v.ToString("e");
        Console.WriteLine(str);

        str = v.ToString("r");
        Console.WriteLine(str);

        str = v2.ToString("p");
        Console.WriteLine(str);

        str = x.ToString("X");
        Console.WriteLine(str);

        str = x.ToString("D12");
        Console.WriteLine(str);

        str = 189.99.ToString("C");
        Console.WriteLine(str);
    }
}
```

## Özel Bir Nümerik Biçim Oluşturmak

Önceden tanımlı biçim belirleyicileri gayet kullanışlı olmalarına rağmen C#, kimi zaman *resim biçim* (*picture format*) olarak adlandırılan bir özelliği kullanarak kendinize ait biçim tanımlamanıza fırsat verir. *Resim biçim* terimi şu gerçekten gelir: Çıktının nasıl görünmesini istiyorsanız, bunu bir örnekle (yani, resim) belirterek kendi biçiminizi oluşturursunuz. Bu yaklaşım Kısım I'de kısaca bahsedilmiştir. Burada ayrıntılarıyla incelenmektedir.

## Özel Biçim Yer Tutucu Karakterleri

Kendi biçiminizi oluştururken verinin nasıl görünmesini istiyorsanız bunun bir örneğini (ya da resmini) oluşturarak biçiminizi belirtirsiniz. Bunun için Tablo 20.5'te gösterilen karakterleri yer tutucu olarak kullanırsınız. Bu karakterlerin her biri sırayla incelenmiştir.

Nokta, ondalık noktanın nereye konumlandırılacağını belirtir.

# yer tutucusu bir basamak konumunu belirtir. #, ondalık noktanın solunda ya da sağında veya tek başına yer alabilir. Ondalık noktanın sağında bir ya da daha fazla # bulunursa bunlar görüntülenecek olan ondalık basamakların sayısını belirtirler. Değer, gerekirse, yuvarlanır. #, ondalık noktanın solunda bulunursa söz konusu değerin tamsayı kısmi için basamak konumlarını belirtir. Gerektiğinde değerin baş tarafına sıfır eklenecektir. Eğer değerin tamsayı kısmı mevcut #'lerden daha fazla basamağa sahipse tamsayı kısmının bütünü görüntülenir. Hiç bir durumda tamsayı kısmının kesilmesi söz konusu değildir. Eğer ondalık nokta yoksa #, söz konusu değerin karşılık gelen tamsayı değere yuvarlanması neden olur. Önemli olmayan sıfır değerleri, söz geliş degerin sonundaki sıfırlar, görüntülenmeyecektir. Bu kimi zaman tuhaf bir duruma neden olur; eğer biçimlendirilen değer sıfır ise ##.## gibi bir biçim ekranda hiç bir şey göstermeyecektir. Sıfır değerinin çıktısını almak için az sonra anlatılan 0 yer tutucusunu kullanın.

0 yer tutucusu, minimum sayıda basamağın mevcut olacağını garanti etmek için değerin başına ya da sonuna 0 eklenmesine neden olur. Ondalık noktanın hem sağında hem de solunda kullanılabilir. Örneğin,

```
Console.WriteLine("{0:00##.#00}", 21.3);
```

aşağıdaki çıktıyı gösterir:

0021.300

Daha çok basamak içeren değerler ondalık noktanın sol tarafında tam olarak ve sağ tarafında ise yuvarlanmış olarak gösterileceklərdir.

# sekansı içine virgül dahil eden bir model belirterek büyük sayılara virgül ekleyebilirsiniz. Örneğin, şu ifade

```
Console.WriteLine("{0:#,###.#}", 3421.3);
```

şunu gösterir:

3,421.3.

Her virgül konumu için virgül belirtmeniz şart değildir. Tek bir virgül belirtmeniz soldan itibaren her üç basamakta bir bir virgül eklenmesine neden olur. Örneğin,

```
Console.WriteLine("{0:#,###.#}", 8763421.3);
```

su çıktıyı üretir:

8,703,421.3.

Virgülerin ikinci bir anlamı daha vardır. Ondalık noktanın hemen solunda bulunurlarsa ölçekleme faktörü olarak davranışırlar. Her virgül, değerin 1,000'e bölünmesine neden olur. Örneğin,

```
Console.WriteLine("Value in thousands: {0:#,###,.#}", 8763421.3);
```

şu çıktıyı üretir:

Value in thousands: 8,763.4

Cıktıdan görüldüğü gibi, değer binlik oranda ölçeklenmiştir.

Yer tutuculara ek olarak kendi oluşturduğunuz biçim belirleyicisi, diğer karakterleri de içerebilir. Diğer karakterler, biçim belirleyicisi içinde nasıl görünüyorlarsa biçimlendirilmiş karakter katarı içinde de tamamen aynı şekilde görünerek aktarılırlar. Örneğin, şu **WriteLine()** ifadesi

```
Console.WriteLine("Fuel efficiency is {0:##.# mpg}", 21.3);
```

şu çıktıyı üretir:

Fuel efficiency is 21.3 mpg

Ayrıca, eğer gerekliyse, \t ya da \n gibi kaçış sekanslarını da kullanabilirsiniz.

*E* ve *e* yer tutucuları, değerin bilimsel notasyonda görüntülenmesine neden olurlar. *E* ya da *e*'nin peşinden en az bir adet, belki daha fazla, 0 gelmelidir. 0'lar görüntülenecek olan ondalık basamak sayısına işaret ederler. Ondalık bileşen biçimde uyması için yuvarlanacaktır. Büyük harf *E* kullanmak büyük harf *E*'nin görüntülenmesine neden olur; küçük harf *e* kullanmak ise küçük harf *e*'nin görüntülenmesine neden olur. Üstün öncesinde bir işaret karakterinin bulunmasını garanti etmek için *E+* ya da *e+* biçimlerini kullanın. Yalnızca negatif değerler için işaret karakterini göstermek için *E*, *e*, *E-* ya da *e-* kullanın.

“;” pozitif, negatif ve sıfır değerleri için farklı biçimler belirtmenizi mümkün kıلان bir ayırcıdır. “;” kullanan ve kendi oluşturduğunuz biçim belirleyicisi genel olarak şu şekildedir:

*pozitif-fmt;negatif-fmt;sıfır-fmt*

İşte bir örnek:

```
Console.WriteLine("{0:#.##; (#.##);0.00}", num);
```

**num** eğer pozitif ise değer iki ondalık basamak ile gösterilir. **num**, negatif ise değer, iki ondalık basamak ile ve parantez içinde gösterilir. **num**, sıfır ise 0.00 karakter katarı gösterilir. Ayırıcıları kullanırken bütün parçaların yer almasına gerek yoktur. Yalnızca pozitif ve negatif değerlerin nasıl görüneceğini belirtmek istiyorsanız, sıfır biçimini dahil etmeyin. Negatif değerler için varsayılan biçim kullanmak amacıyla negatif biçim dahil etmeyin. Bu durumda, pozitif biçim ve sıfır biçim iki noktalı virgül ile ayrılacaktır.

Aşağıdaki program sizin oluşturabileceğiniz olası birçok biçimden yalnızca birkaçını göstermektedir:

```
// ozel bicimleri kullanir.

using System;

class PictureFormatDemo {
    public static void Main() {
        double num = 64354.2345;

        Console.WriteLine("Default format: " + num);

        // 2 ondalik basamakla goster.
        Console.WriteLine("Value with two decimal places: " +
            "{0:#.##}", num);

        // Virgul ve 2 ondalik basamak ile gosterir.
        Console.WriteLine("Add commas: {0:#,###.##}", num);

        // Bilimsel notasyon kullanarak gosterir.
        Console.WriteLine("Use scientific notation: " +
            "{0:#.##e+00}", num);

        // Degeri 1000 ile ölçekler.
        Console.WriteLine("Value in 1,000s: " + "{0:#0,}", num);

        /* Pozitif, negatif ve sifir degerlerini farkli
           farkli gosterir. */
        Console.WriteLine("Display positive, negative, " +
            "and zero values differently.");
        Console.WriteLine("{0:#.#; (#.##);0.00}", num);
        num = -num;
        Console.WriteLine("{0:#.##; (#.##);0.00}", num);
        num = 0.0;
        Console.WriteLine("{0:#.##; (#.##);0.00}", num);

        // Yuzde gosterir.
        num = 0.17;
        Console.WriteLine("Display a percentage: {0:#%}", num);
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Default format: 64354.2345
Value with two decimal places: 64354.23
Add commas: 64,354.23
Use scientific notation: 6.435e+04
Value in 1,000s: 64
Display positive, negative, and zero values differently.
64354.2
(64354.23)
0.00
Display a percentage: 17%
```

### TABLO 20.5: Özel Biçim Yer Tutucu Karakterleri

Yer Tutucu	Anlamı
#	Rakam.
.	Ondalık nokta.
,	Binlik ayırıcı.
%	Yüzde.
0	Değerin başına ya da sonuna sıfır doldurur.
:	Pozitif, negatif ve sıfır değerleri için biçimini tarif eden ayrı bölümler.
E0 E+0 E-0 e0 e+0 e-0	Bilimsel notasyon.

## Tarih ve Saati Biçimlendirmek

Nümerik değerleri biçimlendirmeye ek olarak biçimlendirmenin sıkça uygulandığı bir diğer veri tipi de **DateTime**'dır Bölüm 19'da açıklandığı gibi, **DateTime** tarih ve saatı simgeler. Tarih ve saat değerleri çeşitli şekillerde gösterilebilir. İşte bu örneklerden birkaçı:

02/25/2002  
 Monday, February 25, 2002  
 12:59:00  
 12:59:00 PM

Ayrıca, tarih ve saat gösterimleri ülkeye değişiklik gösterebilir. Bu nedenlerden ötürü, C# tarih ve saat değerleri için çok kapsamlı biçimlendirme alt sistemi sağlamaktadır.

Tarih ve saatin biçimlendirilmesi biçim belirleyicileri aracılığıyla ele alınır. Tarih ve saat için biçim belirleyicileri Tablo 20.6'da gösterilmiştir. Spesifik tarih ve saat gösterimi ülkeye, dilden dile değişebildiği için, hazırlanan kusursuz gösterim, bilgisayarın kültürel ayarlamalarından etkilenecektir.

### TABLO 20.6: Tarih ve Saat Biçim Belirleyicileri

Belirleyici	Biçim
D	Uzun biçimde tarih.
d	Kısa biçimde tarih.
T	Uzun biçimde saat.
t	Kısa biçimde saat.
F	Uzun biçimde tarih ve saat.
f	Kısa biçimde tarih ve saat.
G	Tarih kısa biçimde, saat uzun biçimde.

<b>g</b>	Tarih uzun biçimde, saat kısa biçimde.
<b>M</b>	Ay ve gün.
<b>m</b>	<b>M</b> ile aynı.
<b>R</b>	Tarih ve saat standart, GMT biçiminde.
<b>r</b>	<b>R</b> ile aynı.
<b>s</b>	Tarih ve saat sıralanabilir biçimde.
<b>U</b>	Uzun; evrensel sıralanabilir biçimde tarih ve saat; saat UTC olarak gösterilir.
<b>u</b>	Kısa; evrensel sıralanabilir biçimde tarih ve saat.
<b>Y</b>	Ay ve yıl.
<b>y</b>	<b>Y</b> ile aynı.

İşte, tarih ve saat biçim belirleyicilerini gösteren bir program:

```
// Tarih ve saat bilgilerini bicimlendir.

using System;

class TimeAndDateFormatDemo {
    public static void Main() {
        DateTime dt = DateTime.Now; // su anki saati al

        Console.WriteLine("d format: {0:d}", dt);
        Console.WriteLine("D format: {0:D}", dt);

        Console.WriteLine("t format: {0:t}", dt);
        Console.WriteLine("T format: {0:T}", dt);

        Console.WriteLine("f format: {0:f}", dt);
        Console.WriteLine("F format: {0:F}", dt);

        Console.WriteLine("g format: {0:g}", dt);
        Console.WriteLine("G format: {0:G}", dt);

        Console.WriteLine("m format: {0:m}", dt);
        Console.WriteLine("M format: {0:M}", dt);

        Console.WriteLine("r format: {0:r}", dt);
        Console.WriteLine("R format: {0:R}", dt);

        Console.WriteLine("s format: {0:s}", dt);

        Console.WriteLine("u format: {0:u}", dt);
        Console.WriteLine("U format: {0:U}", dt);

        Console.WriteLine("y format: {0:y}", dt);
        Console.WriteLine("Y format: {0:Y}", dt);
    }
}
```

Örnek çıktı aşağıda gösterilmiştir:

```
d format: 2/28/2002
D format: Thursday, February 28, 2002
t format: 9:32 AM
T format: 9:32:34 AM
f format: Thursday, February 28, 2002 9:32 AM
F format: Thursday, February 28, 2002 9:32:34 AM
g format: 2/28/2002 9:32 AM
G format: 2/28/2002 9:32:34 AM
m format: February 28
M format: February 28
r format: Thu, 28 Feb 2002 09:32:34 GMT
R format: Thu, 28 Feb 2002 09:32:34 GMT
s format: 2002-02-28T09:32:34
u format: 2002-02-28 09:32:34Z
U format: Thursday, February 28, 2002 3:32:34 PM
y format: February, 2002
Y format: February, 2002
```

Bir sonraki program çok basit bir saat geliştirmektedir. Zaman saniyede bir güncellenir. Her saat başı bilgisayarın zili çalar. Biçimlendirilmiş saatin çıktısını almadan önce değerini elde etmek amacıyla **DateTime**'ın **ToString()** metodu kullanılır. Eğer saat başı oldusaya, biçimlendirilmiş saate zilin çalmasını sağlayan alarm karakteri (**\a**) eklenir.

```
// Basit bir saat.

using System;

class SimpleClock {
    public static void Main() {
        string t;
        int seconds;

        DateTime dt = DateTime.Now;
        seconds = dt.Second;

        for(;;) {
            dt = DateTime.Now;

            // saniyeler degisirse saati guncelle
            if(seconds != dt.Second) {
                seconds = dt.Second;

                t = dt.ToString("T");

                if(dt.Minute == 0 && dt.Second == 0)
                    t = t + "\a"; // saat basinda zili cal

                Console.WriteLine();
            }
        }
    }
}
```

## Özel Tarih ve Saat Biçiminizi Oluşturmak

Standart tarih ve saat biçim belirleyicileri olayların büyük çoğunluğuna uygulanacakmasına rağmen özel biçimler de oluşturabilirsiniz. Bu süreç, daha önce bahsedilen, nümerik tipler için kendi biçimlerinizi oluşturmaya benzer. Aslında, basitçe tarih ve saat bilgisinin nasıl görünümesini istiyorsanız, o şekilde bir örnek (resim) oluşturursunuz. Kendinize özgü bir tarih ve saat biçimini oluşturmak için Tablo 20.7'de gösterilen yer göstéricilerinden bir ya da daha fazla kullanacaksınız.

Tablo 20.7'yi incelerseniz, *d*, *f*, *g*, *m*, *M*, *s* ve *t* yer tutucularının Tablo 20.6'da gösterilen tarih ve saat biçim belirleyicileriyle aynı olduğunu göreceksiniz. Genel olarak, bu karakterlerden biri olduğu gibi kullanılırsa, biçim belirleyicisi olarak yorumlanır. Aksi halde, bunun bir yer tutucu olduğu varsayılar. Bu karakterlerden birinin olduğu gibi kullanılmasını, ancak yer tutucu olarak yorumlanması istiyorsanız, karakterin önünde % işaretini kullanın.

Aşağıdaki program, istenilen şekilde oluşturulan tarih ve saat biçimlerinden birkaçını göstermektedir:

```
// Tarih ve saat bilgilerini biçimle.

using System;

class CustomTimeAndDateFormatsDemo {
    public static void Main() {
        DateTime dt = DateTime.Now;

        Console.WriteLine("Time is {0:hh:mm tt}", dt);
        Console.WriteLine("24 hour time is {0:hh:mm}", dt);
        Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", dt);

        Console.WriteLine("Era: {0:gg}", dt);

        Console.WriteLine("Time with seconds: " +
            "{0:HH:mm:ss tt}", dt);

        Console.WriteLine("Use m for day of month: {0:m}", dt);
        Console.WriteLine("use m for minutes: {0:%m}", dt);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Time is 01:49 PM
24 hour time is 01:49
Date is Thu Feb 28, 2002
Era: A.D.
Time with seconds: 13:49:28 PM
Use m far day of month: February 28
use m for minutes: 49
```

**TABLO 20.7: Özel Tarih ve Saat Yer Tutucu Karakterleri**

<b>Yer Tutucu</b>	<b>Ne İle Yer Değiştireceği</b>
<b>d</b>	1 ile 31 arasında bir sayı olarak ayın günü.
<b>dd</b>	1 ile 31 arasında bir sayı olarak ayın günü. 1 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>ddd</b>	Haftanın gününün kısaltılmış ismi.
<b>ddd d</b>	Haftanın gününün kısaltılmamış ismi.
<b>f, ff, fff, ffff, fffff, fffff, ffffff</b>	Kesirli saniyeler. Ondalık basamak sayısı f'lerin sayısı ile belirtilir.
<b>g</b>	Çağ.
<b>h</b>	1 ile 12 arasında bir sayı olarak saat.
<b>hh</b>	1 ile 12 arasında bir sayı olarak saat. 1 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>H</b>	0 ile 23 arasında bir sayı olarak saat.
<b>HH</b>	0 ile 23 arasında bir sayı olarak saat. 0 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>m</b>	Dakika. 0 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>mm</b>	Dakika.
<b>M</b>	1 ile 12 arasında bir sayı olarak ay.
<b>MM</b>	1 ile 12 arasında bir sayı olarak ay. 1 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>MMM</b>	Kısaltılmış ay ismi.
<b>MMMM</b>	Kısaltılmamış ay ismi.
<b>s</b>	Saniye
<b>ss</b>	Saniye. 0 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>t</b>	A ya da P, A.M. ya da P.M. için.
<b>tt</b>	A.M. ya da P.M.
<b>y</b>	Yalnızca tek rakam gerekmiyorsa, iki rakam ile belirtilen yıl.
<b>yy</b>	İki rakam ile belirtilen yıl. 0 ile 9 arasındaki değerlerin önüne 0 eklenir.
<b>yyyy</b>	Dört rakam kullanılarak yıl.
<b>z</b>	Saat dilimi farkı, saat cinsinden.
<b>zz</b>	Saat dilimi farkı, saat cinsinden. 0 ile 9 arasındaki değerlerin önüne 0 eklenir.

---

<b>zzz</b>	Saat dilimi farkı, saat ve dakika cinsinden.
:	Saat bileşenleri için ayırıcı.
/	Tarih bileşenleri için ayırıcı.
% <b>fmt</b>	<b>fmt</b> ile ilişkili standart biçim.

## Numaralandırmaları Biçimlendirmek

C#, bir numaralandırma tarafından tanımlanan değerleri biçimlendirmenize olanak tanır. Numaralandırma değerleri, genel olarak, kendi isimleri ya da değerleri kullanılarak görüntülenebilir. Numaralandırmalar için biçim belirleyicileri Tablo 20.8'de gösterilmiştir. **G** ve **F** biçimlerine özellikle dikkat edin. Bit alanlarını simgelemek için kullanılacak numaralandırmaların önüne **Flags** niteleyicisi yerleştirilmelidir. Tipik olarak; bit alanları, bitleri ayrı ayrı simgeleyen değerleri tutarlar ve ikinin kuvvetleri olarak düzenlenirler. **Flags** niteleyicisi mevcut ise, **G** belirleyicisi, söz konusu değerin geçerli olduğu varsayımyla, değeri oluşturan diğer tüm değerlerin isimlerini görüntüleyecektir. Eğer söz konusu değer numaralandırma ile tanımlanan iki ya da daha fazla alana VEYA uygulanması ile kurulabiliyorsa, **F** belirleyicisi değeri oluşturan diğer tüm değerlerin isimlerini görüntüleyecektir.

Aşağıdaki program numaralandırma belirleyicilerini göstermektedir:

```
// Numaralandırmayı biçimlendirir.

using System;

class EnumFmtDemo {
    enum Direction { North, South, East, West }
    [Flags] enum Status { Ready = 0x1, OffLine = 0x2,
                           Waiting = 0x4, TransmitOK = 0x8,
                           RecieveOK = 0x10, OnLine = 0x20 }

    public static void Main() {
        Direction d = Direction.West;

        Console.WriteLine("{0:G}", d);
        Console.WriteLine("{0:F}", d);
        Console.WriteLine("{0:D}", d);
        Console.WriteLine("{0:X}", d);

        Status s = Status.Ready | Status.TransmitOK;

        Console.WriteLine("{0:G}", s);
        Console.WriteLine("{0:F}", s);
        Console.WriteLine("{0:D}", s);
        Console.WriteLine("{0:X}", s);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```

West
West
3
00000003
Ready, TransmitOK
Ready, TransmitOK
00000009

```

**TABLO 20.8: Numaralandırma Biçim Belirleyicileri**

<b>Belirleyici</b>	<b>Anlamı</b>
<b>G</b>	Sözkonusu değerin ismini görüntüler. Numaralandırmanın öncesinde <b>Flags</b> niteleyicisi yer alıyorsa, söz konusu değerin (değerin geçerli olduğu varsayımyla) bir parçası olan tüm isimler ekranda gösterilecektir.
<b>g</b>	<b>G</b> ile aynı.
<b>F</b>	Sözkonusu değerin ismini görüntüler. Ancak, eğer değer numaralandırma ile tanımlanan iki ya da daha fazla değere VEYA uygulanması ile oluşturulabiliyorsa, söz konusu değerin bir parçası olan tüm isimler ekranda gösterilecektir. Bu durum, <b>Flags</b> niteleyicisi belirtmiş olsun ya da olmasın geçerlidir.
<b>f</b>	<b>F</b> ile aynı.
<b>D</b>	Değeri onluk tabanda bir tamsayı olarak gösterir.
<b>d</b>	<b>D</b> ile aynı.
<b>X</b>	Değeri onaltılık tabanda bir tamsayı olarak gösterir. En az sekiz basamak gösterildiğini garanti etmek için değerin baş tarafına sıfırlar eklenebilir.
<b>x</b>	<b>X</b> ile aynı.

**21**

**YİRMİBİRİNÇİ BÖLÜM**

---

# **ÇOK KANALLI PROGRAMLAMA**

C# birçok yeni özellik içermesine rağmen bu özelliklerden en heyecan verici olanı C#'ın *çok kanallı programlamayı* (*multithreaded programming*) standart olarak desteklemesidir. Çok kanallı bir program paralel zamanlı çalıştırılabilen iki ya da daha fazla parça içerir. Bu tür bir programın her parçasına *kanal* (*thread*) denir ve her kanal farklı bir çalışma yolu tanımlar. Böylece; çok kanallılık, çok görevliliğin (*multitasking*) özelleştirilmiş bir şeklidir.

Çok kanallı programlama, C# dili ve .NET Framework içindeki sınıflar tarafından tanımlanan özelliklerin bir kombinasyonuna dayanır. Çok kanallılık desteği C#'ta standart olarak mevcut olduğu için diğer dillerdeki çok kanallılıkla ilgili problemlerin birçoğu C#'ta minimize edilmiş ya da tamamen ortadan kaldırılmıştır.

## Çok Kanallılıkla İlgili Esaslar

İki ayrı tipte çok görevlilik mevcuttur: Proses tabanlı ve kanal tabanlı. Her ikisi arasındaki farkı anlamak önemlidir. Bir *proses* aslında çalışmakta olan bir programdır. Bu nedenle, *proses tabanlı çok görevlilik* (*process-based multitasking*), bilgisayarınızın iki ya da daha fazla programı eşzamanlı olarak çalıştırmasına olanak tanıyan bir özelliktir. Örneğin, bir elektronik form kullanırken ya da Internet'te gezinirken aynı anda kelime işlemcinizi de çalıştırmanıza imkan veren, proses tabanlı çok görevlilikdir. Proses tabanlı çok görevlilikte bir program, planlayıcı (schedulers) tarafından işleme gönderilen en küçük kod birimidir.

Bir *kanal* işleme gönderilebilen çalıştırılabilir kod birimidir. İsmi, “çalışma kanalı” kavramından gelir. *Kanal tabanlı çok görevlilik* (*thread-based multitasking*) ortamında tüm proseslerin en az bir kanalı vardır, fakat daha fazla kanalları da olabilir. Bu, tek bir program bir kerede iki ya da daha fazla görev gerçekleştirebilir, demektir. Söz gelişi, bir metin editörü metni basarken, aynı anda biçimlendirebilir. Elbette her iki faaliyet iki ayrı kanal tarafından gerçekleştirildiği sürece bu geçerlidir.

Proses tabanlı ve kanal tabanlı çok görevlilik arasındaki farklar şu şekilde özetlenebilir: Proses tabanlı çok görevlilik, programların eşzamanlı olarak çalıştırılmasını kontrol altına alır. Kanal tabanlı çok görevlilik ise aynı programın parçalarının eşzamanlı olarak çalıştırılmasıyla ilgilenir.

Çok kanallılığın başlıca avantajı çok verimli programlar yazmanızı mümkün kılmasıdır, çünkü çok kanallılık, pek çok programda mevcut olan boş zamanı değerlendirmenize imkan verir. Muhtemelen biliyorsunuzdur, I/O cihazlarının birçoğu, ister ağ portu, ister disk sürücüsü ya da klavye olsun, CPU'dan çok daha yavaştır. Bu nedenle, bir program çalışma zamanının büyük çoğunu bir cihaza bilgi göndermek ya da cihazdan bilgi almak için beklemekle harcayacaktır. Çok kanallılık sayesinde, bu boş zamanda programınız bir başka görevi çalıştırabilir. Söz gelişi, programınızın bir bölümü Internet üzerinden bir dosya gönderirken, bir başka bölümü klavyeden girdi okuyabilir, diğer bir bölümü de gönderilecek bir sonraki veri bloğunu tampona yerlestirebilir.

Bir kanal birkaç durumdan birinde olabilir. Kanal *çalışıyor* olabilir. CPU zamanını ele geçirir geçirmez *çalışmaya hazır* olabilir. Çalışmakta olan bir kanal, çalışması geçici olarak

durdurularak *askıya alınmış* olabilir. Çalışmasını daha sonra kaldığı yerden *sürdürebilir*. Bir kaynağı bekliyorken *bloke edilmiş* olabilir. Ya da *sonlandırılmış* olabilir; bu durumda kanalın çalışması biter ve kanal sürdürülemez.

.NET Framework iki tip kanal tanımlar: *ön plan (foreground)* ve arka plan (*background*). Bir kanal oluşturduğunuzda, bu varsayılan değeri ile bir ön plan kanaldır. Fakat bunu arka plan kanala çevirebilirsiniz. Ön plan ve arka plan kanallar arasındaki tek fark, bir arka plan kanalın kendi prosesindeki tüm ön plan kanallar duruktan sonra otomatik olarak sonlandırılmasıdır.

Kanal tabanlı çok görevlilikler birlikte *senkronizasyon (synchronization)* denilen özel tipte bir özelliğe gereksinim duyulmaktadır. Senkronizasyon, iyi tanımlanmış belirli yöntemlerle kanalların çalışmasının koordine edilmesine olanak tanır. C#, senkronizasyona adanmış komple bir alt sisteme sahiptir ve bu sistemin temel özellikleri de ayrıca burada anlatılmaktadır.

Tüm prosesler en azından bir çalışma kanalına sahiptir. Bu kanal genellikle *temel kanal (main thread)* olarak adlandırılır, çünkü programınız başladığında çalıştırılacak olan, bu kanaldır. Öyleyse, bu kitaptaki önceki örnek programların tümünde kullanılmakta olan kanal, temel kanaldır. Temel kanaldan diğer kanalları oluşturabilirsınız.

C# ve .NET Framework hem proses tabanlı çok görevliliği hem de kanal tabanlı çok görevliliği destekler. Böylece, C# kullanarak hem prosesleri hem de kanalları oluşturup, yönetebilirsiniz. Ancak, yeni bir proses başlatmak için biraz programlama yapmanız gerekmektedir, çünkü her proses bir sonrakinden fazlasıyla farklıdır. Aslında, önemli olan C#'ın çok kanallılık desteği sistemde standart olarak mevcut olduğu için C#, yüksek performansa sahip çok kanallı programlarının kurulumunu, C++ (çok kanallılık için standart bir destek içermez) gibi diğer dillere kıyasla daha da kolaylaştırılmıştır.

Çok kanallı programlamayı destekleyen sınıflar **System.Threading** isim uzayında tanımlıdır. Bu nedenle, aşağıdaki ifadeyi genellikle çok kanallı programların başına dahil edeceksiniz:

```
using System.Threading;
```

## Thread Sınıfinin Kullanımı

C#'ın çok kanallılık sistemi, çalışma kanallarını paketleyen **Thread** sınıfı üzerine kurulmuştur. **Thread** sınıfı **sealed** niteliğindedir; yani, kalıtım yoluyla aktarılamaz. **Thread** sınıfında kanalları yönetmeye yardımcı olacak birkaç metot ve özellik tanımlıdır. Bu bölüm içinde bu sınıfın en çok kullanılan üyelerinden birkaçlığını inceleneciktir.

### Bir Kanal Oluşturmak

Bir kanal oluşturmak için **Thread** tipinde bir nesne örneklersiniz. **Thread** aşağıdaki yapılandırıcıyı tanımlar:

```
public Thread(ThreadStart girişNoktası)
```

Burada **girişNoktası**, kanalın çalışmasına başlaması için çağrılacak metodun adıdır. **ThreadStart**, aşağıda gösterildiği şekilde, .NET Framework tarafından tanımlanan bir delegedir:

```
public delegate void ThreadStart()
```

Yani, giriş noktası niteliğindeki metodunuz **void** dönüş değerine sahip olmalı ve hiç argüman almamalıdır.

Bir kez oluşturulduktan sonra yeni kanal, siz onun **Start()** metodunu çağırana kadar çalışmaya başlamayacaktır. **Start()**, **Thread** tarafından tanımlanmıştır; şu şekildedir:

```
public void Start()
```

Bir kez çalışmaya başladıkten sonra kanal, **girişNoktası** ile belirtilen metot dönené kadar çalışmaya devam edecektir. Böylece, **girişNoktası** dönünce kanal otomatik olarak durur. Zaten çalışmaya başlamış bir kanal üzerinde **Start()**'ı çağrımayı denerseniz bir **ThreadStateException** fırlatılacaktır.

**Thread**'in **System.Threading** isim uzayında tanımladığını hatırlınızdan çıkarmayın.

İşte, yeni bir kanalın oluşturulduğu ve çalıştırıldığı bir örnek:

```
// Bir calisma kanali olustur.

using System;
using System.Threading;

class MyThread {
    public int count;
    string thrdName;

    public MyThread(string name) {
        count = 0;
        thrdName = name;
    }

    // kanalin giris noktası.
    public void run() {
        Console.WriteLine(thrdName + " starting.");

        do {
            Thread.Sleep(500);
            Console.WriteLine("In " + thrdName +
                ", count is " + count);
            count++;
        } while(count < 10);

        Console.WriteLine(thrdName + " terminating.");
    }
}

class MultiThread {
```

```

public static void Main() {
    Console.WriteLine("Main thread starting.");

    // Once, bir MyThread nesnesi yapılandır.
    MyThread mt = new MyThread("Child #1");

    // Sonra, bu nesneden bir kanal oluştur.
    Thread newThrd = new Thread(new ThreadStart(mt.run));

    // Son olarak, kanalı çalıştırma başla.
    newThrd.Start();

    do {
        Console.Write(".");
        Thread.Sleep(100);
    } while (mt.count != 10);

    Console.WriteLine("Main thread ending.");
}
}

```

Şimdi gelin bu programa yakından bakalım. **MyThread**, ikinci bir çalışma kanalı oluşturmak için kullanılacak bir sınıf tanımlar. **MyThread**'in **run()** metodu içinde 0'dan 9'a kadar sayan bir döngü kurulur. **Sleep()**'e yapılan çağrıya dikkat edin. **Sleep()**, **Thread** tarafından tanımlanan **static** bir metottur. **Sleep()** metodu, içinden çağrıldığı kanalın çalışmasının belirtilen milisaniye süresince askıya alınmasına neden olur. Programda kullanılan şekli aşağıda gösterilmiştir:

```
public static void Sleep(int milisaniye)
```

Askıda kalma süresi **milisaniye** ile belirtilir. Eğer **milisaniye** sıfır ise, çağrılan kanal yalnızca bekleyen kanalın çalışmasına imkan verecek kadar askıda kalır.

**Main()** içinde, aşağıdaki ifade sekansı tarafından yeni bir **Thread** nesnesi oluşturulur:

```

// Once, bir MyThread nesnesi yapılandır.
My Thread mt = new MyThread("Child #1");

// Sonra, bu nesneden bir kanal oluştur.
Thread newThrd = new Thread(new ThreadStart(mt.run));

// Son olarak, kanalı çalıştırma başla.
newThrd.Start();

```

Açıklamaların gösterdiği gibi, önce bir **MyThread** nesnesi oluşturulur. Bu nesne daha sonra bir **Thread** nesnesi yapılandırmak için kullanılır. Bunun için **mt.run()** metodu giriş noktası olarak aktarılır. Son olarak, **Start()** çağrılarak yeni kanalın çalışması başlatılır. Bu, çocuk kanalın **run()** metodunun başlamasına neden olur. **Start()**'ı çağrıdıktan sonra temel kanalın çalışması **Main()**'e döner ve **Main()**'in **do** döngüsüne girer. Her iki kanal da döngülerini sona erene kadar CPU'yu paylaşarak çalışmalarım sürdürürler. Bu program tarafından üretilen çıktı aşağıdaki gibidir:

```

Main thread starting.
Child #1 starting.
.....In Child #1, count is 0
.....In Child #1, count is 1
.....In Child #1, count is 2
.....In Child #1, count is 3
.....In Child #1, count is 4
.....In Child #1, count is 5
.....In Child #1, count is 6
.....In Child #1, count is 7
.....In Child #1, count is 8
.....In Child #1, count is 9
Child #1 terminating.
Main thread ending.

```

Çok kanallı bir programda genellikle, çalışması biten en son kanalın temel kanal olmasını isteyeceksiniz. Bir program teknik olarak, tüm ön plan kanalları sona erene kadar çalışmaya devam eder. Böylece, temel kanalın çalışmasını en son bitiren kanal olmasını sağlamak bir gereklilik değildir. Yine de, bu uyulmasında fayda olan bir uygulamadır, çünkü programınızın bitiş noktasını net olarak tanımlamaktadır. Önceki program temel kanalın en son olarak sona ereceğini garanti etmektedir, çünkü **do** döngüsü **count 10**'a eşit olunca durmaktadır. **count**, ancak **newThrd** sona erdikten sonra **10**'a eşit olacağı için temel kanal en son olarak sona erer. Bu bölümün ileriki sayfalarında bir kanalın bir diğerinin sona ermesini beklemesiyle ilgili daha iyi yöntemler göreceksiniz.

## Bazı Basit Yenilikler

Önceki program kusursuz olmasına rağmen bazı kolay yenilikler bunu daha da verimli kılacaktır. Öncelikle, bir kanalın oluşturulur oluşturulmaz çalışmaya başlaması mümkün değildir. **MyThread** örneğinde bu, **MyThread**'in yapılandırıcısı içinde bir **Thread** nesnesi örneklenerek gerçekleştirilmektedir. İkincisi, **MyThread**'in kanalın ismini saklamasına gerek yoktur, çünkü **Thread** bu amaç için kullanılabilecek **Name** adında bir özellik tanımlamaktadır. **Name** şu şekilde tanımlanmaktadır:

```
public string Name { get; set; }
```

**Name** salt okunur bir özellik olduğu için bir kanala isim vermek ya da kanalın ismini almak için bunu kullanabilirsiniz.

İşte, önceki program üzerinde bu gelişmeler uygulanarak elde edilen yeni versiyon:

```
// Bir kanalı başlatmak için alternatif yöntem.

using System;
using System.Threading;

class MyThread {
    public int count;
    public Thread thrd;
```

```

public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name; // kanala isim ver.
    thrd.Start(); // kanali baslat.
}

// Kanalin giris noktası.
void run() {
    Console.WriteLine(thrd.Name + " starting.");

    do {
        Thread.Sleep(500);
        Console.WriteLine("In " + thrd.Name +
                          ", count is " + count);
        count++;
    } while(count < 10);

    Console.WriteLine(thrd.Name + " terminating.");
}
}

class MultiThreadImproved {
    public static void Main() {
        Console.WriteLine("Main thread starting.");

        // Once, bir MyThread nesnesi yapılandır.
        MyThread mt = new MyThread("Child #1");
        do {
            Console.Write(".");
            Thread.Sleep(100);
        } while (mt.count != 10);

        Console.WriteLine("Main thread ending.");
    }
}

```

Bu versiyon önceki ile aynı çıktıyı üretir. Kanal nesnesinin **MyThread** içinde **thrd**'de saklandığına dikkat edin.

## Birden Fazla Kanal Oluşturmak

Önceki örnekler yalnızca tek çocuk kanal oluşturmuşlardı. Ancak, sizin programınız gerektiği kadar çok sayıda kanal üretebilir. Örneğin, aşağıdaki program üç çocuk kanal oluşturmaktadır:

```

// Birden fazla calisma kanali olustur.

using System;
using System.Threading;

class MyThread {
    public int count;

```

```

public Thread thrd;

public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
}

// Kanalin giris noktası.
void run() {
    Console.WriteLine(thrd.Name + " starting.");

    do {
        Thread.Sleep(500);
        Console.WriteLine("In " + thrd.Name +
                           ", count is " + count);
        count++;
    } while(count < 10);

    Console.WriteLine(thrd.Name + " terminating.");
}
}

class MoreThreads {
    public static void Main() {
        Console.WriteLine("Main thread starting.");

        // Uc kanal yapılandır.
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        do {
            Console.Write(".");
            Thread.Sleep(100);
        } while (mt1.count == 10 ||
                  mt2.count == 10 ||
                  mt3.count == 10);

        Console.WriteLine(Main thread ending.");
    }
}

```

Bu programın örnek çıktısı aşağıda gösterilmiştir;

```

Main thread starting.
.Child #1 starting.
Child #2 starting.
Child #3 starting.
....In Child #1, count is 0
In Child #2, count is 0
In Child #3, count is 0
.....In Child #1, count is 1
In Child #2, count is 1

```

```

In Child #3, count is 1
.....In Child #1, count is 2
In Child #2, count is 2
In Child #3, count is 2
.....In Child #1, count is 3
In Child #2, count is 3
In Child #3, count is 3
.....In Child #1, count is 4
In Child #2, count is 4
In Child #3, count is 4
.....In Child #1, count is 5
In Child #2, count is 5
In Child #3, count is 5
.....In Child #1, count is 6
In Child #2, count is 6
In Child #3, count is 6
.....In Child #1, count is 7
In Child #2, count is 7
In Child #2, count is 7
.....In Child #1, count is 8
In Child #2, count is 8
In Child #3, count is 8
.....In Child #1, count is 9
Child #1 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #3, count is 9
Child #3 terminating.
Main thread ending.

```

Gördüğünüz gibi, bir kez çalışmaya başladıkten sonra çocuk kanalların üçü de CPU'yu paylaşmaktadır. Sistem konfigürasyonları, işletim sistemleri ve diğer çevresel etmenler arasındaki farklılıklar yüzünden programınızı çalıştırığınızda sizin göreceğiniz çıktı burada gösterilenden bir parça farklı olabilir.

## Çalışma Kanalının Ne Zaman Sona Ereceğiini Belirlemek

Bir çalışma kanalının ne zaman sona ermiş olduğunu bilmek genellikle yararlıdır. Önceki örneklerde bu, `count` değişkeni izlenerek gerçekleştirılmıştı - bu hemen hemen hiç tatmin edici ya da genelleştirilebilir bir çözüm değildir. Neyse ki; `Thread`, bir kanalın sona erip ermediğini belirleyebilmeniz için iki yöntem sağlamaktadır. Birincisi, söz konusu kanal için, salt okunur bir özellik olan `IsAlive`'ı sorgulayabilirisiniz. Bu özellik şu şekilde tanımlanır:

```
public bool IsAlive { get; }
```

`IsAlive`'ın üzerinde çağrıldığı çalışma kanalı halen çalışıyor ise `IsAlive`, `true` döndürür. Aksi halde `false` döndürür. `IsAlive`'ı denemek için `MoreThreads`'in şu versiyonunu önceki programda gösterilenle değiştirin:

```
// Kanalların sona ermelerini beklemek için IsAlive kullanın.
class MoreThreads {
    public static void Main() {
        Console.WriteLine("Main thread starting.");

        // Üç kanal yapılandırılmıştır.
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        do {
            Console.Write(".");
            Thread.Sleep(100);
        }

        while (mt1.thrd.IsAlive &&
               mt2.thrd.IsAlive &&
               mt3.thrdIsAlive);

        Console.WriteLine("Main thread ending.");
    }
}
```

Bu versiyon önceki ile aynı çıktıyı üretir. İki arasındaki tek fark, çocuk kanalların sona ermelerini beklemek için bu versiyonda **IsAlive** kullanılmasıdır.

Bir çalışma kanalının sona ermesini beklemek için bir başka yol **Join()**'ı çağrılmaktır. Bunun en basit şekli aşağıda gösterilmiştir:

```
public void Join()
```

**Join()**, kendisinin üzerinde çağrıldığı kanal sona erene kadar bekler. "Join (katılma)" ismi şu kavramdan gelmektedir: Çağrıda bulunan kanal, belirtilen kanal kendisine *katılana* kadar bekler.

Eğer söz konusu kanal başlatılmamışsa bir **ThreadStateException** fırlatılacaktır. **Join()**'in ilave biçimleri, belirtilen kanalın sona ermesi için beklemek istediğiniz maksimum süreyi belirtmenize olanak tanır.

Temel kanalın sona eren en son kanal olduğunu garanti etmek için **Join()** kullanan bir program aşağıda gösterilmiştir:

```
// Join() kullanır.

using System;
using System.Threading;

class MyThread {
    public int count;
    public Thread thrd;

    public MyThread(string name) {
        count = 0;
```

```
        thrd = new Thread(new ThreadStart(this.run));
        thrd.Name = name;
        thrd.Start();
    }

    // Kanalin giris noktası.
    void run() {
        Console.WriteLine(thrd.Name + " starting.");

        do {
            Thread.Sleep(500);
            Console.WriteLine("In " + thrd.Name +
                ", count is " + count);
            count++;
        }

        while(count < 10);

        Console.WriteLine(thrd.Name + " terminating.");
    }
}

// Kanallarin sona ertmesini beklemek icin Join() kullan.
class JoinThreads {
    public static void Main() {
        Console.WriteLine("Main thread starting.");

        // Uc kanal yapılandır.
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        mt1.thrd.Join();
        Console.WriteLine("Child #1 joined.");

        mt2.thrd.Join();
        Console.WriteLine("Child #2 joined.");

        mt3.thrd.Join();
        Console.WriteLine("Child #3 joined.");

        Console.WriteLine("Main thread ending.");
    }
}
```

Bu programdan elde edilen örnek çıktı aşağıda gösterilmiştir. Programı denediğinizde sizin çıktılarınızın biraz farklı olabileceğini hatırlayın.

```
Main thread starting.
Child #1 starting.
Child #2 starting.
Child #3 starting.
In Child #1, count is 0
In Child #2, count is 0
In Child #3, count is 0
```

```
In Child #1, count is 1
In Child #2, count is 1
In Child #3, count is 1
In Child #1, count is 2
In Child #2, count is 2
In Child #3, count is 2
In Child #1, count is 3
In Child #2, count is 3
In Child #3, count is 3
In Child #1, count is 4
In Child #2, count is 4
In Child #3, count is 4
In Child #1, count is 5
In Child #2, count is 5
In Child #3, count is 5
In Child #1, count is 6
In Child #2, count is 6
In Child #3, count is 6
In Child #1, count is 7
In Child #2, count is 7
In Child #3, count is 7
In Child #1, count is 8
In Child #2, count is 8
In Child #3, count is 8
In Child #1, count is 9
Child #1 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #3, count is 9
Child #3 terminating.
Child #1 joined.
Child #2 joined.
Child #3 joined.
Main thread ending.
```

Gördüğünüz gibi, `Join()`'e yapılan çağrılar geri döndükten sonra kanallar çalışmayı durdurmuşlardır.

## IsBackground Özelliği

Önceden bahsedildiği gibi, .NET Framework iki tür kanal tanımlamaktadır: ön plan ve arka plan. Bu ikisi arasındaki tek fark şudur: Bir proses tüm ön plan kanalları sona erene kadar sonlanmayacaktır; fakat, arka plan kanalları, tüm ön plan kanalları duruktan sonra sona erer. Bir kanalın varsayılan oluşturulma şekli ön plan kanalı biçimindedir. Fakat bu, `Thread` tarafından tanımlanan ve aşağıda gösterilen `IsBackground` özelliği kullanılarak arka plan kanalına dönüştürülebilir:

```
public bool IsBackground { get; set; }
```

Bir kanalı arka plan kanalı olarak ayarlamak için yalnızca `IsBackground`'a `true` değeri atamanız yeterlidir. `false` değeri ön plan kanalına işaret eder.

## Kanal Öncelikleri

Her çalışma kanalı kendisiyle ilintili bir öncelik ayarlamasına sahiptir. Bir kanalın önceliği, kısmen, bir kanalın ne kadar CPU zamanı alacağını belirler. Genel olarak, düşük öncelikli kanallar az CPU zamanı alırlar. Yüksek öncelikli kanallar çok CPU zamanı gerektirirler. Tahmin edebileceğiniz gibi, bir kanalın ne kadar CPU zamanı aldığı kanalın çalışma özelliklerini ve sistemde halen çalışmakta olan diğer kanallarla etkileşimini derinden etkiler.

Bir kanalın ne kadar CPU zamanı alacağını kanalın önceliğinden başka etmenlerin de etkileyebileceğini kavramak önemlidir. Söz gelişi, yüksek öncelikli bir kanal eğer bir kaynağı, belki klavyeden gelecek girdiyi bekliyorsa, bu kanal bloke edilecek ve daha düşük önceliğe sahip bir kanal çalışacaktır. Böylece, bu durumda düşük öncelikli kanal, belirli bir süreç içinde yüksek öncelikli kanala kıyasla daha fazla CPU erişimi kazanabilir.

Bir çocuk kanal başlatılınca bu kanal bir varsayılan öncelik ayarlaması alır. Bir kanalın önceliğini, **Thread**'in bir üyesi olan **Priority** özelliği aracılığıyla değiştirebilirsiniz. Bu özelliğin genel olarak kullanımı şu şekildedir:

```
public Thread Priority Priority{ get; set; }
```

**ThreadPriority**, aşağıdaki beş öncelik ayarlamasını tanımlayan bir numaralandırmadır:

```
ThreadPriority.Highest  
ThreadPriority.AboveNormal  
ThreadPriority.Normal  
ThreadPriority.BelowNormal  
ThreadPriority.Lowest
```

Bir kanal için varsayılan öncelik **ThreadPriority.Normal**'dır.

Önceliklerin, kanalların çalışmasını nasıl etkilediğini anlamak için biri diğerinden daha yüksek önceliğe sahip iki kanal çalıştırın bir örnek kullanacağız. Söz konusu kanallar **MyThread** sınıfının örnekleri olarak oluşturulurlar. **run()** metodu tekrarlamaların sayısını sayan bir döngü içerir. Döngü ya toplam **1,000,000,000**'a ulaşınca ya da **static** değişken **stop**, **true** değerine sahip olunca durur Başlangıçta **stop**, **false** olarak ayarlanır. **1,000.000,000**'a sayan ilk kanal **stop**'a **true** değerini verir. Bu durum ikinci kanalın bir sonraki zaman diliminde sona ermesine neden olur. Döngünün her tekrarında **currentName** içindeki karakter katarı, çalışmakta olan kanalın ismi ile karşılaştırılır. Eğer bunlar eşleşmiyorsa, bir görev değişimi meydana geldi demektir. Her görev değişikliğinde yeni kanalın ismi ekranda görüntülenir ve **currentName**'e yeni kanalın adı verilir. Bu, her kanalın CPU'ya hangi sıklıkla eriştiğini izlemenize imkan verir. Her iki kanal da duruktan sonra her döngünün tekrar sayısı ekranda gösterilir.

```
// Kanal önceliklerini gösterir.  
  
using System;  
using System.Threading;
```

```
class MyThread {
    public int count;
    public Thread thrd;

    static bool stop = false;
    static string currentName;

    /* Yeni bir kanal yapılandır. Dikkat ederseniz,
       bu yapılandırıcı aslında çalışmaktadır
       kanalları başlatmıyor. */
    public MyThread(string name) {
        count = 0;
        thrd = new Thread(new ThreadStart(this.run));
        thrd.Name = name;
        currentName = name;
    }

    // Yeni kanalın çalışmasını başlat.
    void run() {
        Console.WriteLine(thrd.Name + " starting.");
        do {
            count++;

            if(currentName != thrd.Name) {
                currentName = thrd.Name;
                Console.WriteLine("In " + currentName);
            }
        } while(stop == false && count < 1000000000);
        stop = true;

        Console.WriteLine(thrd.Name + " terminating.");
    }
}

class PriorityDemo {
    public static void Main() {
        MyThread mt1 = new MyThread("High Priority");
        MyThread mt2 = new MyThread("Low Priority");

        // Oncelik sırasını ayarla.
        mt1.thrd.Priority = ThreadPriority.AboveNormal;
        mt2.thrd.Priority = ThreadPriority.BelowNormal;

        // Kanalları başlat.
        mt1.thrd.Start();
        mt2.thrd.Start();

        mt1.thrd.Join();
        mt2.thrd.Join();

        Console.WriteLine();
        Console.WriteLine(mt1.thrd.Name + " thread counted to " +
                        mt1.count);
        Console.WriteLine(mt2.thrd.Name + " thread counted to " +
                        mt2.count);
    }
}
```

```
    }  
}
```

Program, 1 GHz Pentium tabanlı bir bilgisayarda, Windows 2000 işletim sistemi altında çalıştırıldığında aşağıdaki çıktı elde edilmiştir:

```
High Priority starting.  
In High Priority  
Low Priority starting.  
In Low Priority  
In High Priority  
In Low Priority  
In High Priority  
In Low Priority  
In High Priority  
In Low Priority  
In High Priority  
In Low Priority  
In High Priority  
High Priority terminating.  
Low Priority terminating.  
  
High Priority thread counted to 1,000,000,000  
Low Priority thread counted to 25,600,064
```

Programın bu çalışmasında yüksek öncelikli kanal, CPU zamanının yaklaşık yüzde 98'ini elinde bulundurdu. CPU'nuzun hızına ve sistemde çalışmakta olan diğer görevlerin sayısına bağlı olarak sizin gördüğünüz tam çıktı elbette farklılık gösterebilir. Windows'un hangi versiyonunu çalıştırığınız bile sonucu etkileyebilmektedir.

Çok kanallı kod farklı ortamlarda farklı davranışları olduğu için, kodunuzu asla tek bir ortamın çalışma özelliklerine dayandırmamalısınız. Örneğin, önceki örnekte yüksek öncelikli kanal sona ermeden önce düşük öncelikli kanalın daima, en azından, kısa bir süre çalışacağını varsayılmak yanlış olur. Farklı bir ortamda, örneğin düşük öncelikli kanal henüz bir kez bile çalışmadan önce yüksek öncelikli kanal çalışmasını tamamlayabilir.

## Senkronizasyon

Birden fazla çalışma kanalı kullanılırken iki ya da daha fazla kanalın faaliyetlerinin bir program tarafından koordine edilmesi kimi zaman gereklidir. Bu tür bir durumun elde edildiği süreç *senkronizasyon* (*synchronization*) olarak adlandırılır. Senkronizasyon kullanımının en yaygın gereklisi, bir anda yalnızca tek bir kanal tarafından kullanılabilen bir kaynağa iki ya da daha fazla kanalın erişmeleri gerektiğinde ortaya çıkar. Örneğin, bir kanal bir dosyaya yazarken, ikinci bir kanalın aynı anda aynı işlemi gerçekleştirmesi önlenmelidir. Senkronizasyonun gerekliliği bir başka durum, bir kanalın başka bir kanalın neden olduğu bir olayı beklemesidir. Bu durumda, olay meydana gelene kadar ilk kanalı asılı durumda tutabilecek bazı yöntemler gereklidir. Sonra, bekleyen kanal çalışmasını sürdürmelidir.

Senkronizasyonun en temel özelliği *kilit* (*lock*) kavramıdır. Kilit, bir nesne içindeki kod bloğuna erişimi kontrol eder. Bir nesne bir kanal tarafından kilitlenince diğer kanalların hiçbir kilitli kod bloğuna erişim elde edemez. Söz konusu kanal kilidi kaldırınca nesne bir başka kanalın kullanımına hazırlıdır.

Kilit özelliği C# diline entegre standart bir özelliktir. Bu sayede, tüm nesneler senkronize edilebilir. Senkronizasyon **lock** anahtar kelimesi ile desteklenir. Senkronizasyon, C#'ın en başında tasarlanmış olduğu için, kullanımını beklediğinizden çok daha kolaydır. Aslında, program için nesnelerin senkronizasyonu hemen hemen açıkrtır.

**lock**'un genel olarak kullanımı aşağıda gösterilmiştir:

```
lock(nesne) {
    // senkronize edilecek ifadeler
}
```

Burada **nesne**, senkronize edilmekte olan nesneye bir referanstır. Yalnızca tek bir ifadeyi senkronize etmek istiyorsanız, küme parantezlerine gerek yoktur. **lock** ifadesi, verilen nesne için kilit ile korunan kod bölümünün yalnızca kilidi alan kanal tarafından kullanılabileceğini garanti eder. Kilit kaldırılana kadar diğer tüm kanallar bloke edilir. Kilit, bloktan çıkışınca kaldırılır.

Aşağıdaki program, **sumIt()** adında bir metoda erişimi kontrol ederek senkronizasyonun kullanımını göstermektedir. **sumIt()**, bir tamsayı dizisinin elemanlarını toplar:

```
// Bir nesneye erisimi senkronize etmek icin kilit kullanir.

using System;
using System.Threading;

class SumArray {
    int sum;

    public int sumIt(int[] nums) {
        lock(this) { // butun metodu kilitle.
            sum = 0; // toplami sifirla.

            for(int i = 0; i < nums.Length; i++) {
                sum += nums[i];
                Console.WriteLine("Running total for " +
                    Thread.CurrentThread.Name +
                    " is " + sum);
                Thread.Sleep(10); // gorev degisimine olanak tanri.
            }
            return sum;
        }
    }

    class MyThread {
        public Thread thrd;
        int[] a;
        int answer;
```

```
/* MyThread'in tum ornekleri icin SumArray nesnesi
   olustur. */
static SumArray sa = new SumArray();

// Yeni bir kanal yapılandır.
public MyThread(string name, int[] nums) {
    a = nums;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start(); // kanali baslat.
}

// Yeni kanalin calismasini baslat.
void run() {
    Console.WriteLine(thrd.Name + " starting.");
    answer = sa.sumIt(a);

    Console.WriteLine("Sum for " + thrd.Name +
                      " is " + answer);

    Console.WriteLine(thrd.Name + " terminating.");
}
}

class Sync {
    public static void Main() {
        int[] a = {1, 2, 3, 4, 5};

        MyThread mt1 = new MyThread("Child #1", a);
        MyThread mt2 = new MyThread("Child #2", a);

        mt1.thrd.Join();
        mt2.thrd.Join();
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Running total for Child #2 is 1
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

Cıktıdan da görüldüğü gibi, her iki kanal da doğru toplamı, 15'i hesaplamaktadır.

Gelin şimdi programı ayrıntılı olarak inceleyelim. Program, üç sınıf oluşturur. İlkî **SumArray**'dır. **SumArray**, bir tamsayı dizisinin elemanlarını toplayan **sumIt()** metodunu tanımlar. İkinci sınıf **MyThread**'dır. **MyThread**, **SumArray** tipinde **sa** olarak adlandırılan **static** bir nesne kullanır. Böylece, **SumArray**'in yalnızca bir nesnesi **MyThread** tipindeki tüm nesneler tarafından paylaşıılır. Bu nesne, bir tamsayı dizisinin elemanlarının toplamını elde etmek amacıyla kullanılır. Dikkat ederseniz, **SumArray**, ara toplamları **sum** adında bir alanda saklamaktadır. Böylece, eğer iki kanal **sumIt()**'i eşzamanlı olarak kullanırsa bu kanalların her ikisi de **sum**'ı ara toplamı tutması için kullanmaya çalışacaktır. Bu durum hatalara neden olacağı için **sumIt()**'e erişim senkronize edilmelidir. Son olarak, **Sync** sınıfı iki çalışma kanalı oluşturur ve bu kanallara tamsayı dizisinin toplamını hesaplatır.

**sumIt()**'in içinde **lock** ifadesi, metodun farklı kanallar tarafından eşzamanlı kullanımını önlüyor. **lock**'un senkronize edilecek nesne olarak **this**'i kullandığına dikkat edin. Çağrıda bulunan nesne kilitleneceği zaman **lock** normal olarak bu şekilde çağrılır. **Sleep()**, eğer mümkünse - bu örnekte mümkün değildir - kasten bir görev değişimine olanak tanımak için çağrılar. **sumIt()**'in içindeki kod kilitli olduğu için bir anda yalnızca tek bir kanal tarafından kullanılabilir. Böylece, ikinci çocuk kanalı çalışmaya başlayınca birinci çocuk kanalının **sumIt()**'le işi bitene kadar ikinci çocuk kanalı **sumIt()**'e girmez. Bu, doğru sonucun üretilmesini garanti eder.

**lock**'un etkilerini tam olarak anlamak için **sumIt()**'in gövdesinden bunu çıkartmayı deneyin. Bunu gerçekleştirdikten sonra, **sumIt()** artık senkronize edilemez ve bunu herhangi bir sayıda kanal aynı nesne üzerinde eşzamanlı olarak kullanabilir. Buradaki problem şudur: **sum** içinde saklanan ara toplamlar **sumIt()**'i çağrıran kanalların her biri tarafından değiştirilecektir. Böylece, iki çalışma kanalı **sumIt()**'i aynı anda aynı nesne üzerinde çağırınca, hatalı sonuçlar üretilir, çünkü **sum**, her iki kanalın birbirine karışmış toplamlarını yansıtmaktadır.

Örneğin, **sumIt()**'ten **lock** kaldırıldıktan sonra programdan alınan örnek çıktı şöyledir:

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #1 is 8
Running total for Child #2 is 11
Running total for Child #1 is 15
Running total for Child #2 is 19
Running total for Child #1 is 24
Running total for Child #2 is 29
Sum for Child #1 is 29
Child #1 terminating.
Sum for Child #2 is 29
Child #2 terminating.
```

Cıktıdan da görüldüğü gibi, her iki çocuk kanal da `sumIt()`'yı aynı anda aynı nesne üzerinde kullanıyorlar ve `sum`'ın değeri bozuluyor.

`lock`'un etkileri aşağıda özetlenmiştir:

1. Verilen herhangi bir nesne için, bir kod bölümüne bir kez bir kilit yerleştirildikten sonra artık nesne kilitlenmiştir ve diğer kanalların hiçbirini kilidi ele geçiremez.
2. Aynı nesne üzerindeki kilidi ele geçirmeye çalışan diğer kanallar kodun kilidi çözüleme kadar bekleme durumuna gireceklerdir.
3. Bir kanal kilitli bloğu terk edince nesnenin kilidi çözülür.

`lock` ile ilgili anlaşılması gereken bir diğer husus şudur: `lock` yalnızca `private` ya da `internal` nesneler üzerinde kullanılmalıdır. Eğer bu tür bir durum söz konusu değilse, programınızın dışındaki herhangi bir program kanalı kilidi ele geçirip, bırakmayabilir.

## Alternatif Yaklaşım

Bir metodun kodunu kilitlemek, önceki örnekte gösterildiği gibi, senkronizasyonu sağlamak için kolay ve etkin bir yol olmasına rağmen her durumda çalışmayıabilir. Söz gelişi, kendinizin oluşturmadığı bir sınıfın senkronize edilmemiş bir metoduna erişimi senkronize etmek isteyebilirsiniz. Üçüncü parti bir sınıfı kullanmak isterseniz ve bu sınıfın kaynak koduna erişim iznine sahip değilseniz böyle bir durum ortaya çıkabilir. Böylece, söz konusu sınıfın içindeki ilgili metoda bir `lock` ifadesi eklemeniz mümkün olmayabilir. Bu sınıfın bir nesnesine erişim nasıl senkronize edilebilir? Neyse ki, bu problemin çözümü basittir: Nesneyi bir `lock` ifadesi içinde belirterek nesnenin dışındaki kodun nesneye erişimini kilitleyin. Örneğin, önceki programın alternatif uygulaması aşağıda yer almaktadır. Dikkat ederseniz, `sumIt()`'in içindeki kod artık kilitli değildir. Bunun yerine, `MyThread` içinden `sumIt()`'e yapılan çağrılar kilitlenmiştir.

```
/* Nesnenin erisimini senkronize etmek icin kilit kullaniminin  
bir baska sekli. */  
  
using System;  
using System.Threading;  
  
class SumArray {  
    int sum;  
  
    public int sumIt(int[] nums) {  
        sum = 0; // toplami sifirla.  
  
        for(int i = 0; i < nums.Length; i++) {  
            sum += nums[i];  
            Console.WriteLine("Running total for " +  
                Thread.CurrentThread.Name +  
                " is " + sum);  
            Thread.Sleep(10); // gorev degisimine olanak tanri.
```

```

        }

        return sum;
    }
}

class MyThread {
    public Thread thrd;
    int[] a;
    int answer;

    /* MyThread'ın tüm örnekleri için bir SumArray nesnesi
       oluştur. */
    static SumArray sa = new SumArray();

    // Yeni bir kanal yapılandır.
    public MyThread(string name, int[] nums) {
        a = nums;
        thrd = new Thread(new ThreadStart(this.run));
        thrd.Name = name;

        thrd.Start(); // kanalı başlat.
    }

    // Yeni kanalın çalışmasını başlat.
    void run() {
        Console.WriteLine(thrd.Name + " starting.");

        // sumIt()'e yapılan çağrıları kilitle.
        lock(sa) answer = sa.sumIt(a);

        Console.WriteLine("Sum for " + thrd.Name +
                          " is " + answer);

        Console.WriteLine(thrd.Name + " terminating.");
    }
}

class Sync {
    public static void Main() {
        int[] a = {1, 2, 3, 4, 5};

        MyThread mt1 = new MyThread("Child #1", a);
        MyThread mt2 = new MyThread("Child #2", a);

        mt1.thrd.Join();
        mt2.thrd.Join();
    }
}

```

Burada **sumIt()**'in içindeki kodun kendisinin yerine **sa.sumIt()**'e yapılan çağrı kilitlenmiştir. Bunu gerçekleştiren kod aşağıda gösterilmiştir:

```
// sumIt()'e yapılan çağrıları kilitle.
lock(sa) answer = sa.sumIt(a);
```

Program, orijinal yaklaşımla aynı doğru sonuçları üretmektedir.

## Bir Statik Metodu Kilitlemek

**lock**, bir nesne ile birlikte çalıştığı için ilk bakışta bir **static** metot içindeki kodu kilitleyemeyeceğinizi düşünebilirsiniz, çünkü kilitlenecek bir nesne yoktur. Ancak bu yanlıştır. Bir **static** metodu kilitlemek için aşağıdaki **lock** şéklini kullanın:

```
lock(typeof(sınıf)) {  
    // kilitlenen blok  
}
```

Burada **sınıf**, **static** metodu içeren sınıfın adıdır.

## Monitör Sınıfı ve lock

C#'ın **lock** anahtar kelimesi aslında **Monitor** sınıfı tarafından tanımlanan senkronizasyon özelliklerini kullanmak için yalnızca bir kestirme yoldur. **Monitor**, **System.Threading** isim uzayında tanımlanmıştır. **Monitor** sınıfında senkronizasyonu kontrol eden ya da yöneten birkaç metot tanımlıdır. Örneğin, bir nesne üzerindeki kilidi elde etmek için **Enter()**'ı çağırın. Bir kilidi kaldırmak için **Exit()**'ı çağırın. Bu metodlar aşağıda gösterilmiştir;

```
public static void Enter(object senkNes)  
public static void Exit(object senkNes)
```

Burada **senkNes**, senkronize edilmekte olan nesnedir. **Enter()** çağrıldığı zaman söz konusu nesne hazır değilse, çağrıda bulunan kanal nesne hazır olana dek bekleyecektir. Microsoft, bir **lock** bloğunun, **Enter()** ve sonra **Exit()**'ı çağrımakla "tamamen eşdeğer" olduğunu bildirmektedir. **lock**, C# içinde standart olarak mevcuttur. Bu nedenle **lock**, C# ile programlama yaparken bir nesne üzerinde kilit elde etmek için tercih edilen bir metottur.

**Monitor** içindeki kullanışlı metodlardan biri **TryEnter()**'dır. Bu metodun kullanım şekillerinden biri aşağıda gösterilmiştir:

```
public static bool TryEnter(object senkNes)
```

Çağrıda bulunan çalışma kanalı **senkNes** üzerinde bir kilit elde ederse metot **true** döndürür. Aksi halde, **false** döndürür. Çağrıda bulunan kanal hiç bir durumda beklemez. İstenilen nesnenin hazır olmadığı durumlarda bu metodu bir alternatif olarak kullanabilirsiniz.

**Monitor**'de ayrıca şu üç metot da tanımlıdır: **Wait()**, **Pulse()** ve **PulseAll()**. Bu metodlar aşağıda anlatılmıştır.

## Wait(), Pulse() ve PulseAll() Kullanarak Kanal İletişimi

Şöyledir bir durum düşünün. **T** adında bir çalışma kanalı bir lock bloğu içinde çalışıyor ve geçici olarak kullanıma hazır **R** adında bir kaynağa erişmek istiyor. **T** ne yapmalıdır? Eğer **T**, **R**'yi bekleyen bir tür yoklama döngüsüne girerse, bu durumda **T**, nesneyi bağlar ve diğer kanalların nesneye erişimini de bloke eder. Bu, optimal çözüm kadar iyi bir çözüm değildir, çünkü çok kanallılık ortamı için gerçekleştirilen programlamanın avantajlarını kısmen mahveder. Daha iyi bir çözüm, **T**'nin geçici olarak nesnenin kontrolünü elinden bırakarak, bir başka kanalın çalışmasına olanak tanımıştır. **R** elverişli olduğunda **T** durumdan haberdar edilir ve çalışmasını kaldığı yerden sürdürbilir. Böyle bir yaklaşım, kanallar arası bir tür iletişimde dayanır. Bu iletişimde, bir kanal bloke edildiğini bir diğerine haber verebilir ve çalışmasını sürdürileceği zaman kendisi durumdan haberdar edilebilir. C# kanallar arasındaki iletişimini **Wait()**, **Pulse()** ve **PulseAll()** metodlarıyla destekler.

**Wait()**, **Pulse()** ve **PulseAll()** metodları **Monitor** sınıfı tarafından tanımlanmıştır. Bu metodlar yalnızca kilitli kod bloğu içinden çağrılabılır. Şu şekilde kullanılır: Bir çalışma kanalının çalışması geçici olarak bloke edilmişse, kanal **Wait()**'ı çağırır. Bu, kanalın uykuya geçmesine neden olur ve ilgili nesnenin kilidi kaldırılır. Böylece, bir başka kanalın nesneyi kullanmasına imkan verilmiş olur. Daha sonraki bir aşamada, bir başka kanal aynı kilide girmeye çalışınca ve **Pulse()** ya da **PulseAll()**'ı çağrıncı uyuyan kanal uyandırılır. **Pulse()**'a yapılan bir çağrı, kilidi bekleyen çalışma kanalları kuyruğundaki ilk kanalın çalışmasını kaldığı yerden sürdürmesini sağlar. **PulseAll()**'a yapılan bir çağrı, kilden kaldırıldığı bekleyen kanalların tümüne işaret eder.

**Wait()**'in en yaygın olarak kullanılan iki şekli şöyledir:

```
public static bool Wait(object bekleNes)
public static bool Wait(object bekleNes, int milisaniye)
```

İlk ifadedeki **Wait()**, kendisine haber verilene kadar bekler. İkinci ifadedeki **Wait()** ise, haber verilene kadar ya da belirtilen **milisaniye** süresi dolana kadar bekler. Her ikisi için de **bekleNes**, üzerinde beklenilecek olan nesneyi belirtir.

**Pulse()** ve **PulseAll()**'ın genel kullanımları şu şekildedir:

```
public static void Pulse(object bekleNes)
public static void PulseAll(object bekleNes)
```

Burada **bekleNes**, serbest bırakılmakta olan nesnedir.

Eğer **Wait()**, **Pulse()** ya da **PulseAll()** bir **lock** bloğun içinde olmayan koddan çağrırlarsa bir **SynchronizationLockException** fırlatılacaktır.

## Wait () ve Pulse () Kullanan Bir Örnek

**Wait()** ve **Pulse()** için duyulan gereksinimi ve bunların uygulamalarını anlamak için ekranda “Tick” ve “Tock” sözcüklerini görüntüleyerek bir saatin çalışmasını canlandıran bir program geliştireceğiz. Bunu gerçekleştirmek için iki metot içeren **TickTock** adında bir sınıf oluşturacağız. Söz konusu metodlar **tick()** ve **tock()** olacak. **tick()** metodu, “Tick” sözcüğünü ve **tock()** da “Tock” sözcüğünü ekranda gösterecek. Saati çalıştırmak için biri **tick()**’i çağırın ve diğeri de **tock()**’u çağırın iki çalışma kanalı oluşturulur. Burada amaç şudur: İki kanalının işleyişini öyle ayarlamalıdır ki, ekranda program çıktısı olarak tutarlı ve sürekli bir “Tick Tock” görüntüsü - yani, her “Tick”ten sonra bir “Tock”un geldiği tekrarlayan bir desen - elde edilebilisin.

```
// Tikleyen bir saat olusturmak icin Wait() ve Pulse() kullanir.

using System;
using.System.Threading;

class TickTock {

    public void tick(bool running) {
        lock(this) {
            if(!running) { // saati durdur
                Monitor.Pulse(this); // bekleyen kanallari haberدار et
                return;
            }

            Console.WriteLine("Tick ");
            Monitor.Pulse(this); // tock()'un calismasina izin ver

            Monitor.Wait(this); // tock()'un tamamlanmasini bekle
        }
    }

    public void tock(bool running) {
        lock(this) {
            if(!running) { // saati durdur
                Monitor.Pulse(this); // bekleyen kanallari haberدار et
                return;
            }

            Console.WriteLine("Tock");
            Monitor.Pulse(this); // tick()'in calismasina izin ver

            Monitor.Wait(this); // tick()'in tamamlanmasini bekle
        }
    }
}

class MyThread {
    public Thread thrd;
    TickTock ttOb;
```

```

// Yeni bir kanal yapılandır.
public MyThread(String name, TickTock tt) {
    thrd = new Thread(new ThreadStart(this.run));
    ttOb = tt;
    thrd.Name = name;
    thrd.Start();
}

// Yeni kanalın çalışmasını başlat.
void run() {
    if(thrd.Name == "Tick") {
        for(int i = 0; i < 5; i++) ttOb.tick(true);
        ttOb.tick(false);
    }
    else {
        for(int i = 0; i < 5; i++) ttOb.tock(true);
        ttOb.tock(false);
    }
}

class TickingClock {
    public static void Main() {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("Tick", tt);
        MyThread mt2 = new MyThread("Tock", tt);

        mt1.thrd.Join();
        mt2.thrd.Join();
        Console.WriteLine("Clock Stopped");
    }
}

```

Programın ürettiği çıktı şu şekildedir:

```

Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Clock Stopped

```

Gelin şimdi bu programa yakından göz atalım. **Main()**'de **tt** adında bir **TickTock** nesnesi oluşturulur; iki çalışma kanalının başlaması için bu nesne kullanılır. **MyThread**'in **run()** metodu içinde, eğer çalışma kanalının ismi "Tick" ise **tick()**'e çağrıda bulunulur. Kanalın ismi "Tock" ise **tock()** metodu çağrılır. Her metoda **true** değerinde bir argüman olarak aktarılan beş çağrı yapılır. Saat, **true** aktarıldığı sürece çalışır.

Metotların her birine **false** değeri aktaran son çağrı saatı durdurur.

Programın en önemli bölümü **tick()** ve **tock()** metodlarında yer almaktadır. **tick()** metodu ile başlayacağız. Kolaylık olsun diye **tick()** metodu aşağıda gösterilmiştir:

```

public void tick(bool running) {
    lock(this) {
        if(!running) { // saatı durdur
            Monitor.Pulse(this); // bekleyen kanalları haberدار et
            return;
        }

        Console.WriteLine("Tick ");

        Monitor.Pulse(this); // tock()'un çalışmasına izin ver

        Monitor.Wait(this); // tock()'un tamamlanmasını bekle
    }
}

```

Öncelikle, **tick()** içindeki kodun bir **lock** bloğu içinde yer aldığına dikkat edin. **Wait()** ve **Pulse()**'ın yalnızca senkronize edilmiş blokların içinde kullanılabileceğini hatırlayın. Metot, çalışmakta olan parametrenin değerini kontrol ederek başlar. Bu parametre saatin düzgün biçimde kapatılmasını sağlamak için kullanılır. Eğer parametre **false** ise saat durdurulmuştur. Böyle bir durum söz konusu ise, bekleyen herhangi bir kanalın çalışmasını mümkün kılmak için **Pulse()**'a bir çağrıda bulunulur. Bu noktaya birazdan geri döneceğiz. Saatin **tick()** gerçeklendiğinde çalıştığını varsayıarak ekranda "Tick" sözcüğü görüntülenir ve sonra **Pulse()**'a bir çağrıda bulunulur; bu çağrıyı **Wait()**'e yapılan çağrı takip eder. **Pulse()**'a yapılan çağrı, aynı nesne üzerinde bekleyen bir kanalın çalışmasına olanak tanır. **Wait()**'e yapılan çağrı ise, bir başka kanal **Pulse()**'ı çağrıiana kadar **tick()**'in askıda kalmasına neden olur. Böylece, **tick()** çağrıldığı zaman ekranda tek bir "Tick" gösterir, bir başka kanalın çalışmasına izin verir, sonra da askıya alınır.

**tock()** metodu, **tick()**'in birebir kopyasıdır; yalnızca, ekranda "Tick" yerine "Tock" görüntüler. Bu nedenle, **tock()** metoduna girilince, metot ekranda "Tock" sözcüğünü görüntüler. **Pulse()**'ı çağrıır ve sonra bekler. Çift olarak bakıldığından **tick()**'e yapılan çağrıyı yalnızca **tock()**'a yapılan çağrı takip edebilir; **tock()**'a yapılan çağrıının ardından da yalnızca **tick()**'e yapılan çağrı gelebilir vs. Bu sayede, her iki metot karşılıklı olarak senkronize edilmiş olur.

Saat durduğunda **Pulse()**'ın çağrılmamasının nedeni, **Wait()**'e yapılan son çağrıının başarıyla tamamlanmasına imkan vermektedir. Hatırlarsanız, **tick()** ve **tock()**'un her ikisi de kendi mesajlarını görüntüledikten sonra **Wait()**'e çağrıda bulunurlar. Buradaki problem, saat durduğunda metotlardan birinin halen bekliyor olmasıdır. Bu nedenle, bekleyen metodun çalışması için **Pulse()**'a yapılan son çağrı gereklidir. Aşılıtmış olarak **Pulse()**'a yapılan bu çağrıyı kaldırmayı deneyin ve ne olduğunu izleyin. Göreceksize ki, program "askıda" kalacaktır. Çıkmak için **CTRL-C** tuşlarına basmak zorunda kalacaksınız. Bunun gerekçesi, **tock()**'a yapılan son çağrı **Wait()**'ı çağrılığında, bu çağrıya karşılık gelen ve **tock()**'un sona ermesine imkan verecek bir **Pulse()** çağrısının mevcut olmamasıdır. Yani, **tock()** sonsuza kadar bekleyerek orada öylece kalır.

Daha fazla ilerlemeden önce, “saatin” doğru çalışmasını sağlamak için **Wait()** ve **Pulse()** çağrılarının gerçekten gerekli olduğu konusunda şüpheleriniz varsa **TickTock**’un aşağıdaki versiyonunu önceki programdaki versiyonunun yerine yerleştirin. Bu versiyonda **Wait()**’e ve **Pulse()**’a yapılan tüm çağrılar kaldırılmıştır.

```
// TickTock'un fonksiyonel olmayan versiyonu.
class TickTock {

    public void tick(bool running) {
        lock(this) {
            if(!running) { // saatı durdur
                return;
            }

            Console.WriteLine("Tick ");
        }
    }

    public void tock(bool running) {
        lock(this) {
            if(!running) { // saatı durdur
                return;
            }

            Console.WriteLine("Tock");
        }
    }
}
```

Bu değişikliğin ardından program tarafından üretilen çıktı aşağıdaki gibi görünecektir:

```
Tick Tick Tick Tick Tick Tock
Tock
Tock
Tock
Tock
Clock Stopped
```

**tick()** ve **tock()** metodlarının artık senkronize olmadıkları açıkça görülmektedir!

## Kilitlenme

Çok kanallı program geliştirirken kilitlenmeleri (deadlock) önlemek için çok dikkatli olmalısınız. *Kilitlenme*, adından da anlaşılacağı gibi, bir çalışma kanalının bir şeyler yapmak için başka bir kanalı beklemesi, ancak diğer kanalın da ilkini bekliyor olması durumudur. Bu yüzden, her iki kanal birbirini bekleyerek askıda kalır ve hiçbiri çalışmaz. Bu durum, aşırı kibar iki kişinin kapıdan bir diğerinin önce geçmesi için birbirlerine ısrar etmelerine benzer!

Kilitlenmeleri önlemek kolay gibi görünür fakat değildir. Söz gelisi, kilitlenme çok dolambaçlı şekillerde ortaya çıkabilir. **TickTock** sınıfını ele alın. Önceden açıklandığı gibi, **Pulse()**’a yapılan son çağrı **tick()** veya **tock()** tarafından gerçeklenmezse, bu

metotlardan biri ya da diğeri sonsuza kadar bekleyecektir; program kilitlenmiştir. Genellikle kilitlenmelerin nedeni programın kaynak koduna bakarak kolayca anlaşılmaz, çünkü programın çalışması sırasında eşzamanlı olarak gerçekleşmekte olan kanallar karmaşık şekillerde etkileşebilirler. Kilitlenmeleri önlemek için dikkatli programlama ve eksiksiz kontrol gereklidir. Genel olarak, eğer çok kanallı bir program arada sırada “askıda” kalıyorsa, büyük olasılıkla bunun nedeni kilitlenmedir.

## MethodImplAttribute Kullanımı

**MethodImplAttribute** niteliğini kullanarak bütün bir metodu senkronize etmek mümkündür. Bu yaklaşım, bir metodun bütün içeriğinin kilitlenmesi gereken durumlarda **lock** ifadesine alternatif olarak kullanılabilir. **MethodImplAttribute**, **System.Runtime.CompilerServices** isim uzayında tanımlanmıştır. Senkronizasyona uygulanan yapılandırıcı aşağıda gösterilmiştir:

```
public MethodImplAttribute(MethodImplOptions opt)
```

Burada **opt**, uygulama niteliğini belirtir. Bir metodу senkronize etmek için **MethodImplOptions.Synchronized**'ı belirtin. Bu nitelik, bütün metodun kilitlenmesine neden olur.

**TickTock** sınıfının senkronizasyon sağlamak için **MethodImplAttribute**'u kullanacak şekilde yeniden yazılmış versiyonu aşağıdaki gibidir:

```
// Bir metodу senkronize etmek icin MethodImplAttribute kullanir.

using System;
using System.Threading;
using System.Runtime.CompilerServices;

class TickTock {

    /* Asagidaki nitelik, butun tick() metodunu senkronize eder. */
    [MethodImplAttribute(MethodImplOptions.Synchronized)]
    public void tick(bool running) {
        if(!running) { // saati durdur
            Monitor.Pulse(this); // bekleyen kanallari haberدار et
            return;
        }

        Console.Write("Tick ");
        Monitor.Pulse(this); // tock()'un calismasina izin ver

        Monitor.Wait(this); // tock()'un tamamlanmasini bekle
    }

    /* Asagidaki nitelik, butun tick() metodunu senkronize eder. */
    [MethodImplAttribute(MethodImplOptions.Synchronized)]
    public void tock(bool running) {
        if(!running) { // saati durdur
```

```

        Monitor.Pulse(this); // bekleyen kanalları haberdar et
        return;
    }

    Console.WriteLine("Tock");
    Monitor.Pulse(this); // tick()'in çalışmasına izin ver

    Monitor.Wait(this); // tick()'in tamamlanmasını bekle
}
}

class MyThread {
    public Thread thrd;
    TickTock ttOb;

    // Yeni bir çalışma kanalı yapılandırır.
    public MyThread(string name, TickTock tt) {
        thrd = new Thread(new ThreadStart(this.run));
        ttOb = tt;

        thrd.Name = name;
        thrd.Start();
    }

    // Yeni kanalın çalışmasını başlat.
    void run() {
        if(thrd.Name == "Tick") {
            for(int i = 0; i < 5; i++) ttOb.tick(true);
            ttOb.tick(false);
        }
        else {
            for(int i = 0; i < 5; i++) ttOb.tock(true);
            ttOb.tock(false);
        }
    }
}

class TickingClock {
    public static void Main() {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("Tick", tt);
        MyThread mt2 = new MyThread("Tock", tt);

        mt1.thrd.Join();
        mt2.thrd.Join();
        Console.WriteLine("Clock Stopped");
    }
}

```

**TickTock**'tan elde edilen doğru çıktı öncekiyle aynıdır.

Bütün bir metodu kilitlerken **lock** ya da **MethodImplAttribute** kullanma tercihi size kalmıştır. Her ikisi de aynı sonucu verir. **lock**, C#'ta standart olarak tanımlı bir anahtar kelime olduğu için bu kitaptaki örneklerde bu yaklaşım kullanılacaktır.

## Kanalları Askıya Almak, Sürdürümek ve Durdurmak

Bir kanalın çalışmasını askıya almak kimi zaman yararlıdır. Örneğin, günün saatini görüntülemek için bir çalışma kanalı kullanılabilir. Eğer kullanıcı saat istemiyorsa, saatı çalıştırın kanal askıda kalabilir. Daha sonra, saat istenildiğinde ilgili kanalın çalışması sürdürülebilir. Ne tür bir durum söz konusu olursa olsun bir kanalı askıya almak ya da çalışmasını sürdürmek basit bir meseledir. Kimi zaman da bir kanalı durdurmak isteyeceksiniz. Bir çalışma kanalını durdurmak askıya almaktan farklıdır, çünkü durdurulan bir kanal sistemden çıkartılır ve yeniden başlatılmaz.

Bir çalışma kanalını askıya almak için **Thread.Suspend()**’ı kullanın. Askıya alınmış bir kanalın çalışmasını sürdürmek için **Thread.Resume()**’u kullanın. Bu metodların genel kullanım şekilleri aşağıda gösterilmiştir:

```
public void Suspend()  
public void Resume()
```

Eğer çağrıda bulunan kanal doğru durumda değilse, bu metodların her ikisi de bir **ThreadStateException** fırlatabilir. Örneğin, askıda olmayan bir kanalın çalışmasını sürdürmeye çalışmak kural dışı bir durumla sonuçlanır.

Bir kanalı durdurmak için **Thread.Abort()**’u kullanın. Bu metodun en basit kullanımı aşağıda gösterilmiştir:

```
public void Abort()
```

**Abort()**, kendisini çağrıran kanala bir **ThreadAbortException** fırlatılmasına neden olur. Bu kural dışı durum, söz konusu kanalın sona ermesine yol açar. Bu kural dışı durum sizin kodunuz tarafından da yakalanabilir (fakat kanalı durdurmak için otomatik olarak yeniden fırlatılır). **Abort()** bir kanalı her zaman anında durduramayabilir. Bu nedenle, programınız çalışmasına devam etmeden önce eğer bir çalışma kanalının durdurulması önemli ise, **Abort()** çağrısını bir **Join()** çağrı ile takip etmeniz gerekecektir. Ayrıca, ender durumlarda, **Abort()**’un bir kanalı durduramaması da mümkündür. Bir **finally** bloğu sonsuz döngüye girerse böyle bir durum ortaya çıkabilir.

Aşağıdaki örnek, bir çalışma kanalının nasıl askıya alınacağını, çalışmasının sürdürileceğini ve durdurulacağını göstermektedir:

```
// Bir calisma kanalini askiya alma, surdurme ve durdurma.  
  
using System;  
using System.Threading;  
  
class MyThread {  
    public Thread thrd;  
  
    public MyThread(string name) {  
        thrd = new Thread(new ThreadStart(this.run));  
    }
```

```

        thrd.Name = name;
        thrd.Start();
    }

    // Burası, kanalın giriş noktasıdır.
    void run() {
        Console.WriteLine(thrd.Name + " starting.");

        for(int i = 1; i <= 1000; i++) {
            Console.Write(i + " ");
            if((i % 10) == 0) {
                Console.WriteLine();
                Thread.Sleep(250);
            }
        }
        Console.WriteLine(thrd.Name + " exiting.");
    }
}

class SuspendResumeStop {
    public static void Main() {
        MyThread mt1 = new MyThread("My Thread");

        Thread.Sleep(1000); // çocuk kanalın çalışmaya başlamasına
                           // izin ver
        mt1.thrd.Suspend();
        Console.WriteLine("Suspending thread.");
        Thread.Sleep(1000);

        mt1.thrd.Resume();
        Console.WriteLine("Resuming thread.");
        Thread.Sleep(1000);

        mt1.thrd.Suspend();
        Console.WriteLine("Suspending thread.");
        Thread.Sleep(1000);

        mt1.thrd.Resume();
        Console.WriteLine("Resuming thread.");
        Thread.Sleep(1000);

        Console.WriteLine("Stopping thread.");
        mt1.thrd.Abort();

        mt1.thrd.Join(); // çalışma kanalının sona ermesini bekle
        Console.WriteLine("Main thread terminating.");
    }
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20

```

```
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
41 42 43 44 45 46 47 48 49 50
Resuming thread.
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
Suspending thread.
91 92 93 94 95 96 97 98 99 100
Resuming thread.
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130
131 132 133 134 135 136 137 138 139 140
Stopping thread.
Main thread terminating.
```

## Alternatif Abort()

Bazı durumlarda **Abort()**'un ikinci şeklini kullanışlı bulabilirsiniz. Genel olarak kullanımı aşağıda gösterilmiştir:

```
public void Abort(object bilgi)
```

Burada **bilgi**, bir kanal durdurulurken bu kanala aktarmak istediğiniz herhangi bilgiyi içerir. Bu bilgi, **ThreadAbortException**'ın **ExceptionState** özelliği aracılığıyla erişilebilir. Bunu, bir çalışma kanalına sonlandırma kodu aktarmak için kullanabilirsiniz. Aşağıdaki program **Abort()**'un bu kullanımını göstermektedir:

```
// Abort (object) kullanımını gösterir.

using System;
using System.Threading;

class MyThread {
    public Thread thrd;

    public MyThread(string name) {
        thrd = new Thread(new ThreadStart(this.run));
        thrd.Name = name;
        thrd.Start();
    }

    // Burası, kanalın giriş noktasıdır.
    void run() {
        try {
            Console.WriteLine(thrd.Name + " starting.");

            for(int i = 1; i <= 1000; i++) {
                Console.Write(i + " ");
                if((i % 10) == 0) {

```

```

        Console.WriteLine();
        Thread.Sleep(250);
    }
}
Console.WriteLine(thrd.Name + " exiting normally.");
}
catch(ThreadAbortException exc) {
    Console.WriteLine("Thread aborting, code is " +
                      exc.ExceptionState);
}
}
}

class UseAltAbort {
    public static void Main() {
        MyThread mt1 = new MyTnread("My Thread");

        Thread.Sleep(1000); // cocuk kanalin calismaya baslamasina
                           // izin ver
        Console.WriteLine("Stopping thread.");
        mt1.thrd.Abort (100);

        mt1.thrd.Join(); // calisma kanalinin sona ermesini bekle
        Console.WriteLine("Main thread terminating.");
    }
}

```

Çıktı aşağıdaki gibidir:

```

My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Stopping thread.
Thread aborting, code is 100
Main thread terminating.

```

Çıktıdan da görüldüğü gibi, **Abort()**'a 100 değeri aktarılmaktadır. Bu değer, daha sonra **ThreadAbortException**'ın **ExceptionState** özelliği aracılığıyla erişilir. **ThreadAbortException**, çalışma kanalı sona erken kanal tarafından yakalanır.

## Abort ()'u İptal Etmek

Bir kanal, bir durdurma isteğini devre dışı bırakabilir. Bunun için söz konusu kanal **ThreadAbortException**'ı yakalamalı ve sonra **ResetAbort()**'u çağrımalıdır. Bu, kanalın kural dışı durum yöneticisi sona erince kural dışı durumun otomatik olarak yeniden fırlatılmasına engel olur. **ResetAbort()** şu şekilde deklare edilir:

```
public static void ResetAbort()
```

Eğer söz konusu kanal, durdurma işlemini iptal etmek için doğru güvenlik ayarlarına sahip değilse **ResetAbort()**'a yapılan çağrı başarısız olabilir.

Aşağıdaki program **ResetAbort()**'u göstermektedir:

```
// ResetAbort() kullanimini gosterir.

using System;
using System.Threading;

class MyThread {
    public Thread thrd;

    public MyThread(string name) {
        thrd = new Thread(new ThreadStart(this.run));
        thrd.Name = name;
        thrd.Start();
    }

    // Burasi, kanalin giris noktasidir.
    void run() {
        Console.WriteLine(thrd.Name + " starting.");

        for(int i = 1; i <= 1000; i++) {
            try {
                Console.Write(i + " ");
                if(i % 10 == 0) {
                    Console.WriteLine();
                    Thread.Sleep(250);
                }
            } catch(ThreadAbortException exc) {
                if((int)exc.ExceptionState == 0) {
                    Console.WriteLine("Abort Cancelled! Code is " +
                        exc.ExceptionState);
                    Thread.ResetAbort();
                }
                else
                    Console.WriteLine("Thread aborting, code is " +
                        exc.ExceptionState);
            }
        }
        Console.WriteLine(thrd.Name + " exiting normally.");
    }
}

class ResetAbort {
    public static void Main() {
        MyThread mt1 = new MyThread("My Thread");
        Thread.Sleep(1000); // cocuk kanalin calismaya baslamasina
                           // izin ver

        Console.WriteLine("Stopping thread.");
        mt1.thrd.Abort(0); // bu, kanali durdurmayacaktir
```

```

        Thread.Sleep(1000); // cocugun biraz daha calismasina
                            // izin ver

        Console.WriteLine("Stopping thread.");
        mt1.thrd.Abort(100); // bu, kanali durduracaktir

        mt1.thrd.Join(); // calisma kanalinin sona ermesini bekle

        Console.WriteLine("Main thread terminating.");
    }
}

```

Çıktı aşağıda gösterilmiştir:

```

My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Stopping thread.
Abort Cancelled! Code is 0
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Stopping thraad.
Thread aborting, code is 100
Main thread terminating.

```

Bu örnekte eğer **Abort()**, sıfır değerindeki bir argüman ile çağrılırsa durdurma isteği çalışma kanalı tarafından **ResetAbort()** çağrılarak iptal edilir ve kanal çalışmasına devam eder. Herhangi başka bir değer kanalın sona ermesine neden olur.

## Çalışma Kanalının Durumunu Belirlemek

Bir kanalın durumu, **Thread** tarafından sağlanan **ThreadState** özelliğinden elde edilebilir. Bu özellik aşağıda gösterilmiştir;

```
public ThreadState ThreadState{ get; }
```

Çalışma kanalının durumu, **ThreadState** numaralandırması içinde tanımlanan bir değer olarak döndürülür. **ThreadState** numaralandırması içinde şu değerler tanımlıdır:

<b>ThreadState.Aborted</b>	<b>ThreadState.AbortRequested</b>
<b>ThreadState.Background</b>	<b>ThreadState.Running</b>
<b>ThreadState.Stopped</b>	<b>ThreadState.StopRequested</b>
<b>ThreadState.Suspended</b>	<b>ThreadState.SuspendRequested</b>
<b>ThreadState.Unstarted</b>	<b>ThreadState.WaitSleepJoin</b>

Bunlardan biri haricinde diğerlerini açıklamaya gerek yoktur, kendileri zaten anlaşılmalıdır. Açıklama gerektiren ise **ThreadState.WaitSleepJoin**'dır. **Wait()**, **Sleep()** ya da **Join()**'e yapılan bir çağrıdan dolayı beklemekte olan bir çalışma kanalı bu bekleme sırasında bu duruma girer.

## Temel Kanalın Kullanımı

Bu bölümün başında bahsedildiği gibi, tüm C# programları, *temel kanal (main thread)* denilen ve program çalışmaya başladığında programa otomatik olarak verilen en azından bir adet kanala sahiptir. Temel kanal tipki diğer kanallar gibi ele alınabilir.

Temel kanala erişmek için bu kanala referansta bulunan bir **Thread** nesnesi elde etmelisiniz. Bunu, **Thread**'in bir üyesi olan **CurrentThread** özelliğini kullanarak gerçekleştirirsiniz. Bu özelliğin genel kullanımı aşağıda gösterilmiştir:

```
public static Thread CurrentThread{ get; }
```

Bu metod, içinde çağrıldığı kanala bir referans döndürür. Bu nedenle, eğer program temel kanal içinde çalışmasını sürdürürken **CurrentThread**'i kullanırsanız, temel kanala bir referans elde edeceksiniz. Bu referansa bir kez sahip olduktan sonra artık temel kanalı tipki diğer kanallar gibi kontrol edebilirsiniz.

Aşağıdaki program, temel kanalı gösteren bir referans elde eder ve temel kanalın adını ve öncelik sırasını alır ve ayarlar:

```
// Temel kanalı kontrol eder.

using System;
using System.Threading;

class UseMain {
    public static void Main() {
        Thread thrd;

        // Temel kanalı al.
        thrd = Thread.CurrentThread;

        // Temel kanalın adını görüntüle.
        if(thrd.Name == null)
            Console.WriteLine("Main thread has no name.");
        else
            Console.WriteLine("Main thread is called: " +
                Thrd.Name);

        // Temel kanalın öncelik sırasını görüntüle.
        Console.WriteLine("Priority: " + thrd.Priority);

        Console.WriteLine();

        // Adını ve öncelik sırasını ayarla.
```

```
Console.WriteLine("Setting name and priority.\n");
thrd.Name = "Main Thread";
thrd.Priority = ThreadPriority.AboveNormal;

Console.WriteLine("Main thread is now called: " +
                  thrd.Name);

Console.WriteLine("Priority is now: " + thrd.Priority);
}
}
```

Programın çıktısı aşağıdaki gibidir:

```
Main thread has no name.
Priority: Normal
```

```
Setting name and priority.
```

```
Main thread is now called: Main Thread
Priority is now: AboveNormal
```

Bir uyarı: Temel kanal üzerinde hangi işlemleri gerçekleştirdiğiniz konusunda dikkatli olmanız gereklidir. Söz gelişi, **Main()**'in sonuna aşağıdaki **Join()** çağrısını eklerseniz

```
thrd.Join();
```

program asla sona ermez, çünkü temel kanalın sona ermesini bekleyecektir!

## Çok Kanallılık Önerileri

Çok kanallılıktan verimli şekilde yararlanmanın temel yolu seri olarak değil paralel olarak düşünmektedir. Örneğin, bir program içinde paralel zamanlı çalışabilen iki alt sisteme sahipseniz, bunları ayrı birer kanal haline gelirsin. Ancak, burada bir uyarı yapmakta da yarar vardır. Eğer haddinden fazla kanal oluşturmuşsanız, programınızın performansını artırmak yerine gerçekleştirebilirsiniz. Hatırlarsanız, içerik değiş tokusu ile ilintili bir ek yük konusudur. Eğer haddinden fazla kanal oluşturursanız, programınızı çalıştmak için gerekli CPU zamanından daha fazlası içerik değiştirmek için harcanacaktır!

## Ayrı Bir Görevi Başlatmak

C#'ta programlama yaparken kanal tabanlı çok görevlilik sizin en sık kullanacağınız yöntem olmasına rağmen, uygun olan yerlerde proses tabanlı çok görevlilikten yararlanmak da mümkündür. Proses tabanlı çok görevlilikten yararlanırken aynı program içinde başka bir kanal başlatmak yerine bir program, bir diğer programın çalışmasını başlatır. C#'ta bunu **Process** sınıfını kullanarak gerçekleştirirsiz. **Process**, **System.Diagnostics** isim uzayında tanımlıdır. Bu bölümü noktalamak için bir başka prosesi başlatmak ve yönetmek konusuna kısaca göz atmanız tavsiye edilir.

Bir başka prosesi başlatmanın en kolay yolu **Process** tarafından tanımlanmış **Start()** metodunu kullanmaktır. Bunun en basit şekillerinden biri aşağıdaki gibidir:

```
public static Process start(string isim)
```

Burada **isim**, çalıştırılacak olan çalıştırılabilir dosyanın ismini ya da çalıştırılabilir bir dosya ile ilişkili bir dosyanın ismini belirtir.

Sizin oluşturduğunuz bir proses sona erdiğinde bu proses ile ilintili belleği serbest bırakmak için **Close()**'u çağırın. **Close()** aşağıda gösterilmiştir:

```
public void Close()
```

Bir prosesi iki şekilde sona erdirebilirisiniz. Eğer söz konusu proses bir Windows GUI uygulaması ise, prosesi sonlandırmak için, aşağıda gösterildiği gibi, **CloseMainWindow()**'u çağırın:

```
public bool CloseMainWindow()
```

Bu metot, prosese durması gerektiğini bildiren bir mesaj gönderir. Mesaj alınmışsa metot **true** döndürür. Eğer uygulama bir GUI programı değilse ya da bir ana pencereye sahip değilse metot **false** döndürür. Üstelik; **CloseMainWindow()**, oturumu kapatmak için yalnızca bir istektir. Eğer uygulama bu isteği dikkate almazsa, uygulama sona ermeyecektir.

Bir prosesi gerçekten sonlandırmak için aşağıda gösterildiği gibi **Kill()**'i çağırın:

```
public void Kill()
```

**Kill()**'i dikkatli kullanın. **Kill()**, söz konusu prosesin kontrolsüz sonlandırılması neden olur. Prosesle ilintili kaydedilmemiş herhangi veri büyük olasılıkla kaybedilecektir.

**WaitForExit()**'i çağrıarak bir prosesin sona ermesini bekleyebilirisiniz. Bu metodun iki şekli aşağıda gösterilmiştir:

```
public void WaitForExit()  
public bool WaitForExit(int milisaniye)
```

İlk ifade, proses sona erene kadar bekler. İkinci ifade yalnızca belirtilen **milisaniye** süresince bekler. Eğer söz konusu proses sona ermişse ikinci ifade **true** döndürür; proses halen çalışıyorsa metot **false** döndürür,

Aşağıdaki program bir prosesin nasıl oluşturulacağını, beklenileceğini ve kapatılacağını göstermektedir. Program, Windows'un standart yardımcı programı olan WordPad.exe'yi çalıştırmaktadır. Program daha sonra WordPad'in sona ermesini bekler.

```
// Yeni bir proses başlatır.  
  
using System;  
using System.Diagnostics;  
  
class StartProcess {
```

```
public static void Main() {  
    Process newProc = Process.Start("wordpad.exe");  
  
    Console.WriteLine("New process started.");  
  
    newProc.WaitForExit();  
  
    newProc.Close(); // serbest kaynaklar  
  
    Console.WriteLine("New process ended.");  
}  
}
```

Bu programı çalıştırığınızda WordPad çalışmaya başlayacaktır ve siz de “New process started.” mesajını ekranda göreceksiniz. Program daha sonra siz WordPad’ı kapatana kadar bekleyecektir. WordPad bir kez sona erdikten sonra artık son mesaj “New process ended.” ekranda gösterilir.

YİRMİ İKİNCİ BÖLÜM

22

---

## KOLEKSİYONLARLA ÇALIŞMAK

C#'ta bir koleksiyon, bir grup nesnedir. System.Collections isim uzayı, çeşitli tiplerde koleksiyonlar tanımlayan ve uygulayan çok sayıda arayüz ve sınıf içerir. Koleksiyonlar pek çok programlama görevini kolaylaştırırlar, çünkü sıkça kullanılan fakat kimi zaman geliştirilmesi sıkıcı olan birkaç veri yapısı için taze çözümler sunarlar. Örneğin, yığın, kuyruk ve “hash” tablolarını destekleyen, standart olarak sisteme entegre koleksiyonlar mevcuttur. Koleksiyonlar, tüm C# programcılarının yakın ilgisine layık en gelişkin teknolojidir.

## Koleksiyonlara Genel Bakış

Koleksiyonların başlıca avantajı, nesne gruplarının programlarınızı tararından ele alınışını standardize etmeleridir. Tüm koleksiyonlar, net olarak tanımlanmış birtakım arayüzler etrafında tasarlanmıştır. Bu arayüzlerin, olduğu gibi kullanabileceğiniz birkaç standart uygulaması mevcuttur; söz gelisi, **ArrayList**, **Hashtable**, **Stack** ve **Queue** bu uygulamalardandır. Ayrıca, kendi koleksiyonunuzu da uygulayabilirsiniz, fakat buna ender olarak gerek duyacaksınız.

.NET Framework, üç genel koleksiyon tipini destekler: Genel amaçlı, özelleştirilmiş ve bit tabanlı. Genel amaçlı koleksiyonlar, aralarında dinamik dizi (dynamic array), yığın (stack) ve kuyruk (queue) da olan birkaç temel veri yapısını uygularlar. Ayrıca, anahtar/değer çiftlerini saklayabileceğiniz sözlükler (dictionary) de içerirler. Böylece, her tipten veriyi saklamak için kullanılabilirler.

Özel amaçlı koleksiyonlar, belirli bir veri tipi üzerinde işlem görürler ya da işlemlerini benzersiz bir biçimde gerçekleştirirler. Örneğin, karakter katarları için özelleştirilmiş koleksiyonlar mevcuttur. Tek bağlı listeyi kullanan özelleştirilmiş koleksiyonlar da ayrıca vardır.

Bit tabanlı koleksiyon sınıfları bir grup biti saklarlar. Bit tabanlı koleksiyonlar, diğer koleksiyon tiplerine kıyasla farklı birtakım işlemleri desteklerler. Söz gelisi, başlıca bit tabanlı koleksiyonlardan biri, VE ya da VEYA gibi bit tabanlı işlemleri destekleyen **BitArray**'dır.

Tüm koleksiyonların esası numaralandırıcı (enumerator) kavramına dayanır. Bu kavram, **IEnumerator** ve **IEnumerable** arayüzleri tarafından desteklenir. Numaralandırıcı, bir koleksiyon içindeki öğelere erişimin bir kerede bir öğeye erişilecek şekilde standardize edilmiş yöntemlerinden biridir. Böylece, koleksiyonun içeriği numaralandırılmış olur. Her koleksiyonun **IEnumerable**'ı uygulaması gereklidir. Bu nedenle, herhangi bir koleksiyon sınıfının öğeleri **IEnumerator** tarafından tanımlanan metodlar aracılığıyla erişilebilir. Bir koleksiyon tipi üzerinde tekrarlayan kod, yalnızca küçük değişikliklerle, bir başkası üzerinde tekrarlayacak şekilde kullanılabilir. İşin ilginç yanı, **foreach** döngüsü, bir koleksiyonun içeriği üzerinde tekrarlamak için numaralandırıcı kullanır.

Son bir husus: Eğer C++'a aşınaysanız, C# koleksiyon sınıflarının C++ tarafından tanımlanan Standart Şablon Kütüphanesi (STL - Standard Template Library) sınıflarıyla aynı anlamda olduğunu bilmek sizin için yararlı olacaktır. C++'ın konteyner olarak adlandırdıklarını C# koleksiyon olarak çağırmaktadır. Aynı durum Java için de geçerlidir. Eğer Java'nın Collections

Framework'üne aşinaysamz, C# koleksiyonlarının kullanımını öğrenirken hiç zorluk yaşamayacaksınız demektir.

## Koleksiyon Arayüzleri

`System.Collections`'ta çok sayıda arayüz tanımlıdır. Koleksiyon arayüzleriyle başlamak gereklidir, çünkü tüm koleksiyon sınıflarında ortak olan işlevsellik bu arayüzler tarafından belirlenmektedir. Koleksiyonları alltan destekleyen arayüzler Tablo 22.1'de özetlenmiştir. Aşağıdaki konularda her arayüz ayrıntılı olarak incelenmektedir.

### ICollection Arayüzü

`ICollection` arayüzü, üzerine tüm koleksiyonların inşa edildiği temel niteliğindedir. `ICollection`'da tüm koleksiyonların sahip olacağı çekirdek metodlar ve özellikler deklare edilir. Ayrıca `IEnumerable` arayüzü de kalıtım yoluyla `ICollection`'a aktarılır. Koleksiyonların ne şekilde çalıştığını net olarak anlamak için `ICollection`'ın anlaşılması olması gereklidir.

`ICollection`'da aşağıdaki özellikler tanımlıdır:

Özellik	Anlamı
<code>int Count { get; }</code>	Koleksiyonda tutulan mevcut öğe sayısını içerdigi için en sık kullanılan özellikdir. Eğer <code>Count</code> sıfır ise koleksiyon boş demektir.
<code>bool IsSynchronized { get; }</code>	Koleksiyon senkronize edilmişse <code>true</code> 'dur; senkronize edilmemişse <code>false</code> 'tur. Koleksiyonlar varsayılan durumda senkronize edilmemişlerdir. Buna rağmen, birçok koleksiyonun senkronize edilmiş bir versiyonunu elde etmek mümkündür.
<code>object SyncRoot { get; }</code>	Üzerinde bir koleksiyonun senkronize edilebileceği bir nesne.

`Count`, bir koleksiyon içinde tutulan mevcut öğe sayısını içerdigi için en sık kullanılan özelliktir. Eğer `Count` sıfır ise koleksiyon boş demektir.

`ICollection`'da aşağıdaki metot tanımlıdır:

```
void CopyTo(Array hedef, int startIdks)
```

`CopyTo()`, bir koleksiyonun içeriğini, `startIdks` ile belirtilen indeksten başlayarak `hedef` ile belirtilen diziye kopyalar. Böylece, `CopyTo()` bir koleksiyon ile standart C# dizisi arasında köprü kurmuş olur.

`IEnumerable`, kalıtım yoluyla `ICollection`'a aktarıldığı için `ICollection`, `IEnumerable` tarafından tanımlanan tek metodu da içerir: `GetEnumerator()`. Bu metot aşağıda gösterilmiştir:

```
IEnumerator GetEnumerator()
```

`GetEnumerator()`, koleksiyon için bir numaralandırıcı döndürür.

**TABLO 22.1: Koleksiyon Arayüzleri**

Arayüz	Açıklama
<b>ICollection</b>	Tüm koleksiyonların sahip olması gereken elemanları tanımlar.
<b>IEnumerable</b>	<code>GetEnumerator()</code> metodunu tanımlar. <code>GetEnumerator()</code> bir koleksiyon sınıfı için bir numaralandırıcı tedarik eder.
<b>IEnumerator</b>	Bir koleksiyonun içeriğinin tek tek elde edilmesini mümkün kıلان metotları sağlar.
<b>IList</b>	Bir indeksleyici üzerinden erişilebilen bir koleksiyon tanımlar.
<b>IDictionary</b>	Anahtar/değer çiftlerinden oluşan bir koleksiyon tanımlar.
<b>IDictionaryEnumerator</b>	<b>IDictionary</b> 'yi uygulayan bir koleksiyon için numaralandırıcı tanımlar.
<b>IComparer</b>	Bir koleksiyonda saklı olan nesneleri karşılaştırın <code>Compare()</code> metodunu tanımlar.
<b>IHashCodeProvider</b>	Bir hash fonksiyonu tanımlar.

## IList Arayüzü

**IList**, kalıtımla **ICollection**'dan türetilir ve elemanlarına sıfır tabanlı bir indeks aracılığıyla erişilmesine izin veren bir koleksiyonun davranışını deklare eder. **ICollection**'da tanımlı metodlara ek olarak **IList**, kendisine ait birkaç metot da tanımlar. Bunlar Tablo 22.2'de özetlenmiştir. Bu metodlardan birkaçı bir koleksiyonun üzerinde yapılan değişiklikleri içermektedir. Eğer söz konusu koleksiyon salt okunur ya da sabit büyüklükte ise bu metodlar bir **NotSupportedException**'ı fırlatacaklardır.

**TABLO 22.2: IList Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<code>int Add(object obj)</code>	Çağrıda bulunan koleksiyona <code>obj</code> 'yi ekler. Nesnenin saklandığı konumun indeksini döndürür.
<code>void Clear()</code>	Çağrıda bulunan koleksiyondaki tüm elemanları siler.
<code>bool Contains(object obj)</code>	Çağrıda bulunan koleksiyon <code>obj</code> üzerinden aktarılan nesneyi içeriyorsa <code>true</code> döndürür; aksi halde <code>false</code> döndürür.
<code>int IndexOf(object obj)</code>	<code>obj</code> , çağrıda bulunan koleksiyon içinde yer alıysa <code>obj</code> 'nin indeksini döndürür. <code>obj</code> mevcut değilse <code>-1</code> döndürür.

<b>void Insert(int idx, object obj)</b>	<i>obj</i> 'yi <i>idx</i> ile belirtilen indekse ekler. <i>idx</i> 'teki ve bunun altındaki tüm elemanlar aşağı doğru kaydırılırak <i>obj</i> için yer açılır.
<b>void Remove(object obj)</b>	Çağrıda bulunan koleksiyondan ilk rastlanılan <i>obj</i> 'yi çıkartır. Çıkarılan elemanın olduğu konumdaki ve aşağıdaki elemanlar yukarı doğru kaydırılırak boşluk kapatılır.
<b>void RemoveAt(int idx)</b>	Çağrıda bulunan koleksiyondan <i>idx</i> ile belirtilen indeksteki nesneyi çıkartır. Çıkarılan elemanın olduğu konumdaki ve aşağıdaki elemanlar yukarı doğru kaydırılırak boşluk kapatılır.

Bir **IList** koleksiyonuna nesneler **Add()** çağrılarak eklenir. **Add()** 'in **object** tipinde bir argüman aldığına dikkat edin. **object**, tüm tipler için bir temel sınıf olduğundan dolayı herhangi tipten **object** bir koleksiyonda saklanabilir. Değer tipleri bu kapsamda yer alır, çünkü kutulama ve kutudan çıkışma otomatik olarak ortaya çıkacaktır.

**Remove()** ya da **RemoveAt()**'ı kullanarak bir öğeyi çıkartabilirsiniz. **Remove()**, belirtilen nesneyi çıkartır. **RemoveAt()**, belirtilen indeksteki nesneyi çıkartır. Koleksiyonu boşaltmak için **Clear()**'ı çağırın.

**Contains()**'ı çağırarak bir koleksiyonun belirli bir nesne içerip, içermediğini belirleyebilirsiniz. Bir nesnenin indeksini **IndexOf()**'u çağırarak elde edebilirsiniz. **Insert()**'ı çağırarak belirli bir indekse bir öğe ekleyebilirsiniz.

**IList**'te aşağıdaki özellikler tanımlıdır;

```
bool IsFixedSize { get; }
bool IsReadOnly { get; }
```

Eğer koleksiyon sabit büyüklükte ise, **IsFixedSize true**'dur. Bu, öğeler eklenemez ya da çıkartılamaz, demektir. Eğer koleksiyon salt okunur ise, **IsReadOnly true** değerine sahiptir. Bu koleksiyonun içeriği değiştirilemez demektir.

**IList**'te aşağıdaki indeksleyici tanımlıdır:

```
object this[int idx] { get; set; }
```

Bir ögenin değerini almak ya da ayarlamak için bu indeksleyiciyi kullanacaksınız. Ancak, koleksiyona yeni bir öğe eklemek için bunu kullanamazsınız. Listeye bir öğe eklemek için **Add()**'ı çağırın. Öğe eklenir eklenmez artık bu öğeye indeksleyici aracılığıyla erişebilirsiniz.

## IDictionary Arayüzü

**IDictionary** arayüzü, benzersiz anahtarları, karşılık gelen değerlerle eşleyen bir koleksiyonunun davranışını tanımlar. Anahtar, daha sonraki bir tarihte bir değere erişmek için kullanacağınız bir nesnedir. Böylece, **IDictionary** arayüzüünü uygulayan bir koleksiyon,

anahtar/değer çiftlerini saklar. Bir kez anahtar/değer çifti saklandıktan sonra artık bu çifti kendi anahtarını kullanarak erişebilirsiniz. **IDictionary**, kalıtımla **ICollection**'dan türetilmiştir. **IDictionary** tarafından tanımlanan metodlar Tablo 22.3'te özetlenmiştir. Bir **null** anahtar belirtilmeye çalışıldığında metodlardan birkaçının **NotSupportedException** fırlatacaktır.

Bir **IDictionary** koleksiyona bir anahtar/değer çifti eklemek için **Add()** kullanın. Anahtar ve anahtara karşılık gelen değerin ayrı ayrı belirtildiğine dikkat edin. Bir öğeyi çıkartmak için söz konusu nesnenin anahtarını **Remove()** çağrıları içinde belirtin. Koleksiyonu çıkartmak için **Clear()**'ı çağırın.

Bir koleksiyonun belirli bir nesneyi içerip, içermediğini **Contains()**'ı istenilen öğenin anahtarı ile birlikte çağırarak belirleyebilirsiniz. **GetEnumerator()**, bir **IDictionary** koleksiyonu ile uyumlu bir numaralandırıcı elde eder. Bu numaralandırıcı anahtar/değer çiftleri üzerinde işlem yapar.

**IDictionary**'de aşağıdaki özellikler tanımlıdır:

Özellik	Açıklama
<b>bool IsFixedSize { get; }</b>	Sözlük sabit büyüklükte ise <b>true</b> değerindedir.
<b>bool IsReadOnly { get; }</b>	Sözlük salt okunur ise <b>true</b> değerindedir.
<b>ICollection Keys { get; }</b>	Anahtarların koleksiyonunu elde eder.
<b>ICollection Values { get; }</b>	Değerlerin koleksiyonunu elde eder.

TABLO 22.3: **IDictionary** Tarafından Tanımlanan Metotlar

Metot	Açıklama
<b>void Add(object k, object v)</b>	<b>k</b> ve <b>v</b> ile belirtilen anahtar/değer çiftini çağrıda bulunan koleksiyona ekler. <b>k</b> , <b>null</b> olmamalıdır. <b>k</b> önceden koleksiyonda saklanmış ise bir <b>ArgumentException</b> fırlatılır.
<b>void Clear()</b>	Çağrıda bulunan koleksiyondan tüm anahtar/değer çiftlerini siler.
<b>bool Contains(object k)</b>	Çağrıda bulunan koleksiyon anahtar olarak <b>k</b> 'yı içeriyorsa <b>true</b> döndürür. Aksi halde <b>false</b> döndürür.
<b>IDictionaryEnumerator GetEnumerator()</b>	Çağrıda bulunan koleksiyon içinde numaralandırıcı döndürür.
<b>void Remove(object k)</b>	Anahtarı <b>k</b> 'ya eşit olan kaydı çıkartır.

Koleksiyon içinde yer alan anahtar ve değerlerin **Keys** ve **Values** özellikleri aracılığıyla ayrı birer liste şeklinde mevcut olduğuna dikkat edin.

**IDictionary**'de aşağıdaki indeksleyici tanımlanmıştır:

```
object this[object anahtar] { get; set; }
```

Bir ögenin değerini almak ya da ayarlamak için bu indeksleyiciyi kullanabilirsiniz. Ayrıca koleksiyona yeni bir öğe eklemek için de bunu kullanabilirsiniz. “İndeks”in aslında bir indeks olmadığına, bunun yerine, söz konusu ögenin anahtarı olduğuna dikkat edin.

## IEnumerable, IEnumerator ve IDictionaryEnumerator

**IEnumerable**, bir sınıfın, eğer numaralandırmaları desteklemesi gerekiyorsa, uygulaması gereken bir arayüzdür. Önceden açıklandığı gibi, **IEnumerable** kalımıla **ICollection**'dan türetildiği için tüm koleksiyon sınıfları **IEnumerable**'ı uygulamak zorundadır. **IEnumerable** tarafından tanımlanan tek metot **GetEnumerator()**'dır. **GetEnumerator()** aşağıda gösterilmiştir:

```
IEnumerator GetEnumerator()
```

Bu metot söz konusu koleksiyon için numaralandırıcı döndürür. Ayrıca, **IEnumerable**'ı uygulamak koleksiyon içeriğinin **foreach** döngüsü ile elde edilmesine olanak tanır.

**IEnumerator**, bir numaralandırıcının işlevsellliğini tanımlayan arayüzdür. **IEnumerator**'ın metodlarını kullanarak koleksiyon içeriği üzerinde tekrarlayarak ilerleyebilirsiniz. Anahtar/değer çiftlerini (sözlükler) saklayan koleksiyonlar için **GetEnumerator()**, **IEnumerator** yerine **IDictionaryEnumerator** tipinde bir nesne döndürür. **IDictionaryEnumerator**, kalım yoluyla **IEnumerator**'dan türer; bir de, sözlüklerin numaralandırılmasını kolaylaştıran işlevsellik katar.

**IEnumerator** tarafından tanımlanan metotlar ve bunun kullanımı için gerekli teknikler bu bölümün ileri ki sayfalarında anlatılmıştır.

## IComparer

**IComparer** arayüzünde, iki nesnenin karşılaştırılma yöntemini tanımlayan **Compare()** adında hir metot tanımlıdır. **Compare()** aşağıda gösterilmiştir:

```
int Compare(object v1, object v2)
```

**v1**, **v2**'den büyükse **Compare()** sıfırdan büyük bir değer döndürmelidir; **v1**, **v2**'den küçükse sıfırdan küçük bir değer, her iki değer eşitse sıfır döndürmelidir. Bu arayüz, koleksiyonun elemanlarının ne şekilde sıralanması gerektiğini belirtmek için kullanılabilir.

## IHashCodeProvider

**IHashCodeProvider**, **GetHashCode()**'un özelleştirilmiş, bir versyonunu belirtmek istediğinizde uygulanan arayüzdür. Hatırlarsanız; tüm nesneler, varsayılan durumda kullanılan hash kod metodu olan **Object.GetHashCode()**'dan türetilirler. **IHashCodeProvider**'ı

uygulayarak alternatif bir metot belirtebilirsiniz. Normalde **IHashCodeProvider**'ı uygulamak için hiç bir neden yoktur.

## The DictionaryEntry Yapısı

**System.Collections.DictionaryEntry** adında bir yapı tipi tanımlar. Anahtar/değer çiftlerini tutan koleksiyonlar, bu çiftleri bir **DictionaryEntry** nesnesi içinde saklarlar. Bu yapıda aşağıdaki iki özellik tanımlanmıştır:

```
public object Key { get; set; }
public object Value { get; set; }
```

Bu özellikler, bir giriş ile ilişkili anahtara ya da değere erişmek için kullanılırlar. Aşağıdaki yapılandırıcıyı kullanarak bir **DictionaryEntry** nesnesi yapılandırılabilirsiniz:

```
public DictionaryEntry(object k, object v)
```

Burada **k**, anahtar; **v** ise değerdir.

## Genel Amaçlı Koleksiyon Sınıfları

Artık koleksiyon arayüzlerini tanıdığınıza göre, bu arayüzleri uygulayan standart sınıfları inceleyebiliriz. Önceden de açıklandığı gibi, koleksiyon sınıfları üç temel alt bölüme ayrılmıştır: Genel amaçlı, bit tabanlı ve özel amaçlı. *Genel amaçlı* sınıflar, herhangi bir tipteki nesneleri saklamak için kullanılabilirler. *Bit tabanlı* koleksiyonlar bitleri saklarlar. *Özel amaçlı* koleksiyonlar ise tipe sıkı sıkıya bağlı (belirli bir veri tipinde işleyecek şekilde tasarlanan) ya da farklı bir şekilde özelleştirilmiş uygulamalar sunarlar. Burada genel amaçlı koleksiyon sınıfları incelenmektedir. Bit tabanlı ve özelleştirilmiş koleksiyon sınıfları bu bölümde daha sonraki sayfalarda anlatılmıştır.

Genel amaçlı koleksiyon sınıfları aşağıda özetlenmiştir:

Özellik	Açıklama
<b>ArrayList</b>	Dinamik dizi. Bu, gerekiğinde genişleyebilen bir dizidir.
<b>Hashtable</b>	Anahtar/değer çiftleri için "hash" tablosu.
<b>Queue</b>	İlk giren- ilk çıkar işleyişine sahip liste.
<b>SortedList</b>	Anahtar/değer çiftlerinin sıralı listesi.
<b>Stack</b>	İlk giren- son çıkar işleyişine sahip liste.

Aşağıda, bu koleksiyon sınıfları incelenmekte ve bunların kullanımı gösterilmektedir.

### ArrayList

**ArrayList** sınıfı, istenildiğinde genişleyebilen ya da daralabilen dinamik dizileri destekler. C#'ta standart diziler sabit uzunluktadır; uzunlukları programın çalışması sırasında

değiştirilemez. Bu, dizinin kaç adet eleman tutacağını önceden bilmeniz gerektiği anlamına gelir. Ancak, kimi zaman ne kadar büyük bir diziye gereksinimin olacağını program çalışmaya başlayana kadar tam olarak bilemeyebilirsınız. Bu durumu kontrol altına almak için **ArrayList** kullanın. **ArrayList**, büyülüğu dinamik olarak artıp azalabilen nesne referanslarından oluşan değişken uzunlukta bir dizidir. **ArrayList**, başlangıç büyülüğu ile birlikte oluşturulur. Bu büyülü aşılınca söz konusu koleksiyon otomatik olarak genişletilir. Nesneler çıkartılınca dizi daraltılabilir. **ArrayList** belki de koleksiyonların en önemlididir; burada bunu derinlemesine inceleyeceğiz.

**ArrayList**; **ICollection**, **IList**, **IEnumerable** ve **ICloneable** arayüzlerini uygular. **ArrayList** aşağıda gösterilen yapılandırıcıları içerir:

```
public ArrayList()
public ArrayList(ICollection c)
public ArrayList(int kapasite)
```

İlk yapılandırıcı, başlangıç kapasitesi **16** olan boş bir **ArrayList** yapılandırır. İkinci yapılandırıcı, **c** ile belirtilen koleksiyonun eleman sayısına ve kapasitesine sahip bir **ArrayList** yapılandırır. Üçüncü yapılandırıcı ise belirtilen **kapasite**'ye sahip bir dizi listesi yapılandırır. Kapasite, elemanları saklamak için kullanılan altta yatan dizinin büyülüğündür. Elemanlar bir **ArrayList**'e eklendikçe kapasite otomatik olarak büyür. Listenin genişletilmesi gerekiğinde listenin kapasitesi ikiye katlanır.

Uyguladığı arayüzler tarafından tanımlanan metodlara ek olarak **ArrayList**, kendisi de birkaç metot tanımlar. Bunlardan en yaygın olarak kullanılan birkaçı Tablo 22.4'te gösterilmiştir. Bir **ArrayList**, **Sort()** çağrıları ile sıralanabilir. Bir kez sıralandıktan sonra **BinarySearch()** ile verimli biçimde aranabilir. **ArrayList**'in içeriği **Reverse()** çağrıları ile tersine çevrilebilir.

**ArrayList**, bir koleksiyon içindeki elemanların bir bölümü üzerinde işlem yapan birkaç metodu da destekler. **InsertRange()**'ı çağırarak bir **ArrayList**'e bir başka koleksiyon ekleyebilirsınız. **RemoveRange()**'ı çağırarak bir bölümü listeden çıkartabilirsiniz. **SetRange()**'ı çağırarak da bir **ArrayList** içindeki bir bölümün üzerine bir başka koleksiyonun elemanlarını yazabilirsınız. Ayrıca, bütün bir koleksiyonu sıralamak ya da aramak yerine koleksiyon içindeki bir bölümü sıralayıp, arayabilirsiniz.

**ArrayList** varsayılan şekliyle senkronize edilmemiştir. Bir koleksiyon etrafında senkronize edilmiş bir kilit elde etmek için **Synchronized()**'ı çağırın.

TABLO 22.4: **ArrayList** Tarafından Tanımlanan Metotlardan Sıkça Kullanılan Birkaçı

Metot	Açıklama
<b>public virtual void AddRange(ICollection c)</b>	<b>c</b> 'nin içindeki elemanları çağrıda bulunan <b>ArrayList</b> 'in sonuna ekler.

```
public virtual int BinarySearch(object v)
```

Çağrıda bulunan koleksiyon içinde **v** üzerinden aktarılan değeri arar. Eşlenen elemanın indeksi döndürülür. Eğer değer bulunamazsa negatif bir değer döndürülür. Çağrıda bulunan liste sıralı olmalıdır.

```
public virtual int BinarySearch(object v, IComparer comp)
```

**comp** ile belirtilen karşılaştırma nesnesini kullanarak çağrıda bulunan koleksiyon içinde **v** üzerinden aktarılan değeri arar. Eşlenen elemanın indeksi döndürülür. Eğer değer bulunamazsa negatif bir değer döndürülür. Çağrıda bulunan liste sıralı olmalıdır.

```
public virtual int BinarySearch(int startIdx, int count, object v, IComparer comp)
```

**comp** ile belirtilen karşılaştırma nesnesini kullanarak çağrıda bulunan koleksiyon içinde **v** üzerinden aktarılan değeri arar. Arama **startIdx**'ten başlar ve **count** sayıda elemanı kapsar. Eşlenen elemanın indeksi döndürülür. Eğer değer bulunamazsa negatif bir değer döndürülür. Çağrıda bulunan liste sıralı olmalıdır.

```
public virtual void CopyTo(Array ar, int startIdx)
```

**startIdx**'ten başlayarak çağrıda bulunan koleksiyonun içeriğini **ar** ile belirtilen dizeye kopyalar. **ar**, koleksiyondaki elemanların tipiyle uyumlu tek boyutlu bir dizidir.

```
public virtual void CopyTo(int srcIdx, Array ar, int destIdx, int count)
```

**startIdx**'ten başlayarak **count** sayıda eleman için geçerli olmak üzere çağrıda bulunan koleksiyonun bir bölümünü **ar** ile belirtilen dizeye kopyalar. **ar**, koleksiyondaki elemanların tipiyle uyumlu tek boyutlu bir dizidir.

```
public virtual ArrayList GetRange(int idx, int count)
```

Çağrıda bulunan **ArrayList**'in bir bölümünü döndürür. Döndürülen aralık **idx** ile başlar ve **count** sayıda elemanı kapsar. Döndürülen nesne, çağrıda bulunan nesne ile aynı elemanlara referansta bulunur.

```
public static ArrayList FixedSize(ArrayList ar)
```

**ar**'ı sabit büyüklükte bir **ArrayList** içine sarar ve sonucu döndürür.

```
public virtual void InsertRange(int startIdx, ICollection c)
```

**c**'nin elemanlarını **startIdx** ile belirtilen indeksten başlayarak çağrıda bulunan koleksiyona ekler.

```
public virtual int LastIndexOf(object v)
```

Çağrıda bulunan koleksiyon içinde **v**'ye son rastlanılan konumun indeksini döndürür. **v** mevcut değilse **-1** döndürür.

```
public static ArrayList ReadOnly(ArrayList ar)
```

**ar**'ı salt okunur bir **ArrayList** içine sarar ve sonucu döndürür.

<code>public virtual void RemoveRange(int idx, int count)</code>	<code>idx</code> 'ten başlayarak çağrıda bulunan koleksiyondan <code>count</code> sayıda elemanı çıkartır.
<code>public virtual void Reverse()</code>	Çağrıda bulunan koleksiyonun içeriğini ters çevirir.
<code>public virtual void Reverse(int startIdx, int count)</code>	<code>startIdx</code> 'ten başlayarak çağrıda bulunan koleksiyonda <code>count</code> sayıda elemanı ters çevirir.
<code>public virtual void SetRange(int startIdx, ICollection c)</code>	<code>startIdx</code> 'ten başlayarak çağrıda bulunan koleksiyon içindeki elemanları <code>c</code> ile belirtilen elemanlarla değiştirir.
<code>public virtual void Sort()</code>	Koleksiyonu artan sırada sıralar.
<code>public virtual void Sort(IComparer comp)</code>	Belirtilen karşılaştırma nesnesini kullanarak koleksiyonu sıralar. Eğer <code>comp null</code> ise her nesne için varsayılan karşılaştırma kullanılır.
<code>public virtual void Sort(int startIdx, int endIdx, IComparer comp)</code>	Belirtilen karşılaştırma nesnesini kullanarak koleksiyonu sıralar. Sıralama <code>startIdx</code> 'ten başlar ve <code>endIdx</code> 'te sona erer. Eğer <code>comp null</code> ise her nesne için varsayılan karşılaştırma kullanılır.
<code>public static ArrayList Synchronized(ArrayList list)</code>	Çağrıda bulunan <code>ArrayList</code> 'in senkronize edilmiş versiyonunu döndürür.
<code>public virtual object[] ToArray()</code>	Çağrıda bulunan nesnenin elemanlarının kopyalarını içeren bir dizi döndürür.
<code>public virtual Array ToArray(Type type)</code>	Çağrıda bulunan nesnenin elemanlarının kopyalarını içeren bir dizi döndürür. Dizinin elemanlarının tipi <code>type</code> ile belirtilir.
<code>public virtual void TrimToSize()</code>	<code>Capacity</code> 'ye <code>Count</code> değerini verir.

Uyguladığı arayüzler tarafından tanımlanan özelliklere ek olarak `ArrayList`, aşağıda gösterilen `Capacity` özelliğini ekler:

```
public virtual int Capacity { get; set; }
```

`Capacity`, çağrıda bulunan `ArrayList`'in kapasitesini alır ya da ayarlar. Kapasite, söz konusu `ArrayList`'in genişletilmesi gerekmeden önce tutabileceği eleman sayısıdır. Önceden de bahsedildiği gibi, bir `ArrayList` otomatik olarak büyür; bu nedenle, kapasiteyi elle ayarlamaya gerek yoktur. Yine de, verimliliği sağlamak adına listenin kaç adet eleman içereceğini önceden bildiğiniz durumlarda, kapasiteyi ayarlamak isteyebilirsiniz. Bu, daha fazla bellek alanı ayırmakla ilintili ek yükü önler.

Öte yandan, bir `ArrayList`'in altında yatan dizinin büyütülüğünü azaltmak istediğinizde `Capacity`'ye daha küçük bir değer verebilirsiniz. Ancak, bu değer `Count`'tan daha küçük olmamalıdır. Hatırlarsanız; `Count`, bir koleksiyon içinde halihazırda saklı olan nesnelerin sayı-

sını tutan, **ICollection** tarafından tanımlı bir özelliktir. **Capacity**'ye **Count**'tan daha küçük bir değer vermeye çalışmak bir **ArgumentOutOfRangeException**'ın üretilmesine neden olur. Tam olarak, bir **ArrayList**'in halen tutmakta olduğu öğe sayısı kadar bir büyülüğu sahip bir **ArrayList** elde etmek için **Capacity**'yi **Count**'a eşitleyin. Ayrıca **TrimToSize()**'ı da çağırabilirsiniz.

Aşağıdaki program **ArrayList**'i göstermektedir. Program, bir **ArrayList** oluşturur ve buna karakterler ekler. Liste sonra ekranda görüntülenir. Elemanların bazıları çıkartılır ve liste tekrar ekranda görüntülenir. Daha sonra, daha fazla eleman eklenir: bu sayede, listenin kapasitesinin artırılması zorlanır. Son olarak, elemanların içerikleri değiştirilir.

```
// ArrayList kullanimini gosterir.

using System;
using System.Collections;

class ArrayListDemo {
    public static void Main() {
        // Bir dizi listesi olustur
        ArrayList al = new ArrayList();

        Console.WriteLine("Initial capacity: " +
                          al.Capacity);
        Console.WriteLine("Initial number of elements: " +
                          al.Count);

        Console.WriteLine();

        Console.WriteLine("Adding 6 elements");
        // Dizi listesine eleman ekle
        al.Add('C');
        al.Add('A');
        al.Add('E');
        al.Add('B');
        al.Add('D');
        al.Add('F');

        Console.WriteLine("Current capacity: " +
                          al.Capacity);
        Console.WriteLine("Number of elements: " +
                          al.Count);

        // Dizi indeksleme kullanarak dizi listesini goruntule.
        Console.Write("Current contents: ");
        for(int i = 0; i < al.Count; i++)
            Console.Write(al[i] + " ");
        Console.WriteLine("\n");

        Console.WriteLine("Removing 2 elements");
        // Dizi listesinden eleman cikart.
        al.Remove('F');
        al.Remove('A');
```

```
Console.WriteLine("Current capacity: " + al.capacity);
Console.WriteLine("Number of elements: " + al.Count);

// Listeyi goruntulemek icin foreach dongusu kullan.
Console.Write("Contents: ");
foreach(char c in al)
    Console.Write(c + " ");
Console.WriteLine("\n");

Console.WriteLine("Adding 6 elements");
// al'in buyumesini zorlamak icin yeterli eleman ekle.
for(int i = 0; i < 20; i++)
    al.Add((char)('a' + i));
Console.WriteLine("Current capacity: " +
                  al.Capacity);
Console.WriteLine("Number of elements after adding 20: " +
                  al.Count);
Console.Write("Contents: ");
foreach(char c in al)
    Console.Write(c + " ");
Console.WriteLine("\n");

// Dizi indeksleme kullanarak icerigi degistir.
Console.WriteLine("Change first three elements");
al[0] = 'X';
al[1] = 'Y';
al[2] = 'Z';
Console.WriteLine("Contents: ");
foreach(char c in al)
    Console.Write(c + " ");
Console.WriteLine();
}
}
```

Bu programın çıktısı aşağıda gösterilmiştir:

```
Initial capacity: 16
Initial number of elements: 0
```

```
Adding 6 elements
Current capacity: 16
Number of elements: 6
Current contents: C A E B D F
```

```
Removing 2 elements
Current capacity: 16
Number of elements: 4
Contents: C E B D
```

```
Adding 20 more elements
Current capacity: 32
Number of elements after adding 20: 24
Contents: C E B D a b c d e f g h i j k l m n o p q r s t
```

```
Change first three elements
```

---

Contents: X Y Z D a b c d e f g h i j k l m n o p q r s t

Dikkat ederseniz; koleksiyon, 16 başlangıç kapasitesi ile boş olarak başlamaktadır. İhtiyaç oldukça kapasite artırmaktadır. Her artırılışında kapasite ikiye katlanır.

## ArrayList'i Sıralamak ve Aramak

**ArrayList**, **Sort()** ile sıralanabilir. Bir kez sıralandıktan sonra **BinarySearch()** ile verimli biçimde aranabilir. Aşağıdaki program bu metodları göstermektedir:

```
// ArrayList'i sıralamak ve aramak.

using System;
using System.Collections;

class SortSearchDemo {
    public static void Main() {
        // bir dizi listesi oluştur
        ArrayList al = new ArrayList();

        // Dizi listesine eleman ekle
        al.Add(55);
        al.Add(43);
        al.Add(-4);
        al.Add(88);
        al.Add(3);
        al.Add(19);

        Console.WriteLine("Original contents: ");
        foreach(int i in al)
            Console.Write(i + " ");
        Console.WriteLine("\n");

        // Sirala
        al.Sort();

        // Listeyi görüntülemek için foreach dongusu kullan.
        Console.WriteLine("Contents after sorting: ");
        foreach(int i in al)
            Console.Write(i + " ");
        Console.WriteLine("\n");

        Console.WriteLine("Index of 43 is " + al.BinarySearch(43));
    }
}
```

Çıktı aşağıda gösterilmiştir:

Original contents: 55 43 -4 88 3 19

Contents after.sorting: -4 3 19 43 55 88

Index of 43 is 3

Bir **ArrayList** aynı liste içinde herhangi tipte nesneler saklayabilmesine rağmen, bir listeyi sıralarken ya da ararken bu nesnelerin karşılaştırılabilir olmaları gereklidir. Söz gelişi, önceki programda eğer liste bir karakter katarı içermiş olsaydı, program kural dışı bir durum üretmiş olurdu. (Her şeye rağmen, karakter katarlarını ve tamsayıları karşılaştırmaya imkan veren özel karşılaştırma metotları oluşturmak mümkündür. Özel karşılaştırıcılar bu bölümün ileriki sayfalarında anlatılmıştır.)

## Bir ArrayList'ten Bir Dizi Elde Etmek

**ArrayList** ile çalışırken kimi zaman söz konusu listenin içeriğini içeren gerçek anlamda bir dizi elde etmek isteyeceksiniz. Bunu **ToArray()**'i çağırarak gerçekleştirebilirsiniz. Bir koleksiyonu bir diziye dönüştürme isteğinizin ardından birkaç neden olabilir. Bunlardan ikisi şöyledir: Belirli işlemler için daha hızlı işlem süreleri elde etmek isteyebilirsiniz ya da bir diziyi, bir koleksiyonu kabul edecek şekilde aşırı yüklenmemiş bir metoda aktarmanız gerekebilir. Gerekçeniz ne olursa olsun, bir **ArrayList**'i bir diziye dönüştürmek, aşağıdaki örnekte de görüldüğü gibi, önemsiz bir meseledir:

```
// Bir ArrayList'i bir diziye donusturmek.

using System;
using System.Collections;

class ArrayListToArray {
    public static void Main() {
        ArrayList al = new ArrayList();

        // Dizi listesine eleman ekle.
        al.Add(1);
        al.Add(2);
        al.Add(3);
        al.Add(4);

        Console.WriteLine("Contents: ");
        foreach(int i in al)
            Console.Write(i + " ");
        Console.WriteLine();

        // Diziyi al.
        int[] ia = (int[]) al.ToArray(typeof(int));
        int sum = 0;

        // dizi elemanlarını topla
        for(int i = 0; i < ia.Length; i++)
            sum += ia[i];

        Console.WriteLine("Sum is: " + sum);
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

Contents: 1 2 3 4

Sum is: 10

Program, tamsayılardan oluşan bir koleksiyon oluşturarak başlar. Sonra, `int` olarak belirtilen tip ile birlikte `ToArray()` çağrılır. Bu, bir tamsayı dizisinin oluşturulmasına neden olur. `ToArray()`'in dönüş tipi `Array` olduğu için dizinin içeriği yine de `int[]`'e dönüştürülmelidir. Son olarak, değerler toplanır.

## Hashtable

**Hashtable**, depolama için bir hash tablosu kullanan bir koleksiyon oluşturur. Okuyucuların birçoğu bilecektir, bir *hash tablosu*, *hashing* denilen bir mekanizmayı kullanarak bilgileri saklar. Hashing'de bir anahtarın bilgi taşıyan içeriği, *hash kod* denilen benzersiz bir değeri belirlemek için kullanılır. Hash kod daha sonra söz konusu anahtar ile ilişkili verilerin tablo içinde saklanacağı konumun indeksi olarak kullanılır. Anahtarın karşılık gelen hash koduna dönüşümü otomatik olarak gerçekleştirilir - asla hash kodun kendisini görmezsiniz. Hashing'in avantajı; arama, alma ve ayarlama işlemlerinin çalışma sürelerinin büyük kümeler için bile sabit olarak kalmasına olanak tanımıştır.

**Hashtable**; `IDictionary`, `ICollection`, `IEnumerable`, `ISerializable`, `IDeserializationCallback` ve `ICloneable` arayüzlerini uygular.

**Hashtable**'da sıkça kullanılan şu fonksiyonlar da dahil olmak üzere birçok yapılandırıcı tanımlanmıştır:

```
public Hashtable()
public Hashtable(IDictionary c)
public Hashtable(int kapasite)
public Hashtable(int kapasite, float dolulukOrani)
```

İlk ifade varsayılan **Hashtable**'ı yapılandırır. İkinci ifade, `c`'nin elemanlarını kullanarak **Hashtable**'a başlangıç değeri atar. Üçüncü ifade, **Hashtable**'ın kapasitesini `kapasite` olarak ayarlar. Dördüncü ifade ise hem kapasiteye hem de doluluk oranına ilk değer atar. Doluluk oranı (buna *yük faktörü* de denir) **0.1** ile **1.0** arasında olmalıdır. Doluluk oranı, hash tablosunun yukarı yönde yeniden boyutlandırmasından önce ne kadar dolu olacağını belirler. Spesifik olarak, eleman sayısı, tablonun kapasitesinin doluluk oranı ile çarpımından büyükse, tablo genişletilir. Doluluk oranı almayan yapılandırıcılar için **1.0** kullanılır.

**Hashtable**'ın uyguladığı arayüzler tarafından tanımlanan metodlara ek olarak **Hashtable**'ın kendisi de birkaç metod tanımlamaktadır. Yaygın olarak kullanılanlardan bazıları Tablo 22.5'te gösterilmiştir. Bir **Hashtable**'ın bir anahtar içerip içermediğini belirlemek için `ContainsKey()`'ı çağırın. Belirli bir değerin saklanıp saklanmadığını anlamak için `ContainsValue()`'yu çağırın. Bir **Hashtable**'ın içeriğini numaralandırmak için `GetEnumerator()`'ı çağırarak bir `IDictionaryEnumerator` elde edin. Hatırlarsanız, anahtar/değer çiftlerini saklayan bir koleksiyonun içeriğini numaralandırmak için `IDictionaryEnumerator` kullanılmaktadır.

TABLO 22.5: Hashtable Tarafından Tanımlanan Metotlardan Sıkça Kullanılan Birkaçı

Metot	Açıklama
<code>public virtual bool ContainsKey(object k)</code>	<b>k</b> , çağrıda bulunan <b>Hashtable</b> içinde bir anahtar ise <b>true</b> döndürür. Aksi halde, <b>false</b> döndürür.
<code>public virtual bool ContainsValue(object v)</code>	<b>v</b> , çağrıda bulunan <b>Hashtable</b> içinde bir değer ise <b>true</b> döndürür. Aksi halde, <b>false</b> döndürür.
<code>public virtual IDictionaryEnumerator Get Enumerator()</code>	Çağrıda bulunan <b>Hashtable</b> için bir <b>IDictionaryEnumerator</b> döndürür.
<code>public static Hashtable Synchronized(Hashtable ht)</code>	<b>ht</b> üzerinden aktarılan <b>Hashtable</b> 'in senkronize edilmiş versiyonunu döndürür.

**Hashtable**'ın uyguladığı arayüzler tarafından tanımlanan bu özelliklere ek olarak, **Hashtable**'ın kendisi de iki açık özellik daha ekler. Aşağıda gösterilen özellikleri kullanarak **Hashtable**'ın anahtarlarından ya da değerlerinden oluşan bir koleksiyon elde edebilirsiniz:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

**Hashtable** ile sıralı bir koleksiyon desteklenmediği için elde edilen anahtar ya da değer koleksiyonunun belirli bir sırası yoktur. **Hashtable** ayrıca **Hcp** ve **Comparer** adlarında iki korumalı özellik de tanımlar. Her iki özellik, kendilerinden türetilen sınıfların kullanımlarına açıktır.

**Hashtable**, anahtar/değer çiftlerini bir **DictionaryEntry** yapısı şeklinde saklar; fakat, çoğu zaman bu durumdan direkt olarak haberiniz olmayacaktır, çünkü söz konusu özellikler ve metodlar, anahtar ve değerlerle ayrı ayrı çalışmaktadır. Söz gelisi, bir **Hashtable**'a bir eleman eklerseniz, iki argüman alan **Add()**'ı çağrırsınız. **Add()**'in argümanları anahtar ve değerdir.

**Hashtable**'ın elemanlarının sırasını garanti etmediğine dikkat çekmek önemlidir. Bunun nedeni, hashing sürecinin genellikle sıralı tablo oluşumuna kendisini katmamasıdır.

İşte, **Hashtable** kullanımını gösteren bir örnek;

```
// Hashtable kullanımını gösterir.

using System;
using System.Collections;

class HashtableDemo {
    public static void Main() {
        // Bir hash tablo oluştur.
        Hashtable ht = new Hashtable();

        // Tabloya eleman ekle.
        ht.Add("house", "Dwelling");
```

```

ht.Add("car", "Means of transport");
ht.Add("book", "Collection of printed words");
ht.Add("apple", "Edible fruit");

// Indeksleyiciyi kullanarak da ekleme yapılabilir.
ht["tractor"] = "Farm implement";

// Anahtarlardan oluşan bir koleksiyon al.
ICollection c = ht.Keys;

// Değerleri elde etmek için anahtarları kullan.
foreach(string str in c)
    Console.WriteLine(str + ":" + ht[str]);
}
}

```

Bu programın çıktısı aşağıda gösterilmiştir:

```

tractor: Farm implement
book: Collection of printed words
apple: Edible fruit
car: Means of transport
house: Dwelling

```

Cıktıdan da görüldüğü gibi, anahtar/değer çiftleri sıralı olarak saklanmaktadır. **ht** isimli hash tablosunun içeriğinin nasıl elde edildiğine ve görüntülenmeye dikkat edin. Önce, **Keys** özelliği kullanılarak anahtarlardan oluşan bir koleksiyon alınmıştır. Her anahtar daha sonra **ht**'yi indekslemek için kullanılmış, her anahtara karşılık gelen değer, sonuç olarak elde edilmiştir. Hatırlarsanız, **IDictionary** tarafından tanımlanan ve **Hashtable** tarafından uygulanan indeksleyici, indeks olarak bir anahtar kullanmaktadır.

## SortedDictionary

**SortedDictionary**, anahtarların değerlerine bağlı olarak anahtar/değer çiftlerini sıralı olarak saklayan bir koleksiyon oluşturur. **SortedDictionary**; **IDictionary**, **ICollection**, **IEnumerable** ve **ICloneable** arayüzlerini uygular.

**SortedDictionary**, aşağıda gösterilenler de dahil olmak üzere, birkaç yapılandırıcıya sahiptir:

```

public SortedDictionary()
public SortedDictionary(IDictionary c)
public SortedDictionary(int kapasite)
public SortedDictionary(IComparer karşılaştır)

```

İlk yapılandırıcı, başlangıç kapasitesi olarak **16** değerine sahip boş bir koleksiyon yapılandırır. İkinci yapılandırıcı, kendisine ilk değer olarak **c**'nin elemanları ve kapasitesi atanan bir **SortedDictionary** yapılandırır. Üçüncü yapılandırıcı, **kapasite** ile belirtilen başlangıç kapasitesine sahip boş bir **SortedDictionary** kurar. Kapasite, elemanları saklamak için kullanılan altta yatan dizinin büyüklüğündür. Dördüncü ifade, listenin içinde yer alan nesneleri

karşılaştırmak için kullanılacak bir karşılaştırma metodu belirtmenize olanak tanır. Bu ifade, başlangıç kapasitesi **16** olan boş bir koleksiyon oluşturur.

Bir dizi listesine elemanlar eklendikçe **SortedList**'in kapasitesi gerektiğinde otomatik olarak büyür. Mevcut kapasitenin aşılması durumunda kapasite iki katına çıkartılır. Bir **SortedList** oluştururken kapasite belirtmenin avantajı, koleksiyonun büyüklüğünü yeniden ayarlama işlemiyle ilintili ek yükü önleyebilmeniz ya da asgariye indirmenizdir. Elbette, başlangıç kapasitesi belirtmek, ancak kaç adet eleman saklanacağı hakkında bir fikriniz varsa mantıklıdır.

**SortedList**'in uyguladığı arayüzler tarafından tanımlanan metodlara ek olarak **SortedList**'in kendisi de ayrıca birkaç metod tanımlar. Yaygın olarak kullanılanlardan bazıları Tablo 22.6'da gösterilmiştir. Bir **SortedList**'in bir anahtar içerip içermediğini belirlemek için **ContainsKey()**'ı çağırın. Belirli bir değerin saklanıp saklanmadığını anlamak için **ContainsValue()**'yu çağırın. Bir **SortedList**'in içeriğini numaralandırmak için **GetEnumerator()**'ı çağırarak bir **IDictionaryEnumerator** elde edin. Hatırlarsanız, anahtar/değer çiftlerini saklayan bir koleksiyonun içeriğini numaralandırmak için **IDictionaryEnumerator** kullanılır. **Synchronized()**'ı çağırarak bir **SortedList** etrafında senkronize edilmiş bir kilit elde edebilirsiniz.

**TABLO 22.6: SortedList Tarafından Tanımlanan Metotlardan Sıkça Kullanılan Bırkaçı**

Metot	Açıklama
<code>public virtual bool ContainsKey(object k)</code>	<i>k</i> , çağrıda bulunan <b>SortedList</b> içinde bir anahtar ise <b>true</b> döndürür. Aksi halde, <b>false</b> döndürür.
<code>public virtual bool ContainsValue(object v)</code>	<i>v</i> , çağrıda bulunan <b>SortedList</b> içinde bir değer ise <b>true</b> döndürür. Aksi halde, <b>false</b> döndürür.
<code>public virtual object GetByIndex(int idx)</code>	<i>idx</i> ile belirtilen indeksteki değeri döndürür.
<code>public virtual IDictionaryEnumerator GetEnumerator()</code>	Çağrıda bulunan <b>SortedList</b> için bir <b>IDictionaryEnumerator</b> döndürür.
<code>public virtual object GetKey(int idx)</code>	<i>idx</i> ile belirtilen indeksteki anahtarı döndürür.
<code>public virtual IList GetKeyList()</code>	Çağrıda bulunan <b>SortedList</b> içindeki anahtarların bir <b>IList</b> koleksiyonunu döndürür.
<code>public virtual IList GetValueList()</code>	Çağrıda bulunan <b>SortedList</b> içindeki değerlerin bir <b>IList</b> koleksiyonunu döndürür.

<code>public virtual int IndexOfKey(object k)</code>	<code>k</code> ile belirtilen anahtarın indeksini döndürür. Anahtar listede değilse <code>-1</code> döndürür.
<code>public virtual int IndexOfValue(object v)</code>	<code>v</code> ile belirtilen değere ilk rastlanılan konumun indeksini döndürür. Değer listede değilse <code>-1</code> döndürür.
<code>public virtual void SetByIndex(int idx, object v)</code>	<code>idx</code> ile belirtilen indeksteki değere <code>v</code> üzerinden aktarılan değeri atar.
<code>public static SortedList Synchronized(SortedList sl)</code>	<code>sl</code> üzerinden aktarılan <code>SortedList</code> 'in senkronize edilmiş versiyonunu döndürür.
<code>public virtual void TrimToSize()</code>	<code>Capacity</code> 'ye <code>Count</code> değerini atar.

Bir değer ya da anahtarı ayarlamak veya elde etmek için çeşitli yöntemler mevcuttur. Belirli bir indeks ile ilişkili değeri elde etmek için `GetByIndex()`'ı çağırın. İndeksi verilen bir değeri ayarlamak için `SetByIndex()`'ı çağırın. Tüm anahtarların bir listesini elde etmek için `GetKeyList()`'ı kullanın. Tüm değerlerin bir listesini almak için `GetValueList()`'ı kullanın. `IndexOfKey()`'ı çağrıarak bir anahtarın indeksini ve `IndexOfValue()`'yu çağrıarak da değerin indeksini elde edebilirsiniz. `SortedList`, elbette, `IDictionary` tarafından tanımlanan indeksleyiciyi de ayrıca destekler. Bu sayede, anahtarı verilen bir değeri ayarlamana ya da elde etmenize olanak tanır.

`SortedList`'in uyguladığı arayüzler tarafından tanımlanan bu özelliklere ek olarak `SortedList`'in kendisi de iki özellik ekler. Aşağıda gösterilen özellikleri kullanarak bir `SortedList`'in anahtarlarından ya da değerlerinden oluşan salt okunur bir koleksiyon elde edebilirsiniz:

```
public virtual ICollection Keys { get; }  
public virtual ICollection Values { get; }
```

Anahtar ve değerlerin sırası `SortedList`'in sırasını yansıtır.

Tıpkı `Hashtable` gibi, `SortedList` de anahtar/değer çiftlerini bir `DictionaryEntry` yapısı şeklinde saklar, fakat genellikle anahtarlara ve değerlere `SortedList` tarafından tanımlanan metotları ve özellikleri kullanarak ayrı ayrı erişeceksiniz.

Aşağıdaki program `SortedList` kullanımını göstermektedir. Bu program, önceki bölümdeki `Hashtable` kullanımını gösteren programın üzerinden gidip, `Hashtable` yerine `SortedList`'i yerleştirerek o programı genişletmiştir. Çıktıyı incelediğinizde `SortedList` versiyonunun anahtara göre sıralanmış olduğunu göreceksiniz.

```
// SortedList kullanımını gösterir.  
  
using System;  
using System.Collections;  
  
class SLDemo {
```

```
public static void Main() {
    // Sirali bir SortedList olustur.
    SortedList sl = new SortedList();

    // Tabloya eleman ekle.
    sl.Add("house", "Dwelling");
    sl.Add("car", "Means of transport");
    sl.Add("book", "Collection of printed words");
    sl.Add("apple", "Edible fruit");

    // Indeksleyiciyi kullanarak da ekleme yapabilirsiniz.
    sl["tractor"] = "Farm implement";

    // Anahtarlarin koleksiyonunu al.
    ICollection c = sl.Keys;

    // Degerleri elde etmek icin anahtarları kullan.
    Console.WriteLine("Contents of list via indexer.");
    foreach(string str in c)
        Console.WriteLine(str + ": " + sl[str]);

    Console.WriteLine();

    // Tamsayi indeksleri kullanarak listeyi goruntule.
    Console.WriteLine("Contents by integer indexes.");
    for(int i =0; i < sl.Count; i++)
        Console.WriteLine(sl.GetByIndex(i));

    Console.WriteLine();

    // Ogelerin tamsayi indekslerini goster.
    Console.WriteLine("Integer indexes of entries.");
    foreach(string str in c)
        Console.WriteLine(str + ": " + sl.IndexOfKey(str));
}
```

Cıktı aşağıdadır:

```
Contents of list via indexer.
apple: Edible fruit
book: Collection of printed words
car: Means of transport
house: Dwelling
tractor: Farm implement
```

```
Contents by integer indexes.
Edible fruit
Collection of printed words
Means of transport
Dwelling
Farm implement
```

```
Integer indexes of entries.
apple: 0
```

```
book: 1
car: 2
house: 3
tractor: 4
```

## Stack

Okuyucuların birçoğunun bildiği gibi yiğin, ilk giren son çıkar işleyişine sahip bir listedir. Yiğini göz önünde canlandırmak için bir masa üzerinde bir tabak yiğini hayal edin. Masaya ilk konan tabak, en son alınacak tabaktır. Yiğin, hesaplama amaçlı kullanılan en önemli veri yapılarından biridir. Sistem yazılımları, derleyiciler ve yapay zeka tabanlı geri izleme (backtracking) rutinleri yiğinin sıkça kullanıldığı alanlardan yalnızca birkaçıdır.

Bir yiğini destekleyen koleksiyon sınıfı **Stack** olarak adlandırılır. **Stack** sınıfı **ICollection**, **IEnumerable** ve **ICloneable** arayüzlerini uygular. **Stack**, saklaması gereken elemanları barındırabilmesi için gerektiğinde büyüyen dinamik bir koleksiyondur. Kapasitenin artırılması gerektiğinde kapasite ikiye katlanır.

**Stack**'te aşağıdaki yapılandırıcılar tanımlıdır:

```
public Stack()
public Stack(int kapasite)
public Stack(ICollection c)
```

İlk ifade, **10** değerinde başlangıç kapasitesine sahip boş bir yiğin oluşturur. İkinci ifade, **kapasite** ile belirtilen başlangıç kapasitesine sahip boş bir yiğin oluşturur. Üçüncü ifade ise **c** ile belirtilen koleksiyonun eleman sayısını ve kapasitesini içeren bir yiğin kurar.

**Stack**'in uyguladığı arayüzler tarafından tanımlanan metodlara ek olarak **Stack**'in kendisi de Tablo 22.7'de gösterilen metodları tanımlar. Genel olarak, **Stack**'ı işte şu şekilde kullanırsınız. Yiğinin en üstüne bir nesne eklemek için **Push()**'u çağırın. Yiğinin en üstündeki elemanı çıkarmak ve döndürmek için **Pop()**'u çağırın. **Pop()**'u çağrıdığınızda yiğin eğer boş ise, bir **InvalidOperationException** fırlatılır. Yiğinin en üstündeki nesneyi yiğinden çıkartmadan döndürmek için **Peek()**'i kullanabilirsiniz.

**TABLO 22.7: Stack Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<b>public virtual bool Contains(object v)</b>	<b>v</b> , çağrıda bulunan yiğinde ise <b>true</b> döndürür. Eğer <b>v</b> mevcut değilse, <b>false</b> döndürür.
<b>public virtual void Clear()</b>	<b>Count</b> 'a sıfır değerini atar. Bu, etkin biçimde yiğini temizler.
<b>public virtual object Peek()</b>	Çağrıda bulunan yiğinin en üstündeki nesneyi döndürür, fakat nesne yiğinden çıkartılmaz.

<b>public virtual object Pop()</b>	Çağrıda bulunan yiğinin en üstündeki nesneyi döndürür. Nesne bu işlem sırasında yiğinden çıkartılır.
<b>public virtual object Push(object v)</b>	<b>v</b> 'yi yiğine yerleştirir.
<b>public static Stack Synchronized(Stack stk)</b>	<b>stk</b> üzerinden aktarılan <b>Stack</b> 'ın senkronize edilmiş versiyonunu döndürür.
<b>public virtual object[] ToArray()</b>	Çağrıda bulunan yiğinin elemanlarının kopyalarını içeren bir dizi döndürür.

İşte, bir yiğin oluşturan, bu yiğine birkaç **Integer** nesnesi yerleştiren ve sonra bunları tekrar yiğinden çıkartan bir örnek:

```
// Stack sınıfını gösterir.

using System;
using System.Collections;

class StackDemo {
    static void showPush(Stack st, int a) {
        st.Push(a);
        Console.WriteLine("Push(" + a + ")");

        Console.Write("stack: ");
        foreach(int i in st)
            Console.Write(i + " ");

        Console.WriteLine();
    }

    static void showPop(Stack st) {
        Console.Write("Pop -> ");
        int a = (int) st.Pop();

        Console.WriteLine(a);

        Console.Write("stack: ");
        foreach(int i in st)
            Console.Write(i + " ");

        Console.WriteLine();
    }
}

public static void Main() {
    Stack st = new Stack();

    foreach(int i in st)
        Console.Write(i + " ");

    Console.WriteLine();

    showPush(st, 22);
    showPush(st, 65);
```

```

        showPush(st, 91);
        showPop(st);
        showPop(st);
        showPop(st);

        try {
            showPop(st);
        } catch (InvalidOperationException) {
            Console.WriteLine("Stack empty.");
        }
    )
}

```

Program tarafından üretilen çıktı aşağıdaki gibidir. Yığın boşken yiğinden eleman alınmaya çalışılınca ortaya çıkan **InvalidOperationException**'ı, kural dışı durum yöneticisinin nasıl kontrol altına aldığına dikkat edin.

```

Push(22)
stack: 22
Push(65)
stack: 65 22
Push(91)
stack: 91 65 22
Pop -> 91
stack: 65 22
Pop -> 65
stack: 22
Pop -> 22
stack:
Pop -> Stack empty.

```

## Queue

Bir başka tanık veri yapısı ise kuyruktur. Kuyruk, ilk giren ilk çıkar işleyişine sahip bir listedir. Yani, bir kuyruğa yerleştirilen ilk öğe kuyruktan çıkartılacak ilk öğedir. Kuyruklar gerçek hayatı sıkça karşımıza çıkar. Söz gelişi, bir bankadaki ya da fast-food restoranındaki sıralar kuyruktur. Programlamada kuyruklar; sistemde halen çalışmakta olan prosesler, askıda olan veri tabanı işlemleri ya da Internet üzerinden alınan veri paketleri gibi şeyleri tutmak için kullanılırlar. Ayrıca simülasyonlarda da sıkça kullanılmaktadır.

Bir kuyruğu destekleyen koleksiyon sınıfı **Queue** olarak adlandırılır. **Queue** sınıfı **ICollection**, **IEnumerable** ve **ICloneable** arayüzlerini uygular. **Queue**, saklaması gereken elemanları barındırabilmesi için gerektiğinde büyüyen dinamik bir koleksiyondur. Daha fazla yere ihtiyaç olduğunda kuyruğun büyüklüğü büyümeye faktörü oranında artırılır. Büyümeye faktörünün varsayılan değeri **2.0**'dır.

**Queue**'da aşağıdaki yapılandırmalar tanımlıdır:

```

public Queue()
public Queue (int kapasite)
public Queue (int kapasite, float büyümefaktörü)

```

```
public Queue (ICollection c)
```

İlk ifade, başlangıç kapasitesi **32** olan boş bir kuyruk oluşturur ve **2.0** değerindeki varsayılan büyümeye faktörünü kullanır. İkinci ifade, başlangıç kapasitesi **kapasite** ile belirtilen ve **2.0** büyümeye faktörüne sahip boş bir kuyruk oluşturur. Üçüncü ifade, **büyümeFaktörü** içinde bir büyümeye faktörü belirtmenize olanak tanır. Dördüncü ifade ise, **c** ile belirtilen koleksiyonun eleman sayısına ve kapasitesine sahip bir kuyruk yapılandırır. Bu ifadede **2.0** değerindeki varsayılan büyümeye faktörü kullanılır.

**Queue**'nun uyguladığı arayüzler tarafından tanımlanan metodlara ek olarak **Queue**'nun kendisi de Tablo 22.8'de gösterilen metodları tanımlar. Genel olarak, **Queue**'yu şu şekilde kullanırsınız. Kuyruğa bir nesne yerleştirmek için **Enqueue ()** 'yu çağırın. Kuyruğun başındaki nesneyi çıkarıp döndürmek için **Dequeue ()** 'yu kullanın. **Dequeue ()** ile çağrılan kuyruk boş ise bir **InvalidOperationException** fırlatılır. Bir sonraki nesneyi kuyruktan çıkartmadan döndürmek için **Peek ()** kullanabilirsiniz.

**TABLO 22.8: Queue Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<b>public virtual bool Contains(object v)</b>	<b>v</b> , çağrıda bulunan kuyruk içinde ise <b>true</b> döndürür. Eğer <b>v</b> mevcut değilse, <b>false</b> döndürür.
<b>public virtual void Clear()</b>	<b>Count</b> 'a sıfır değerini atar. Bu, etkin biçimde kuyruğu temizler.
<b>public virtual object Dequeue()</b>	Çağrıda bulunan kuyruğun önündeki nesneyi döndürür. Nesne bu işlem sırasında kuyruktan çıkartılır.
<b>public virtual void Enqueue(object v)</b>	<b>v</b> 'yi kuyruğun sonuna ekler.
<b>public virtual object Peek()</b>	Çağrıda bulunan kuyruğun önündeki nesneyi döndürür, fakat nesne kuyruktan çıkartılmaz.
<b>public static Queue Synchronized(Queue q)</b>	<b>q</b> 'nın senkronize edilmiş versiyonunu döndürür.
<b>public virtual object[] ToArray()</b>	Çağrıda bulunan kuyruğun elemanlarının kopyalarını içeren bir dizi döndürür.
<b>public virtual void TrimToSize()</b>	<b>Capacity</b> 'ye <b>Count</b> 'un değerini atar.

İşte, **Queue** sınıfını gösteren bir örnek:

```
// Queue sınıfını gösterir.

using System;
using System.Collections;
```

```
class QueueDemo {
    static void showEnq(Queue q, int a) {
        q.Enqueue(a);
        Console.WriteLine(".Enqueue(" + a + ")");
        Console.Write("queue: ");
        foreach(int i in q)
            Console.Write(i + " ");
        Console.WriteLine();
    }

    static void showDeq(Queue q) {
        Console.Write("Dequeue -> ");
        int a = (int) q.Dequeue();
        Console.WriteLine(a);

        Console.Write("queue: ");
        foreach(int i in q)
            Console.Write(i + " ");
        Console.WriteLine();
    }

    public static void Main() {
        Queue q = new Queue();

        foreach(int i in q)
            Console.Write(i + " ");

        Console.WriteLine();

        showEnq(q, 22);
        showEnq(q, 65);
        showEnq(q, 91);
        showDeq(q);
        showDeq(q);
        showDeq(q);

        try {
            showDeq(q);
        } catch (InvalidOperationException) {
            Console.WriteLine("Queue empty.");
        }
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
.Enqueue(22)
queue: 22
.Enqueue(65)
queue: 22 65
.Enqueue(91)
queue: 22 65 91
```

```
Dequeue -> 22
queue: 65 91
Dequeue -> 65
queue: 91
Dequeue -> 91
queue:
Dequeue -> Queue empty.
```

## BitArray İle Bitleri Saklamak

**BitArray** sınıfı, bitlerden oluşan bir koleksiyonu destekler. **BitArray**, nesneler yerine bitleri sakladığı için diğer koleksiyonlardan daha farklı becerilere sahiptir. Yine de, **BitArray**; **ICollection** ve **IEnumerable**'ı uygulayarak temel koleksiyon esaslarını da destekler. Ayrıca **ICloneable**'ı da uygulamaktadır.

**BitArray**'de birkaç yapılandırıcı tanımlıdır. Aşağıdaki yapılandırıcıyı kullanarak Boolean değerlerinden oluşan bir diziden bir **BitArray** yapılandırabilirsiniz:

```
public BitArray(bool[] bitler)
```

Bu durumda **bitler**'in her elemanı söz konusu koleksiyonun biti olur. Böylece, koleksiyondaki her bit, **bitler**'in bir elemanına karşılık gelir. Üstelik, **bitler** içindeki elemanların sırası ile koleksiyondaki bitlerin sırası aynıdır.

Aşağıdaki yapılandırıcıyı kullanarak bir byte dizisinden bir **BitArray** oluşturabilirsiniz:

```
public BitArray(byte[] bitler)
```

Burada **bitler** içindeki bit modeli, koleksiyon içindeki bitlerin modeli olur. **bitler[0]**, ilk 8 biti belirtir; **bitler[1]** ikinci 8 biti belirtir vs. Benzer şekilde, aşağıdaki yapılandırıcıyı kullanarak bir tamsayı dizisinden bir **BitArray** yapılandırabilirsiniz:

```
public BitArray(int[] bitler)
```

Bu durumda **bitler[0]** ilk 32 biti; **bitler[1]** ikinci 32 biti vs belirtir.

Şu yapılandırıcıyı kullanarak belirli bir büyülükte bir **BitArray** oluşturabilirsiniz:

```
public BitArray(int büyülük)
```

Burada **büyülük**, bit sayısını belirtir. Koleksiyon içindeki bitlere ilk değer olarak **false** atanır. Büyünlüğü ve bitler için ilk değerleri belirtmek için aşağıdaki yapılandırıcıyı kullanın:

```
public BitArray(int büyülük, bool v)
```

Bu durumda koleksiyondaki tüm bitler **v** üzerinden aktarılan değere göre ayarlanacaktır.

Son olarak, aşağıdaki yapılandırıcıyı kullanarak mevcut olan bir **BitArray**'den yeni bir **BitArray** oluşturabilirsiniz:

```
public BitArray(BitArray bitler)
```

Yeni nesne, **bitler** ile aynı bit koleksiyonunu içerecektir, fakat iki koleksiyon diğer bakımlardan ayrı olacaktır.

**BitArray**'ler indekslenebilir. Her indeks ayrı bir biti belirtir. Sıfır değerine sahip alt bite işaret eder.

**BitArray**'in uyguladığı arayüzler tarafından belirlilen metotlara ek olarak **BitArray**'in kendisi de Tablo 22.9'da gösterilen metotları tanımlar. **BitArray**'in **Synchronized()** metodunu sağlamadığını dikkat edin. Bu nedenle, senkronize edilmiş bir kılıf mevcut değildir ve **IsSynchronized** özelliği daima **false**'tur. Ancak, **SyncRoot** tarafından sağlanan nesne üzerinde senkronizasyon yaparak **BitArray**'e erişimi kontrol edebilirsiniz.

**BitArray**'in uyguladığı arayüzler tarafından belirtilen özelliklere **BitArray** aşağıda gösterilen **Length** özelliğini de ekler:

```
public int Length { get; set; }
```

**Length**, koleksiyon içindeki bitlerin sayısını ayarlar ya da elde eder. Böylece, **Length**; tüm koleksiyonlar için tanımlı olan standart **Count** özelliği ile aynı değeri verir. Ancak, **Count** salt okunur bir özelliktir; fakat **Length** değildir. Böylece, bir **BitArray**'in büyüklüğünü değiştirmek için **Length** kullanabilir. Eğer bir **BitArray**'ı kısaltırsanız, üst uçtaki bitler kesilir. Bir **BitArray**'ı uzatırsanız, üst uca sıfır değerinde bitler eklenir.

**TABLO 22.9: BitArray Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<b>public BitArray And(BitArray ba)</b>	Çağrıda bulunan nesnenin bitleri ile <b>ba</b> ile belirtilen bitlere VE uygular. Sonucu içeren bir <b>BitArray</b> döndürür.
<b>public bool Get(int idx)</b>	<b>idx</b> ile belirtilen indeksteki bitin değerini döndürür.
<b>public BitArray Not()</b>	Çağrıda bulunan koleksiyon üzerinde bit tabanlı mantıksal DEĞİL işlemi gerçekleştirir ve sonucu içeren bir <b>BitArray</b> döndürür.
<b>public BitArray Or(BitArray ba)</b>	Çağrıda bulunan nesnenin bitleri ile <b>ba</b> ile belirtilen bitlere VEYA uygular. Sonucu içeren bir <b>BitArray</b> döndürür.
<b>public void Set(int idx, bool v)</b>	<b>idx</b> ile belirtilen indeksteki bite <b>v</b> değerini verir.
<b>public void SetAll(bool v)</b>	Tüm bitlere <b>v</b> değerini verir.
<b>public BitArray Xor(BitArray ba)</b>	Çağrıda bulunan nesnenin bitleri ile <b>ba</b> ile belirtilen bitlere XOR uygular. Sonucu içeren bir <b>BitArray</b> döndürür.

**BitArray**'de aşağıdaki indeksleyici tanımlıdır:

```
object this[int idks] { get; set; }
```

Bir elemanın değerini almak ya da ayarlamak için bu indeksleyiciyi kullanabilirsiniz.

İşte, **BitArray**'ı gösteren bir örnek:

```
// BitArray'i gösterir.

using System;
using System.Collections;

class BADemo {
    public static void showbits(string rem, BitArray bits) {
        Console.WriteLine(rem);

        for(int i = 0; i < bits.Count; i++)
            Console.Write("{0, -6} ", bits[i]);

        Console.WriteLine("\n");
    }

    public static void Main() {
        BitArray ba = new BitArray(8);
        byte[] b = { 67 };
        BitArray ba2 = new BitArray(b);

        showbits("Original contents of ba:", ba);

        ba = ba.Not();

        showbits("Contents of ba after Not:", ba);

        showbits("Contents of ba2:", ba2);

        BitArray ba3 = ba.Xor(ba2);

        showbits("Result of ba XOR ba2:", ba3);
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Original contents of ba:
False  False  False  False  False  False  False  False

Contents of ba after Not:
True   True   True   True   True   True   True   True

Contents of ba2:
True   True   False  False  False  True   True   False

Result of ba XOR ba2:
False  False  True   True   True   True   False  True
```

## Özelleştirilmiş Koleksiyonlar

.NET Framework belirli bir veri tipi üzerinde ya da belirli bir şekilde çalışmak üzere optimize edilmiş bazı özelleştirilmiş koleksiyonlar sağlar. Bu koleksiyon sınıfları **System.Collections.Specialized** isim uzayında tanımlanmışlardır. Aşağıdaki tabloda bunların bir özeti yer almaktadır:

Özelleştirilmiş Koleksiyon	Açıklama
<b>CollectionsUtil</b>	Karakter katarlarında büyük-küçük harf ayrimını dikkate almayan bir koleksiyon.
<b>HybridDictionary</b>	Koleksiyon içinde birkaç eleman olduğu zaman anahtar/değer çiftlerini saklamak için bir <b>ListDictionary</b> kullanan bir koleksiyon. Koleksiyon belirli bir büyülüüğün ötesine geçerse elemanları saklamak için otomatik olarak bir <b>Hashtable</b> kullanılır.
<b>ListDictionary</b>	Anahtar/değer çiftlerini bir bağlı liste içinde saklayan bir koleksiyon. Yalnızca küçük koleksiyonlar için önerilir.
<b>NameValueCollection</b>	Anahtar ve değerin her ikisinin de <b>string</b> tipinde olduğu anahtar/değer çiftlerinden oluşan bir koleksiyon.
<b>StringCollection</b>	Karakter katarlarını saklamak için optimize edilmiş bir koleksiyon.
<b>StringDictionary</b>	Anahtar ve değerin her ikisinin de <b>string</b> tipinde olduğu anahtar/değer çiftlerinden oluşan bir hash tablo.

**Systems.Collections**'da ayrıca üç özet temel sınıf daha tanımlıdır: **CollectionBase**, **ReadOnlyCollectionBase** ve **DictionaryBase**. Bu sınıflar kalıtım yoluyla aktarılabilir ve kullanımına uygun özelleştirilmiş koleksiyonların geliştirilmesi için başlangıç noktası olarak kullanılabilirler.

## Bir Numaralandırıcı Üzerinden Bir Koleksiyona Erişmek

Genellikle bir koleksiyonun elemanları üzerinde ilerlemek isteyeceksiniz. Söz gelişi, elemanların her birini görüntülemek isteyebilirsiniz. Bunu gerçekleştirmenin bir yolu, önceki örneklerde yaptığı gibi bir **foreach** döngüsü kullanmaktır. Bir diğer yöntem ise bir numaralandırıcı kullanmaktır. Numaralandırıcı, **IEnumerator** arayüzüünü uygulayan bir nesnedir.

**IEnumerator**'da **Current** adından bir tek özellik tanımlıdır. **Current** aşağıda gösterilmiştir:

```
object Current { get; }
```

**Current**, halihazırda numaralandılmakta olan elemanı elde eder. **Current** salt okunur bir özellik olduğu için, bir numaralandırıcı yalnızca koleksiyon içindeki nesneleri almak için kullanılabilir, değiştirmek için kullanılamaz.

**IEnumerator** iki metot tanımlar, İlk **MoveNext()**'tir:

```
bool MoveNext()
```

**MoveNext()**'e yapılan her çağrı, numaralandırıcının mevcut konumunu koleksiyon içindeki bir sonraki elemana kaydırır. Bir sonraki eleman mevcut ise metot **true** döndürür, eğer koleksiyonun sonuna gelinmişse **false** döndürür. **MoveNext()**'e yapılan ilk çağrıdan önce **Current**'ın değeri tanımsızdır.

**Reset()**'ı çağrılmak suretiyle numaralandırıcıyı koleksiyonun başına kaydırarak sıfırlayabilirsiniz:

```
void Reset()
```

**Reset()**'ı çağrıdıktan sonra numaralandırma koleksiyonun başından başlar; ilk elemanı elde etmek için **MoveNext()**'ı çağrımalısınız.

## Numaralandırıcı Kullanmak

Bir koleksiyona bir numaralandırıcı aracılığıyla erişebilmeniz için öncelikle bir numaralandırıcı elde etmiş olmalısınız. Koleksiyon sınıflarının her biri, koleksiyonun başına bir numaralandırıcı döndüren **GetEnumerator()** metodunu sağlarlar. Bu numaralandırıcıyı kullanarak, bir kerede bir elemana erişmek suretiyle koleksiyonun elemanlarının her birine erişebilirisiniz. Genel olarak, bir koleksiyonun içeriği üzerinde dönerek ilerlemek amacıyla bir numaralandırıcı kullanmak için aşağıdaki adımları takip edin:

1. Koleksiyonun **GetEnumerator()** metodunu çağrıarak koleksiyonun başı için bir numaralandırıcı elde edin.
2. **MoveNext()**'e çağrıda bulunan bir döngü kurun. **MoveNext()**, **true** döndürdüğü sürece döngünün tekrarlamasını sağlayın.
3. Döngü içinde **Current** aracılığıyla elemanların her birini elde edin.

İşte, bu adımları uygulayan bir örnek. Bu program bir **ArrayList** kullanmaktadır, fakat genel prensipler her türlü koleksiyon tipi için geçerlidir.

```
// Bir numaralandırıcı gösterir.  
  
using System;  
using System.Collections;  
  
class EnumeratorDemo {  
    public static void Main() {  
        ArrayList list = new ArrayList(1);
```

```

        for(int i = 0; i < 10; i++)
            list.Add(i);

        // Listeye erismek icin numaralandirici kullan.
        IEnumator etr = list.GetEnumerator();
        while(etr.MoveNext())
            Console.Write(etr.Current + " ");

        Console.WriteLine();

        // Listeyi yeniden numaralandir.
        etr.Reset();
        while(etr.MoveNext())
            Console.Write(etr.Current + " ");

        Console.WriteLine();
    }
}

```

Çıktı aşağıda gösterilmiştir:

```

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

```

Genel olarak, bir koleksiyonun üzerinde ilerlemeniz gereklince **foreach** döngüsü numaralandırmadan daha kullanışlıdır. Bununla birlikte; numaralandırıcı kullanımı, isteğe bağlı olarak numaralandırıcıyı sıfırlamanıza olanak tanıyarak küçük bir ek kontrol sunmaktadır.

## IDictionaryEnumerator Kullanımı

**IDictionary**'yi uygulayan bir koleksiyon sınıfı anahtar/değer çiftlerini saklar. Bu tür bir koleksiyonun elemanları üzerinde tekrarlama yaparken **IEnumator** yerine **IDictionaryEnumerator** kullanacaksınız. **IDictionaryEnumerator**, **IEnumator**'dan kalıtımla türetilmiştir; ayrıca üç özellik daha içerir. Bu özelliklerin ilki şudur:

```
DictionaryEntry Entry { get; }
```

**Entry**, bir **DictionaryEntry** yapısı şeklinde olan numaralandırıcıdan bir sonraki anahtar/değer çiftini alır. **DictionaryEntry**'nin **Key** ve **Value** adında iki özellik tanımladığını ve bu özelliklerin koleksiyon girdisinin içeriği anahtar ya da değere erişmek için kullanılabilirliğini hatırlayın. **IDictionaryEnumerator** tarafından tanımlanan diğer iki özellik aşağıda gösterilmiştir:

```
object Key { get; }
object Value { get; }
```

Bunlar anahtar ve değere doğrudan erişmenize imkan verirler.

**IDictionaryEnumerator** típkı sıradan numaralandırıcı gibi kullanılır. Tek fark, bu kez mevcut değeri **Current** yerine **Entry**, **Key** ve **Value** özellikleri aracılığıyla elde

edebileceksiniz. Böylece, bir **IDictionaryEnumerator** elde ettikten sonra ilk elemana erişmek için **MoveNext()**'i çağrılmışınız. Koleksiyondaki kalan elemanları da elde etmek için **MoveNext()**'i çağrımeye devam edin. Artık hiç eleman kalmadığında **MoveNext()** **false** döndürür.

İşte, bir **Hashtable** içindeki elemanları bir **IDictionaryEnumerator** aracılığıyla numaralandıran bir örnek:

```
// IDictionaryEnumerator'i gösterir.

using System;
using System.Collections;

class IDicEnumDemo {
    public static void Main() {
        // Bir hash tablosu oluştur.
        Hashtable ht = new Hashtable();

        // Tabloya eleman ekle.
        ht.Add("Tom", "555-3456");
        ht.Add("Mary", "555-9876");
        ht.Add("Todd", "555-3452");
        ht.Add("Ken", "555-7756");

        // Numaralandırıcıyı göster.
        IDictionaryEnumerator etr = ht.Getenumerator();
        Console.WriteLine("Display info using through Entry.");
        while(etr.MoveNext())
            Console.WriteLine(etr.Entry.Key + ":" + 
                etr.Entry.Value);

        Console.WriteLine();

        Console.WriteLine("Display info using Key and Value
                        directly.");
        etr.Raset();
        while(etr.MoveNext())
            Console.WriteLine(etr.Key + ":" + etr.value);
    }
}
```

Çıktı aşağıda gösterilmiştir:

```
Display info using through Entry.
Tom: 555-3456
Todd: 555-3452
Ken: 555-7756
Mary: 555-9876

Display info using Key and Value directly.
Tom: 555-3456
Todd: 555-3452
Ken: 555-7756
Mary: 555-9876
```

## Koleksiyonlar İçinde Kullanıcı Tarafından Tanımlanan Sınıfları Saklamak

Basit olsun diye önceki örnekler bir koleksiyon içinde **int**, **string** ya da **char** gibi standart tipleri saklamışlardı. Koleksiyonlar elbette standart nesnelerin depolanması ile sınırlı değildir. Aslında amaç bununla oldukça zittir. Koleksiyonların gücü, kendi oluşturduğunuz sınıflara ait nesneler de dahil olmak üzere herhangi tipteki nesneyi saklayabilmesindedir. Örneğin, aşağıdaki programı ele alın. Bu program, **Inventory** sınıfı içine paketlenen envanter bilgilerini saklamak için bir **ArrayList** kullanmaktadır.

```
// Basit bir envanter programı.

using System;
using System.Collections;

class Inventory {
    string name;
    double cost;
    int onhand;

    public Inventory(string n, double c, int h) {
        name = n;
        cost = c;
        onhand = h;
    }

    public override string ToString() {
        return
            String.Format("{0,-10}Cost: {1,6:C}          Onhand: {2}",
                          name, cost, onhand);
    }
}

class InventoryList {
    public static void Main() {
        ArrayList inv = new ArrayList();

        // Listeye eleman ekle.
        inv.Add(new Inventory("Pliers", 5.95, 3));
        inv.Add(new Inventory("Wrenches", 8.29, 2));
        inv.Add(new Inventory("Hammers", 3.50, 4));
        inv.Add(new Inventory("Drills", 19.88, 8));

        Console.WriteLine("Inventory list:");
        foreach(Inventory i in inv) {
            Console.WriteLine(" " + i);
        }
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

Inventory list:

Pliers	Cost:	\$5.95	On hand:	3
Wrenches	Cost:	\$8.29	On hand:	2
Hammers	Cost:	\$3.50	On hand:	4
Drills	Cost:	\$19.88	On hand:	8

Programda, **Inventory** tipinde nesneleri bir koleksiyon içinde saklamak için özel bir işleme gerek olmadığına dikkat edin. Tüm tipler **object**'ten türetildikleri için herhangi tipten bir nesne genel amaçlı herhangi bir koleksiyon içinde saklanabilir.

Yukarıdaki programla ilgili dikkat çekici bir başka husus daha söz konusudur: Program oldukça kısıdadır. Bu programın 40 satırdan daha az bir kod ile envanter bilgilerini saklayabilen, geri alabilen ve işleyebilen dinamik bir dizi kurduğunu ele alırsak, koleksiyonların gücü netlik kazanmaya başlar. Okuyucuların birçoğu bilecektir, bu işlevsellisinin tümü elle kodlanmış olsaydı, program birkaç kat daha uzun olurdu. Koleksiyonlar çok çeşitli programlama problemlerine kesin çözümler sunarlar. Gerektiğinde bunları kullanmalısınız.

Önceki programın yanında fark edilemeyen kısıtlayıcı bir yönü de vardır: Söz konusu koleksiyon sıralanamaz. Bunun gerekçesi, **ArrayList**'in iki **Inventory** nesnesini nasıl karşılaştıracağı hakkında bilgi sahibi olmamasıdır. Bu durumu düzeltmek için iki yöntem mevcuttur. Öncelikle **Inventory**, **IComparable** arayüzüünü uygulayabilir. Bu arayüz, bir sınıfın ait iki nesnenin nasıl karşılaştırılacağını tanımlar. İkincisi, karşılaştırma gerekli olduğunda bir **IComparer** nesnesi belirtilebilir. Aşağıdaki bölümlerde her iki yaklaşım örneklenmektedir.

## IComparable'i Uygulamak

Kullanıcı tarafından tanımlanan nesnelerden oluşan bir **ArrayList**'i sıralamak isterseniz (ya da bu nesneleri bir **SortedList** içinde saklamak isterseniz), söz konusu koleksiyon bu nesneleri nasıl karşılaştıracağını bilmelidir. Bunu gerçekleştirmenin bir yolu, saklanmakta olan nesnenin **IComparable** arayüzüünü uygulamasıdır. **IComparable** yalnızca tek bir metod tanımlar: **CompareTo()**. **CompareTo()**, karşılaştırmaların nasıl gerçekleştirileceğini belirler. **CompareTo()**'nın genel kullanım şekli aşağıda gösterilmiştir:

```
int CompareTo(object nesne)
```

**CompareTo()**, metodu çağrıran nesneyi **nesne** ile karşılaştırır. Artan sırada sıralamak için uygulamanız, nesneler eşitse sıfır döndürmelidir; çağrıda bulunan nesne **nesne**'den büyükse sıfırdan büyük bir değer döndürmeli; çağrıda bulunan nesne **nesne**'den küçükse sıfırdan küçük bir değer döndürmelidir. Karşılaştımanın sonucunu tersinden ele alarak azalan sırada sıralama yapabilirsiniz. Eğer **nesne**'nin tipi, çağrıda bulunan nesnenin tipi ile karşılaştırılmak için uyumlu değilse metod, bir **ArgumentException** fırlatabilir.

İşte, **IComparable**'ın nasıl uygalandığını gösteren bir örnek. Program, önceki bölümde geliştirilen **Inventory** sınıfına **IComparable**'ı ekler. **IComparable**'ı uygulayarak program **Inventory** nesnelerinden oluşan bir koleksiyonunun sıralanmasına olanak tanır.

```
// IComparable'i uygular.

using System;
using System.Collections;

class Inventory : IComparable {
    string name;
    double cost;
    int onhand;

    public Inventory(string n, double c, int h) {
        name = n;
        cost = c;
        onhand = h;
    }

    public override string ToString() {
        return
            String.Format("{0,-10}Cost: {1,6:C}      On hand: {2}",
                          name, cost, onhand);
    }

    // IComparable arayuzunu uygula.
    public int CompareTo(object obj) {
        Inventory b;
        b = (Inventory) obj;
        return name.CompareTo(b.name);
    }
}

class IComparableDemo {
    public static void Main() {
        ArrayList inv = new ArrayList();

        // listeye eleman ekle
        inv.Add(new Inventory("Pliers", 5.95, 3));
        inv.Add(new Inventory("Wrenches", 8.29, 2));
        inv.Add(new Inventory("Hammers", 3.50, 4));
        inv.Add(new Inventory("Drills", 19.88, 8));

        Console.WriteLine("Inventory list before sorting:");
        foreach(Inventory i in inv) {
            Console.WriteLine(" " + i);
        }
        Console.WriteLine();

        // Listeyi sırala.
        inv.Sort();

        Console.WriteLine("Inventory list after sorting:");
        foreach(Inventory i in inv) {
            Console.WriteLine(" " + i);
        }
    }
}
```

Çıktı şu şekildedir. Dikkat ederseniz, `Sort()`'a yapılan çağrıdan sonra envanter isme göre sıralanmıştır.

Inventory list before sorting:

Pliers	Cost: \$5.95	On hand: 3
Wrenches	Cost: \$8.29	On hand: 2
Hammers	Cost: \$3.50	On hand: 4
Drills	Cost: \$19.88	On hand: 8

Inventory list after sorting:

Drills	Cost: \$19.88	On hand: 8
Hammers	Cost: \$3.50	On hand: 4
Pliers	Cost: \$5.95	On hand: 3
Wrenches	Cost: \$8.29	On hand: 2

## IComparer'i Belirtmek

Kendi oluşturduğunuz sınıflara ait nesnelerin bir koleksiyon tarafından sıralanmasına imkan vermek amacıyla bu sınıflar için `IComparable`'ı uygulamak en kolay yöntem olmasına rağmen bu probleme `IComparer` kullanarak farklı bir açıdan yaklaşabilirsiniz. `IComparer`'ı kullanmak için öncelikle `IComparer`'ı uygulayan bir sınıf oluşturun, sonra karşılaşmalar gereklili olduğunda bu sınıfa ait bir nesne belirtin. `IComparer`'de yalnızca bir tek metod tanımlıdır: `Compare()`. `Compare()` aşağıda gösterilmiştir:

```
int Compare(object nesne1, object nesne2)
```

`Compare()`, `nesne1`'i `nesne2` ile karşılaştırır. Uygulamanız, artan sırada sıralamak için, nesneler eşit ise sıfır döndürmelidir; `nesne1`, `nesne2`'den büyükse pozitif bir değer döndürmeli; `nesne1`, `nesne2`'den küçükse negatif bir değer döndürmelidir. Karşılaştırmanın sonucunu tersinden ele alarak azalan sırada sıralama yapabilirsiniz. Çağrıda bulunan nesne ile `nesne` tipi karşılaştırma için uyumlu değilse metot, bir `ArgumentException` fırlatabilir.

`IComparer`; bir `SortedList` yapılandırırken, `ArrayList.Sort(IComparer)` çağrılarında ve koleksiyon sınıfları içinde çeşitli diğer konumlarda belirtilebilir. `IComparer` kullanmanın başlıca avantajı, `IComparable`'ı uygulamayan sınıfların nesnelerini de sıralayabiliyor olmanızdır.

Aşağıdaki program, envanter programının, envanter listesini sıralamak için `IComparer` kullanacak şekilde yeniden çalışılmış bir versiyonudur. Program öncelikle `IComparer`'ı uygulayan `CompInv` adında bir sınıf oluşturur ve iki `Inventory` nesnesini karşılaştırır. Bu sınıfa ait bir nesne daha sonra envanter listesini sıralamak için `Sort()` çağrılarında kullanılır.

```
// IComparer kullanır.

using System;
using System.Collections;

// Inventory nesneleri için bir IComparer oluştur.
class CompInv : IComparer {
```

```
// IComparable arayuzunu uygula.
public int Compare(object obj1, object obj2) {
    Inventory a, b;
    a = (Inventory) obj1;
    b = (Inventory) obj2;
    return a.name.CompareTo(b.name);
}

class Inventory {
    public string name;
    double cost;
    int onhand;

    public Inventory(string n, double c, int h) {
        name = n;
        cost = c;
        onhand = h;
    }

    public override string ToString() {
        return
            String.Format("{0,-10}Cost: {1,6:C}      On hand: {2}",
                          name, cost, onhand);
    }
}

class MailList {
    public static void Main() {
        CompInv comp = new CompInv();
        ArrayList inv = new ArrayList();

        // Elemanları listeye ekle.
        inv.Add(new Inventory("Pliers", 5.95, 3));
        inv.Add(new Inventory("Wrenches", 8.29, 2));
        inv.Add(new Inventory("Hammers", 3.50, 4));
        inv.Add(new Inventory("Drills", 19.88, 8));

        Console.WriteLine("Inventory list before sorting:");
        foreach(Inventory i in inv) {
            Console.WriteLine(" " + i);
        }
        Console.WriteLine();

        // IComparer kullanarak listeyi sırala.
        inv.Sort(comp);

        Console.WriteLine("Inventory list after sorting:");
        foreach(Inventory i in inv) {
            Console.WriteLine(" " + i);
        }
    }
}
```

Çıktı programın bir önceki versiyonunun çıktısı ile aynıdır.

## Koleksiyonların Özeti

Koleksiyonlar, siz programcılara, programlamanın en yaygın görevlerinden bazıları için iyi tasarlanmış, güçlü birtakım çözümler sunar. Bilgileri saklamamanız ve geri almanız gereken bir sonraki işlemde bir koleksiyon kullanmayı düşünün. Koleksiyonların yalnızca “büyük işler” için ayrılmaları gerekmekz. Söz gelişi, kurumsal veri tabanları, posta listeleri ya da envanter sistemleri gibi. Koleksiyonlar daha küçük işlere uygulandıklarında da etkilidirler. Örneğin, bir **SortedList**, günlük randevuların listesini tutmak için mükemmel bir koleksiyon oluşturur. Koleksiyon temelli çözümlerden yararlanacağınız problem türleri, yalnızca hayal gücünüzle sınırlıdır.

**23**

**YİRMİÜÇÜNCÜ BÖLÜM**

---

# **INTERNET İLE AĞ UYGULAMALARI**

C#, modern hesaplama ortamları için tasarlanmış bir dildir. Öyle ki, bu ortamların önemli bir parçasını internet oluşturmaktadır. Bu yüzden, C#'ın tasarımındaki ana kriter, Internet kullanımı için gerekli özellikleri dahil etmekti. Internet'e erişmek, dosyaları indirmek ve kaynakları elde etmek için C ve C++ gibi önceki diller de kullanılabilir olmasına rağmen bu süreç, birçok programcının beğenisi kazanacak kadar standardize edilmemişti. C# bu durumu çözüme kavuşturmuştur. C#'ın ve .NET Library'nin standart özelliklerini kullanılarak uygulamalarınızı "Internet-olanaklı" kılmak kesinlikle kolaydır.

Ağ oluşturma (networking) desteği iki isim uzayında yer almaktadır. Birincisi **System.Net**'tir. **System.Net**, Internet ortamında yaygın, çeşitli tipte işlemleri destekleyen yüksek düzeyli, kullanımı kolay çok sayıda sınıf tanımlar. İkincisi, **System.Net.Sockets**'dir. Bu isim uzayı, ağ oluşturma üzerinde düşük düzeyli kontrol sunan soketleri destekler. **System.Net** tarafından sağlanan sınıflar, sundukları rahatlıktan dolayı bir çok uygulama için daha iyi bir seçenekdir; bu bölümde biz de bu isim uzayını kullanacağız.

**NOT** Sunucu tarafı için destek, ASP.NET tabanlı ağ uygulamaları **System.Web** isim uzayında bulunur.

## System.Net Üyeleri

**System.Net** birçok üye içeren büyük bir isim uzayıdır. Bunların tümünü burada incelemeyecek olmamıza rağmen, nelerin sizin kullanımınıza hazır olduğu hakkında bir fikriniz olması açısından, bu üyeleri listeleye zahmetine değer. **System.Net** tarafından tanımlanan sınıflar aşağıda gösterilmiştir:

<b>AuthenticationManager</b>	<b>Authorization</b>
<b>Cookie</b>	<b>CookieCollection</b>
<b>CookieContainer</b>	<b>CookieException</b>
<b>CredentialCache</b>	<b>Dns</b>
<b>DnsPermission</b>	<b>DnsPermissionAttribute</b>
<b>EndPoint</b>	<b>EndpointPermission</b>
<b>FileWebRequest</b>	<b>FileWebResponse</b>
<b>GlobalProxySelection</b>	<b>HttpVersion</b>
<b>HttpWebRequest</b>	<b>HttpWebResponse</b>
<b>IPAddress</b>	<b>IPEndPoint</b>
<b>IPHostEntry</b>	<b>NetworkCredential</b>
<b>ProtocolViolationException</b>	<b>ServicePoint</b>
<b>ServicePointManager</b>	<b>SocketAddress</b>
<b>SocketPermission</b>	<b>SocketPermissionAttribute</b>
<b> WebClient</b>	<b>WebException</b>
<b>WebHeaderCollection</b>	<b>WebPermission</b>
<b>WebPermissionAttribute</b>	<b>WebProxy</b>
<b>WebRequest</b>	<b>WebResponse</b>

**System.Net**'te aşağıdaki arayüzler tanımlıdır:

**IAuthenticationModule**  
**ICredentials**  
**IWebRequestCreate**

**ICertificatePolicy**  
**IWebProxy**

**System.Net** dört adet numaralandırma tanımlar:

<b>HttpStatusCode</b>	<b>NetworkAccess</b>
<b>TransportType</b>	<b>WebExceptionStatus</b>

Son olarak, **System.Net**'te bir tek delege tanımlanmıştır: **HttpContinueDelegate**.

**System.Net**'te birçok üye tanımlanmış olmasına rağmen, Internet programlama görevlerinin birçoğunu gerçekleştirmek için bunlardan yalnızca birkaç gereklidir. Ağ oluşturmanın çekirdeğinde **WebRequest** ve **WebResponse** isimli özet sınıflar yer almaktadır. Bu sınıflar, belirli bir ağ protokolünü destekleyen sınıflardan kalıtmıla türetilmişlerdir. (Protokol, bir ağ üzerinden bilgi göndermek için kullanılan kuralları tanımlar.) Örneğin, standart HTTP protokolünü destekleyen türetilmiş sınıflar **HttpWebRequest** ve **HttpWebResponse**'tur.

**WebRequest** ve **WebResponse** bazı görevler için kolaylıkla kullanılabilir olsalar bile siz  **WebClient**'a dayanan çok daha basit bir yöntem de kullanabilirsiniz. Söz gelişi, yalnızca bir dosyayı göndermeniz ya da indirmeniz gerekiyorsa, bunu gerçekleştirmenin en iyi yolu genellikle  **WebClient** kullanmaktır.

## Uniform Resource Identifier'lar

Internet programlamanın temelinde Uniform Resource Identifier (URI - Düzgün Kaynak Tanımlayıcısı) yer alır. *URI*, ağ üzerindeki bir kaynağın konumunu tarif eder. URI genellikle *URL* olarak da adlandırılır. URL, Uniform Resource Locator (Düzgün Kaynak Konumlandırıcısı) ifadesinin kısaltılmış şeklidir. Microsoft, **System.Net**'in üyelerini tarif ederken URI terimini kullandığı için, bu kitapta da aynı şekilde ele alınacaktır, URI'ları kuşkusuz biliyorsunuzdur, çünkü ne zaman Internet tarayıcınıza bir adres girerseniz aslında bir URI kullanırsınız.

Bir URI genel olarak şu şekildedir:

Protocol://ServerID/FilePath?Query

**Protocol**, kullanılmakta olan protokolü belirtir; örneğin, bu HTTP olabilir. **ServerID**, belirli bir sunucuyu - söz gelişi Osborne.com ya da Weather.com - belirler. **FilePath**, belirli bir dosya için dosya yolunu belirtir. **FilePath** belirtilmezse, belirtilen **ServerID** üzerindeki varsayılan sayfa elde edilir. Son olarak; **Query**, sunucuya gönderilecek bilgiyi belirtir. **Query** isteğe bağlıdır.

C#'ta URI'lar **Uri** sınıfı içine paketlenmiştir. Bu sınıf, bölüm içinde daha sonra incelenecektir.

## Internet Erişim Esasları

**System.Net** içinde yer alan sınıflar, Internet etkileşiminin istek/yanıt modelini desteklerler. Bu yaklaşımında; programınız, yani istemci, sunucudan bilgi talep eder, sonra sunucunun yanıtını bekler. Söz gelisi, programınız bir Web sitesinin URI'ını sunucuya gönderebilir. Sizin alacağınız yanıt söz konusu URI ile ilişkili hypertext'tir. Bu istek/yanıt yaklaşımının kullanımını hem rahat hem de basittir, çünkü ayrıntıların çoğu sizin için ele alınmıştır.

En üstünde **WebRequest** ve **WebResponse**'un yer aldığı sınıf hiyerarşisi, Microsoft'un takılabilir (pluggable) protokoller olarak adlandırdığı protokoller uygular. Okuyucuların birçoğunun bildiği gibi, birkaç farklı tipte ağ iletişim protokolü mevcuttur. Internet kullanımında en yaygın olanı HTTP'dir (HyperText Transfer Protocol). Bir diğeri ise FTP'dir (File Transfer Protocol). Bir URI kurulduğu zaman URI'nin öneki ilgili protokolü belirtir. Örneğin, `HTTP://MyWebSite.com`, HTTP önekini kullanır. Bu, HyperText Transfer Protocol'ünü belirtir.

Önceden de bahsedildiği gibi, **WebRequest** ve **WebResponse**, tüm protokollerde ortak olan genel istek/yanıt işlemlerini tanımlayan özel sınıflardır. Bu sınıflardan, spesifik protokoller uygulayan somut sınıflar türetilir. Türetilmiş sınıflar, **static** metot **RegisterPrefix()**'i kullanarak kendilerini kaydederler. **RegisterPrefix()**, **WebRequest** tarafından tanımlanmıştır. Bir **WebRequest** nesnesi oluşturduğunuz zaman URI'nin önekinde belirtilen protokol, eğer mevcutsa, otomatik olarak kullanılacaktır. Bu "takılabilir" yaklaşımın sağladığı avantaj şudur: Ne tür bir protokol kullanıyor olursanız olun, kodunuzun büyük bölümü değişmeden aynı kalır.

.NET çalışma sırasında HTTP protokolünü otomatik olarak tanımlar. Böylece, HTTP önekini kullanan bir URL belirtirseniz otomatik olarak bunu destekleyen HTTP uyumlu bir sınıf elde edersiniz. HTTP'yi destekleyen sınıflar **HttpWebRequest** ve **HttpWebResponse**'dur. Bu sınıflar **WebRequest** ve **WebResponse**'dan kalıtımıla türetilmiştir; ayrıca HTTP protokolüne uygulanan birkaç metot daha eklerler.

**System.Net**, hem senkronize hem de asenkron iletişimini destekler. Internet uygulamalarının birçoğunda senkronize işlemler en iyi seçenekdir, çünkü bunların kullanımı kolaydır. Senkronize iletişimde programınız bir istekte bulunur, sonra bir yanıt alınana kadar bekler. Yüksek performanslı bazı uygulama tipleri için asenkron iletişim daha iyidir. Programınız asenkron yaklaşım kullanarak bir yandan bilgilerin iletilmesini beklerken diğer yandan işlemlerine devam edebilir. Ancak, asenkron iletişimini uygulanması daha zordur. Üstelik, asenkron yaklaşımından tüm programlar yararlanamaz. Örneğin, Internet'ten bilgi almaya gerek olduğunda, genellikle bilgi alınana kadar yapılacak bir şey yoktur. Bu tür durumlarda asenkron yaklaşımından elde edilecek potansiyel kazançlar ayırmamız. Senkronize Internet erişimi hem kullanım kolay olduğu için, hem de evrensel olarak daha çok kabul gördüğünden dolayı bu bölümde incelenen tek erişim tipidir.

**WebRequest** ve **WebResponse**, **System.Net**'in kalbinde yer aldığı için bir sonraki adımda bunlar ele alınacaktır.

## WebRequest

**WebRequest** sınıfı, ağ isteklerini yönetir. Bu özel bir sınıfır, çünkü belirli bir protokolü uygulamaz. Ancak, tüm isteklerde ortak olan metotları ve özellikleri uygular. Senkronize iletişimini destekleyen ve **WebRequest** tarafından tanımlanan metotlar Tablo 23.1'de gösterilmiştir, **WebRequest** tarafından tanımlanan özellikler ise Tablo 23.2'de gösterilmiştir. Özelliklerin varsayılan değerleri türetilmiş sınıflarla belirlenir. **WebRequest** açık yapılandırıcı tanımlamaz.

**TABLO 23.1: WebRequest Tarafından Tanımlanan ve Senkronize İletişimi Destekleyen Metotlar**

Metot	Açıklama
<code>public static WebRequest Create(string uri);</code>	<i>uri</i> tarafından aktarılan karakter katarı ile belirtilen URI için bir <b>WebRequest</b> nesnesi oluşturur. Döndürülen nesne, URI'in öneki ile belirtilen protokolü uygulayacaktır. Böylece nesne, <b>WebRequest</b> 'ten kalıtımla türetilen bir sınıfa ait olacaktır. İstenilen protokol mevcut değilse bir <b>NotSupportedException</b> fırlatılır. URI biçimi geçerli değilse, bir <b>UriFormatException</b> fırlatılır.
<code>public static WebRequest Create(Uri uri);</code>	<i>uri</i> tarafından aktarılan karakter katarı ile belirtilen URI için bir <b>WebRequest</b> nesnesi oluşturur. Döndürülen nesne, URI'in öneki ile belirtilen protokolü uygulayacaktır. Böylece nesne, <b>WebRequest</b> 'ten kalıtımla türetilen bir sınıfa ait olacaktır. İstenilen protokol mevcut değilse bir <b>NotSupportedException</b> fırlatılır.
<code>public virtual Stream GetRequestStream()</code>	Önceden istenilen URI ile ilişkili bir çıktı akışı döndürür.
<code>public virtual WebResponse GetResponse()</code>	Önceden oluşturulan isteği gönderir ve yanıt bekler. Bir yanıt alındığında, bu yanıt bir <b>WebResponse</b> nesnesi olarak döndürülür. Programınız belirtilen URI'dan bilgi edinmek için bu nesneyi kullanacaktır. Yanıtı elde ederken bir hata meydana gelirse bir <b>WebException</b> fırlatılır.

**TABLO 23.2: WebRequest Tarafından Tanımlanan Özellikler**

Özellik	Açıklama
<code>public virtual string ConnectionGroupName { get; set; }</code>	Bağlantı grup adını elde eder ya da ayarlar. Bağlantı grupları, bir dizi istek oluşturmanın yoludur. Basit Internet iletişimleri için gerekli değildir.

<code>public virtual long ContentLength { get; set; }</code>	İçeriğin uzunluğunu elde eder ya da ayarlar.
<code>public virtual string ContentType { get; set; }</code>	İçeriğin uzunluğunu elde eder ya da ayarlar.
<code>public virtual ICredentials Credentials { get; set; }</code>	Güven belgesini elde eder ya da ayarlar. Güven belgeleri, kullanıcı kimlik denetimi gerektiren siteler için gereklidir.
<code>public virtual WebHeaderCollection Headers { get; set; }</code>	Başlıklarını elde eder ya da ayarlar.
<code>public virtual string Method { get; set; }</code>	Protokolü elde eder ya da ayarlar.
<code>public virtual bool PreAuthenticate { get; set; }</code>	<code>true</code> ise, istek gönderildiğinde kimlik denetimi bilgisi de gönderilir. <code>false</code> ise, kimlik denetimi bilgisi sadece URI tarafından istendiğinde sağlanır.
<code>public virtual IWebProxy Proxy { get; set; }</code>	Proxy sunucu elde eder ya da ayarlar. Bu sadece, bir proxy sunucu kullanılan ortamlarda geçerlidir.
<code>public virtual Uri RequestUri { get; }</code>	İsteğin URI'ını elde eder.
<code>public virtual int Timeout { get; set; }</code>	İsteğin yanıt vermek için bekleyeceği süreyi milisaniye olarak elde eder ya da ayarlar. Sonsuza kadar beklemek için <code>Timeout.Infinite</code> kullanın.

Bir URI'a bir istek göndermek için öncelikle istenilen protokolü uygulayan ve `WebRequest`'ten türetilen bir sınıfa ait bir nesne oluşturmalısınız. Bu, `Create()` çağrılarıarak gerçekleştirilir. `Create()`, `WebRequest` tarafından tanımlanan `static` bir metottur. `Create()`, `WebRequest`'ten kalıtım yoluyla türetilen ve belirli bir protokolü uygulayan bir sınıfa ait bir nesne döndürür.

**TABLO 23.3: WebResponse Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<code>public virtual void Close()</code>	Yanıtı kapatır. Ayrıca, <code>GetResponseStream()</code> tarafından döndürülen yanıt akışını da kapatır.
<code>public virtual Stream GetResponseStream()</code>	İstenilen URI'a bağlı bir girdi akışı döndürür. Bu akış kullanılarak URI'dan veri okunabilir.

## WebResponse

`WebResponse`, bir isteğin sonucu olarak elde edilen yanıt sınıfına paketler. `WebResponse` özet bir sınıfıdır. `WebResponse`'tan kalıtımıla türetilen sınıflar, `WebResponse`'un bir protokol destekleyen spesifik, somut versiyonlarını oluştururlar. Bir

**WebResponse** nesnesi normal olarak **WebRequest** tarafından tanımlanan **GetResponse()** metodu çağrılarak elde edilir. Bu nesne, spesifik bir protokol uygulayan **WebResponse**'tan türetilen somut bir sınıfın bir örneği olacaktır: **WebResponse** tarafından tanımlanan metodlar yukarıda Tablo 23.3'te gösterilmiştir. **WebResponse** tarafından tanımlanan özellikler ise Tablo 23.4'te gösterilmiştir. Bu özelliklerin değerleri, her yanıt ayrı ayrı bağlı olarak ayarlanır. **WebResponse** açık yapılandırıcı tanımlamaz.

**TABLO 23.4: WebResponse Tarafından Tanımlanan Özellikler**

Özellik	Açıklama
<code>public virtual long ContentLength { get; set; }</code>	Alınmakta olan içeriğin uzunluğunu elde eder ya da ayarlar. İçerik uzunluğu mevcut değilse bunun değeri -1 olacaktır.
<code>public virtual string ContentType { get; set; }</code>	İçeriğin tamamını elde eder.
<code>public virtual WebHeaderCollection Headers { get; }</code>	Söz konusu URI ile ilintili bir başlık koleksiyonu elde eder.
<code>public virtual Uri ResponseUri { get; }</code>	Yanıtı üreten URI'ı ilde eder. Eğer yanıt bir başka URI'a yönlendirilmişse elde edilen, istenilenden farklı olabilir.

## HttpWebRequest ve HttpWebResponse

**HttpWebRequest** ve **HttpWebResponse** sınıfları **WebRequest** ve **WebResponse** sınıflarından kalıtım yoluyla türetilirler ve HTTP protokolünü uygularlar. Bu süreç içinde her iki sınıf, bir HTTP işlemi hakkında ayrıntılı bilgi veren birkaç özellik de ekler. Bu özelliklerin bir kısmı bu bölümün ileriki sayfalarında kullanılmıştır. Ancak, basit Internet işlemleri için bu ek becerileri genellikle kullanmanız gerekmeyecektir.

## İlk Basit Örnek

Internet erişimi **WebRequest** ve **WebResponse** etrafında odaklanır. Bu süreci ayrıntılı olarak incelemeden önce istek/yanıt yaklaşımı ile Internet erişimini gösteren bir örnek görmek yararlı olacaktır. Bu sınıfları çalışırken gördükten sonra, bunların neden bu şekilde organize edildiklerini anlamak daha da kolaylaşır.

Aşağıdaki program, basil ama aynı zamanda çok da yaygın bir Internet işlemini gerçekleştirmektedir. Program, belirli bir URI'da mevcut olan hypertext'i alır. Bu örnekte McGraw-Hill'in (elinizdeki kitabın orijinalinin yayıcısı) Osborne bölümünün web sitesi olan Osborne.com'un içeriği alınmaktadır; fakat siz, bunun yerine - herkes tarafından erişilebilir olması kaydıyla - bir başka web sitesini yerleştirebilirsiniz. Program, hypertext'i ekranda 400 karakterlik parçalar halinde görüntüler; böylece, Internet üzerinden nelerin indirildiğini ekran kaymadan önce görebilirsiniz.

```
// Internet'e erisir.

using System;
using System.Net;
using System.IO;

class NetDemo {
    public static void Main() {
        int ch;

        // Once, bir URI'a bir WebRequest olusturun.
        HttpWebRequest req = (HttpWebRequest)
            WebRequest.Create("http://www.osborne.com");

        // Sonra bu istegi gonderin ve yaniti bekleyin.
        HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

        // Gelen yanittan bir girdi akisi elde edin.
        Stream istrm = resp.GetResponseStream();

        /* Simdi, belirtilen URI'da mevcut olan html'i okuyun ve
           goruntuleyin. Boylece, ekranda neyin gosterilmekte
           oldugunu anlayabilirsiniz; veri, bir kerede 400 karakter
           seklinde gosterilir. Her 400 karakter gosterildikten
           sonra bir sonraki 400 karakteri almak icin ENTER'a
           basmalisiniz. */

        for(int i = 1; ; i++) {
            ch = istrm.ReadByte();
            if(ch == -1) break;
            Console.Write((char) ch);
            if((i % 400) == 0) {
                Console.WriteLine("\nPress a key.");
                Console.Read();
            }
        }

        // Yaniti kapatin. Bu ayni zamanda istrm'i de kapatir.
        resp.Close();
    }
}
```

Çıktının ilk kısmı aşağıda gösterilmiştir. (Tam içerik kuşkusuz zaman içinde değişebilir.)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Find all the right computer books and learning tools at
Osborne McGraw-
Hill</TITLE>
<META NAME="Title" CONTENT="Find all the right computer books and
learning tools at Osborne McGraw-Hill">
<META NAME="Keywords" CONTENT="osborne, mcgraw-hill, mcgraw hill,
it books, computer books, database books, programming"
```

```
Press a key.  
books, networking books, certification books, computing books,  
computer application books, hardware books, information  
technology books, operating systems, web development, oracle  
press, communications, complete refarence, how to do everything,  
yellow pages, book publisher, certification study guide,  
reference book, security, network security, ebusiness,  
e-business, a+, network+, i-net+, cisco ce  
Press a key.  
. . .
```

Bu, hypertext'in Osborne.com web sitesi ile ilintili parçasıdır. Program, içeriği karakter karakter görüntülediği için, tarayıcı çıktısındaki gibi biçimlendirilmemiştir. Ham haliyle görüntülenmiştir.

Gelin şimdi bu programı satır satır inceleyelim. Öncelikle, **System.Net** isim uzayının kullanıldığına dikkat edin. Önceden açıklandığı gibi, ağ oluşturma sınırlarını içeren isim uzayı budur. Ayrıca, **System.IO**'nun da dahil edildiğine dikkat edin. Web sitesinden alınan bilgiler bir **Stream** nesnesi kullanılarak okunduğu için bu isim uzayını kullanmak gereklidir.

Program, istenilen URI'ı içeren bir **WebRequest** nesnesi oluşturarak başlar. Bu amaçla bir yapılandırıcı yerine **Create()** metodunun kullanıldığına dikkat edin. **Create()**, **WebRequest**'in **static** üyesidir. **WebRequest** aslında özel sınıf olsa bile bu sınıfın **static** bir metodunu çağırma her şeye rağmen mümkün değildir. Söz konusu URI'ın protokol önekine bağlı olarak **Create()**, kendisine "takılmış" doğru protokole sahip bir **WebRequest** nesnesi döndürür. Bu örnekte söz konusu protokol HTTP'dir. Böylece, **Create()** bir **HttpWebRequest()** nesnesi döndürür. Kuşkusuz, **Create()**'in döndürdüğü değer, **req** adında **HttpWebRequest** referansına atanırken her şeye rağmen tip ataması yoluyla **HttpWebRequest**'e dönüştürülmelidir. Bu aşamada, söz konusu istek oluşturulmuştur, fakat belirtilen URI'a henüz gönderilmemiştir.

İsteği göndermek için program, **WebRequest** nesnesi üzerinde **GetResponse()**'u çağrıır. İstek gönderildikten sonra **GetResponse()** bir yanıt bekler. Bir kez yanıt alındıktan sonra **GetResponse()**, yanıtı içine saran bir **WebResponse** nesnesi döndürür. Bu nesne **resp**'e atanır. Bu örnekte yanıt, HTTP protokolünü kullandığı için sonuç **HttpWebResponse**'a dönüştürülür. Diğer özelliklerinin yanı sıra yanıt, URI'dan veri okumak için kullanılabilcek bir akış içerir.

Sonra, **resp** üzerinde **GetResponseStream()** çağrılarak bir girdi akışı elde edilir. Bu, diğer herhangi bir girdi akışının tüm nitelik ve özelliklerine sahip standart bir **Stream** nesnesidir. Akışı gösteren bir referans **istrm**'e atanır. **istrm** kullanılarak, belirtilen URI'daki veriler, tıpkı bir dosyadan okunur gibi okunabilir.

Daha sonra program, Osborne.com'dan verileri okur ve bunları ekranда gösterir. Çok fazla miktarda bilgi mevcut olduğu için, ekran her 400 karakterde bir durur ve sizin ENTER tuşuna

basmanızı bekler. Bu şekilde, bilgilerin ilk bölümü ekrandan kayıp gitmeyecektir. Karakterlerin **ReadByte()** kullanılarak okunduğuuna dikkat edin. Hatırlarsanız, bu metot, girdi akışındaki bir sonraki karakteri bir **int** olarak döndürür; bu **int** değer tip ataması yoluyla **char**'a dönüştürülmelidir. Akışın sonuna ulaşıldığında metot, **-1** döndürür.

Son olarak; yanıt, **resp** üzerinde **Close()** çağrılarıarak kapatılır. Yanıt akışını kapatmak otomatik olarak girdi akışını da kapatır. Her istekten sonra yanıt kapatmak önemlidir. Eğer kapatmazsanız, ağ kaynaklarını tüketmek ve bir sonraki bağlantıyı önlemek mümkündür.

Bu örneği terk etmeden önce vurgulanması gereken bir önemli husus daha vardır: Osborne.com web sitesinden okunan hypertext'i görüntülemek için bir **HttpWebRequest** ya da **HttpWebResponse** nesnesi kullanmak aslında gerekli değildi. Önceki program HTTP'ye özgü özellikleri kullanmadığı için **WebRequest** ve **WebResponse** tarafından tanımlanan standart metotlar bu işlemi ele almak için yeterliydi. Böylece, **Create()**'e ve **GetResponse()**'a yapılan çağrılar şu şekilde de yazılabildi:

```
// Once, bir URI'a bir WebRequest oluşturun.  
WebRequest req = WebRequest.Create("http://www.osborne.com");  
  
// Sonra bu isteği gönderin ve yanıt bekleyin.  
WebResponse resp = req.GetResponse();
```

Microsoft, belirli tipte bir protokol uygulaması için tip ataması kullanmanıza gerek olmayan durumlarda **WebRequest** ve **WebResponse** kullanmanın daha iyi olacağını öneriyor. Çünkü bu sayede kodunuza etkilemeden protokollerini değiştirmeniz mümkündür. Ancak, bu bölümdeki örneklerin tümü HTTP ve çok azı HTTP'ye özgü özellikleri kullanıyor olacağı için, programlar **HttpWebRequest** ve **HttpWebResponse** kullanacaklardır.

## Ağ Hatalarını Kontrol Altına Almak

Önceki bölümdeki program doğru olmasına rağmen esnek değildir. En basit ağ hatası bile programın aniden sona ermesine neden olacaktır. Bu bölümde gösterilen örneklerde böyle bir problem söz konusu olmamasına rağmen bu, gerçek dünyaya ait uygulamalarda önlemeniz gereken bir durumdur. Programın üretebileceği ayla ilgili tüm kural dışı durumları tam olarak kontrol altına almak için **Create()**'e, **GetResponse()**'a ve **GetResponseStream()**'e yapılan çağrıları izlemelisiniz. Her tipten olası hata bir sonraki bölümde anlatılmaktadır.

## Create() Tarafından Üretilen Kural Dışı Durumlar

**WebRequest** tarafından tanımlanan **Create()** metodu üç kural dışı durum üretебilir. URI öneki ile belirtilen protokol eğer desteklenmiyorsa **NotSupportedException** fırlatılır. Eğer URI biçimi geçerli değilse, **UriFormatException** fırlatılır. **Create()** ayrıca, eğer **null** referans ile çağrılmışsa, bir **ArgumentNullException** da fırlatabilir, ancak bu, ağa bağlanma ile ilintili olarak üretilen bir hata değildir.

## GetResponse() Tarafından Üretilen Kural Dışı Durumlar

`GetResponse()` çağrılarak bir yanıt elde edilirken eğer bir hata ortaya çıkarsa bir `WebException` fırlatılır. Tüm kural dışı durumlar için tanımlanan üyelerin ek olarak `WebException`, ağ hatalarıyla bağlantılı iki özellik daha ekler: `Response` ve `Status`.

`Response` özelliği aracılığı ile bir kural dışı durum yöneticisi içinde `WebResponse` nesnesine atıfta bulunan bir referans elde edebilirsiniz. Bir kural dışı durum ortaya çıkmış olsaydı, `GetResponse()` tarafından döndürülen nesne bu olurdu. `Response` şu şekilde tanımlanır:

```
public WebResponse Response { get; }
```

Bir hata ortaya çıktığında problemin nereden kaynaklandığını bulmak için `WebException`'ın `Status` özelliğini kullanabilirsiniz. `Status` şu şekilde tanımlanır:

```
public WebExceptionStatus { get; }
```

`WebExceptionStatus`, aşağıdaki değerleri içeren bir numaralandırmadır:

<code>ConnectFailure</code>	<code>ConnectionClosed</code>
<code>KeepAliveFailure</code>	<code>NameResolutionFailure</code>
<code>Pending</code>	<code>PipelineFailure</code>
<code>ProtocolError</code>	<code>ProxyNameResolutionFailure</code>
<code>ReceiveFailure</code>	<code>RequestCanceled</code>
<code>SecureChannelFailure</code>	<code>SendFailure</code>
<code>ServerProtocolViolation</code>	<code>Success</code>
<code>Timeout</code>	<code>TrustFailure</code>

Hatanın nedeni belirlenir belirlenmez, artık programınız uygun bir eylemde bulunabilir.

## GetResponseStream() Tarafından Üretilen Kural Dışı Durumlar

`GetResponse`'un `GetResponseStream()` metodu bir `ProtocolViolationException` fırlatabilir. `ProtocolViolationException`, genel olarak, belirtilen protokole bağlı olarak bir hatanın meydana geldiği anlamına gelir. `ProtocolViolationException`, `GetResponseStream()` ile bağlantılı olduğu için mevcut yanıt akışlarından hiç birinin geçerli olmadığı anlamını taşır. Ayrıca, akışın okunması sırasında bir `IOException` da meydana gelebilir.

## Kural Dışı Durum Yönetimini Kullanmak

Aşağıdaki program, daha önce gösterilen örneğe, ağ ile ilgili olası tüm kural dışı durumlar için kural dışı durum yöneticileri ekler:

```
// Ağ ile ilgili kural dışı durumları kontrol altına almak.

using System;
using System.Net;
using System.IO;

class NetExcDemo {
    public static void Main() {
        int ch;

        try {
            // Önce, bir URI'a bir WebRequest oluşturun.
            HttpWebRequest req = (HttpWebRequest)
                WebRequest.Create("http://www.osborne.com");

            // Sonra bu isteği gönderin ve yanıtını bekleyin.
            HttpWebResponse resp = (HttpWebResponse)
                req.GetResponse();

            // Yanıtın bir girdi akışı elde edin.
            Stream istrm = resp.GetResponseStream();

            /* Şimdi, belirtilen URI'da mevcut olan html'i okuyun ve
               görüntüleyin. Böylece, ekranda neyin gösterilmekte
               olduğunu anlayabilirsiniz; veri, bir kerede 400
               karakter şeklinde gösterilir. Her 400 karakter
               gösterildikten sonra bir sonraki 400 karakteri almak
               için ENTER'a basmalısınız. */

            for(int i = 1; ; i++) {
                ch = istrm.ReadByte();
                if(ch == -1) break;
                Console.Write((char) ch);
                if((i % 400) == 0) {
                    Console.WriteLine("\nPress a key.");
                    Console.Read();
                }
            }

            // Yanınızı kapatın. Bu aynı zamanda istrm'i de kapatır.
            resp.Close();
        } catch(WebException exc) {
            Console.WriteLine("Network Error: " + exc.Message +
                "\nStatus code: " + exc.Status);
        } catch(ProtocolViolationException exc) {
            Console.WriteLine("Protocol Error: " + exc.Message);
        } catch(UriFormatException exc) {
            Console.WriteLine("URI Format Error: " + exc.Message);
        } catch(NotSupportedException exc) {
            Console.WriteLine("Unknown Protocol: " + exc.Message);
        } catch(IOException exc) {
            Console.WriteLine("I/O Error: " + exc.Message);
        }
    }
}
```

```
}
```

Ağ ile ilgili metodların üretebileceği kural dışı durumlar artık yakalanmıştır. Örneğin, `Create()`'e yapılan çağrıyı aşağıdaki gibi değiştirirseniz,

```
WebRequest.Create("http://www.osborne.com/moonrocket");
```

sonra programı yeniden derleyip, çalıştırırsanız şu çıktıyı göreceksiniz:

```
Network Error: The remote server returned an error: (404) Not Found.  
Status code: ProtocolError
```

Osborne.com web sitesinde “moonrocket” isimli bir dizin mevcut olmadığı için böyle bir URI bulunamaz. Çıktı da bunu doğrulamaktadır.

Örnekleri kısa ve sade tutmak amacıyla bu bölümdeki programların birçoğu eksiksiz bir kural dışı yönetim mekanizması içermeyecektir. Ancak, sizin gerçek dünyaya yönelik uygulamalarınız mutlaka içermelidir.

## URI Sınıfı

Tablo 23.1'de `WebRequest.Create()`'in iki farklı versiyonu olduğu dikkatinizi çekerectir. Bunlardan biri, söz konusu URI'ı bir karakter katarı şeklinde kabul eder. Önceki programlarda bu versiyon kullanılmıştır. Diğer ise, URI'ı `Uri` sınıfının bir örneği olarak alır. `Uri` sınıfı bir URI'ı paketler. `Uri` kullanarak, `Create()`'e aktarılabilen bir URI kurabilirsiniz. Ayrıca, bir `Uri`'ı dikkatle inceleyip, parçalarını elde edebilirsiniz. Birçok basit Internet işlemleri için `Uri` kullanmanız gerekmese de, daha sofistike durumlarda `Uri` kullanımının çok değerli olduğunu fark edebilirsiniz.

`Uri`'da birkaç yapılandırıcı tanımlıdır. Bunlardan yaygın olarak kullanılan ikisi aşağıda gösterilmiştir:

```
public Uri(string uri)  
public Uri(string baz, string rel)
```

İlk ifade, bir karakter katarı şeklinde verilen URI'dan bir `Uri` kurar. İkinci ifade, `rel` ile belirtilen bir bağıl URI'ı, `baz` ile belirtilen mutlak baz konumundaki URI'a ekleyerek bir `Uri` yapılandırır. Mutlak URI, komple bir URI tanımlar. Bağıl URI ise yalnızca erişim yolunu tanımlar.

`Uri`, URI'ları yönetmenize yardımcı olan ya da bir URI'ın çeşitli parçalarına erişmenize imkan veren birçok alan, özellik ve metod tanımlar. Kayda değer ölçüde ilgi çekici özellikler aşağıda gösterilmiştir:

### Özellik

```
public string Host { get; }  
public string LocalPath { get; }
```

### Açıklama

Sunucunun adını elde eder.  
Dosya yolunu elde eder.

<code>public string PathAndQuery { get; }</code>	Dosya yolunu ve sorgu karakter katarını elde eder.
<code>public int Port { get; }</code>	Belirtilen protokol için port (kapı) numarasını elde eder. http için port 80'dir.
<code>public string Query { get; }</code>	Sorgu karakter katarını elde eder.
<code>public string Scheme { get; }</code>	Protokolü elde eder.

Bu özellikler bir URI'ı kendisini oluşturan parçalara ayırmak için kullanışlıdır. Aşağıdaki program bunların kullanımını göstermektedir:

```
// Uri kullanımını gösterir.

using System;
using System.Net;

class UriDemo {
    public static void Main() {
        Uri sample =
            new Uri("http://MySite.com/somefile.txt?SomeQuery");
        Console.WriteLine("Host: " + sample.Host);

        Console.WriteLine("Port: " + sample.Port);
        Console.WriteLine("Scheme: " + sample.Scheme);
        Console.WriteLine("Local Path: " + sample.LocalPath);
        Console.WriteLine("Query: " + sample.Query);
        Console.WriteLine("Path and query: " + sample.PathAndQuery);
    }
}
```

Cıktı aşağıda gösterilmiştir:

```
Host: mysite.com
Port: 80
Scheme: http
Local Path: /somefile.txt
Query: ?SomeQuery
Path and query: /somefile.txt?SomeQuery
```

## Ek HTTP Yanıt Bilgilerine Erişim

**HttpResponse**'u kullanırken belirtilen kaynağın içeriği haricindeki diğer bilgilere de erişim iznine sahip olursunuz. URI'nın en son güncellenme zamanı, sunucunun adı bu tür bilgiler kapsamında yer alır. Bu bilgiler yanıt ile ilintili çeşitli özellikler aracılığıyla kullanılabilirler. Bu özellikler, ki **WebResponse** tarafından tanımlanan dördü de buna dahildir, Tablo 23.5'te gösterilmişlerdir. Aşağıdaki bölümlerde tipik örneklerin nasıl kullanılacağı açıklanmaktadır.

TABLO 23.5: HttpWebResponse Tarafından Tanımlanan Özellikler

Özellik	Açıklama
<code>public string CharacterSet { get; }</code>	Kullanılmakta olan karakter setinin adını elde eder.
<code>public string ContentEncoding { get; }</code>	Kodlama yönteminin adını elde eder.
<code>public long ContentLength { get; }</code>	Alınmakta olan içeriğin uzunluğunu elde eder. İçerik uzunluğu mevcut değilse bu değer <code>-1</code> 'dir.
<code>public string ContentType { get; }</code>	İçeriğin tamamını elde eder.
<code>public CookieCollection Cookies { get; set; }</code>	Yanıtla ilişirilmiş çerezlerin listesini elde eder.
<code>public WebHeaderCollection Headers { get; }</code>	Yanıtla ilişirilmiş başlıkların koleksiyonunu elde eder.
<code>public DateTime LastModified { get; }</code>	URI'ın en son değiştirildiği saati elde eder.
<code>public string Method { get; }</code>	Yanıt metodunu belirten bir karakter katarı elde eder.
<code>public version ProtocolVersion { get; }</code>	İşlemde kullanılan HTTP'nin versyonunu tarif eden bir <code>version</code> nesnesi elde eder.
<code>public Uri ResponseUri { get; }</code>	Yanıtı üreten URI'ı elde eder. Eğer yanıt bir başka URI'a yönlendirilmişse elde edilen, istenilenden farklı olabilir.
<code>public string Server { get; }</code>	Sunucunun ismini simgeleyen bir karakter katarı elde eder.
<code>public HttpStatusCode StatusCode { get; }</code>	İşlemin durumunu tarif eden bir <code>HttpStatusCode</code> nesnesi elde eder.
<code>public string StatusDescription { get; }</code>	İşlemin durumunu okunabilir şekilde simgeleyen bir karakter katarı elde eder.

## Başlığa Erişim

`HttpWebResponse` tarafından tanımlanan `Headers` özelliği aracılığıyla bir HTTP yanıtı ile ilintili başlık bilgilerine erişebilirsiniz. `Headers` aşağıda gösterilmiştir:

```
public WebHeaderCollection Headers{ get; }
```

Bir HTTP başlığı, karakter katarı olarak simgelenen isim ve değer çiftlerinden oluşur. Her isim/değer çifti bir `WebHeaderCollection` içinde saklanır. Bu özelleştirilmiş koleksiyon anahtar/değer çiftlerini saklar ve diğer herhangi bir koleksiyon gibi kullanılabilir. (Bölüm 22'ye bakın.) İsimlerden oluşan bir `string` dizisi `AllKeys` özelliği ile elde edilebilir. Bir isim ile ilişkili değeri indeksleyiciyi kullanarak elde edebilirsiniz. İndeksleyici, ya bir nümerik indeks ya da söz konusu ismi kabul edecek şekilde aşırı yüklenmiştir.

Aşağıdaki program, Osborne.com ile ilintili tüm başlıklarını görüntüler:

```
// Basliklari inceler.

using System;
using System.Net;

class HeaderDemo {
    public static void Main() {

        // Bir URI'a bir WebRequest olustur.
        HttpWebRequest req = (HttpWebRequest)
            WebRequest.Create("http://www.osborne.com");

        // Bu istegi gonder ve yaniti dondur.
        HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

        // Isimlerin listesini al.
        string[] names = resp.Headers.AllKeys;

        // Baslikta yer alan isim/deger ciftlerini goster.
        Console.WriteLine("{0,-20}{1}\n", "Name", "Value");
        foreach(string n in names)
            Console.WriteLine("{0,-20}{1}", n, resp.Headers[n]);

        // Yaniti kapat.
        resp.Close();
    }
}
```

Üretilen çıktı aşağıdadır. (Osborne.com'daki başlık bilgilerinin değiştirilebileceğini hatırlınızdan çıkarmayın. Bu nedenle, çıktıda biraz farklı bir şey görebilirsiniz.)

Name	Value
Date	Mon, 14 Jan 2002 17:45:50 GMT
Server	Apache/1.3.9 (Unix) PHP/3.0.14
Keep-Alive	timeout=30, max=500
Connection	Keep-Alive
Transfer-Encoding	chunked
Content-Type	text/html

## Çerezlere Erişim

**HttpWebResponse** tarafından tanımlanan **Cookies** özelliği aracılığıyla bir HTTP yanıtı ile ilintili cerezlere erişim elde edebilirsiniz. Çerezler, bir tarayıcı tarafından saklanan bilgileri içerir. Çerezler, isim/değer çiftlerinden oluşur ve belirli tipte Web erişimini kolaylaştırırlar. **Cookies** şu şekilde tanımlanır:

```
public CookieCollection Cookies { get; set; }
```

**CookieCollection**, **ICollection** ve **IEnumerable** arayüzlerini uygular ve diğer herhangi bir koleksiyon gibi kullanılabilir. (Bölüm 22'ye bakın.) Ayrıca, cerezin indeksini ya da ismini belirterek cerezi elde etmeye olanak tanıyan bir indeksleyici de içerir.

**CookieCollection**, **Cookie** tipinde nesneler saklar. **Cookie**, bir cerez ile ilişkili bilginin çeşitli parçalarına erişmenize imkan veren birkaç özellik tanımlar. Bunlardan burada kullanacağımız ikisi **Name** ve **Value**'dur; şu şekilde tanımlanırlar:

```
public string Name { get; set; }
public string Value { get; set; }
```

Çerezin ismi **Name** içinde yer alır ve değeri de **Value** içindedir.

Bir yanıt ile ilintili cerezlerin listesini elde etmek için söz konusu istek içinde bir cerez konteyneri sağlamalısınız. Bu amaçla, **HttpWebRequest** aşağıda gösterilen **CookieContainer** özelliğini tanımlar:

```
public CookieContainer CookieContainer { get; set; }
```

**CookieContainer**, cerezleri saklamanıza imkan veren çeşitli alan, özellik ve metodlar sunar. Ancak, birçok uygulamada **CookieContainer** ile doğrudan çalışmanıza gerek olmayacağından emin olun. Bunun yerine, yanittan elde ettiğiniz **CookieCollection**'ı kullanacaksınız. **CookieContainer** yalnızca cerezler için alta yatan bir depolama mekanızması sağlar.

Aşağıdaki program, komut satırında belirtilen URI ile ilişkili cerezlerin isimlerini ve değerlerini görüntüler. Bütün web sitelerinin cerez kullanmadığını hatırlınızdan çıkarmayın; bu nedenle, cerez kullanan bir site bulana kadar birkaçını denemek zorunda kalabilirsiniz.

```
/* Cerezleri inceler.

    Bir Web sitesinin kullandığı cerezleri görmek için
    Web sitesinin ismini komut satırında belirtin.
    Ornegin, eger bu programa Cookie adını verirseniz,
        Cookie http://MSN.COM

    MSN.COM ile ilintili cerezi görüntuler.
 */

using System;
using System.Net;

class CookieDemo {
    public static void Main(string[] args) {

        if(args.Length != 1) {
            Console.WriteLine("Usage: CookieDemo <uri>");
            return;
        }

        // Belirtilen URI'a bir WebRequest oluştur.
        HttpWebRequest req = (HttpWebRequest)
```

```
WebRequest.Create(args[0]);  
  
// Bos bir cerez konteyneri al.  
req.CookieContainer = new CookieContainer();  
  
// Isteği gönder ve yaniti dondur.  
HttpWebResponse resp = (HttpWebResponse)  
    req.GetResponse();  
  
// Cerezleri goruntule.  
Console.WriteLine("Number of cookies: " +  
    resp.Cookies.Count);  
Console.WriteLine("{0,-20}{1}", "Name", "Value");  
  
for(int i = 0; i < resp.Cookies.Count; i++)  
    Console.WriteLine("{0, -20}{1}",  
        resp.Cookies[i].Name,  
        resp.Cookies[i].Value);  
  
// Yaniti kapat.  
resp.Close();  
}  
}
```

## LastModified Özelliğinin Kullanımı

Kimi zaman bir URI'ın en son ne zaman güncellendigini bilmek isteyeceksiniz. **HttpWebResponse** kullanılırken bunu keşfetmek kolaydır, çünkü **HttpWebResponse**, **LastModified** özelliğini tanımlar. Bu özellik aşağıda gösterilmiştir:

```
public DateTime LastModified { get; }
```

**LastModified**, söz konusu URI'ın içeriğinin değiştirildiği en son zamanı elde eder.

Aşağıdaki program, Microsoft.com web sitesinin en son güncellendiği saatı ve tarihi görüntüler:

```
// LastModified'i kullanır.  
  
using System;  
using System.Net;  
  
class HeaderDemo {  
    public static void Main() {
```

```
        return uri;
    }

    public static void Main(string[] args) {
        string link = null;
        string str;
```

```
string answer;

int curloc; // yanit icinde mevcut konumu tutar

if(args.Length != 1) {
    Console.WriteLine("Usage: MiniCrawler <uri>");
    return;
}

string uristr = args[0]; // mevcut URI'i tutar

try {

    do {
        Console.WriteLine("Linking to " + uristr);

        // Belirtilen URI'a bir WebRequest olusturur.
        HttpWebRequest req = (HttpWebRequest)
            WebRequest.Create(uristr);

        uristr = null; // bu URI'in daha fazla kullanimina
                        // izin verme
        // Bu istegi gonder ve yaniti dondur.
        HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

        // Yanittan bir girdi akisi elde et.
        Stream istrm = resp.GetResponseStream();

        // Girdi akisini bir StreamReader icine sar.
        StreamReader rdr = new StreamReader(istrm);

        // Butun sayfayı oku.
        str = rdr.ReadToEnd();

        curloc = 0;

        do {
            // Baglanilacak bir sonraki URI'i bul.
            link = FindLink(str, ref curloc);

            if(link != null) {
                Console.WriteLine("Link found: " + link);

                Console.Write("Link, More, Quit?");
                answer = Console.ReadLine();

                if(string.Compare(answer, "L", true) == 0) {
                    uristr = string.Copy(link);
                    break;
                }
                else
                    if(string.Compare(answer, "Q", true) == 0) {
                        break;
                    }
            }
        }
    }
}
```

```

        else
            if(string.Compare(answer, "M", true) == 0) {
                Console.WriteLine("Searching for another
                                  link.");
            }
        } else {
            Console.WriteLine("No link found.");
            break;
        }

    } while(link.Length == 0);

    // Yaniti kapat.
    resp.Close();
} while(uristr != null);

} catch(WebException exc) {
    Console.WriteLine("Network Error: " + exc.Message +
                      "\nStatus code: " + exc.Status);
} catch(ProtocolViolationException exc) {
    Console.WriteLine("Protocol Error: " + exc.Message);
} catch(UriFormatException exc) {
    Console.WriteLine("URI Format Error: " + exc.Message);
} catch(NotSupportedException exc) {
    Console.WriteLine("Unknown Protocol: " + exc.Message);
} catch(IOException exc) {
    Console.WriteLine("I/O Error: " + exc.Message);
}

Console.WriteLine("Terminating MiniCrawler.");
}
}

```

İşte oturumun bir parçası:

```

C:\MiniCrawler http://osborne.com
Linking to http://osborne.com
Link found: http://www.osborne.com/aboutus/aboutus.shtml
Link, More, Quit? M
Searching for another link.
Link found: http://www.osborne.com/downloads/downloads.shtml
Link, More, Quit? L
Linking to http://www.osborne.com/downloads/downloads.shtml
.
.
.

```

Gelin şimdi, Mini Crawler'in nasıl çalıştığını yakından göz atalım. MiniCrawler'in çalışmaya başlayacağı URI, komut satırında belirtilir. Bu URI **Main()**'de **uristr** denilen bir karakter katarı içinde saklanır. Bu URI'a bir istek oluşturulur, sonra **uristr**, **null** olacak şekilde ayarlanır. Bu, bu URI'ın daha önce kullanılmış olduğuna işaret eder. Daha sonra, istek gönderilir ve buna karşılık gelen yanıt elde edilir. **GetResponseStream()** tarafından döndürülen akış, bir **StreamReader** içine sarılarak ve sonra **ReadToEnd()** çağrılarıarak söz

konusu içerik okunur. `ReadToEnd()`, söz konusu akışın bütün içeriğini bir karakter katarı olarak döndürür.

Program daha sonra bu içeriği kullanarak bir bağlantı arar. Bunu, ayrıca MiniCrawlers tarafından da tanımlanan `static` bir metot olan `FindLink()`'ı çağırarak gerçekleştirir. `FindLink()`, içeriği tutan karakter katarı ve aramaya başlanılacak olan başlangıç konumu ile birlikte çağrılır.

Bu değerleri alan parametreler sırasıyla `htmlstr` ve `startloc`'tur. `startloc`'un bir `ref` parametresi olduğuna dikkat edin. `FindLink()` önce içeriği tutan karakter katarının küçük harflerden oluşan bir kopyasını çıkarır, sonra `href="http` - bu bir bağlantıya işaret eder - ile eşlenen bir alt karakter katarı arar. Eşlenen bir değer bulunursa söz konusu URI, `uri`'a kopyalanır ve `startloc`'un değeri bağlantının sonuna işaret edecek şekilde güncellenir. `startloc` bir `ref` parametresi olduğu için bu işlem, `startloc`'un `Main()`'de karşılık gelen argümanının güncellenmesine neden olur. Böylece, bir sonraki aramanın bir öncekinin kaldığı yerden başlaması mümkün kılınır. Son olarak, `uri` döndürülür. `uri`'a başlangıçta `null` değeri atandığı için, bir eşleşme olmaması durumunda, başarısızlık anlamına gelen bir `null` referans döndürülmüş olur.

`Main()`'e geri dönersek, `FindLink()` tarafından döndürülen bağlantı `null` değilse bu bağlantı görüntülenir ve kullanıcıya ne yapacağı sorulur. Kullanıcı `L` tuşuna basarak bu bağlantıya gidebilir, `M` tuşuna basarak mevcut içerik içinde bir başka bağlantı arayabilir ya da `Q` tuşuna basarak programdan çıkabilir. Kullanıcı eğer `L` tuşuna basarsa, söz konusu bağlantı takip edilir ve bağlantının içeriği elde edilir. Sonra, yeni bağlantı içinde bir bağlantı aranır. Bu işlem, olası tüm bağlantılar tükenene kadar sürer.

MiniCrawler'ın gücünü artırmayı ilginç bulabilirsiniz. Söz gelişi, crawler'ın kullanıcı ile etkileşime girmeden, bulunduğu her bağlantı kendisinin gitmesini sağlayacak şekilde, programı tamamen otomatize etmeyi deneyin. Yani, bir başlangıç sayfasından başlayarak bulunduğu ilk bağlantıya gitmesini sağlayın. Sonra, yeni sayfa içinde ilk bağlantıya gitmesini sağlayın vs. Bir kez bir kör noktaya ulaşıldığında artık bir seviye geri gelmesini (backtrack), bir sonraki bağlantıyı bulmasını ve bağlamaya devam etmesini sağlayın. Bu planı başarıyla gerçekleştirmek için URI'ları ve bir URI içindeki aramanın mevcut konumunu tutmak için yiğin kullanmanız gerekecektir. Bunu gerçekleştirmenin bir yolu bir `Stack` koleksiyonu kullanmaktır. Ekstra zorluk katmak için, bağlantıları bir ağaç şeklinde gösteren bir çıktı oluşturmayı deneyin.

## Web Client Kullanımı

Bu bölümü noktalamadan önce `WebClient`'ın kısa bir özeti yapmaya değer. Bu bölümün başlarında bahsedildiği gibi, uygulamanızın Internet üzerinden yalnızca gönderme ya da indirme yapmasına gerek varsa `WebRequest` ve `WebResponse` yerine `WebClient`'i kullanabilirsiniz. `WebClient`'ın avantajı, birçok ayrıntıyı sizin yerinize ele alıyor olmasıdır.

`WebClient`'ta, aşağıda gösterilen, tek yapılandırıcı tanımlıdır:

```
public WebClient()
```

**WebClient**, Tablo 23.6'da gösterilen özellikleri ve Tablo 23.7'de gösterilen metodları tanımlar. Belirtilen URI geçerli değilse metodların tümü bir **UriFormatException** fırlatırlar. Transmisyon sırasında bir hata meydan gelir ise, bir **WebException** fırlatılır.

**TABLO 23.6: WebClient Tarafından Tanımlanan Özellikler**

Özellik	Açıklama
<code>public string BaseAddress { get; set; }</code>	İstenilen URI'ın baz adresini elde eder ya da ayarlar. Bu özelliğin varsayılan değeri <code>null</code> 'dır. Bu özellik eğer ayarlanmışsa <b>WebClient</b> metodları tarafından belirtilen metodlar, baz adresine bağlı olacaktır.
<code>public ICredentials Credentials { get; set; }</code>	Kimlik denetimi (authentication) bilgilerini elde eder ya da ayarlar. Bu özelliğin varsayılan değeri <code>null</code> 'dır.
<code>public WebHeaderCollection Headers { get; set; }</code>	İstek başlıklarının koleksiyonunu elde eder ya da ayarlar.
<code>public NameValueCollection QueryString { get; set; }</code>	Bir isteğe ilişirilebilir isim/değer çiftlerinden oluşan bir sorgu karakter katarı elde eder ya da ayarlar. Sorgu karakter katarı, URI'dan ? ile ayrılır. Eğer birden fazla isim/değer çifti mevcutsa her çift @ ile birbirinden ayrılır.
<code>public WebHeaderCollection ResponseHeaders { get; }</code>	Yanıt başlıklarının koleksiyonunu elde eder.

**TABLO 23.7: WebClient Tarafından Tanımlanan Metotlar**

Metot	Açıklama
<code>public byte[] DownloadData(string uri)</code>	<code>uri</code> ile belirtilen URI'daki bilgileri indirir ve sonucu bir byte dizisinde döndürür.
<code>public void DownloadFile(string uri, string fname)</code>	<code>uri</code> ile belirtilen URI'daki bilgileri indirir ve <code>fname</code> ile belirtilen dosyada saklar.
<code>public Stream OpenRead(string uri)</code>	<code>uri</code> ile belirtilen URI'daki bilgilerin okunabileceği bir girdi akışı döndürür. Okuma sona erdikten sonra bu akış kapatılmalıdır.

<code>public Stream OpenWrite(string uri)</code>	<code>uri</code> ile belirtilen URI'daki bilgilerin yazılabileceği bir çıktı akışı döndürür. Yazma sona erdikten sonra bu akış kapatılmalıdır.
<code>public Stream OpenWrite(string uri, string how)</code>	<code>uri</code> ile belirtilen URI'daki bilgilerin yazılabileceği bir çıktı akışı döndürür. Yazma sona erdikten sonra bu akış kapatılmalıdır. <code>how</code> üzerinden aktarılan karakter katarı, bilgilerin nasıl yazılacağını belirtir.
<code>public byte[] UploadData(string uri, byte[] info)</code>	<code>info</code> ile belirtilen bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür.
<code>public byte[] UploadData(string uri, string how, byte[] info)</code>	<code>info</code> ile belirtilen bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür. <code>how</code> üzerinden aktarılan karakter katarı, bilgilerin nasıl yazılacağını belirtir.
<code>public byte[] UploadFile(string uri, string fname,)</code>	<code>fname</code> ile belirtilen dosya içindeki bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür.
<code>public byte[] UploadFile(string uri, string how, string fname,)</code>	<code>fname</code> ile belirtilen dosya içindeki bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür. <code>how</code> üzerinden aktarılan karakter katarı, bilgilerin nasıl yazılacağını belirtir.
<code>public byte[] UploadValues(string uri, NameValueCollection vals)</code>	<code>vals</code> ile belirtilen koleksiyon içindeki bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür.
<code>public byte[] UploadValues(string uri, string how, NameValueCollection vals)</code>	<code>vals</code> ile belirtilen koleksiyon içindeki bilgileri <code>uri</code> ile belirtilen URI'a yazar. Yanıt döndürülür. <code>how</code> üzerinden aktarılan karakter katarı, bilgilerin nasıl yazılacağını belirtir.

Aşağıdaki program, verileri Internetten bir dosyaya indirmek için `WebClient`'in nasıl kullanıldığını göstermektedir:

```
// Bilgileri bir dosyaya indirmek icin WebClient kullanir.

using System;
using System.Net;
using System.IO;

class WebClientDemo {
    public static void Main() {
        WebClient user = new WebClient();
        string uri = "http://www.osborne.com";
        string fname = "data.txt";

        try {
```

```
        Console.WriteLine("Downloading data from " +
                           uri + " to " + fname);
        user.DownloadFile(uri, fname);
    } catch (WebException exc) {
        Console.WriteLine(exc);
    } catch (UriFormatException exc) {
        Console.WriteLine(exc);
    }

    Console.WriteLine("Download complete.");
}
}
```

Bu program, Osborne.com'daki bilgileri indirir ve **data.txt** adında bir dosyaya yerleştirir. Programda, özellikle, olası kural dışı durumların her ikisinin de kontrol altına alındığı düşünülürse, ne kadar az kod satırının kullanıldığına dikkat edin. Ayrıca, uri tarafından belirtilen karakter katarını değiştirerek herhangi bir URI'dan bilgi indirebilirsiniz.

**WebRequest** ve **WebResponse** size daha fazla kontrol verir ve daha fazla bilgiye erişim olanağı tanır. Ancak, buna rağmen bir çok uygulamanın gerektirdiği **WebClient** olacaktır. Bir uygulamanın tüm gereksinim duyduğu Web'den bilgi indirmek olduğunda **WebClient** özellikle kullanışlıdır. Söz gelisi, bir uygulamanın belgelerle ilgili güncellemeleri elde etmesine olanak tanımak için **WebClient** kullanabilirsiniz.

# C#'I UYGULAMAYA GEÇİRMEK

Kısım III'te C#'ın üç uygulamasına yer verilmektedir. Bunlardan ilki, bileşenlerin oluşturulması ve yönetilmesidir. C#'ın pek çok özelliği doğrudan bileşenlerin oluşturulmasına olanak vermek amacıyla tasarlandığı için bileşenler, C# ile programlamanın önemli bir parçasıdır. İkinci uygulamada, System.Windows.Forms tarafından tanımlanan sınıflar kullanılarak bir Windows programı oluşturulmaktadır. Nihayet kitabın son uygulaması olarak, C#, bir “saf kod” örneğine uygulanmaktadır: Cebirsel deyimlerin hesaplanması için yinelerek inen bir ayırtıcı hazırlanmaktadır.

**24**

**YİRMİ DÖRDÜNCÜ BÖLÜM**

---

# **BİLEŞENLERİN OLUŞTURULMASI**

C#'ın hemen hemen her türlü uygulama yazımında kullanılabilmesine rağmen, C#'ın kullanıldığı en önemli uygulamalardan biri bileşenlerdir. Bileşen tabanlı programlama C# ile öylesine bütünleşmiştir ki, C#'a bazen *bileşen yönetimli* bir dil denir. C# ve .NET Framework, bileşenler üzerine tasarlanmıştır ve bu yapılrken bileşenlerle programlama modeli, eski yaklaşımlara göre önemli ölçüde basitleştirilmiş ve standardize edilmiştir. Örneğin, *bileşen* terimi size COM bileşenlerini ve bunların neden olduğu sıkıntıları anımsatıyorsa, endişe etmeyin, C# tabanlı bileşenleri yazması çok daha kolaydır.

## Bileşen Nedir?

Öncelikle *bileşen* terimini tanımlamakla işe başlayalım. İşte genel bir tanım: Bir bileşen; bağımsız ve yeniden kullanılabilen ikili işlevsellik birimidir. Bu tanım, bir bileşenin dört temel özelliğini tarif etmektedir. Her birini tek tek ele alacağız.

Bir bileşen, bağımsızdır. Her bileşen kendi başına vardır (yani kendi *paketini*) oluşturur. Bir başka ifade ile, bir bileşen, ihtiyacı olan tüm işlevselligi kendi üzerinde taşıır. Ayrıca, bir bileşenin kendi içinde nasıl çalıştığı dış dünyaya açık değildir. Bileşenin tam olarak nasıl uygulandığı, o bileşeni kullanan kodu etkilemeden değiştirilebilir.

Bir bileşen yeniden kullanılabilir. Yani bir bileşen, onun işlevselligine ihtiyaç duyan herhangi bir program tarafından kullanılabilir. Bir bileşeni kullanan programa *istemci (client)* denir. Bir bileşen, istenen sayıda istemci tarafından kullanılabilir.

Bir bileşen tek bir işlevsellik birimidir. Bu, temel bir kavramdır, istemcinin açısından bakıldığında, bir bileşen, spesifik bir işlevi ya da işlevleri yerine getirir.

Bir bileşenin sağladığı işlevsellik, herhangi bir program tarafından kullanılabilir ama bir bileşen kendi başına bir program değildir.

Son olarak, bir bileşenin yeniden kullanılabilir olması ikili biçimde sağlanır. Bu, temel bir kavramdır. Bir bileşeni istenen sayıda istemci kullanabilir ama istemciler bileşenin kaynak koduna erişemez. Bir bileşenin işlevselligi istemcilere o bileşenin **public** üyeleri aracılığı ile sağlanır. Yani bir bileşen, istemcilerine hangi işlevselligi sağladığını ve hangi işlevselligi **private** olarak sakladığını kontrol edebilir.

## Bileşen Modeli

Yukarıda sunulan tanım bir yazılım bileşenini gerçekten de doğru bir şekilde tanımlıyor olsa da, bileşeni temelden etkileyen bir konu daha vardır: Bileşenin uyguladığı model. Bir istemcinin bir bileşeni kullanması için; hem istemcinin hem de bileşenin aynı kural dizisini kullanması gereklidir. Bir bileşenin şeklini ve doğasını belirleyen kurallar dizisine *bileşen modeli* denir. Bir bileşenle bir istemci arasında nasıl bir etkileşim olacağını işte bu bileşen modeli tanımlar.

Bileşen modeli önemlidir; çünkü yeniden kullanılabilir ikili işlevsellik herhangi bir şekilde oluşturulabilir. Örneğin, parametreleri aktarmanın ve değer döndürmenin farklı yolları vardır.

Ayrıca, bellek ya da sistem kaynaklarını ayırmadan ve serbest bırakmanın farklı yolları mevcuttur. İstemcilerin bileşenleri serbestçe kullanabilmesi için, her ikisi de bileşen modeli tarafından tanımlanan kurallara uymalıdır. Aslında, bileşen modeli, istemci ile bileşen arasında her iki tarafın da uymayı kabul ettiği sözleşmeyi tanımlar.

İlginç bir husus şudur: C# ve .NET Framework öncesinde çoğu bileşen COM *bileşeniydi*. Component Object Model anlamına gelen COM, geleneksel Windows ortamı ve C++ için tasarlanmıştı. Bu yüzden COM, C# ve .NET Framework tarafından sağlanan modern bellek yönetiminden yararlanamadı. Sonuçta, COM sözleşmesi, uygulaması zor ve hatalara açtı. Neyse ki, C# ve .NET Framework tüm bu sıkıntıları ortadan kaldırdı. Dolayısıyla, eğer geçmişte

COM'la ilgili kötü deneyimler yaşadıysanız, bileşenlerin C#'ta ne kadar kolay oluşturulabildiğini görmek sizin için hoş bir sürpriz olacaktır.

## C# Bileşeni Nedir?

C#'ın çalışma şeklinden dolayı, herhangi bir sınıf, genel bileşen tanımına uyar. Örneğin, bir sınıf derlendikten sonra ikili şekilde herhangi bir sayıda uygulama tarafından kullanılabilir. Bu, her sınıfın bir bileşen olduğu anlamına gelir mi? Hayır. Bir sınıfın bir bileşen olabilmesi için, o sınıf .NET Framework tarafından tanımlanan bileşen modeline uymalıdır. Neyse ki, bu son derece kolay yerine getirilebilecek bir koşuldu: `System.ComponentModel.IComponent` arayüzüünü uygulamak bunun için yeterlidir.

`IComponent`'in uygulanması zor olmasa da, pek çok durumda daha iyi bir alternatif vardır: Bir sınıf, `System.ComponentModel.Component`'i kalıtım yoluyla alabilir. `Component` sınıfı `IComponent`'in varsayılan bir uygulamasını sağlar. Bu sınıf aynı zamanda bileşenle ilgili başka yararlı özellikler de sağlar. Çoğu bileşen `IComponent`'i kendileri uygulamak yerine `Component`'i kalıtım yoluyla alır, çünkü işin sıkıcı kısmının bir bölümü sizin adınıza yapılmıştır. Böylece, C#'ta bir bileşen oluştururken sizin üzerinize ağır bir iş düşmez.

## Konteynerler ve Siteler

C# bileşenleri ile yakından ilgili iki yapı daha vardır: *Konteynerler* ve *siteler*. Konteyner, bir bileşen grubudur. Konteynerler, birden fazla bileşen kullanan programları sadeleştirir. Bir site ise konteynerlerle bileşenleri birbirine bağlamaya yarar. Bunların her ikisi de, ileride ayrıntılı olarak ele alınmaktadır.

## C# ile COM Bileşenlerinin Karşılaştırması

C# bileşenlerinin uygulanması ve kullanılması, COM tabanlı bileşenlerden daha kolaydır. COM'u tanıyanlar bir COM bileşeni kullanırken bileşenin bellekte kalmasını sağlamak için *referans sayımı* yapmak gerektiğini bilirler. Böyle bir çerçevede, bir bileşene yeni bir referans geldikçe, kodunuz `AddRef()`'i çağrırmalıdır. Bir referans ortadan kalktığında da kodunuz

**Release()**'i çağrımalıdır. Bu yaklaşımın problemi, hataya açık olmasıdır. Neyse ki referans sayımı C# bileşenlerinde gerekli değildir. C#, anlamsız verilerin toplanmasını (garbage collection) kullandığı için bir bileşen, kendisine herhangi bir referans kaldığı sürece bellekte durur ve referans kalmadığı anda bellekten silinir.

**IComponent** ve **Component**, bileşenlerle programlamanın temelini oluşturduğu için şimdi de bunları ele alacağız.

## IComponent

**IComponent**, bileşenlerin uymak zorunda olduğu sözleşmeyi tanımlar. **IComponent**, tek bir özellik ve bir olay belirler. Burada sözü edilen özellik **Site**'dir ve aşağıda gösterilmiştir:

```
ISite Site { get; set; }
```

Site, bileşenin sitesini elde eder ya da atar. Bir site, bileşenin kimliğini belirler. Eğer bileşen bir konteynerde saklanmıyorsa bu özellik **null** değerine sahiptir.

**IComponent** tarafından tanımlanan olay aşağıda gösterildiği gibi **Disposed**'dur.

```
event EventHandler Disposed
```

Bir bileşen yok edildiğinde bunun kendisine bildirilmesini isteyen işlemci **Disposed** olayı ile bir olay yöneticisi kaydeder.

**IComponent**, **Dispose()** metodunu aşağıda gösterildiği şekilde tanımlayan **System.IDisposable**'ı da kalıtım yoluyla alır:

```
void Dispose()
```

Bu metot, nesnenin kullanıyor olabileceği tüm kaynakları serbest bırakır.

## Bileşen

Bir bileşen oluşturmak için **IComponent**'i uygulayabilirsiniz ama kalıtım yoluyla **Component**'ı almak çok daha kolaydır. Çünkü **Component**'te **IComponent**'in varsayılan bir uygulaması mevcuttur. Bu bölümdeki örneklerde bu yaklaşımı kullanacağımız. **Component**'ı kalıtım yoluyla aldığı için, sınıfınız .NET uyumlu bir bileşen oluşturmak için gerekli kuralları otomatik olarak yerine getirmiş olacaktır.

**Component** sınıfı sadece varsayılan yapılandırıcıyı tanımlar. Normal olarak bir **Component** nesnesini direkt olarak yapılandıramazsınız. Çünkü **Component**'in asıl kullanım yeri, oluşturduğunuz bileşenler için bir temel sınıf olmaktadır.

**Component**, iki açık özellik tanımlar. Bunlardan ilki aşağıda gösterilen **Container**'dır.

```
public IContainer Container { get; }
```

**Container**, çağrıran bileşeni tutan konteyneri elde eder. Eğer bileşen orada değilse **null** değeri döndürülür. **Container**'in ataması bileşen değil konteyner tarafından yapılır.

İkinci özellik, **IComponent** tarafından tanımlanan **Site**'tir. **Site**, aşağıda gösterildiği gibi **Component** tarafından sanal bir özellik olarak uygulanır.

```
public virtual ISite Site { get; set; }
```

Bu şekilde bileşenle bağlantılı **ISite** nesnesi elde edilir ya da ona değer atanır. Eğer bileşen bir konteynerde tutulmuyorsa **Site**, **null**'dır. **Site**'ın değeri bileşen değil konteyner tarafından atanır.

**Component**, iki açık metot tanımlar. Bunlardan ilki **ToString()**'i devre dışı bırakın bir metottur. İkincisi de **Dispose()** metodudur. Bunun iki biçimi vardır. İlk şöyledir:

```
public void Dispose()
```

Bu kod, çağrıran bileşen tarafından kullanılan tüm kaynakları serbest bırakır. Metot, **IDisposable** arayüzü tarafından belirlenen **Dispose()** metodunu uygulamaktadır. Bir bileşenle onun kaynaklarını serbest bırakmak için, istemci **Dispose()**'un bu versiyonunu çağıracaktır.

**Dispose()** metodunun ikinci şekli de şöyledir:

```
protected virtual public void Dispose(bool nasil)
```

Eğer **nasil true** ise, metodun bu versiyonu, çağrıran bileşen tarafından kullanılan yönetilen ve yönetilmeyen tüm kaynakları serbest bırakır. Eğer **nasil false** ise, sadece yönetilmeyen kaynaklar serbest bırakılır. **Dispose()**'un bu versiyonu korunmuş olduğundan, istemci kodu tarafından çağrılamaz. Onun yerine, önce **Dispose()**'un ilk versiyonu tarafından çağrılır. Bir başka ifade ile, **Dispose()**'a yapılan ilk çağrı, **Dispose(bool)**'a bir çağrı oluşturmaktadır.

Genelde, sizin yazdığınız bileşen, artık kendisine ihtiyaç kalmadığında, serbest bırakılması gereken kaynakları hala elinde tutuyorsa, **Dispose(bool)**'u devre dışı bırakacaktır. Eğer bileşeninizin elinde herhangi bir kaynak yoksa, **Dispose(bool)**'un **Component** tarafından sağlanan varsayılan uygulaması yeterlidir.

**Component**, bir bileşenin kendi yerel uygulama ortamının dışında örneklenmesi durumunda kullanılan **MarshalByRefObject** nesnesini kalıtım yoluyla alır. Böyle bir uygulama, bir bileşenin bir başka prosesle ya da ağla bağlı başka bir bilgisayarda oluşturulması durumunda ortaya çıkabilir. Metot argümanları ve geri dönüş değerleri gibi verinin alınıp verilebilmesi için, verinin nasıl gönderildiğini tanımlayan bir mekanizma olmalıdır. Varsayılan durumda, bilgi, değer olarak alınıp verilir. Ancak **MarshalByRefObject**'in kalıtım yoluyla alınmasıyla veri, referans olarak aktarılır. Böylece, bir C# bileşeni veriyi referans olarak alıp verir.

## Basit Bir Bileşen

Bu noktada daha fazla teori sunmadan önce, basit bir örnek üzerinde çalışmak yararlı olacaktır. Aşağıdaki programda çok basit bir şifreleme stratejisini uygulayan **CipherComp** adlı bir bileşen oluşturulmaktadır. Bir karakter, kendisine **1** eklenerek kodlanmaktadır. Kodun çözülmesi de **1** çıkarılarak yapılmaktadır. Bir karakter katarım şifrelemek için **Encode()**, **plaintext** argümanı ile çağrılır. Şifrelenmiş bir karakter katarının kodunu çözmek için **Decode()**, **ciphertext** argümanı ile çağrılmaktadır. Her iki durumda da çevrilmiş karakter katarı döndürülmektedir.

```
// Basit bir Cipher bileseni. Bu dosyaya CipherLib.cs adini verin.

using System.ComponentModel;

namespace CipherLib { // bileseni kendi isim uzayina koyun

    /* CipherComp'un Component'i kalitim yoluyla aldigina
       dikkat edin. */
    public class CipherComp : Component {

        // Bir karakter katarini kodla.
        public string Encode(string msg) {
            string temp = "";

            for(int i = 0; i < msg.Length; i++)
                temp += (char) (msg[i] + 1);

            return temp;
        }

        // Bir karakter katarinin kodunu coz.
        public string Decode(string msg) {
            string temp = "";

            for(int i = 0; i < msg.Length; i++)
                temp += (char) (msg[i] - 1);

            return temp;
        }
    }
}
```

Şimdi bu kodu yakından inceleyelim. Önce, dosyanın en üstündeki açıklamanın da işaret ettiği gibi, örneği takip edebilmek için bu dosyaya **CipherLib.cs** adını verin. Bu sayede, eğer Visual Studio IDE kullanıyorsanız, bileşeni daha rahat kullanabilirsiniz. **System.ComponentModel**'in dahil edilmiş olmasına da dikkat edin. Daha önce açıkladığımız gibi, bileşenlerle programlamayı destekleyen isim uzayı budur.

**CipherComp** sınıfı, **CipherLib** adlı bir isim uzayı içinde tutulmaktadır. Bir bileşeni kendi isim uzayı içine koyarak global isim uzayının dağınık bir hal alması önlenir. Bu, takip edilmesi iyi olacak bir uygulamadır, ama teknik olarak gerekli değildir.

**CipherComp** sınıfı **Component**'ı kalıtım yoluyla alır. Yani, **CipherComp**, .NET uyumlu bir bileşen olmak için gerekli kontratın gereklerini yerine getirmektedir. **CipherComp** çok basit olduğundan, kendi başına bileşenle ilgili işlevler yerine getirmesine gerek yoktur. Formaliteye dair tüm bu ayrıntıları **Component** halletmektedir.

Sonra, **CipherComp**'un herhangi bir kaynak ayırmadığına dikkat edin. Spesifik olarak, diğer nesnelere herhangi bir referans da tutmamaktadır. **CipherComp** sadece **Encode()** ve **Decode()** adlı iki metod tanımlamaktadır. **CipherComp** herhangi bir kaynak tutmadığı için **Dispose(bool)**'u uygulamak zorunda değildir. Kuşkusuz hem **Encode()** hem de **Decode()** karakter katarı referansları döndürmektedir ama bu referanslar **CipherComp** nesnesinin değil çağrıran kodundur.

## CipherLib'i Derlemek

Bir bileşenin **exe** dosyası olarak değil **dll** olarak derlenmesi gereklidir. Visual Studio IDE kullanıyorsanız, **CipherLib** için bir Class Library projesi oluşturmanız gerekecektir. Komut satırı derleyicisini kullanıyorsanız, **/t:library** seçeneğini belirtmelisiniz. Örneğin, **CipherLib**'i derlemek için şu komut satırını kullanabilirsiniz:

```
csc /t:library CipherLib.cs
```

Bu komut, **CipherComp** bileşenini tutan **CipherLib.dll** adlı bir dosya oluşturur.

## CipherComp Kullanan Bir İstemci

Oluşturulan bileşen artık kullanıma hazırır. Örneğin, aşağıdaki program **CipherComp**'un bir istemcisidir ve bir karakter katarını şifrelemek ve onun şifresini çözmek için **CipherComp**'u kullanmaktadır.

```
// CipherComp kullanan bir istemci.

using System;
using CipherLib; // CipherComp'un isim uzayını al

class CipherCompClient {
    public static void Main() {
        CipherComp cc = new CipherComp();

        string text = "This is a test";
        string ciphertext = cc.Encode(text);
        Console.WriteLine(ciphertext);

        string plaintext = cc.Decode(ciphertext);
        Console.WriteLine(plaintext);
```

```

        cc Dispose(); // kaynakları serbest bırak
    }
}

```

İstemcinin **CipherLib** isim uzayını dahil edidine dikkat edin. Böylece **CipherComp** bileşeni görünür hale gelmektedir. Kuşkusuz **CipherComp**'a yapılan her referansı uzun uzun ve tam olarak yazmak da mümkündür ama onun isim uzayını dahil etmek daha kolaydır, Daha sonra, **CipherComp** herhangi bir sınıf gibi kullanılmaktadır.

Dikkat ederseniz, programın sonunda **Dispose()** çağrılmaktadır Daha önce açıklanıldığı gibi, **Dispose()**'u çağrıarak istemci, bileşenin elinde bulunması mümkün tüm kaynakların serbest bırakılmasını sağlamaktadır. Bileşenler, diğer C# nesne tipleri ile aynı *anlamsız verilerin toplanması* mekanizmalarını kullansalar da anlamsız verilerin toplanması işi son derece seyrek olarak yapılır. **Dispose()**'u çağrıarak bileşenin kaynaklarını derhal serbest bırakmasını sağlarsınız.

Bu, bir bileşenin bir ağ bağlantısı gibi kısıtlı bir kaynağı elinde tutması gibi bazı durumlarda önemli olabilir. **CipherComp** kendisine ait bir kaynağı elinde tutmadığı için, **Dispose()**'a yapılan çağrı aslında bu örnekte gerekli değildir. Ancak, **Dispose()**, bileşen sözleşmesinin bir parçasını oluşturduğu için bir bileşenle işiniz bittiğinde bu metodu çağrımayı adet edinmeniz iyi bir fikirdir.

İstemci programı derlemek için derleyiciye **CipherLib.dll**'e referansta bulunmasını söylemeniz gereklidir. Bunu sağlamak için /r seçeneğini kullanın. Örneğin, aşağıdaki komut satırı, istemci programın derlenmesini sağlamaktadır.

```
csc /r:CipherLib.dll client.cs
```

Eğer Visual Studio IDE kullanıyorsanız, istemciye referans olarak **CipherLib.dll**'i eklemeniz gereklidir.

Programı çalıştırığınızda aşağıdaki çıktıyı göreceksiniz:

```
Uijt!jt!b!uftu
This is a test
```

## Dispose()’u Devre Dışı Bırakmak

Yukarıda gördüğümüz **CipherComp** bileşeni herhangi bir sistem kaynağı tutmadığı gibi herhangi bir nesne de oluşturmaz ve tutmaz. Bu nedenle, **Dispose(bool)**'u devre dışı bırakmaya gerek kalmamıştır. Ancak, bileşeniniz kaynak tutarsa, kaynakların deterministik (aynı koşullarda her seferinde tam olarak aynı sonucu veren) bir şekilde serbest bırakılmasını sağlamak için genellikle **Dispose(bool)**'u devre dışı bırakmanız gerekecektir. Neyse ki, bu kolay bir iştir.

Başlamadan önce, C#'ın anlamsız verileri toplama mekanizmasına güvenmek yerine, neden bir bileşenin kendi kaynaklarını genellikle kendisinin serbest bırakması gerektiğini

anlamak önemlidir. Bu kitapta daha önce de açıklandığı gibi, sizin programınızın gözüyle bakıldığından, anlamsız verilerin toplanması deterministik bir olay değildir. Anlamsız veriler, nesnelerin geri dönüşümü gerekiğinde değil ihtiyaç oldukça (ya da anlamsız veri toplayıcısının ihtiyaç gördüğü zamanlarda) toplanmaktadır. Bu nedenle, eğer bir bileşen örneğin açık bir dosya gibi bir kaynağı elinde tutuyorsa ve bu kaynağın bir başka program tarafından kullanılabilmesi için serbest bırakılması gerekiyorsa, o zaman kaynağı kullanan bileşenin işi bittiği zaman kaynağı deterministik olarak serbest bırakılmasının bir yolu olmalıdır. Bileşene yönelik tüm referansları kaldırmak problemi çözmez. Çünkü bileşen, ihtiyaç duyulan kaynağa yaptığı referansı bir sonraki anlamsız veri toplama turuna kadar elinde tutmaya devam eder. Problemin çözümü, bileşenin **Dispose (bool)** 'u uygulamasıdır.

**Dispose (bool)** 'u devre dışı bırakırken bazı kurallara uymanız gereklidir:

1. **Dispose (bool)** değeri **true** olan bir argümanla çağrıldığında, sizin versiyonunuz, bileşenle ilgili yönetilen ve yönetilmeyen tüm kaynakları serbest bırakmalıdır. Eğer metod **false** bir argümanla çağrılsa, sizin versiyonunuz -eğer varsa- sadece yönetilmeyen kaynakları serbest bırakmalıdır.
2. **Dispose (bool)**, herhangi bir sakınca doğmadan tekrar tekrar çağrılabilmelidir.
3. **Dispose (bool)**, **Dispose (bool)** 'un temel sınıf uygulamasını çağırmalıdır.
4. Bileşeninizin yok edicisi sadece **Dispose (false)** 'u çağırmalıdır.

İkinci kurala uymak için, bileşeniniz, ne zaman dispose edildiğini takip etmelidir. Bu, genellikle dispose konumunu tutan özel bir alan tutarak yapılır.

Aşağıda **Dispose (bool)** 'u uygulayan iskelet bir bileşene yer verilmektedir.

```
/* Dispose(bool) kullanan bir bilesenin iskeletsel (basit) bir
   uygulaması. */
class MyComp : Component {
    bool isDisposed; // bilesen dispose edildiyse true'dur

    public MyComp {
        isDisposed = false;
        // ...
    }

    ~MyComp () {
        Dispose(false);
    }

    protected override void Dispose (bool dispAll) {
        if (!isDisposed) {
            if (dispAll) {
                // yönetilen kaynakları burada serbest bırak
                isDisposed = true; // bileseni dispose edilmiş olarak
                // işaretle
            }
            // yönetilmeyen kaynakları burada serbest bırak
        }
    }
}
```

```
        base.Dispose(disposing);  
    }  
}
```

**Dispose()** 'u bir bileşenin bir örneği üzerinde çağrıdığınızda, **Dispose(bool)** bileşenin elinde bulunuyor olabilecek tüm kaynakları temizlemek için otomatik olarak çağrılır.

### **Dispose(bool)'un Kullanımı**

**Dispose (bool)**'un kullanımını görmek için, **CipherComp**'u, tüm şifreleme işlemlerinin kaydını tutacak şekilde geliştireceğiz. Program bunu yapmak için bir dosyanın **Encode ()** ve **Decode ()** edilmesi için gelen tüm çağrıların sonucunu kaydedecektil. Bu ek işlevsellik, **CipherComp**'un kullanıcısına saydamdır, yani kullanıcının konudan haberdar olması gerekmekz. Amaca ulaşmak için, **Dispose (bool)**'un bileşene olan ihtiyaç ortadan kalktığında dosyayı kapatması sağlanmıştır. **Dispose (bool)**'un ne zaman ve nasıl çağrıldığını göstermek için koda **WriteLine ()** çağrıları eklenmiştir.

```
// Kutuk dosyasi tutacak sekilde gelistirilmis bir sifreleme
// bileseni.

using System;
using System.ComponentModel;
using System.IO;

namespace CipherLib {

    // Kutuk dosyasi tutan bir Cipher bileseni.
    public class CipherComp : Component {
        static int useID = 0;
        int id; // ornek (instance) kodu
        bool isDisposed; // bilesen dispose edilmis ise true
                         // degerine sahiptir.

        FileStream log;

        // Yapilandirici
        public CipherComp() {
            isDisposed = false; // bilesen dispose edilmemis
            try {
                log = new FileStream("CipherLog" +
                                      useID, FileMode.Create);
                id = useID;
                useID++;
            } catch (FileNotFoundException exc) {
                Console.WriteLine(exc);
                log = null;
            }
        }

        // Yok edici
        ~CipherComp() {
            Console.WriteLine("Destructor for component " + id);
        }
    }
}
```

```

        Dispose(false);
    }

    // Dosyayı kodla. Sonucu dondur ve sakla.
    public string Encode(string msg) {

        string temp = "";

        for(int i = 0; i < msg.Length; i++)
            temp += (char) (msg[i] + 1);

        // Kutuk dosyasına kaydet.
        for(int i = 0; i < temp.Length; i++)
            log.WriteByte((byte) temp[i]);

        return temp;
    }

    // Mesajın kodunu çöz. Sonucu dondur ve sakla.
    public string Decode(string msg) {

        string temp = "";

        for(int i = 0; i < msg.Length; i++)
            temp += (char) (msg[i] - 1);

        // Kutuk dosyasına kaydet.
        for(int i = 0; i < temp.Length; i++)
            log.WriteByte((byte) temp[i]);

        return temp;
    }

    protected override void Dispose(bool dispAll) {
        Console.WriteLine("Dispose(" + dispAll +
                           ") for component " + id);

        if(!isDisposed) {
            if(dispAll) {
                Console.WriteLine("Closing file for " +
                                  "component " + id);
                log.Close(); // sifrelenmiş dosyayı kapat
                isDisposed = true;
            }
            // serbest bırakılacak yönetilmeyen bir kaynak yok.
            base.Dispose(dispAll);
        }
    }
}
}

```

**CipherComp**'un bu versiyonunu yakından inceleyelim. Kod, şu alanlarla başlıyor:

```

static int useID = 0;
int id; // örnek (instance) kodu

```

```
bool isDOdisposed; // bilesen dispose edilmişse true.
FileStream log;
```

İlk alan, statik bir **int**'tir ve **CipherComp**'un her bir örneğinin kimliğini belirlemek için kullanılacaktır. **useID**'deki değer kütük dosyası ismine eklenecektir ve böylece **CipherComp**'un her örneği kendi kütük dosyasını kullanacaktır. Bileşenin kimliği **id** alanında tutulmaktadır. Bu, bileşenin oluşturulduğu andaki **useID** değeridir. Bileşenin dispose edilip edilmediği bilgisi **isDisposed** alanında tutulmuştur. Dördüncü alan daha sonra kütük dosyasına atıfta bulunacak olan bir **FileStream** referansı olan **log**'dur.

Ardından **CipherComp**'un yapılandırıcısı gelmektedir:

```
public CipherComp() {
    isDisposed = false; // bilesen dispose edilmemiş
    try {
        log = new FileStream("CipherLog" + useID, FileMode.Create);
        id = useID;
        useID++;
    } catch (FileNotFoundException exc) {
        Console.WriteLine(exc);
        log = null;
    }
}
```

Yapılandırıcıda **isDisposed**'a başlangıç değeri olarak **false** verilmiştir. Bu, **CipherComp** nesnesinin kullanılabilir olduğunu göstermektedir. Ardından kütük dosyası açılmıştır. Dikkat ederseniz, dosya ismi "CipherLog" ile **useID**'nin karakter katarı cinsinden ifadesinin birleştirilmesiyle oluşturulmaktadır. Daha sonra, **useID**'nin değeri **id**'ye atanmakta ve peşinden **useID** artırılmaktadır. Böylece, **CipherComp**'un her örneği ayrı bir kütük dosyası kullanır ve her örneğin benzersiz bir kimlik numarası vardır. Burada söz konusu olan önemli noktalardan biri, bir **CipherComp** nesnesi oluşturulması ile bir dosya açılmış olmasıdır. Bu dosya, bileşenin artık ona ihtiyaç duymadığı an serbest bırakılması gereken bir sistem kaynağıdır. Ancak istemcinin dosyayı serbest bırakma konusunda doğrudan bir yeteneği yoktur. Hatta, istemci bir dosya açıldığından bile habersizdir. Bu nedenle, dosya kapatma işi **Dispose(bool)** tarafından halledilmelidir.

**Encode()**, karakter katarı argümanını şifrelemekte ve sonucu döndürmektedir. Bu metot aynı zamanda sonucu bir kütük dosyasına da kaydetmektedir. Kütük dosyası açık kaldığı için **Encode()**'a gelen yeni çağrılar, dosyaya yeni bilgiler eklenmesiyle sonuçlanır. Örneğin, **Encode()**, üç farklı karakter katlarını şifrelemek için kullanılırsa, şifrelenmiş üç karakter katlarını da tutan bir kütük dosyası ortaya çıkacaktır. **Decode()** da aynı şekilde çalışmaktadır. Tek farkı, argümanını deşifre etmesidir.

Şimdi, **CipherComp** tarafından devre dışı bırakılan **Dispose(bool)**'a yakından bakalım. Kolaylık olsun diye kod aşağıda gösterilmiştir:

```
protected override void Dispose(bool dispAll) {
    Console.WriteLine("Dispose(" + dispAll +
```

```

        ") for component " + id);

    if(!isDisposed) {
        if(dispAll) {
            Console.WriteLine("Closing file for " +
                "component " + id);
            log.Close(); // sifrelenmis dosyayı kapat
            isDisposed = true;
        }
        // serbest bırakılacak yönetilmeyen bir kaynak yok
        base.Dispose(dispAll);
    }
}

```

**Dispose(bool)**'un **protected** olduğuna dikkat edin. Bu metot, istemci kodundan çağrılmayacaktır. Bunun yerine, **Component** tarafından uygulanan ve açık olarak erişilebilen **Dispose()** metodu tarafından çağrılmacaktır. **Dispose(bool)**'un içinde **isDisposed** değeri kontrol edilir. Nesne önceden dispose edilmişse, hiç bir eylemde bulunulmaz. Eğer **isDisposed**, **false** ise **dispAll** parametresi kontrol edilir. Eğer bu parametre **true** ise kütük dosyası kapatılır ve **isDisposed**, **true** değerini alacak şekilde ayarlanır. **dispAll**, **true** olduğunda, kural gereği tüm kaynakların serbest bırakılması gerektiğini hatırlayın. **dispAll**, **false** iken yalnızca yönetilmeyen kaynaklar (bu örnekte bu tür kaynaklar mevcut değildir) serbest bırakılmalıdır. Son olarak, temel sınıf (bu örnekte, **Component**) tarafından uygulanan **Dispose(bool)** çağrılrı. Bu temel sınıf tarafından kullanılan her türlü kaynağın serbest bırakılmasını garanti eder. **WriteLine()** çağrıları yalnızca örnek göstermek amacıyla dahil edilmiştir. Gerçek dünyaya ait bir uygulamada kullanılan **Dispose(bool)** metodu bu çağrıları içemeyecektir.

Şimdi **CipherCom** için yok ediciye göz atalım; bu fonksiyon aşağıda gösterilmiştir:

```

-CipherComp() {
    Console.WriteLine("Destructor for component " + id);
    Dispose(false);
}

```

Söz konusu yok edici, **false** argümanı ile birlikte **Dispose(bool)**'u çağırmaktadır. Bunun gerekçesi basittir. Eğer söz konusu bileşen için ilgili yok edici çalışmakta ise, bileşen anlamsız veri toplayıcısı tarafından geri dönüştürülmüyor dernektr. Bu durumda, yönetilen kaynakların tümü otomatik olarak serbest bırakılacaktır. Yok edicinin yapması gereken tek şey, yönetilmeyen kaynakları serbest bırakmaktır. **WriteLine()** çağrısı yalnızca örnek göstermek amacıyla kullanılmıştır; gerçek bir programda bu çağrı yer almayacaktır.

**CipherComp**'a yapılan değişiklikler **CipherComp**'un açık arayüzüne etkilemediği için **CipherComp** daha önce kullanıldığı şekilde kullanılabilir. Örneğin, iki karakter katarını şifreleyen ve şifrelerini çözen bir istemci programı aşağıda yer almaktadır:

```

// CipherComp kullanan başka bir istemci.

using System;

```

```

using CipherLib; // CipherComp'un isim uzayini al

class CipherCompClient {
    public static void Main() {
        CipherComp cc = new CipherComp();

        string test = "Testing";

        string ciphertext = cc.Encode(text);
        Console.WriteLine(ciphertext);

        string plaintext = cc.Decode(ciphertext);
        Console.WriteLine(plaintext);

        text = "Components are powerful.";

        ciphertext = cc.Encode(text);
        Console.WriteLine(ciphertext);

        plaintext = cc.Decode(ciphertext);
        Console.WriteLine(plaintext);

        cc.Dispose(); // kaynaklari serbest bırak
    }
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

Uftujoh
Testing
Dpnqpopfout!bsf!qpxfsgvm/
Components are powerful.
Dispose(True) for component 0
Closing file for component 0

```

Program çalıştırıldıktan sonra **CipherLogo** adında bir kütük dosyası aşağıdakileri içerecektir:

```
UftujohTestingDpnqpopfout!bsf!qpxfsgvm/Components are powerful.
```

Bu, şifrelenen ve şifreleri çözülen iki karakter katarının peş peşe eklenmiş halidir.

**Dispose(bool)**'un **true** argümanı ile çağrıldığına çıktıda dikkat edin. Bunun nedeni, programın **Dispose()**'u **CipherComp** nesnesi üzerinde çağrımasıdır. Önceden de açıklandığı gibi, **Dispose()** daha sonra **Dispose(bool)**'u **true** argümanı ile çağırır; bu, tüm kaynakların serbest bırakılmasını işaret eder. Alıştırma olarak istemci programındaki **Dispose()** çağrısını açıklama yoluyla programdan çıkarın, sonra programı derleyin ve çalıştırın. Artık şu çıktıyi göreceksiniz:

```

Uftujoh
Testing
Dpnqpopfout!bsf!qpxfsgvm/
Components are powerful.
Destructor for component 0
Dispose(False) for component 0

```

```
Dispose(False) for component 0
```

**Dispose()**'a çağrı yapılmadığı için **CipherComp** bileşeni açıkça dispose edilmez. Ancak, program sona erdiğinde kuşkusuz yok edilir. Böylece, çıktıdan da görüldüğü gibi, bileşenin kendi yok edicisi çağrılr; bu da sırası geldiğinde **Dispose(bool)**'u bir **false** argümanla çağırır. **Dispose(bool)**'a yapılan ikinci çağrı, **CipherComp**'un **Dispose(bool)** metodu içindeki **Dispose()**'un temel sınıf versiyonuna yapılan çağrıdan dolayı meydana gelir. Bu durum **Dispose(bool)**'un ikinci kez çağrımasına neden olur. Bu örnekte bu gereksizdir, fakat **Dispose()**, nasıl çağrıldığının farkında olamayacağı için bu önlenemez ancak zararsızdır.

**Dispose(bool)**'un uygulanışı ile ilgili aynı temel yaklaşım kendi oluşturduğunuz herhangi bir bileşen tarafından da kullanılabilir.

## Dispose Edilmiş Bir Bileşenin Kullanımını Önlemek

Yukarıda gösterildiği üzere **CipherComp** kendi kendisini tam anlamıyla dispose ediyor olsa bile, yine de, herhangi bir gerçek dünya bileşeni tarafından yönlendirilmesi gereken bir problem içermektedir: **CipherComp**, bir istemcinin dispose edilmiş bir bileşeni kullanma girişimini önlemektedir. Söz gelişi, bir istemcinin bir **CipherComp** bileşenini dispose etlikten sonra bu bileşen üzerinde **Encode()**'u çağrıma girişiminden bu istemciyi hiç bir şey alıkoyamaz. Neyse ki, bunun çözümü kolaydır: Söz konusu bileşenin her kullanımında **isDisposed** alanını kontrol etmesi yeterlidir. Örneğin, **Encode()** ve **Decode()**'u daha iyi yazmanın yolları aşağıda gösterilmiştir:

```
// Dosyayı şifrele. Sonucu dondur ve sakla.
public string Encode(string msg) {

    // Dispose edilmiş bir bilesenin kullanılmasını engelle.
    if(isDisposed) {
        Console.WriteLine("Error: Component disposed.");
        return null;
    }

    string temp = "";

    for(int i = 0; i < msg.Length; i++)
        temp += (char) (msg[i] + 1);

    // Bir kutuk dosyasında sakla
    for(int i = 0; i < temp.Length; i++)
        log.WriteByte((byte) temp[i]);

    return temp;
}

// Mesajın şifresini çöz. Sonucu dondur ve sakla.
public string Decode(string msg) {

    // Dispose edilmiş bir bilesenin kullanımını engelle.
```

```

    if(isDisposed) {
        Console.WriteLine("Error: Component disposed.");
        return null;
    }

    string temp = "";

    for(int i = 0; i < msg.Length; i++)
        temp += (char) (msg[i] - 1);

    // Bir kutuk dosyasında sakla
    for(int i = 0; i < temp.Length; i++)
        log.WriteByte((byte) temp[i]);

    return temp;
}

```

Her iki metotta da, eğer **isDisposed**, **true** ise bir hata mesajı görüntülenir ve başka bir eylemde bulunulmaz. Kuşkusuz gerçek dünya ile ilgili bir bileşende, eğer dispose edilmiş bir nesne kullanılmaya çalışılırsa normal olarak programınız bir kural dışı durum fırlatacaktır.

## using İfadesini Kullanmak

Bölüm 18'de açıklandığı gibi, bir nesneyi serbest bırakmak için açıkça **Dispose()** metodunu çağrırmak yerine, bu işlemin otomatik olarak gerçekleşmesi için **using** ifadesi kullanılabilir. **using** ifadesinin aşağıdaki gibi genel kullanımı olduğunu hatırlayın:

```

using (nesne) {
    // nesne'yi kullanır
}

using (tip nesne = ilkDeğer) {
    // nesne'yi kullanır
}

```

Burada **nesne**, **using** bloğu içinde kullanılmakta olan bir nesnedir. İlk ifadede, söz konusu nesne **using** ifadesinin dışında deklare edilir, ikinci ifadede, nesne **using** ifadesinin içinde deklare edilir. Blok sona ererken **Dispose()** metodu otomatik olarak çağrılır. **using** ifadesi yalnızca **System.IDisposable** arayüzünu uygulayan nesnelere uygulanır. (Elbette tüm bileşenler bu arayüzü uygular.)

İşte, **CipherComp** için, doğrudan **Dispose()**'u çağrırmak yerine **using**'ı kullanan bir istemci:

```

// Using ifadesini kullanır.

using System;
using CipherLib; // CipherComp'un isim uzayını import et

class CipherCompClient {
    public static void Main() {

```

```
// cc bu blogun sonunda dispose edilecektir.
using(CipherComp cc = new CipherComp()) {

    string text = "The using statement.";

    string ciphertext = cc.Encode(text);
    Console.WriteLine(ciphertext);

    string plaintext = cc.Decode(ciphertext);
    Console.WriteLine(plaintext);
}

}
```

Programın çıktısı aşağıda gösterilmiştir:

```
Uif!vtjoh!tubufnfou/
The using statement.
Dispose(True) for component 0
Closing file for component 0
```

Çıktıdan da görüldüğü gibi, **using** bloğu sona erince **Dispose()** otomatik olarak çağrılmıştır. **using**'i kullanmak ya da açıkça **Dispose()**'u çağrırmak size kalmıştır. Ancak, **using** kesinlikle kodunuzu kolaylaştırır.

## Konteynerler

Bileşenlerle çalışırken bunları bir konteyner içinde saklamanın yararlı olabileceğini fark edebilirsiniz. Önceden de açıklandığı gibi, bir konteyner bir grup bileşen tanımlar. Konteynerin başlıca avantajı, bir grup bileşenin ortaklaşa yönetilmesine olanak tanımasıdır. Söz gelişî, bir konteyner üzerinde **Dispose()**'u çağrıarak konteyner içindeki tüm bileşenleri dispose edebilirsiniz. Genel olarak konteynerler, birden fazla bileşenle çalışmayı çok daha kolaylaştırırlar.

Bir konteyner oluşturmak için **System.ComponentModel** isim uzayında tanımlanan **Container** sınıfına ait bir nesne oluşturacaksınız. Konteyner için yapılandırıcı şu şekildedir:

```
public Container()
```

Bu ifade boş bir konteyner oluşturur.

Bir kez bir **Container** nesnesi oluşturulduktan sonra artık **Add()**'i çağrıarak buna bileşenler eklersiniz. **Add()**'in aşağıda gösterilen iki kullanımı vardır:

```
public virtual void Add(IComponent bileşen)
public virtual void Add(IComponent bileşen, string bileşenAdı)
```

İlk ifade **bileşeni** konteynere ekler. İkinci ifade **bileşeni** konteynere ekler ve buna **bileşenAdı** ile belirtilen ismi verir. İsim benzersiz olmalıdır ve büyük küçük harf ayrimı dikkatli olmalıdır.

cate alımamalıdır. Zaten konteynerde olan bir ismi kullanmaya çalışırsanız, bir **ArgumentException** fırlatılır. Bu isim, bileşenin **site** özelliği aracılığıyla elde edilebilir.

Bir konteynerden bir bileşen çıkarmak için aşağıda gösterilen **Remove()**'u kullanın:

```
public virtual void Remove(IComponent bileşen)
```

Bu metodun başarılı olması beklenir, çünkü ya metot **bileşen**'i konteynerden çıkartır ya da **bileşen** ilk etapta konteynerde değildir. Her iki durumda da, **Remove()**'a yapılan çağrıdan sonra bileşen artık konteynerde mevcut değildir.

**Container**, **Dispose()**'u uygular. **Dispose()** bir konteyner üzerinde çağrıldığı zaman konteyner içindeki tüm bileşenler kendi **Dispose()** metodlarının çağrılmasını sağlayacaklardır. Böylece, tek bir **Dispose()** çağrıları ile bir konteyner içindeki tüm bileşenleri dispose edebilirsiniz.

**Container**, **Components** adında bir özellik tanımlar. **Components** şu şekilde tanımlanır:

```
public virtual ComponentCollection Components { get; }
```

Bu özellik, çağrıda bulunan konteyner içinde saklanan bileşenlerin bir koleksiyonunu elde eder.

**Component**'ın **Container** ve **Site** adında iki özellik tanımladığını hatırlayın. Bu özellikler tüm türetilmiş bileşenlerde yer alacaktır. Bir bileşen bir konteyner içinde saklandığı zaman söz konusu nesnenin **Container** ve **Site** özellikleri otomatik olarak ayarlanır. **Container** özelliği, söz konusu bileşenin saklandığı konteynere referansta bulunur. **Site**, bileşenle ilgili bilgilere referansta bulunur. Bileşenin adı, bileşenin içinde saklandığı konteynerin adı ve bileşenin tasarım kipinde olup olmadığı bu bilgiler kapsamında yer alır. **Site** özelliği bir **ISite** referansı döndürür. Bu arayüzde aşağıdaki özellikler tanımlıdır:

Özellik	Açıklama
<b>IComponent Component { get; }</b>	Söz konusu bileşen için bir referans elde eder.
<b>IContainer Container { get; }</b>	Söz konusu konteyner için bir referans elde eder.
<b>bool DesignMode { get; }</b>	Eğer bileşen bir tasarım aracı tarafından kullanılıyorsa bu, doğru değerine sahiptir.
<b>string Name { get; set; }</b>	Söz konusu bileşenin adını alır ya da ayarlar.

Çalışma sırasında kullanılan konteyner hakkında ya da bileşenin çalışma sırasında durumu hakkında bilgi edinmek için bu özelliklerini kullanabilirsiniz.

## Konteynerin Gösterilmesi

Aşağıdaki program, iki **CipherComp** bileşenini saklamak için bir konteyner kullanmaktadır. İlk bileşen bir isim belirtmeden konteynere eklenir. İkinci bileşene “Second Component” adı verilir. Bir sonraki adımda, her iki bileşen üzerinde işlemler gerçekleştirilir. Sonra, ikinci bileşenin adı kendi **Site** özelliği aracılığı ile görüntülenir. Son olarak, konteyner üzerinde **Dispose()** çağrılarak her iki bileşen serbest bırakılır. (Burada bir **using** ifadesi kullanılmış olabilirdi. Ancak, örnek vermek amacıyla **Dispose()** açık olarak çağrılmıştır.)

```
// Bir bilesen konteynerini gösterir.

using System;
using System.ComponentModel;
using CipherLib; // CipherComp'un isim uzayini al

class UseContainer {
    public static void Main(string[] args) {
        string str = "Using containers.";
        Container cont = new Container();

        CipherComp cc = new CipherComp();
        CipherComp cc2 = new CipherComp();

        cont.Add(cc);
        cont.Add(cc2, "Second Component");

        Console.WriteLine("First message: " + str);
        str = cc.Encode(str);
        Console.WriteLine("First message encoded: " + str);

        str = cc.Decode(str);
        Console.WriteLine("First message decoded: " + str);

        str = "one, two, three";
        Console.WriteLine("Second message: " + str);

        str = cc2.Encode(str);
        Console.WriteLine("Second message encoded: " + str);

        str = cc2.Decode(str);
        Console.WriteLine("Second message decoded: " + str);

        Console.WriteLine("\ncc2's name: " + cc2.Site.Name);

        Console.WriteLine();

        // Her iki bileseni de serbest bırak.
        cont.Dispose();
    }
}
```

Programın çıktısı aşağıda gösterilmiştir:

```
First message: Using containers.  
First message encoded: Vtjoh!dpoubjofst/  
First message decoded: Using containers.  
Second message: one, two, three  
Second message encoded: pof -!uxp-!uisff  
Second message decoded: one, two, three  
  
cc2's name: Second Component  
Dispose(True) for component 1  
Closing file for component 1  
Dispose(True) for component 0  
Closing file for component 0
```

Gördüğünüz gibi, konteyner üzerinde `Dispose()` çağrıları ile konteyner içindeki tüm bileşenler dispose edilmektedir. Birçok bileşenle ya da bileşen örneği ile çalışırken bu çok büyük yarar sağlar.

## Bileşenler, Programlamanın Geleceği mi?

Bir uygulamayı birtakım bileşenler üzerine inşa etmek programlama işine son derece güçlü bir yaklaşım oluşturmaktadır, çünkü bileşenler, daha karmaşık programlara hakim olmanızı olanak sağlamaktadır. Program karmaşıklığının artması uzun süredir programcılardan karşısındaki en büyük sorun olarak kabul edilmektedir. Yeni başlayanlar, programcılık kariyerlerinin başından itibaren program uzadıkça hatadan arındırma süresinin de arttığını görmektedirler. Program uzadıkça, programın karmaşıklığı da artar. Biz insanların yönetebileceği karmaşıklık düzeyinin de bir sınırı vardır. Sadece sayı açısından bakılırsa, programın satır sayısı arttıkça yan etkilerin ve istenmeyen etkileşimlerin ortaya çıkma olasılığı da çoğalmaktadır. Bugün çoğu programcının da bildiği gibi, programlar giderek aşın karmaşık bir hal almaktadır.

Yazılım bileşenleri, karmaşıklığı “böl ve yönet” stratejisiyle yönetmemize yardımcı olur. İşlevsellik birimlerinin bağımsız bileşenlere ayrılması sayesinde programcılar programın yansittığı karmaşıklığı azaltabilir. Bileşen yönelimli bir yaklaşımla, bir program, net olarak tanımlanmış ve uygulama ayrıntılarıyla ilgilenmeye gerek kalmadan kullanılabilen yapı taşları (bileşenler) üzerine inşa edilir. Bu yaklaşımın net sonucu, karmaşıklığın azaltılmasıdır. İki mantıksal sonucuna götürürsek, bir programın büyük bölümü birbiriyle bağlanmış, aralarında veri alışverişi yapan bileşenlerden ibaret olacaktır. Böyle bir yaklaşımı *bileşen yönelimli programlama* denilebilir.

Bileşenlerin gücü ve C#'ta ne kadar kolay oluşturulabildikleri dikkate alınırsa, acaba programcılığının geleceği bileşenlerde midir? Pek çok programcı açısından bu sorunun cevabı şartsız olarak “Evet!”tır.

**25**

**YİRMİBEŞİNÇİ BÖLÜM**

---

**FORM TABANLI  
WINDOWS  
UYGULAMALARI  
GELİŞTİRMEK**

Bu kitapta gösterilen programların çoğu konsol uygulamalarıdır. Konsol uygulamaları, C# dilinin öğelerini göstermek için iyi bir yöntemdir; ayrıca, dosya filtreleri gibi bazı yardımcı program türleri için kullanıma uygundur. Kuşkusuz, modern uygulamaların pek çoğu Windows grafik kullanıcı arayüzü (GUI) ortamına yönelik tasarılmaktadır. C#'ın bir Windows uygulaması oluşturmak için nasıl kullanıldığını göstermeden elinizdeki kitap eksik olurdu.

Geçmişte Windows uygulamaları geliştirmek zorlu bir çabaydı. Yeni başlayanların bir Windows uygulamasının yalnızca temel öğelerini öğrenmek için birkaç hafta harcamaları garip değildi. Neyse ki, C# ve .NET bütün bunları tümden değiştirdi. .NET kütüphanesi, Windows Formlarını destekleyen bütün bir alt sistem içermektedir. Windows Formları, bir Windows programı geliştirmeyi adamatıllı basitleştirir. C# ve **System.Windows.Forms** kütüphanesini kullanarak Windows uygulamaları çok daha kolay geliştirilir ve bütün geliştirme süreci önemli ölçüde standardize edilmiş olur.

Windows programlama, bu konuya adanmış seriler dolusu kitap içeren çok büyük bir konudur. Bunun bütün yönlerini tek bir bölümde anlatmak elbette mümkün değildir. Bunun yerine, bu bölüm form tabanlı Windows programlamaya “atlayarak” bir giriş yapar. Bu bölümde bir pencerenin, bir menünün nasıl oluşturulacağı; bir düğmenin nasıl uygulanacağı ve bir mesaja nasıl yanıt verileceği açıklanmaktadır. Bu bölümün başlan sona çalışıktan sonra, form tabanlı Windows programlamanın diğer yönlerine de kolaylıkla ilerleyebileceksiniz.

## Windows Programlamanın Kısa Tarihçesi

C# ve .NET Framework'ün Windows programlamaya getirdiği avantajları takdir etmeniz için Windows programlamanın tarihini biraz bilmeniz gereklidir. Windows ilk geliştirildiğinde programlar doğrudan Windows API (Application Programming Interface - Uygulama Programlama Arayüzü) ile etkileşirdi. Windows API, Windows tarafından tanımlanan ve programların Windows'un sağladığı çeşitli fonksiyonlara erişmek için çağrıdıkları kapsamlı metotlar bütünüdür. API tabanlı programlar çok uzun ve karmaşıktır. Söz gelisi, API tabanlı bir program taslağı bile yaklaşık 50 satır uzunluğunda kod gerektirir. İşe yarar herhangi bir işlevde bulunan API temelli programlar *en azından* birkaç yüz satır kod içerirler; gerçek uygulamalar ise binlerce satırlık koda sahiptir. Bu nedenle, ilk günlerde Windows programlarını yazmak ve bakımını yapmak zordu.

Bu probleme yanıt olarak API'nin işlevsellliğini bir sınıf içine paketleyen sınıf kütüphaneleri geliştirildi. Bunlardan en önemlisi MFC'dir (Microsoft Foundation Classes - Microsoft Temel Sınıfları). Okuyucuların birçoğu MFC ile aşina olacaklardır. MFC, C++'ta yazılmıştır; MFC tabanlı programlar da C++'ta yazılmıştır. MFC, nesne yönelimli avantajlar sağladığı için, bu sayede, Windows programı geliştirme süreci basitleşmiştir. Ancak, MFC programları, ayrı başlık dosyaları, kod dosyaları ve kaynak dosyaları gerektirerek her şeye rağmen oldukça karmaşık işlerdi. Üstelik MFC, API'nin etrafında yalnızca “ince bir zar” gibiydi. Bu nedenle, birçok Windows tabanlı etkinlik halen önemli miktarda açık program ifadesi gerektiriyordu.

C# ve .NET Framework'ün Forms kütüphanesi, Windows programlama yaklaşımına tamamen nesne yönelimli bir boyut katmaktadır. Forms kütüphanesi API'nin etrafında yalnızca bir ambalaj sağlamak yerine, bir Windows uygulaması geliştirilirken, bu sürecin standardize edilmiş, entegre ve mantıksal açıdan tutarlı bir şekilde yönetilmesini sağlayan bir yöntem tanımlar. Bü tür bir entegrasyon düzeyi C#'ın delegeler ve olaylar gibi benzersiz özellikleri sayesinde mümkün kılınır. Ayrıca, C#'ın anlamsız veri toplayıcısı (garbage collector) sayesinde özellikle sıkıntı veren “bellek sizıntısı” problemi neredeyse tamamen ortadan kalkmıştır.

API ya da MFC kullanarak daha önceden Windows için programlama yapmışsanız, C#'ın yaklaşımını dikkat çekici ölçüde ferahlatıcı bulacaksınız. Windows'un doğusundan beri ilk kez bir Windows uygulaması geliştirmek, hemen hemen bir konsol uygulaması geliştirmek kadar kolay olmaktadır.

## Form Tabanlı Bir Windows Uygulaması Yazmanın İki Yöntemi

Başlamadan önce önemli bir noktayı vurgulamamız gereklidir. Visual Studio, bir Windows uygulaması oluşturma sürecinin büyük kısmını otomatize eden tasarım araçlarından oluşan sofistike bir set içerir. Bu araçları kullanarak, uygulamanız tarafından kullanılan çeşitli kontrolleri ve menüleri görsel açıdan kurabilir ve konumlandırlırsınız. Visual Studio ayrıca her özellik için gerekli sınıfları ve metotları da ayarlar. Açığçası, Visual Studio tasarım araçları, gerçek dünyaya ait Windows uygulamalarının birçoğunun geliştirilmesinde iyi bir tercihtir. Ancak, bu araçları kullanmanız için hiçbir gerek yoktur. Tıpkı konsol tabanlı uygulamalarda olduğu gibi bir metin editörü kullanarak ve sonra bunu derleyerek de bir Windows programı geliştirebilirsiniz.

Bu kitap Visual Studio yerine C# ile ilgili olduğu için ve bu bölümde yer alan Windows programları oldukça kısa olduğu için tüm programlar, bir metin editörü kullanılarak kodlanabilecek şekilde gösterilecektir. Yine de, programların genel yapısı, tasarımını ve organizasyonu tasarım araçları tarafından geliştirilen programlarla aynıdır. Bu nedenle, bu kitaptaki malzeme her iki yöntem için de geçerlidir.

## Windows'un Kullanıcı ile Etkileşimi

Windows programlama ile ilgili öğrenmeniz gereken ilk şey kullanıcının ve Windows'un nasıl etkileşikleridir, çünkü bu, tüm Windows programlarının paylaştığı ortak mimariyi tanımlamaktadır. Bu etkileşim, bu kitabın diğer bölümlerinde gösterilen konsol tabanlı programlardan esasen farklıdır. Bir konsol programı yazdığınızda, işletim sistemi ile etkileşimi başlatan sizin programınızdır. Söz gelisi, `Read()` ya da `WriteLine()`'ı çağırarak girdi ya da çıktı gibi isteklerde bulunan programın kendisidir. Yani, “geleneksel biçimde” yazılan programlar işletim sistemini çağrırlar. İşletim sistemi sizin programınızı çağrırmaz. Ancak, geniş ölçekte, Windows bunun tam ters biçiminde çalışır. Sizin programınızı çağrıran Windows'dur. Bu süreç şu şekilde gerçekleşir: Bir program kendisine Windows tarafından bir *mesaj* gönderilene kadar bekler. Mesaj alınır alınmaz programınızın uygun bir eylemde

bulunması beklenir. Programınız mesaja yanıt verirken Windows tarafından tanımlanan bir metodu çağırabilir, ancak etkinliği başlatan her şeye rağmen Windows'tur. Tüm Windows programlarının genel şeklini belirleyen, her şeyden çok, Windows ile gerçekleştirilen mesaja dayalı etkileşimdir.

Windows'un programınıza gönderebileceği farklı tipte birçok mesaj mevcuttur. Söz gelişi, programınıza ait bir pencere içinde ne zaman fare tıklansa, "fare tıklandı" anlamında bir mesaj gönderilecektir. Bir düğmeye basıldığında ya da bir menü seçeneği işaretlendiğinde başka tipte bir mesaj gönderilir. Bir gerçeği beyninize kazıyın: Sizin programınız söz konusu olduğu sürece mesajlar rasgele gelir. Windows programlarının, kesme güdümlü (interrupt driven) programlara benzemesinin nedeni budur. Bir sonraki mesajın nasıl bir mesaj olacağını bilemezsiniz.

## Windows Formları

Bir C# Windows programının çekirdeğinde *form* yer alır. Form, bir pencere oluşturmak, bunu ekranда görüntülemek ve mesajları almak için gerekli temel işlevselligi bir bütün halinde paketler. Bir form her türlü pencereyi simgeleyebilir. Uygulamanın ana penceresi, çocuk pencere, hatta bir iletişim kutusu bile bu kapsamda yer alır.

Form ilk oluşturulduğunda boştur, işlevsellik katmak için, söz gelişi, menüler ve düğmeler, listeler ve onay kutuları gibi kontroller eklersiniz. Bu nedenle, bir formu, diğer Windows nesneleri için bir konteyner gibi düşünebilirsiniz.

Pencereye bir mesaj gönderildiğinde bu bir olaya dönüştürülür. Bu yüzden, bir Windows mesajını kontrol altına almak amacıyla bu mesaj için formla birlikte bir olay yöneticiyi kaydetmeniz yeterli olacaktır. Sonra, mesaj ne zaman alınırsa alınsın olay yöneticiniz otomatik olarak çağrılr.

### Form Sınıfı

Bir form **Form** sınıfından ya da **Form**'dan türetilen herhangi bir sınıfın bir nesne örneklenerek oluşturulur. **Form** kendisine özgü önemli bir işlevselligi sahiptir; ayrıca, kalıtım yoluyla ek işlevsellik de edinir. **Form**'un en önemli temel sınırlarından ikisi Bölüm 24'te ele alınan **System.ComponentModel.Component** ve **System.Windows.Forms.Control**'dır. **Control** sınıfı tüm Windows kontrollerinde ortak olan özellikleri tanımlar. **Form**, kalıtım yoluyla **Control**'den türediği için **Form** da aslında bir kontroldür. Bu gerçek sayesinde formların kontrol oluşturmak amacıyla kullanılmaları mümkün olur. **Form** ve **Control**'ün üyelerinden birkaçı takip eden örneklerde kullanılmıştır.

## Form Tabanlı Bir Windows Program Taslağı

Küçük çapta bir form tabanlı Windows uygulaması geliştirerek başlayacağız. Bu uygulama yalnızca bir pencere oluşturur ve bunu görüntüler. Başka özellikler içermez. Ancak, bu taslak

tümüyle fonksiyonel bir pencere oluşturmak için gerekli adımları kesinlikle göstermektedir. Bu taslak, birçok tipte Windows uygulamasının üzerine inşa edileceği bir başlangıç noktasıdır. Form tabanlı Windows programının taslağı aşağıda gösterilmiştir:

```
// Form tabanlı Windows program taslağı.

using System;
using System.Windows.Forms;

// WinSkel, Form'dan turetilmiştir.
class WinSkel : Form {

    public WinSkel() {
        // Pencereye bir isim ver.
        Text = "A Windows Skeleton";
    }

    // Main, sadece uygulamayı başlatmak için kullanılır.
    [STAThread]
    public static void Main() {
        WinSkel skel = new WinSkel(); // bir form olustur

        // pencereyi calistirmaya basla.
        Application.Run(skel);
    }
}
```

Bu programla oluşturulan pencere Şekil 25.1'de gösterilmiştir.

Gelin şimdi, bu programı satır satır inceleyelim. Öncelikle, hem **System**'in hem de **System.Windows.Forms**'un programa dahil edildiğine dikkat edin. **System**, **Main()**'in hemen öncesinde yer alan **STAThread** niteliğinden dolayı gereklidir. **System.Windows.Forms** az önce bahsedildiği gibi Windows Forms alt sistemini destekler.

Daha sonra, **WinSkel** adında bir sınıf oluşturulur. Bu sınıf, **Form**'dan kalıtım yoluyla türetilir. Böylece, **WinSkel** belirli tipte bir form tanımlar. Bu örnekte, bu küçük çaplı bir formdur.

**WinSkel** yapılandırıcısı içinde aşağıdaki kod satırı mevcuttur:

```
Text = "A Windows Skeleton";
```

**Text**, pencerenin başlığını ayarlayan özelliktir. Böylece, bu atama, pencerenin başlık çubuğuun **A Windows Skeleton** yazısını içermesine neden olur. **Text** şu şekilde tanımlanır:

```
public virtual string Text { get; set; }
```

**Text**, **Control**'den türetilmiştir.



**ŞEKİL 25.1:** Form tabanlı pencere taslağı.

Sonra, **Main()** metodu gelir. Bu, elinizdeki kitabın geri kalan bölümlerinde yer alan **Main()**'e çok benzer şekilde deklare edilmiştir. Bu, programın çalışmaya başladığı metottur. Ancak, dikkat ederseniz, **Main()** öncesinde **STAThread** özelliği gelmektedir. Microsoft, bir Windows programının **Main()** metodunun bu özelliğe sahip olması gerektiğini bildirmektedir. Bu, programın kanal modelini (threading model) *single-threaded apartment* (STA) olarak ayarlar. (Kanal modelleri ve apartmanlar bu kitabın kapsamı dışındadır, ancak kısaca bahsetmek gerekirse, bir Windows uygulaması iki farklı kanal modelinden birini kullanabilir: tek kanallı apartman [single-threaded apartment] ya da çok kanallı apartman [multi-threaded apartment].)

**Main()** içinde bir **WinSkel** nesnesi oluşturulur. Bu nesne, sonra **Applications** sınıfı tarafından tanımlanan **Run()** metoduna, aşağıda gösterildiği şekliyle aktarılır:

```
Application.Run(skel);
```

Bu, pencerenin çalışmasını başlatır. **Application** sınıfı **System.Windows.Form** içinde tanımlıdır ve tüm Windows uygulamalarında ortak olan özellikleri bütünüyle paketler. Taslakta kullanılan **Run()** metodu aşağıda gösterilmiştir:

```
public static void Run(Form nesne)
```

Metot, bir forma atıfta bulunan bir referansı parametre olarak alır. **WinSkel**, **Form**'dan türetildiği için **WinSkel** tipinde bir nesne **Run ()**'a aktarılabilir.

Program çalıştırıldığında Şekil 25.2'de gösterilen pencereyi oluşturur. Pencere varsayılan boyutlara sahiptir: 300 piksel genişlik ve 300 piksel yükseklik. Pencere tam anlamıyla fonksiyoneldir. Yeniden boyutlandırılabilir, taşınabilir, küçültülebilir, büyütülebilir ve kapatılabilir. Böylece, hemen hemen tüm pencereler için gerekli olan temel özellikler, forma dayalı birkaç satır kod yazılarak elde edilmiştir. Öte yandan, aynı programı C dili kullanarak ya da doğrudan Windows API'sını çağırarak yazmak yaklaşık beş katı kadar daha fazla kod satırı gerektirirdi!

Önceki taslak, forma dayalı Windows uygulamalarının birçoğunu şeklini alacağı anahatları tanımlamaktadır. Genel olarak, bir form oluşturmak için **Form**'dan türetilen bir sınıf oluşturursunuz. Formu gereksinimlerinize uygun biçimde kullanıma hazırlayın, türetilmiş sınıfınızdan bir nesne oluşturun ve bu nesne üzerinde **Application.Run ()**'ı çağırın.

## Windows Taslağını Derlemek

Bir Windows programını komut satırı derleyicisi ya da Visual Studio kullanarak derleyebilirsiniz. Bu bölümdeki çok kısa programlar için komut satırı derleyicisi en kolay yöntemdir, fakat gerçek uygulamalarda muhtemelen IDE'yi kullanmak isteyeceksiniz. (Ayrıca, bu bölümün başında da açıklandığı gibi, Visual Studio tarafından sunulan tasarım araçlarını da muhtemelen kullanacaksınız.) Her iki yöntem de burada gösterilmiştir.

### Komut Satırından Derlemek

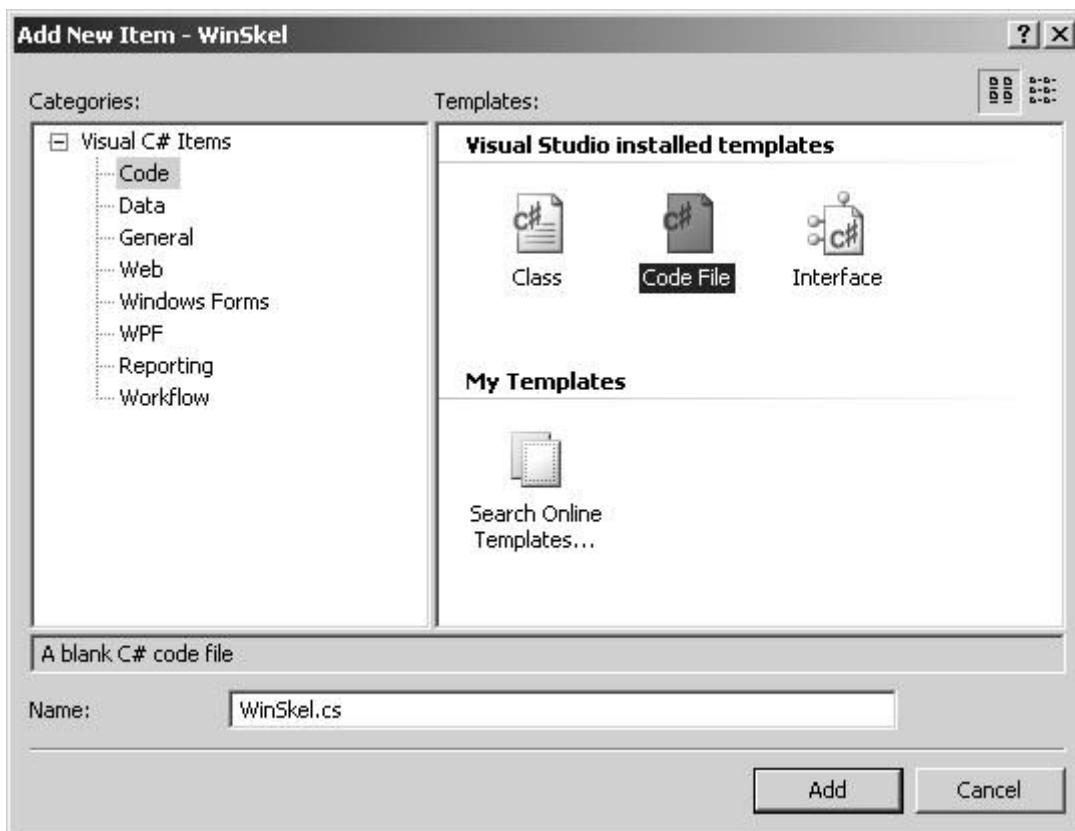
Taslağa **WinSkel.cs** adını verdığınızı varsayarsak, taslağı komut satırından derlemek için şu komutu kullanın:

```
csc /t:winexe WirtSkel.cs
```

**/t:winexe** seçeneği derleyiciye bir konsol programı yerine bir Windows uygulaması oluşturmasını bildirir. Programı çalıştmak için komut satırından yalnızca **WinSkel**'i girmeniz yeterlidir.

### IDE'den Derlemek

Programı Visual Studio IDE kullanarak derlemek için öncelikle yeni bir Windows Application projesi oluşturun. Bunu File | New | Project komutunu seçerek gerçekleştirin. Sonra, New Project iletişim kutusundan Windows Application seçeneğini işaretleyin. Projeye **WinSkel** adını verin. Bu proje, kendisi ile ilişkilendirilmiş **Form1.cs** adında bir dosya içerecektir. Bu dosyayı silin. Sonra, **WinSkel** proje ismi üzerinde farenin sağ tuşu ile tıklayın. Add seçeneğini, sonra da Add New Item seçeneğini işaretleyin. Aşağıda gösterilen iletişim kutusunu göreceksiniz:



C# Code File seçeneğini işaretleyin ve dosyayı **WinSkel.cs** olarak isimlendirin. Şimdi, taslak kodu aynen gösterildiği gibi girin ve çözümü kurun. Projeyi çalıştırmak için Debug | Start Without Debugging komutunu seçin.

## Düğme Ekleme

Genel olarak, bir pencerenin işlevselliği iki tip öğe ile ifade edilir: Kontroller ve menüler. Kullanıcı bu öğeler aracılığı ile programınızla eklileşebilir. Menüler bu bölümün ileriki sayfalarında anlatılmaktadır. Burada bir pencereye nasıl kontrol ekleyebileceğinizi öğreneceksiniz.

Windows birçok farklı tipte kontrol tanımlar. Bunlardan birkaçı sıralayacak olursak, düğmeler, onay kutuları, radyo düğmeleri ve liste kutuları bu kapsamda yer alır. Her kontrol tipi farklı olmasına rağmen bunların tümü az çok aynı şekilde çalışırlar. Burada, bir pencereye bir düğme ekleyeceğiz. Fakat, diğer kontrol tiplerini eklemek için de aynı temel prosedür kullanılabilir.

## Button Sınıfinın Esasları

Bir düğme **Button** sınıfı içinde paketlenmiştir. **Button**, **ButtonBase** özet sınıfından kalıtım yoluyla türetilir. Kendisi bir kontrol olduğu için **Control** sınıfının içeriğini kalıtım yoluyla elde eder. **Button**'da yalnızca tek bir yapılandırıcı tanımlıdır:

```
public Button()
```

Bu, varsayılan bir büyülüğe sahip ve pencere içinde bir konumu olan bir düğme oluşturur. Bir düğme kullanılmadan önce, söz konusu düğmenin **Text** özelliğine bir karakter katarı atarak düğmeye bir tanım verilmesi gerekecektir.

Düğmenin pencere içindeki konumunu belirtmek için **Location** özelliğine düğmenin sol üst köşesinin koordinatlarını atamalısınız. **Location** özelliği kalıtımıla **Control**'den alınır ve şu şekilde tanımlanır:

```
public Point Location { get; set; }
```

Koordinatlar bir **Point** yapısı içinde tutulur. **Point**, **System.Drawing** isim uzayında tanımlıdır. **Point**, iki özellik içerir:

```
public int X { get; set; }
public int Y { get; set; }
```

Böylece, “Press Here” tanımını içeren ve 100,200 konumuna yerleştirilmiş bir düğme oluşturmak için aşağıdaki sekansı kullanın:

```
Button MyButton = new Button();
MyButton.Text = "Press Here";
MyButton.Location = new Point(100, 200);
```

## Bir Forma Düğme Ekleme

Bir düğme oluşturduktan sonra bunu bir forma eklemelisiniz. Bunu, söz konusu forma bağlı kontrollerden oluşan koleksiyon üzerinde **Add()** metodunu çağırarak yaparsınız. Bu koleksiyon, **Control** sınıfından kalıtımıla elde edilen **Controls** özelliği aracılığıyla kullanılabilir. **Add()** metodu şu şekilde tanımlanır:

```
public virtual void Add(Control knt1)
```

Burada **knt1**, eklenmekte olan kontroltür. Bir kez bir forma bir kontrol eklenince form görüntülendiğinde artık kontrol de görüntülenecektir.

## Basit Bir Düğme Örneği

Aşağıdaki program daha önce gösterilen taslağa bir düğme ekler. Şu aşamada düğme bir işe yaramaz, fakat formda mevcuttur ve tıklanabilir.

```
// Bir dugme ekler.

using System;
using System.Windows.Forms;
using System.Drawing;
```

```
class ButtonForm : Form {  
    Button MyButton = new Button();  
  
    public ButtonForm() {  
        Text = "Using a Button";  
  
        MyButton = new Button();  
  
        MyButton.Text = "Press Here";  
  
        MyButton.Location = new Point(100, 200);  
  
        Controls.Add(MyButton);  
    }  
  
    [STAThread]  
    public static void Main() {  
        ButtonForm skel = new ButtonForm();  
  
        Application.Run(skel);  
    }  
}
```

Bu program, **Form**'dan türetilen **ButtonForm** adında bir sınıf oluşturur. **ButtonForm**, **MyButton** adında, **Button** tipinde bir alan içerir. Yapılandırıcı içinde düğme oluşturulur, düğmeye ilk değer atanır ve düğme forma eklenir. Program çalıştırıldığında Şekil 25.2'de gösterilen pencereyi görüntüler. Düğmeyi tıklayabilirsiniz, fakat hiçbir şey meydana gelmez. Düğmenin bir şeyler yapmasını sağlamak için, bir sonraki konuda anlatıldığı gibi bir mesaj yöneticisi eklemelisiniz.

## Mesajları Yönetmek

Bir programın tıklanan bir düğmeye yanıt vermesi (ya da diğer herhangi tipte bir kontrol etkileşimi) için söz konusu düğme tarafından üretilen mesajı ele alması gereklidir. Genel olarak, bir kullanıcı bir programla etkileştiği zaman bu etkileşimler sizin programınıza mesaj olarak aktarılır. Form tabanlı bir C# programında bu mesajlar olay yöneticileri tarafından üretilir. Bu nedenle, mesajları almak için, bir mesaj üretildiğinde çağrılan yöneticiler listesine programınız kendi olay yöneticisini ekler. Bu, düğme tıklandığında üretilen mesajlar açısından, yöneticinizi **Click** olayına eklemeniz anlamına gelir.

**Click** olayı **Button** tarafından tanımlanır. (**Click**, kalıtımla **Control**'den elde edilir.) **Click** aşağıda gösterilen şekilde sahiptir:

```
public Event EventHandler Click;
```



**ŞEKİL 25.2:** Bir düğme eklemek.

**EventHandler** delegesi aşağıdaki gibi tanımlanır:

```
public delegate void EventHandler(object kim, EventArgs arglar)
```

Söz konusu olayı üreten nesne **kim** üzerinden aktarılır. Bu olayla ilintili tüm bilgiler **arglar** üzerinden geçirilir. Birçok olay için **arglar**, **EventArgs**'dan türetilen bir sınıfın nesnesi olacaktır. Düğmeyi tıklamak ek bilgi gerektirmeden için, düğmeyi ele alırken olay argümanlarını dert etmemize gerek yoktur.

Aşağıdaki program, önceki programdaki düğmeye bir yanıt (tepki) kodu ekler. Düğmenin her tıklanışı, düğmenin konumunun değişmesine neden olur.

```
// Dugme mesajlirini ele almak.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
    Button MyButton = new Button();

    public ButtonForm() {
        Text = "Respond to a Button";

        MyButton = new Button();
        MyButton.Text = "Press Here";
        MyButton.Location = new Point(100, 200);

        // Olay yoneticisini listeye ekle.
        MyButton.Click += new EventHandler(MyButtonClick);

        Controls.Add(MyButton);
    }
}
```

```

    }

    [STAThread]
    public static void Main() {
        ButtonForm skel = new ButtonForm();

        Application.Run(skel);
    }

    // MyButton icin yönetici.
    protected void MyButtonClick(object who, EventArgs e) {

        if(MyButton.Top == 200)
            MyButton.Location = new Point(10, 10);
        else
            MyButton.Location = new Point(100, 200);
    }
}

```

Gelin, bu programdaki olay yönetimini gerçekleştiren koda yakından bakalım. Düğmenin tıklanmasıyla ilişkili olay yöneticisi aşağıda gösterilmiştir:

```

// MyButton icin yönetici.
protected void MyButtonClick(object who, EventArgs e) {

    if(MyButton.Top == 200)
        MyButton.Location = new Point(10, 10);
    else
        MyButton.Location = new Point(100, 200);
}

```

**MyButtonClick()**, daha önce gösterilen **EventHandler** delegesi ile aynı imzayı kullanır. Bu, **MyButtonClick()**'in de **Click** olay zincirine eklenebileceği anlamına gelir. Metodun **protected** olarak nitelendiğine dikkat edin. Bu teknik olarak gerekli değildir; fakat, iyi bir fikirdir, çünkü olay yöneticilerinin olaylara yanıt olarak kullanıllarının haricinde çağrılmaları düşünlmez.

Olay yöneticisi içinde, düğmenin üst konumu **Top** özelliği ile belirlenir. Tüm kontroller aşağıdaki özelliklerini tanımlar. Bu özellikler, sol üst ve sağ alt köşelerin koordinatlarını belirtir.

```

Public int Top { get- set; }
public int Bottom { get; }
public int Left { get; set; }
public int Right { get ; }

```

Dikkat ederseniz, kontrolün konumu **Top** ve **Left** ayarlanarak değiştirilebilir: fakat, **Bottom** ve **Right** ayarlanarak değiştirilemez, çünkü bunlar salt okunur özelliklerdir. (Bir kontrolün büyülüüğünü değiştirmek için **Width** ve **Height** özelliklerini kullanabilirsiniz.)

Düğmenin tıklanması ile ilişkili olay alındığı zaman, kontrol eğer orijinal 200 konumundaysa konum 10, 10 olacak şekilde değiştirilir. Aksi halde, orijinal konumu olan 100, 200 noktasına döndürülür. Bu nedenle, düğmeyi her tıklayışınızda düğmenin konumu değişir.

**MyButtonClick()**'in mesajları alabiliyor olması için, düğmenin **Click** olayına bağlı olay yöneticileri listesine kendisinin eklenmesi gereklidir. Bu, **ButtonForm** yapılandırıcısı içinde aşağıdaki ifade kullanılarak gerçekleştirilir:

```
MyButton.Click += new EventHandler(MyButtonClick);
```

Bu işlem tamamlandıktan sonra, düğmenin her iki tıklamışında **MyButtonClick()** çağrılr.

## Alternatif Uygulama

İlginc bir husus şudur: **MyButtonClick()** biraz farklı bir şekilde de yazılabilirdi. Hatırlarsanız, bir olay yöneticisinin **kim** parametresi, çağrıyı üreten nesneye atıfta bulunan bir referans alır. Düğme tıklama olayında bu, tıklanan düğmedir. Bu nedenle, **MyButtonClick()** şu şekilde de yazılabilirdi:

```
// Alternatif dugme yonetici.
protected void MyButtonClick(object who, EventArgs e) {
    Button b = (Button) who;

    if(b.Top == 200)
        b.Location = new Point(10, 10);
    else
        b.Location = new Point(100, 200);
}
```

Bu versiyonda **who**, **Button** tipine dönüştürülür ve bu referans (**MyButton** alanı yerine) düğme nesnesine erişmek için kullanılır. Bu örnekte bu yaklaşımın sağladığı bir avantaj olmamasına rağmen, bunun oldukça değerli olabileceği durumlar kolaylıkla hayal edilebilir. Örneğin, bu tür bu yaklaşım, herhangi spesifik bir düğme için bağımsız bir düğme olay yöneticisi yazılmasına olanak tanır.

## Mesaj Kutusunun Kullanımı

Windows'un en kullanışlı standart özelliklerinden biri *mesaj kutusudur*. Mesaj kutusu, bir mesaj görüntülemenize imkan veren, sistemde önceden tanımlı bir penceredir. Kullanıcıdan Evet, Hayır ya da Tamam gibi basit yanıtları da alabilirsiniz. Forma dayalı bir programda bir mesaj kutusu **MessageBox** sınıfı tarafından desteklenir. Ancak, bu sınıfın ait bir nesne oluşturmazsınız.

Bunun yerine, bir mesaj kutusunu görüntülemek için **MessageBox** tarafından tanımlanan **static** metot **Show()**'u çağırın.

**Show()** metodunun birkaç farklı kullanımı mevcuttur. Bizim kullanacağımız şekli aşağıda gösterilmiştir:

```
public static DialogResult Show(string msj, string altyazı,
                                MessageBoxButtons mbb)
```

**msj** üzerinden aktarılan karakter katarı kutunun gövdesinde görüntülenir. Mesaj kutusu penceresinin ismi **altyazı** ile aktarılır. Görüntülenecek olan düğmeler **mbb** ile belirtilir. Kullanıcının yanıtı döndürülür.

**MessageBoxButtons** aşağıdaki değerleri tanımlayan bir numaralandırmadır:

<b>AbortRetryIgnore</b>	<b>OK</b>	<b>OKCancel</b>
<b>RetryCancel</b>	<b>YesNo</b>	<b>YesNoCancel</b>

Bu değerlerin her biri, bir mesaj kutusu içine dahil edilecek düğmeleri tarif eder. Örneğin, eğer **mbb YesNo** değerini içeriyorsa, Yes ve No düğmeleri mesaj kutusuna dahil edilir.

**Show()**'un döndürdüğü değer hangi düğmenin basıldığını gösterir. Bu, şu değerlerden biri olacaktır:

<b>Abort</b>	<b>Cancel</b>	<b>Ignore</b>	<b>No</b>
<b>None</b>	<b>OK</b>	<b>Retry</b>	<b>Yes</b>

Programınız, kullanıcının istediği eylemin biçimini belirlemek amacıyla döndürülen değeri inceleyebilir. Söz gelişi, mesaj kutusu bir dosyanın üzerine yazmadan önce kullanıcıya haber veriyorsa, kullanıcı Cancel düğmesini (İptal) tıkladığı takdirde programınız üzerine yazma işlemini önleyebilir. Kullanıcı OK düğmesini tıklarsa üzerine yazma işlemine izin verilebilir.

Aşağıdaki program, önceki programa bir Stop düğmesi ve bir mesaj kutusu ekler. Stop düğmesi yönetici içinde, kullanıcıya programı durdurmak isteyip istemediğini soran bir mesaj kutusu görüntülenir. Kullanıcı Evet düğmesini tıklarsa program durdurulur. Hayır düğmesini tıklarsa program çalışmaya devam eder.

```
// Stop dugmesi ekler

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
    Button MyButton;
    Button StopButton;

    public ButtonForm() {
        Text = "Adding a Stop Button";

        // Dugmeleri olustur.
        MyButton = new Button();
        MyButton.Text = "Press Here";
        MyButton.Location = new Point(100, 200);

        StopButton = new Button();
        StopButton.Text = "Stop";
        StopButton.Location = new Point(100, 100);

        // Dugme ile iliskili olay yoneticilerini pencereye ekle.
        MyButton.Click += new EventHandler(MyButtonClick);
    }
}
```

```

        Controls.Add(MyButton);
        StopButton.Click += new EventHandler(StopButtonClick);
        Controls.Add(StopButton);
    }

    [STAThread]
    public static void Main() {
        ButtonForm skel = new ButtonForm();

        Application.Run(skel);
    }

    // MyButton icin yonetici.
    protected void MyButtonClick(object who, EventArgs e) {

        if(MyButton.Top == 200)
            MyButton.Location = new Point(10, 10);
        else
            MyButton.Location = new Point(100, 200);
    }

    // StopButton icin yonetici.
    protected void StopButtonClick(object who, EventArgs e) {

        // Kullanici Yes yanitini verirse, programi sona erdir.
        DialogResult result = MessageBox.Show("Stop Program?",
                                              "Terminate",
                                              MessageBoxButtons.YesNo);

        if(result == DialogResult.Yes) Application.Exit();
    }
}

```

Gelin şimdi mesaj kutusunun nasıl kullanıldığına yakından bakalım. **ButtonForm** yapılandırıcısına içine ikinci bir düğme eklenmiştir. Bu düğme “Stop” metnini içerir ve bununla ilgili olay yöneticisi **StopButtonClick()**'e bağlanır.

**StopButtonClick()** içinde mesaj kutusu aşağıdaki ifade kullanılarak görüntülenir:

```

// Kullanici Evet yanitini verirse, programi sona erdir.
DialogResult result = MessageBox.Show("Stop Program?",
                                      "Terminate",
                                      MessageBoxButtons.YesNo);

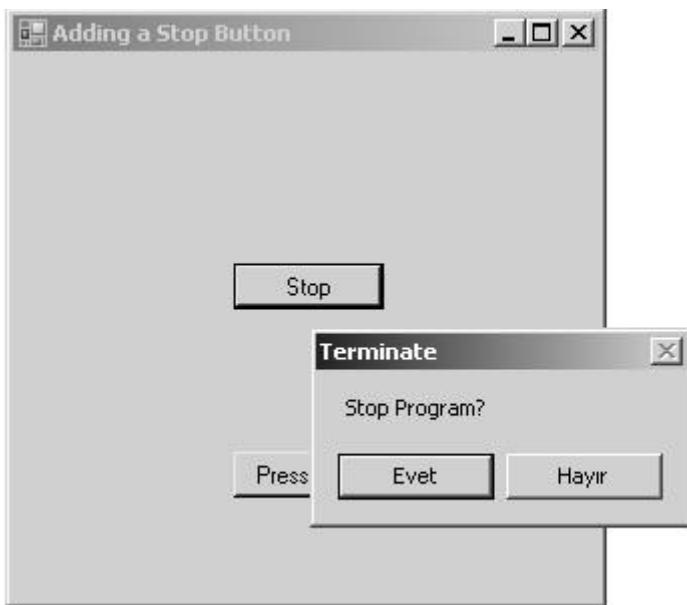
```

Burada kutunun içindeki mesaj “Stop Program?”dır. Kutunun ismi “Terminate” ve görüntülenecek düğmeler Evet ve Hayır’dır. **Show()** döndüğünde kullanıcının yanıtı **result**'a atanır. Bu yanıt daha sonra eylemin biçimini belirlemek için aşağıdaki kod tarafından incelenir:

```
if(result == DialogResult.Yes) Application.Exit();
```

Kullanıcı eğer Evet düğmesini tıklarsa program, `Application.Exit()` çağrılarak durdurulur. Bu, programın derhal durdurulmasına neden olur. Aksi halde, bir eylem gerçekleştirilemez ve program çalışmaya devam eder.

Örnek çıktı Şekil 25.3'te gösterilmiştir.



**ŞEKİL 25.3:** Stop Düğmesi programının örnek çıktısı.

## Menü EklemeK

Hemen hemen tüm Windows uygulamalarının ana penceresinin en üstünde bir menü yer alır. Buna *ana menü* denir. Ana menü genellikle File, Edit ya da Tools gibi üst düzey kategoriler içerir. Ana menüden aşağıya doğru *açılan* (drop down) menüler inen Bu menüler, kategorilerle ilintili asıl seçenekleri içerirler. Bir menü öğesi seçildiğinde bir mesaj üretilir. Bu nedenle, bir menü seçimini işlemek için programınız her menü öğesine bir olay yöneticisi atayacaktır.

Bir ana menü iki sınıfın bileşimiyle kurulur. Bunlardan birincisi `MainMenu`'dır. `MainMenu`, menünün bütün yapısını sınıf içine paketler. İkincisi, yalnızca tek bir seçimi sınıf içine paketleyen `MenuItem`'dır. Bir menü seçimi ya Close gibi nihai bir eylemi simgeler ya da bir başka açılan menüyü etkin kılar. `MainMenu` ve `MenuItem`'nın her ikisi de `Menu` sınıfından kalıtımıla türetilir. Menüler, Windows programlamada temel bir kaynak oldukları için, menülerle ilgili pek çok seçenek mevcuttur ve menü konusu oldukça genişdir. Neyse ki, standart bir ana menü oluşturmak kolaydır.

Bir menü öğesi seçildiğinde bir `Click` olayı üretilir. `Click`, `MenuItem` tarafından tanımlanır. Bu nedenle, programınız bir menü seçimini ele almak için bu öğe ile ilişkili `Click` olay listesine kendi yöneticisini ekleyecektir.

Her form şu şekilde tanımlanan bir `Menu` özelliğine sahiptir:

```
public MainMenu Menu { get; set; }
```

Bu özelliğe varsayılan değer olarak bir menü atanmaz. Bir menü öğesini görüntülemek için bu özellik, sizin oluşturduğunuz menüye ayarlanmalıdır.

Bir ana menü oluşturmak basittir, fakat birkaç adım içerir:

1. Bir **MainMenu** nesnesi oluşturun.
2. **MainMenu** nesnesine üst düzey kategorileri tarif eden **MenuItem**'ı ekleyin. Bu menü öğeleri, ana menü ile ilişkili **MenuItem** koleksiyonuna eklenir.
3. Her üst düzey **MenuItem**'ına, bu üst düzey öğe ile ilişkili açılan menülerini tarif eden **MenuItem** listesi ekleyin. Bu menü öğeleri, her üst düzey menü ile ilişkili **MenuItem** koleksiyonuna eklenir.
4. Her seçenek için olay yöneticilerini ekleyin.
5. **MainMenu** nesnesini formla ilişkili **Menu** özelliğine atayın.

Aşağıdaki sekans, üç seçenek içeren bir File menüsünün nasıl oluşturulduğunu göstermektedir. Seçenekler Open, Close ve Exit'tir.

```
// Ana menu nesnesi olusturun.  
MainMenu MyMenu = new MainMenu();  
  
// Menuye ust düzey menu ogesi ekleyin.  
MenuItem m1 = new MenuItem("File");  
MyMenu.MenuItems.Add(m1);  
  
// File icin bir alt menu olusturun.  
MenuItem subm1 = new MenuItem("Open");  
m1.MenuItems.Add(subm1);  
  
MenuItem subm2 = new MenuItem("Close");  
m1.MenuItems.Add(subm2);  
  
MenuItem subm3 = new MenuItem("Exit");  
m1.MenuItems.Add(subm3);
```

Şimdi bu sekansı çok dikkatli inceleyelim. Sekans, **MyMenu** adında bir **MainMenu** nesnesi oluşturarak başlamaktadır. Bu nesne, menü yapısı içinde en üstte olacaktır. Sonra, **m1** adında bir menü öğesi oluşturulur. Bu, dosya başlığıdır. Doğrudan ana menüye eklenir ve üst düzey seçenek konumunu alır. Sonra, File ile ilintili açılan menü oluşturulur. Bu menü öğelerinin File menü öğesi olan **m1**'e eklendiğine dikkat edin. Bir **MenuItem** bir diğerine eklendiğinde eklenen öğe, eklendiği öğe ile ilişkili açılan menünün bir parçası olur. Böylece, **subm1**'den **subm3**'e kadarki öğeler **m1**'e eklendikten sonra File'ı seçmek Open, Close ve Exit'i içeren bir açılan menünün ekranda görüntülenmesine neden olur.

Menü oluşturulur olşturulmaz her öğe ile ilişkili olay yöneticilerine de değerleri atanmalıdır. Daha önce açıklandığı gibi, seçim yapan bir kullanıcı bir **Click** olayı üretir. Bu nedenle, aşağıdaki sekans **subm1**'den **subm3**'e kadar yöneticilere değer atamaktadır:

```
// Menu ogeleri icin olay yoneticileri ekle.  
subm1.Click += new EventHandler(MMOpenClick);  
subm2.Click += new Eventnandler(MMCcloseClick);  
subm3.Click += new EvantHandler(MMExitClick);
```

Bu yüzden, kullanıcı eğer Exit'i seçerse, **MMExitClick()** çalıştırılır.

Son olarak, **MainMenu** nesnesi, aşağıda gösterildiği gibi, formun **Menu** özelliğine atanmalıdır:

```
Menu = MyMenu;
```

Bu atama gerçekleşikten sonra, pencere oluşturulduğunda söz konusu menü görüntülenecektir ve tercihler doğru yöneticiye gönderilecektir.

Aşağıdaki program tüm parçaları birleştirir ve bir ana menünün nasıl oluşturulacağını ve menü seçeneklerinin nasıl ele alınacağını gösterir:

```
// Ana menu ekler.  
  
using System;  
using System.Windows.Forms;  
  
class MenuForm : Form {  
    MainMenu MyMenu;  
  
    public MenuForm() {  
        Text = "Adding a Main Menu";  
  
        // Bir ana menu nesnesi olusturun.  
        MyMenu = new MainMenu();  
  
        // Menuye ust düzey ogeler ekleyin.  
        MenuItem m1 = new MenuItem("File");  
        MyMenu.MenuItems.Add(m1);  
  
        MenuItem m2 = new MenuItem("Tools");  
        MyMenu.MenuItems.Add(m2);  
  
        // File icin bir alt menu olusturun  
        MenuItem subm1 = new MenuItem("Open");  
        m1.MenuItems.Add(subm1);  
  
        MenuItem subm2 = new MenuItem("Close");  
        m1.MenuItems.Add(subm2);  
  
        MenuItem subm3 = new MenuItem("Exit");  
        m1.MenuItems.Add(subm3);
```

```
// Tools icin bir alt menu olusturun.
MenuItem subm4 = new MenuItem("Coordinates");
m2.MenuItems.Add(subm4);

MenuItem subm5 = new MenuItem("Change Size");
m2.MenuItems.Add(subm5);

MenuItem subm6 = new MenuItem("Restore");
m2.MenuItems.Add(subm6);

// Menu ogeleri icin yoneticiler ekleyin.
subm1.Click += new EventHandler(MMOpenClick);
subm2.Click += new EventHandler(MMCloseClick);
subm3.Click += new EventHandler(MMExitClick);
subm4.Click += new EventHandler(MMCoordClick);
subm5.Click += new EventHandler(MMChangeClick);
subm6.Click += new EventHandler(MMRestoreClick);

// Menuyu forma ekleyin.
Menu = MyMenu;
}

[STAThread]
public static void Main() {
    MenuForm skel = new MenuForm();

    Application.Run(skel);
}

// Ana menudeki Coordinates secimi icin yonetici.
protected void MMCoordClick(object who, EventArgs e) {
    // Koordinatlari iceren bir karakter katari olusturun.
    string size =
        String.Format("{0}: {1}, {2}\n{3}: {4}, {5},
                      "Top, Left", Top, Left,
                      "Bottom, Right", Bottom, Right);

    // Bir mesaj kutusu goruntuleyin.
    MessageBox.Show(size, "Window Coordinates",
                    MessageBoxButtons.OK);
}

// Ana menudeki Change secimi icin yonetici.
protected void MMChangeClick(object who, EventArgs e) {
    Width = Height = 200;
}

// Ana menudeki Restore secimi icin yonetici.
protected void MMRestoreClick(object who, EventArgs e) {
    Width = Height = 300;
}

// Ana menudeki Open secimi icin yonetici.
protected void MMOpenClick(object who, EventArgs e) {
```

```
        MessageBox.Show("Inactive", "Inactive",
                         MessageBoxButtons.OK);
    }

    // Ana menudeki Close secimi icin yonetici.
    protected void MMCcloseClick(object who, EventArgs e) {

        MessageBox.Show("Inactive", "Inactive",
                         MessageBoxButtons.OK);
    }

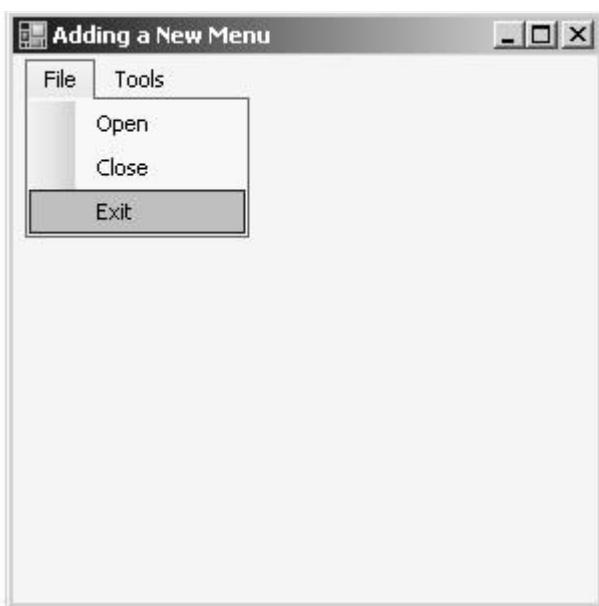
    // Ana menudeki Exit secimi icin yonetici.
    protected void MMExitClick(object who, EventArgs e) {

        DialogResult result = MessageBox.Show("Stop Program?",
                                              "Terminate",
                                              MessageBoxButtons.YesNo);

        if(result == DialogResult.Yes) Application.Exit();
    }
}
```

Örnek çıktı Şekil 25.4'te gösterilmiştir.

Bu programda iki açılan menü tanımlanmaktadır. Bunlardan ilki File menüsü ile erişilir. Bu menü Open, Close ve Exit seçeneklerini içerir. Open ve Close ile ilişkili menü yöneticileri, ekranda bir mesaj kutusu görüntülemekten başka bir işlevi olmayan yalnızca birer yer göstERICİSİdir. Close yönetici programı durdurmak isteyip istemediğİNİZİ sorar. Evet yanıtını verirseniz, program sonlandırılır.



ŞEKİL 25.4: Menü programının örnek çıktısı.

Tools menusu üç seçenek içerir: Coordinates, Change Size ve Restore. Coordinates'in seçilmesi, bir mesaj kutusu içinde görüntülenecek pencerenin sol üst ve sağ alt köşelerinin

koordinatlarının görüntülenmesine neden olur. Pencereyi taşıyıp, koordinatlarını görüntülemeyi deneyin. Pencere ne zaman yeni bir konuma taşınsa koordinatları değişir.

Change Size seçildiğinde pencerenin boyutlarının, eni ve boyu **200** piksel uzunluğunda olacak şekilde küçültülmesine neden olur. Bu, aşağıda gösterilen **Width** ve **Height** özelliklerini aracılığıyla gerçekleştirilir:

```
public int width { get; set; }
public int Height { get; set; }
```

Form tabanlı bir pencerenin varsayılan boyutu **300x300**'dür. Restore seçilerek pencere varsayılan boyutuna döndürülür.

## Şimdi Ne Yapabilirsiniz?

Başlarken bildirdiği gibi Windows programlama çok geniş bir konudur. Formlar kuşkusuz bu süreci standardize ediyor olsa da, sizin yaklaşımınız ne şekilde olursa olsun bu doğrudur. Windows, programcılardan önemli taleplerde bulunan zengin özelliklere sahip bir ortamdır. Windows programlamaya yeni başlıyorsanız, Windows'un çeşitli alt sistemlerini öğrenmek için birkaç hafta harcayacağınızı umabilirisiniz. Çalışmaya **System.Windows.Forms** içinde tanımlı kontrollerden başlayabilirsiniz. Ayrıca, Windows'un sıkça kullanılan mesajlarından birkaçını, söz gelisi bir pencerenin yeniden çizilmesi isteminde bulunan bir mesaj gibi, nasıl ele alacağınızı da öğrenmeniz gerekecektir. Bir diğer önemli alt sistem **System.Drawing**'de yer alır. **System.Drawing**, çıktıının pencere içinde nasıl görüntülendiğini kontrol eden çeşitli sınıfları tanımlar. **System.Drawing**, Windows GDI (Grafik Cihaz Arayüzü - Graphics Device Interface) tarafından sağlanan işlevselliği sınıf içine paketler. Birçok kısa program yazmayı deneyin; her yeni seçenek ya da teknikle ilintili olarak bu programların her birinin tam davranışını ve nasıl göründüğünü gözleyin.

**26**

**YİRMİALTINCI BÖLÜM**

---

# **YİNELENEREK İNEN BİR DEYİM AYRIŞTIRICISI**

(10 - 59) \*3 gibi bir nümerik deyim içeren bir karakter katarını girdi olarak alan ve doğru yanıt hesaplayan bir programı nasıl yazarsınız? Programcılar arasında hala programcılığın “piri” kavramı kaldıysa, bunlar, bu işin nasıl yapıldığını bilen az sayıda kişi olmalıdır. Bu konu haricinde becerikli birçok programcı, yüksek düzeyli bir dilin, cebirsel deyimleri bir bilgisayarın çalıştırabileceği komutlara dönüştürme şeklärinden hayrete düşerler. Bu prosedüre *deyim ayrıştırma* (*expression parsing*) denir ve bu, tüm dil derleyicilerinin ve yorumlayıcılarının, elektronik formların ve nümerik deyimleri bir bilgisayarın kullanabileceği bir forma dönüştürmeyi gerektiren her türlü şeyin omurgasını oluşturur.

Yeni başlayanlar için gizemli gelse de deyim ayrıştırma, oldukça zarif bir çözüm sunan net olarak tanımlanmış bir görevdir. Bunun nedeni, deyim ayrıştırma işleminin katı cebir kurallarına göre çalışıyor olmasıdır. Bu bölümde, çoğunlukla *yinelenerek inen ayrıştırıcı* (*recursive-descent parser*) olarak bilinen bir ayrıştırıcı geliştirilmektedir. Ayrıca, nümerik deyimleri hesaplamanızı mümkün kıyan gerekli tüm destek rutinleri de geliştirilmektedir. Ayrıştırıcının işleyışı üzerinde uzmanlaştktan sonra bunu kendi ihtiyaçlarınıza uyacak şekilde kolayca geliştirip, değiştirebilirsiniz.

Burada geliştirilen ayrıştırıcı, kendi başına yararlı bir kod parçası olmasının yanı sıra, ayrıca ikinci bir amaca da hizmet eder: C# dilinin gücünü ve menzilini gözler önüne serer. Bir ayrıştırıcı bir “saf kod” uygulamasıdır. Bir GUI olarak ağa arayüz olmaz ya da sınırlı sistem kaynaklarından yararlanmaz. Bu, geçmişte normal olarak C++’ta yazılacak türde bir koddur. Bir ayrıştırıcının C# kullanılarak kolaylıkla oluşturulabilir olması gerçeği, C#’in her türlü programlama görevi için uygun olduğunu kanıtlamaktadır.

## Deyimler

Bir deyim ayrıştırıcısı cebirsel bir deyimin değerini hesapladığı için bir deyimi oluşturan parçaları anlamanız önemlidir. Deyimler her türde bilgiden oluşturulabilir olsalar da, bu bölümde yalnızca nümerik deyimler ele alınmaktadır. Kendi amacımız açısından nümerik deyimler aşağıdaki öğelerden oluşur:

- Sayılar
- Operatörler +, -, /, \*, ^, %, =
- Parantezler
- Değişkenler

Burada karet operatörü (^) üstel işlemleri (aynı operatörün C#'taki XOR işlevi ile karıştırmayıñ) işaret eder. Eşittir işaretti (=) de atama operatördür. Bu öğeler, deyimler içinde cebir kurallarına göre birleştirilebilirler. İşte birkaç örnek:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10 ^ 5
a = 10 - b
```

Her operatörün şu şekilde önceliği olduğunu varsayıın:

<b>en yüksek</b>	<b>+</b> - (tekli)
	<b>^</b>
	<b>*</b> / %
	<b>+</b> -
<b>en düşük</b>	<b>=</b>

Eşit öncelik sırasına sahip operatörler soldan sağa doğru hesaplanırlar.

Bu bölümde geliştirilen ayırtıcı birkaç kısıtlamaya tabi tutulacaktır. Öncelikle, tüm değişkenler tek harften oluşur (bir başka deyişle, **A**'dan **z**'ye 26 değişken söz konusudur). Değişkenlerde büyük-küçük harf ayrımı yapılmaz (**a** ve **A** aynı değişken olarak ele alınır). İkincisi, tüm nümerik değerlerin **double** olduğu varsayıılır. Ayırtıcıyı kuşkusuz diğer değer tiplerini ele alacak şekilde değiştirmeniz mümkünündür. Son olarak, ayırtıcının mantığını net ve kolay anlaşılabilir tutmak için yalnızca temel düzeyde hata kontrolü dahil edilmiştir.

## Deyimleri Ayırtırmak: Problem

Deyim ayırtma problemi hakkında çok fazla düşünmemişseniz, bunun basit bir iş olduğunu farz edebilirsiniz, fakat değildir. Problemi daha iyi anlamak için şu örnek deyimi hesaplamayı deneyin:

10 - 2 \* 3

Bu deyimin **4** değerine eşit olduğunu biliyorsunuz. Bu spesifik deyimi hesaplayacak bir programı kolaylıkla oluşturabilirsiniz. Ancak asıl problem, herhangi *keyfi* bir deyim için doğru yanıt veren bir programın nasıl oluşturulacağıdır. İlk önce şu tür bir rutin düşünebilirsiniz:

```
a = ilk operandi al
while(operand mevcut ise) {
    op = operatoru al
    b = ikinci operandi al
    a = a op b
}
```

Bu rutin, ilk işlemi gerçekleştirmek için birinci operandı, operatörü ve ikinci operandı alır. Sonra, bir sonraki işlemi gerçekleştirmek için bir sonraki operatörü ve operandı alır vs. Ancak, bu temel yaklaşımı kullanırsanız eğer, **10 - 2 \* 3** deyimi **4** yerine **24** (yani, **8 \* 3**) olarak hesaplanır, çünkü bu prosedür, operatörlerin öncelik sırasını dikkate almamaktadır. Operandları ve operatörleri soldan sağa doğru hesaplayamazsınız, çünkü cebir kuralları çarpmanın çıkartmadan önce yapılmasını dikte ekmektedir. Yeni başlayanların bir kısmı bu problemin kolaylıkla üstesinden gelineceğini düşünürler; bazen çok sınırlı durumlarda gelinebilir de. Ancak, parantezleri, üstel ifadeleri, değişkenleri, tekli operatörleri ve benzerini eklerseniz problem yalnızca daha da kötüye gider.

Deyimleri hesaplayan bir rutin yazmanın çok sayıda yolu olsa da, burada geliştirilen, birisi tarafından yazılabilecek en kolay yaklaşımındır. Bu yaklaşım *yinelenerek inen ayırtıcı*

(recursive-descent parser) olarak adlandırılır. Bu bölümün akışı içinde bu yaklaşımın isminin nereden geldiğini anlayacaksınız. (Ayırıcı yazmak için kullanılan diğer metotlardan bazıları, bir başka bilgisayar programı tarafından üretilmesi gereken karmaşık tablolar kullanırlar. Bunlara kimi zaman *tablo güdümlü* (table-driven) *ayırıcı* da denir.)

## Bir Deyimi Ayırıutmak

Bir deyimi ayırıtmadan hesaplamanın bir çok yolu mevcuttur. Yinelerek inen bir ayırıcı açısından deyimleri, *yinelenen veri yapıları* - yani, kendi kendilerince tanımlanan deyimler olarak düşünün. Simdilik deyimlerin yalnızca **+**, **-**, **\***, **/** ve parantezleri kullandığını varsayırsak, tüm deyimler aşağıdaki kurallara göre tanımlanabilir:

```

deyim -> terim [+ terim] [- terim]
terim -> faktör [* faktör] [/ faktör]
faktör -> değişken, sayı ya da (deyim)

```

Köşeli parantezler isteğe bağlı bir elemanı işaret eder; **->** işaretini ise üretir anlamına gelir. Aslında bu kurallar genellikle deyim *üretur kuralları* olarak adlandırılır. Bu nedenle, terimin tanımını şu şekilde ifade edebilirsiniz: "Terim, faktör çarpı faktör ya da faktör bölü faktör sonucunu üretir." Dikkat ederseniz, operatörlerin öncelik sırası, deyimin tanımlanma şeklinde gizlidir.

**10 + 5 \* B**

Yukarıdaki deyim iki terime sahiptir: **10** ve **5 \* B**. İlk terim iki faktör içerir: **5** ve **B**. Bu faktörler bir sayı ve bir de değişkenden ibarettir. Öte yandan, şu deyim iki faktöre sahiptir:

**14 \* (7 - C)**

Bu faktörler **14** ve **(7 - C)**'dır. Faktörler bir sayı ve bir de parantez içine alınmış bir deyimden oluşur. Parantez içine alınmış deyim iki terim içerir: Bir sayı ve bir değişken.

Bu süreç, yinelerek inen bir ayırıcının temelini oluşturur. Yinelerek inen bir ayırıcı, zincirleme bir düzende çalışan ve üretim kurallarını uygulayan, karşılıklı yinelenen metotlar bütünüdür. Ayırıcı, her uygun adımda, belirtilen işlemleri cebirsel olarak doğru sekansta gerçekleştirir. Üretim kurallarının bir deyimi ayırıutmak için nasıl kullanıldığını anlamak amacıyla gelin şimdi aşağıdaki deyimi kullanarak bir örnek üzerinden gidelim:

**9/3-(100 + 56)**

İşte takip edeceğiniz adımlar:

1. İlk terimi alın: **9/3**.
2. Faktörlerin her birini alın ve tamsayıları bölün. Elde edilen sonuç **3**'tür.
3. İkinci terimi alın: **(100 + 56)**. Bu noktada, ikinci alt deyimi yineleyerek analiz etmeye başlayın.
4. Terimlerin her birini atın ve toplayın. Elde edilen sonuç **156**'dır.

5. YineLEN çağrısından dönün ve 156'yı 3'ten çıkarın. Yanıt –153'tür.

Bu noktada aklınız biraz karıştıysa kendinizi kötü hissetmeyin. Bu, alışması biraz zaman alan oldukça karmaşık bir kavramdır. Deyimlerin bu yinelenen görüntüsünden hatırlamanız gereken iki temel husus vardır. Birincisi, operatörlerin öncelik sıraları üretim kurallarının tanımlanma şeklinde gizlidir. İkincisi, deyimleri bu şekilde ayırtmak ve hesaplamak, biz insanların matematiksel deyimleri hesaplamamıza çok benzemektedir.

Bu bölümün geri kalanında iki ayırtıcı geliştirilmektedir. İlk, yalnızca literal değerlerden oluşan **double** tipinde kayan noktalı deyimleri ayırtırıp, hesaplayacaktır. Bu ayırtıcı, yinelenerek inen ayırtırma metodunun esaslarını gözler önüne serer. ikinci ayırtıcı ise değişken kullanma becerisini ekler.

## Bir Deyimi Parçalara Ayırmak

Bir deyimin değerini hesaplamak için söz konusu deyimin bileşenlerini ayırtıcıya ayrı ayrı beslemek gereklidir. Örneğin, şu deyim

A \* B - (W + 10)

ayrı ayrı şu bölümleri içerir: **A**, **\***, **B**, **-**, **(**, **W**, **+**, **10** ve **)**. Ayırtırma dilinde bir deyimin her bileşenine *simge* (token) denir ve her simge deyimin ayrı bir birimini temsil eder. Bir deyimi *simgeselleştirme*, ayırtırmanın temelidir. Bu nedenle, ayırtıcının kendisini incelemeden önce gelin simgeselleştirmeye göz atalım.

Bir deyimi simgelere ayırmak için deyimin başından başlayıp sonuna kadar tek tek ilerleyerek sırayla karşılık gelen simgeleri döndüren bir metoda ihtiyacınız vardır. Metot ayrıca boşlukları atlayabilmeli ve deyimin sonunu algılayabilmelidir. Burada geliştirilen ayırtıcıda bu görevi gören metot **GetToken()** olarak adlandırılmıştır.

Bu bölümdeki her iki ayırtıcı da **Parser** sınıfı içine pakellenmişlerdir. Bu sınıf daha sonra ayrıntılı olarak ele alınacak olsa da, **GetToken()**'ın nasıl çalıştığını açıklayabilmek için söz konusu sınıfın ilk bölümünün gösterilmesi gereklidir **Parser**, aşağıda gösterilen numaralandırmaları ve alanları tanımlayarak başlar:

```
class Parser {
    // Simge tiplerini numaralandır..
    enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
    // Hata tiplerini numaralandır..
    enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

    string exp;      // deyimi tutan karakter katarını gösterir
    int expIdx;     // deyim içindeki mevcut indeks
    string token;   // mevcut simgeyi tutar
    Types tokType;  // simgenin tipini tutar
```

**Parser** iki numaralandırma tanımlayarak başlar. Birincisi **Types**'dır. Bir deyimi ayırtırırken bir simge çoğunlukla bir tip ile ilişkilendirilir. Bu bölümde geliştirilen ayırtıcılar için yalnızca üç tip gereklidir: değişken, sayı ve sınırlayıcı (delimiter), Bunlar **Types** numaralandırması tarafından tanımlanan **VARIABLE**, **NUMBER** ve **DELIMITER** değerleri ile simgelenirler. **DELIMITER** kategorisi hem operatörler hem de parantezler için kullanılır. **NONE** tipi yalnızca tanımsız bir simge için bir yer göstericisidir. **Errors** numaralandırması çeşitli ayırtırma hatalarını simgeler.

Ayırtılmakta olan deyimi tutan karakter katarına atıfta bulunan referans **exp** içinde saklanır. Dolayısıyla, **exp** "10+4" gibi bir karakter katarına atıfta bulunacaktır. Söz konusu karakter katarı içindeki bir sonraki sembolün indeksi **expIdx** içinde tutulur. **expIdx**'in ilk değeri sıfırdır. Elde edilen simge **token** içinde; tipi ise **tokType** içinde saklanır.

Ayırtıcı tarafından kullanılan **GetToken()** metodu aşağıda gösterilmiştir. Bu metot ne zaman çağrılısa, **expIdx**'ten başlayarak, **exp** tarafından atıfta bulunan karakter katarı içindeki deyimde yer alan bir sonraki sembolü elde eder. Başka bir ifadeyle, **GetToken()** ne zaman çağnlsa, **exp[expIdx]**'teki bir sonraki sembolü alır. Metot daha sonra bu sembolü **token** alanına yerleştirir. Sembolün tipini de **tokType**'a yerleştirir. **GetToken()**, **isDelim()** metodunu kullanır. Bu metot da aşağıda gösterilmiştir:

```
// Bir sonraki sembolu elde et.
void GetToken()
{
    tokType = Types.NONE;
    token = "";

    if(expIdx == exp.Length) return; // deyimin sonu

    // boslukları atla
    while(expIdx < exp.Length &&
        Char.IsWhiteSpace(exp[expIdx])) ++expIdx;

    /* deyimin sonundaki bosluklar ve gorunmeyen karakterler
       deyimi sonlandirir */
    if(expIdx == exp.Length) return;

    if(IsDelim(exp[expIdx])) { // operator mu?
        token += exp[expIdx];
        expIdx++;
        tokType = Types.DELIMITER;
    }
    else if(Char.IsLetter(exp[expIdx])) { // degisken mi?
        while(!IsDelim(exp[expIdx])) {
            token += exp[expIdx];
            expIdx++;
            if(expIdx >= exp.Length) break;
        }
        tokType = Types.VARIABLE;
    }
    else if(Char.IsDigit(exp[expIdx])) { // sayi mi?

```

```

        while(IsDelim(exp[expIdx])) {
            token += exp[expIdx];
            expIdx++;
            if(expIdx >= exp.Length) break;
            tokType = Types.NUMBER;
        }
    }

    // c bir sınırlayıcı ise true dondur.
    bool IsDelim(char c)
    {
        if((" +/*%^()".IndexOf(c) != -1))
            return true;
        return false;
    }
}

```

**GetToken()**'a yakından göz atın. İlk birkaç değer atamadan sonra **GetToken()**, deyimin sonuna ulaşılıp ulaşılmadığını belirlemek için **expIdx**'in **exp.Length**'e eşit olup olmadığını kontrol eder. **expIdx** incelenmekte olan deyimin indeksi olduğu için eğer o, deyimi tutan karakter katarının uzunluğuna eşitse deyim tamamıyla ayırtılmıştır.

Eğer deyimden alınması gereken daha hala simge varsa, **GetToken()** önce baştaki boşlukları atlar. Boşluklar atlandıktan sonra **exp[expIdx]** artık ya bir rakam, bir değişken, bir operatör ya da deyimi sondaki boşluklar sonlandıryorsa bir boşluk içerir. Bir sonraki karakter eğer bir operatör ise bu, **token** içinde bir karakter katarı olarak döndürülür ve **DELIMITER**, **tokType** içinde saklanır. Eğer bir sonraki karakter bir harf ise, bu, değişkenlerden biri olarak kabul edilir. **token** içinde bir karakter katarı olarak döndürülür ve **tokType**'a **VARIABLE** değeri atanır. Bir sonraki karakter bir rakam ise bütün sayı okunur ve karakter katarı şekliyle **token**'da saklanır. Tipi **NUMBER**'dır. Son olarak, eğer karakter bunların hiçbirini değilse, **token** bir **null** karakter katarı içerecektir.

**GetToken()**'un kodunun anlaşılabilirliğini korumak için belirli miktar hata kontrolü atlanmıştır ve birtakım varsayımlarda bulunulmuştur. Söz geliş, önünde bir boşluk olması kaydıyla tanınmayan herhangi bir karakter deyimi sona erdirebilir. Ayrıca, bu versiyonda değişkenler herhangi uzunlukta olabilir, fakat yalnızca ilk harfleri önem taşır. Sizin spesifik uygulamanıza bağlı olarak daha fazla hata kontrolü ya da diğer ayrıntıları ekleyebilirsiniz.

**GetToken()**'ın nasıl çalıştığını daha iyi anlamak için şu deyimi ele alın:

A + 100 - (B \* C) / 2

Bu deyim simgeselleştirildiğinde **GetToken()** aşağıdaki simgeleri ve simge tiplerini elde eder:

Simge	Simge tipi
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER

```

(      DELIMITER
B      VARIABLE
*
C      DELIMITER
)      VARIABLE
/      DELIMITER
2      NUMBER

```

Şunu hatırlınızda tutun: **token** yalnızca tek bir karakter içerse bile daima bir karakter katarı tutar.

## Basit Bir Deyim Ayrıştırıcısı

İşte, ayırtırıcının ilk versiyonu. Bu, yalnızca literal, operatör ve parantezlerden oluşan deyimlerin değerini hesaplayabilir. **GetToken()** değişkenleri işleyebiliyor olsa da ayırtırıcının bunlarla bir işi yoktur. Bu basitleştirilmiş ayırtırıcının nasıl çalıştığını bir kez anladıkten sonra değişkenleri ele alma becerisini de ekleyeceğiz.

```

/*
    Bu nodul, degiskenleri kullanmayan,
    yinelerek inen bir ayrristirici icerir.
*/

using System;

// Ayrristirici hatalari icin kural disi durum sinifi.
class Parser(Exception : ApplicationException {
    public ParserException(string str) : base(str) { }

    public override string ToString() {
        return Message;
    }
}

class Parser {
    // Simge tiplerini numaralandir..
    enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
    // Hata tiplerini numaralandir..
    enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

    string exp;          // deyimi tutan karakter katarini gosterir
    int expIdx;         // deyim icindeki mevcut indeks
    string token;        // mevcut simgeyi tutar
    Types tokType;       // simgenin tipini tutar

    // Ayrristiricinin giris noktasi.
    public double Evaluate(string expstr)
    {
        double result;

        exp = expstr;
        expIdx = 0;
    }
}

```

```
try {
    GetToken();
    if(token == "") {
        SyntaxErr(Errors.NOEXP); // Deyim mevcut degil
        return 0.0;
    }

    EvalExp2(out result);

    if(token != "") // son simge null olmali
        SyntaxErr(Errors.SYNTAX);

    return result;
} catch (ParserException exc) {
    // istege gore baska bir hata yonetimi ekle.
    Console.WriteLine(exc);
    return 0.0;
}
}

// iki terimi ekle ya da cikart.
void EvalExp2(out double result)
{
    string op;
    double partialResult;

    EvalExp3(out result);
    while((op = token) == "+" || op == "-") {
        GetToken ();
        EvalExp3(out partialResult);
        switch(op) {
            case "-":
                result = result - partialResult;
                break;
            case "+":
                result = result + partialResult;
                break;
        }
    }
}

// Iki faktoru carp ya da bol.
void EvalExp3(out double result)
{
    string op;
    double partialResult = 0.0;

    EvalExp4(out result);
    while((op = token) == "*" ||
          op == "/" || op == "%") {
        GetToken();
        EvalExp4(out partialResult);
        switch(op) {
            case "*":
                result = result * partialResult;
```

```
        break;
    case "/":
        if(partialResult == 0.0)
            SyntaxErr(Errors.DIVBYZERO);
        result = result / partialResult;
        break;
    case "%":
        if(partialResult == 0.0)
            SyntaxErr(Errors.DIVBYZERO);
        result = (int) result % (int) partialResult;
        break;
    }
}
}

// Ustel ifadeyi hesapla.
void EvalExp4 (out double result)
{
    double partialResult, ex;
    int t;

    EvalExp5(out result);
    if(token == "^") {
        GetToken();
        EvalExp4(out partialResult);
        ex = result;
        if(partialResult == 0.0) {
            result = 1.0;
            return;
        }
        for(t=(int)partialResult-1; t > 0; t--)
            result = result * (double)ex;
    }
}

// Tekli + ya da -'nin degerini hesapla.
void EvalExp5(out double result)
{
    string op;

    op = "";
    if((tokType == Types.DELIMITER) &&
       token == "+" || token == "-") {
        op = token;
        GetToken();
    }

    EvalExp6(out result);
    if(op == "-") result = -result;
}

// Parantez içindeki deyimi hesapla.
void EvalExp6(out double result)
{
    if((token == "(")) {
```

```

        GetToken();
        EvalExp2(out result);
        if(token != ")")
            SyntaxErr(Errors.UNBALPARENS);
        GetToken();
    }
    else Atom(out result);
}

// Bir sayinin degerini al.
void Atom(out double result)
{
    switch(tokType) {
        case Types.NUMBER:
            try {
                result = Double.Parse(token);
            } catch (FormatException) {
                result = 0.0;
                SyntaxErr(Errors.SYNTAX);
            }
            GetToken();
            return;
        default:
            result = 0.0;
            SyntaxErr(Errors.SYNTAX);
            break;
    }
}

// Soz dizimi hatasini kontrol altina al.
void SyntaxErr(Errors error)
{
    string[] err = {
        "Syntax Error",
        "Unbalanced Parentheses",
        "No Expression Present",
        "Division by Zero"
    };

    throw new ParserException(err[(int)error]);
}

// Bir sonraki simbolu elde et.
void GetToken()
{
    tokType = Types.NONE;
    token = "";

    if(expIdx == exp.Length) return;
    // deyim sonu ise bosluklari ve gorunmeyen karakterleri atla
    while(expIdx < exp.Length &&
          Char.IswhiteSpace(exp[expIdx])) ++expIdx;

    /* deyimin sonundaki bosluklar ve gorunmeyen karakterler
       deyimi sonlandirir */
}

```

```

        if(expIdx == exp.Length) return;

        if(IsDelim(exp[expIdx])) { // operator mu?
            token += exp[expIdx];
            expIdx++;
            tokType = Types.DELIMITER;
        }
        else if(Char.IsLetter(exp[expIdx])) { // degisken mi?
            while(!IsDelim(exp[expIdx])) {
                token += exp[expIdx];
                expIdx++;
                if(expIdx >= exp.Length) break;
            }
            tokType = Types.VARIABLE;
        }
        else if(Char.IsDigit(exp[expIdx])) { // sayi mi?
            while(!IsDelim(exp[expIdx])) {
                token += exp[expIdx];
                expIdx++;
                if(expIdx >= exp.Length) break;
            }
            tokType = Types.NUMBER;
        }
    }

    // c bir sinirlandirici ise true dondur.
    bool IsDelim(char c)
    {
        if((" +/*%^()".IndexOf(c) != -1))
            return true;
        return false;
    }
}

```

Gördüğü gibi ayırtıcı, şu operatörleri ele alabilmektedir: **+**, **-**, **\***, **/** ve **%**. İlaveten, üstel tamsayı operatörünü (**^**) ve tekli eksi operatörünü de ele alabilmektedir. Ayırtıcı ayrıca parantezlerle de doğru biçimde ilgilenebilir.

Ayırtıcıyı kullanmak için öncelikle **Parser** tipinde bir nesne oluşturun. Sonra, değerinin hesaplanması istediğiniz deyimi tutan karakter katarını argüman olarak aktararak **Evaluate()**'ı çağırın. **Evaluate()** sonucu döndürür. Aşağıdaki program ayırtıcıyı göstermektedir:

```

// Ayrustiriciyi gosterir.

using System;

class ParserDemo {
    public static void Main() {
        string expr;
        Parser p = new Parser();

        Console.WriteLine("Enter an empty expression to stop.");

```

```

        for(;;) {
            Console.Write("Enter expression: ");
            expr = Console.ReadLine();
            if(expr == "") break;
            Console.WriteLine("Result: " + p.Evaluate(expr));
        }
    }
}

```

İşte örnek çıktı:

Enter an empty expression to stop.

Enter expression: 10.2\*3

Result: 4

Enter expression: (10-2)\*3

Result: 24

Enter expression: 10/3.5

Result: 2.85714285714286

## Ayrıştırıcıyı Anlayalım

Gelin şimdi, **Parser**'a ayrıntılı olarak göz atalım. **GetToken()** anlatılırken de bahsedildiği gibi, **Parser** dört özel alan içermektedir. Sonucu hesaplanacak deyimi içeren karakter katarı **exp** ile gösterilir. **Evaluate()**'e yapılan her çağrıda bu alan ayarlanır. Ayrıştırıcının, standart C# karakter katarlarında saklanan deyimleri hesapladığı hatırlınızda tutmak önemlidir. Örneğin, aşağıdaki karakter katarları ayrıştırıcının hesaplayabileceği deyimleri içermektedeler:

```
"10 - 5"
"2 * 3.3 / (3.1416 * 3.3)"
```

**exp** içindeki mevcut indeks konumu **expIdx**'te saklanır. Ayrıştırma işlemi başlayınca **expIdx**'e sıfır değeri verilir. Ayrıştırıcı deyim ürünlerinde ilerledikçe **expIdx** birer birer artırılır. **token** alanı mevcut sembolü tutar. **tokType** ise simge tipini içerir.

Ayrıştırıcının giriş noktası **Evaluated()** içindendir. **Evaluate()**, analiz edilecek deyimi içeren karakter katarı ile birlikte çağrılmalıdır. **EvalExp2()**'den **EvalExp6()**'ya kadar yer alan metodlar **Atom()** ile birlikte, yineLENEREK inen ayrıştırıcıyı oluştururlar. Bu metodlar, daha önce anlatılan deyim üretim kurallarının geliştirilmiş bir bütününyi uygularlar. Metotların üstünde yer alan açıklamalar, onların gerçekleştirdikleri işlevleri tarif etmektedir. Ayrıştırıcının bir sonraki versiyonunda **EvalExp1()** adında bir metot daha eklenecektir.

**SyntaxErr()**, deyim içindeki söz dizimi halalarını kontrol altına alır. **GetToken()** ve **isDelim()** metodları, önceden anlatıldığı şekliyle deyimi bileşenlerine ayırır. Ayrıştırıcı, deyimin başından başlayıp sonuna kadar ilerlerken deyimden simgeleri elde etmek için **GetToken()**'ı kullanır. Elde edilen sembolün tipine göre farklı eylemlerde bulunulur.

Ayristiricinin bir deyimin degerini tam olarak ne sekilde hesapladigini anlamak icin asagidaki deyim üzerinde calisın:

10 - 3 \* 2

Ayristiriciya giriş noktası olan **Evaluate()** çağrılinca metot ilk simbolü alır. Eğer simge bir **null** karakter katarı ise **No Expression Present** mesajı ekranda görüntülenir ve **Evaluate()** 0.0 döndürür. Ancak, bu örnekte simge 10 değerini içermektedir. Sonra, **EvalExp2()** çağrılr. **EvalExp2()**, **EvalExp3()**'ü çağrırlar; **EvalExp3()**, **EvalExp4()**'ü çağrırlar; **EvalExp4()** ise sırası geldiğinde **EvalExp5()**'i çağrırlar. Daha sonra, **EvalExp5()**, söz konusu simbolün tekli artı ya da eksı olup olmadığını belirler. Bu örnekte simge, tekli artı ya da eksı değildir; dolayısıyla, **EvalExp6()** çağrırlar. Bu noktada **EvalExp6()**, sayının değerini bulmak için ya yinelenecek **EvalExp2()**'yi (parantezli bir deyim söz konusu olduğunda) ya da **Atom()**'u çağrırlar. Simge, sol parantez olmadığı için **Atom()** gerçekleşir ve **result'a** 10 değeri atanır. Sonra, bir başka simge alınır ve metotlar zincir boyunca yukarı yönde geri dönmeye başlar. Şimdi simge - operatörü olduğu için metotlar **EvalExp2()**'ye kadar döner.

Bir sonraki aşamada meydana gelenler çok önemlidir. Söz konusu simge - olduğu için bu simge **op** içinde saklanır. Ayristirici sonraki simbolü alır; bu 3'tür ve zincir boyunca tekrar aşağı yönde inmeye başlar. Onceki gibi yine **Atom()**'a girilir, **result** içinde 3 değeri döndürülür ve \* simgesi okunur. Bu, zincir boyunca yukarı yönde **EvalExp3()**'e kadar geri dönmeye neden olur. Burada en son simge olan 2 okunur. Bu noktada ilk aritmetik işlem meydana gelir: 2 ile 3'ün çarpımı. Sonuç **EvalExp2()**'ye döndürülür ve çıkartma işlemi gerçekleştirilir. Çıkartma işleminden 4 sonucu elde edilir. Bu süreç başlangıcta karmaşık gibi görünse de, bu metodun her zaman doğru olarak işlediğini doğrulamak için diğer birkaç örnek üzerinde bunu deneyebilirsiniz.

Ayristirma sırasında bir hata meydana gelirse, **SyntaxErr()** metodu çağrırlar. Bu metot, hatayı tarif eden bir **ParserException** fırlatır. **ParserException**, ayristirici dosyasının başında tanımlanan kendi oluşturduğumuz bir kural dışı durumdur.

Bu ayristirici, onceki örnekte gösterildiği gibi, basit masa üstü hesap makinesinde kullanmak için uygun olur. Ancak, bir bilgisayar dilinde, veri tabanında ya da sofistike bir hesap makinesinde kullanılabilmesi için öncelikle değişkenleri ele alabilmesi gereklidir. Bu, bir sonraki bölümün konusudur.

## Ayristiriciya Değişken Eklemek

Tüm programlama dilleri, pek çok hesap makinesi ve elektronik form değerlerin daha sonradan kullanılabilmesi için değişken kullanırlar. Ayristiricinin bu tür uygulamalarda kullanılabilmesi için öncelikle değişkenleri dahil edecek şekilde genişletilmesi gereklidir. Bunu gerçekleştirmek için ayristiriciya birkaç öğe eklemeniz gereklidir. Öncelikle, kuşkusuz, değişkenlerin kendileri eklenmelidir. Önceden de bildirildiği gibi, değişkenler için **A**'dan **z**'ye

kadar harfleri kullanacağınız. Değişkenler **Parser** sınıfı içinde bir dizide saklanırlar. Her değişken, 26 elemanlı bir **double** dizisinde tek dizi konumunu kullanır. Bu nedenle, **Parser** sınıfına şu satırı ekleyin:

```
double[] vars = new double[26];
```

Ayrıca, değişkenlere ilk değerlerini atayan aşağıdaki **Parser** yapılandırıcısını da eklemeniz gerekecek:

```
public Parser() {
    // Degiskenlere sifir degeri ata.
    for(int i = 0; i < vars.Length; i++)
        vars[i] = 0.0;
}
```

Kullanıcıya yardımcı olmak adına değişkenlere ilk değer olarak **0.0** atanmıştır.

Verilen bir değişkenin değerini aramak için de ayrıca bir metoda ihtiyacınız olacaktır. Değişkenler, **A**'dan **Z**'ye isimlendirildikleri için **vars** dizisini indekslemek amacıyla kolaylıkla kullanılabilirler. **A**'nın ASCII değerini değişkenin isminden çıkartmak yeterlidir. Aşağıda gösterilen **FindVars()** metodu bunu gerçekleştirir:

```
// Degiskenin degerini dondur.
double FindVar(string vname)
{
    if(!Char.IsLetter(vname[0])) {
        SyntaxErr(Errors.SYNTAX);
        return 0.0;
    }
    return vars[Char.ToUpper(vname[0]) - 'A'];
}
```

Bu metod yazıldığı şekliyle aslında uzun değişken isimlerini de kabul edecektir, söz gelisi **sA12** ya da **test**. Fakat, yalnızca ilk harf önemlidir. Kendi ihtiyaçlarınıza uyacak şekilde bu özelliği değiştirebilirsiniz.

**Atom()** metodunu da hem sayıları hem de değişkenleri ele alacak şekilde değiştirmelisiniz. Yeni versiyon aşağıda gösterilmiştir:

```
// Sayi ya da degiskenin degerini al.
void Atom(out double result)
{
    switch(tokType) {
        case Types.NUMBER:
            try {
                result = Double.Parse(token);
            } catch (FormatException) {
                result = 0.0;
                SyntaxErr(Errors.SYNTAX);
            }
            GetToken();
            return;
    }
}
```

```

        case Types.VARIABLE:
            result = FindVar(token);
            GetToken();
            return;
        default:
            result = 0.0;
            SyntaxErr(Errors.SYNTAX);
            break;
    }
}

```

Aynışırıcıının değişkenleri doğru kullanması için gerekli olan eklemeler teknik olarak bunlardan ibarettir. Ancak, bu değişkenlere değer atanması hiçbir şekilde mümkün değildir. Bir değişkene bir değer verilebilmesi için ayırtıcıının atama operatörünü (=) ele alabilmesi gereklidir. Atamaları gerçekleştirmek için **Parser** sınıfına bir başka metod daha ekleyeceğiz: **EvalExp1()**. Bu metod artık yineLENEREK inen zinciri başlatacaktır. Bunun anlamı şudur: Deyimi ayırtırma işlemini başlatmak için **Evaluate()** tarafından çağrılması gereken bu metottur: **EvalExp2()** değildir. **EvalExp1()** aşağıda gösterilmiştir:

```

// Atama ifadesini hesapla.
void EvalExp1(out double result)
{
    int varIdx;
    Types ttokType;
    string temptoken;

    if(tokType == Types.VARIABLE) {
        // eski simbolu sakla
        temptoken = String.Copy(token);
        ttokType = tokType;

        // Degiskenen indeksini hesapla.
        varIdx = Char.ToUpper(token[0]) - 'A';

        GetToken();
        if(token != "=") {
            PutBack(); // mevcut simbolu dondur
            // eski simbolu geri al - bu bir atama degil
            token = String.Copy(temptoken);
            tokType = ttokType;
        }
        else {
            GetToken(); // deyimin bir sonraki parcasini al
            EvalExp2(out result);
            Vars[varIdx] = result;
            return;
        }
    }

    EvalExp2(out result);
}

```

**EvalExp1()**, esasen bir atamanın yapılmışlığını belirlemek için ileriye bakmalıdır. Bunun nedeni şudur: Bir değişken ismi daima bir atamanın önünde gelir, fakat bir değişken ismi, peşinden bir atama deyiminin geldiğini garanti etmez. Yani, ayristirıcı **A = 100** deyimini bir atama olarak alacaktır, fakat **A/10** deyiminin bir atama olmadığını bilecek kadar da akıllıdır. Bunu gerçekleştirmek için **EvalExp1()**, girdi akışından bir sonraki simbolü okur. Eğer simge eşittir işaretini değilse, simge **PutBack()** çağrılarak daha sonra kullanılmak üzere girdi akışına geri döndürülür. **PutBack()** aşağıda gösterilmiştir

```
// Sembolu girdi akisina dondur.
void PutBack()
{
    for(int i = 0; i < token.Length; i++) expIdx--;
}
```

Gerekli tüm değişiklikler yapıldıktan sonra ayristirıcı artık şu şekilde görünecektir:

```
/*
    Bü modul, degiskenleri algilanay,
    yinelenerek inen bir ayristirici icerir.
*/

using System;

// Ayrustirici hatalari icin kural disi durum sinifi.
class ParserException : ApplicationException {
    public ParserException(string str) : base(str) { }

    public override string ToString() {
        return Message;
    }
}

class Parser {
    // Simge tiplerini numaralandir..
    enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
    // Hata tiplerini numaralandir..
    enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

    string exp;      // deyimi tutan karakter katarini gosterir
    int expIdx;     // deyim icindeki mevcut indeks
    string token;   // mevcut simgeyi tutar
    Types tokType;  // simgenin tipini tutar

    // Degiskenler için dizi.
    double[] vars = new double[26];

    public Parser() {
        // Degiskenlere sifir degerini ata.
        for(int i = 0; i < vars.Length; i++)
            vars[i] = 0.0;
    }

    // Ayrustiricinin giris noktasι.
```

```
public double Evaluate(string expstr)
{
    double result;

    exp = expstr;
    expIdx = 0;

    try {
        GetToken();
        if(token == "") {
            SyntaxErr(Errors.NOEXP); // deyim mevcut degil
            return 0.0;
        }

        EvalExp1(out result); // simdi baslamak icin
                               // EvalExp1()'i cagir

        if(token != "") // son simge null olmali
            SyntaxErr(Errors.SYNTAX);

        return result;
    } catch (ParserException exc) {
        // Istege gore baska bir hata yonetimi ekle.
        Console.WriteLine(exc);
        return 0.0;
    }
}

// Atama ifadesini hesapla.
void EvalExp1(out double result)
{
    int varIdx;
    Types ttokType;
    string temptoken;

    if(tokType == Types.VARIABLE) {
        // eski simbolu sakla
        temptoken = String.Copy(token);
        ttokType = tokType;

        // Degiskenin indeksini hesapla.
        varIdx = Char.ToUpper(token[0]) - 'A';

        GetToken();
        if(token != "=") {
            PutBack(); // mevcut simgeyi dondur
            // eski simbolu geri al - bu bir atama degil
            token = String.Copy(temptoken);
            tokType = ttokType;
        }
        else {
            GetToken(); // deyimin bir sonraki parcasini al
            EvalExp2(out result);
            vars[varIdx] = result;
            return;
        }
    }
}
```

```
        }

    }

    EvalExp2(out result);
}

// iki terimi ekle ya da cikart.
void EvalExp2(out double result)
{
    string op;
    double partialResult;

    EvalExp3(out result);
    while((op = token) == "+" || op == "-") {
        GetToken();
        EvalExp3(out partialResult);
        switch(op) {
            case "-":
                result = result - partialResult;
                break;
            case "+":
                result = result + partialResult;
                break;
        }
    }
}

// İki faktoru carp ya da bol.
void EvalExp3(out double result) {
{
    string op;
    double partialResult = 0.0;

    EvalExp4(out result);
    while((op = token) == "*" || op == "/" || op == "%") {
        GetToken();
        EvalExp4(out partialResult);
        switch(op) {
            case "*":
                result = result * partialResult;
                break;
            case "/":
                if(partialResult == 0.0)
                    SyntaxErr(Errors.DIVBYZERO);
                result = result / partialResult;
                break;
            case "%":
                if(partialResult == 0.0)
                    SyntaxErr(Errors.DIVBYZERO);
                result = (int) result % (int) partialResult;
                break;
        }
    }
}
```

```
// Ustel ifadeyi hesapla.  
void EvalExp4(out double result)  
{  
    double partialResult, ex;  
    int t;  
  
    EvalExp5(out result);  
    if(token == "*") {  
        GetToken();  
        EvalExp4(out partialResult);  
        ex = result;  
        if(partialResult == 0.0) {  
            result = 1.0;  
            return;  
        }  
        for(t = (int) partialResult - 1; t > 0; t--)  
            result = result * (double) ex;  
    }  
}  
  
// Tekli + ya da -'nin degerini hesapla.  
void EvalExp5(out double result)  
{  
    string op;  
  
    op = "";  
    if((tokType == Types.DELIMITER) &&  
        token == "+" || token == "-") {  
        op = token;  
        GetToken();  
    }  
    EvalExp6(out result);  
    if(op == "-") result = -result;  
}  
  
// Parantez icindeki deyimi hesapla.  
void EvalExp6(out double result)  
{  
    if((token == "(")) {  
        GetToken();  
        EvalExp2(out result);  
        if(token != ")")  
            SyntaxErr(Errors.UNBALPARENS);  
        GetToken();  
    }  
    else Atom(out result);  
}  
  
// Sayinin ya da degiskenin degerini al.  
void Atom(out double result)  
{  
    switch(tokType) {  
        case Types.NUMBER:  
            try {  
                result = Double.Parse(token);  
            }  
    }  
}
```

```
        } catch (FormatException) {
            result = 0.0;
            SyntaxErr(Errors.SYNTAX);
        }
        GetToken();
        return;
    case Types.VARIABLE:
        result = FindVar(token);
        GetToken();
        return;
    default:
        result = 0.0;
        SyntaxErr(Errors.SYNTAX);
        break;
    }
}

// Degiskenin degerini dondur.
double FindVar(string vname)
{
    if(!Char.IsLetter(vname[0])) {
        SyntaxErr(Errors.SYNTAX);
        return 0.0;
    }
    return vars[Char.ToUpper(vname[0]) - 'A'];
}

// Girdi akisina bir simge dondur.
void PutBack()
{
    for(int i = 0; i < token.Length; i++) expIdx--;
}

// Soz dizimi hatasini kontrol altina al.
void SyntaxErr(Errors error)
{
    string[] err = {
        "Syntax Error",
        "Unbalanced Parentheses",
        "No Expression Present",
        "Division by Zero"
    };

    throw new ParserException(err[(int)error]);
}

// Bir sonraki simgeyi elde et.
void GetToken()
{
    tokType = Types.NONE;
    token = "";

    if(expIdx == exp.Length) return; // deyim sonu ise

    // bosluklari atla
```

```

        while(expIdx < exp.Length &&
              Char.IsWhiteSpace(exp[expIdx])) ++expIdx;

        // deyimin sonundaki bosluklar deyimi sonlandirir
        if(expIdx == exp.Length) return;

        if(IsDelim(exp[expIdx])) { // operator mu?
            token += exp[expIdx];
            expIdx++;
            tokType = Types.DELIMITER;
        }
        else if(Char.IsLetter(exp[expIdx])) { // degisen mi?
            while(!IsDelim(exp[expIdx])) {
                token += exp[expIdx];
                expIdx++;
                if(expIdx >= exp.Length) break;
            }
            tokType = Types.VARIABLE;
        }
        else if(Char.IsDigit(exp[expIdx])) { // sayi mi?
            while(!IsDelim(exp[expIdx])) {
                token += exp[expIdx];
                expIdx++;
                if(expIdx == exp.Length) break;
            }
            tokType = Types.NUMBER;
        }
    }

    // c bir sinirlandirici ise true dondur.
    bool IsDelim(char c)
    {
        if((" +/*%^()".IndexOf(c) != -1))
            return true;
        return false;
    }
}

```

Geliştirilmiş ayrıştırıcıyı denemek için basit ayrıştırıcıda kullandığınız programın aynısını kullanabilirsiniz. Geliştirilmiş ayrıştırıcıyı kullanarak artık şu tür deyimleri girebilirsiniz:

```

A = 10/4
A - B
C = A * (F - 21)

```

## Yinelenerek İnen Ayrıştırıcıda Söz Dizimi Kontrolü

Deyim ayrıştırma işleminde bir söz dizimi hatası, basitçe, ayrıştırıcının gerektirdiği katı kurallara girdi deyiminin uymaması durumudur. Buna çoğunlukla insanların hataları neden olur - bunlar genellikle yazım hatalarıdır. Örneğin, aşağıdaki deyimler bu bölümde yer alan ayrıştırıcılar için geçerli değildir:

```
10 ** 8e
```

```
( (10 - 5) * 9  
/8
```

İlk deyim aynı satırda iki operatör içermektedir; ikincisinde eşlenmeyen parantez vardır ve sonucusu, deyimin başında bolu işaretine sahiptir. Ayırıştırıcı, bu koşulların hiçbirine izin vermez. Söz dizimi hataları ayırıştırıcının hatalı sonuçlar vermesine neden olabildiği için bunlara karşı tedbirli olmalısınız.

Ayırıştırıcı kodu üzerinde çalıştiysanız, yalnızca belirli durumlarda çağrılan **Syntax()** metodunu muhtemelen fark etmişsinizdir. Diğer ayırıştırıcı tiplerinden farklı olarak yinelenecek inen ayırıştırıcı metodu, söz dizimi kontrolünü kolaylaştırmaktadır, çünkü söz dizimi kontrolü çoğu kez **Atom()**'da, **FindVar()**'da ya da parantezlerin kontrol edildiği **EvalExp6()**'da meydana gelmektedir.

**SyntaxErr()** çağrıldığı zaman hatanın tanımını içeren bir **ParserException** fırlatır. **Parser**'ın mevcut şekliyle bu kural dışı durum, **Evaluate()** içinde yakalanır. Dolayısıyla; ayırıştırıcı, bir hata ile karşılaşınca hemen durur. Kuşkusuz, siz kendi ihtiyaçlarınıza uygun olacak şekilde bu davranışları değiştirebilirsiniz.

## Denenecek Birkaç Şey

Bu bölümün başlarında bahsedildiği gibi, ayırıştırıcı tarafından yalnızca asgari düzeyde hata kontrolü gerçekleştirilmektedir. Siz ayrıntılı hata bildirimini eklemek isteyebilirsiniz. Örneğin, deyim içinde hatanın tespit edildiği noktanın işaretlenmesini sağlayabilirsiniz. Bu, kullanıcının bir söz dizimi hatasını bulup, düzeltmesine olanak tanıyacaktır.

Ayırıştırıcı şu haliyle yalnızca nümerik deyimleri hesaplayabilir. Ancak, birkaç eklemeyle diğer tipte deyimleri de hesaplamasını sağlayabilirsiniz. Örneğin, karakter katarları, uzaysal koordinatlar ya da karmaşık sayılar gibi. Ayırıştırıcının, söz geliş, karakter katarı nesnelerinin değerini hesaplaması için aşağıdaki değişiklikleri yapmanız gereklidir.

1. **STRING** adında yeni bir simge tanımlayın.
2. **GetToken()**'ı karakter katarlarını algılayacak şekilde geliştirin.
3. **Atom()**'un içine **STRING** tipinde simgeleri ele alan yeni bir “case” ekleyin.

Bu adımlan uyguladıktan sonra ayırıştırıcı, karakter katarı şeklindeki şu tür deyimleri ele alabilir:

```
a = "one"  
b = "two"  
c = a + b
```

**c**'deki sonuç, **a** ve **b**'nin peş peşe eklenmiş şekli, ya da “onetwo”, olmalıdır.

İşte, ayırtıcı için iyi bir uygulama: Kullanıcının girdiği bir deyimi alan ve sonra sonucu ekranında gösteren basit, açılır bir mini hesap makinesi oluşturun. Bu, hemen hemen tüm ticari uygulamalar için mükemmel bir ek olacaktır.

Son olarak, **Parser**'ı bir bileşene dönüştürmeyi deneyin. Bunu yapmak kolaydır. Önce, **Parser**'ın **Component**'tan kalıtımıla elde edilmesini sağlayın. Sonra, **Dispose(bool)**'u uygulayın. İşte bu kadar! Bunu yaparsanız, yazdığınız tüm uygulamalarda kullanabileceğiniz bir ayırtıcıya sahip olacaksınız.

E K A

A

---

# XML AÇIKLAMALARI REFERANSI

C#'ta üç tane açıklama işaretti vardır. Bunların ilk ikisi // ve /\* ... \*/ işaretleridir. Üçüncü tip ise XML imlerine dayalıdır ve *XML açıklaması* olarak adlandırılır. (XML açıklamalarına *belgeleme açıklamaları* da denir.) Bir XML açıklamasının her satırı /// ile başlar. XML açıklamaları; sınıf, isim uzayı, metot, özellik ve olay gibi öğelerin deklarasyonlarından önce gelir. XML açıklamalarını kullanarak programlarındaki bilgileri programın içine gömebilirsiniz. Programınızı derlerken XML açıklamalarını bir XML dosyasının içine yerleştirebilirsınız. XML açıklamaları ayrıca Visual Studio'nun IntelliSense özelliği tarafından kullanılmaya da uygundur.

## XML Açıklama İmleri

C#, Tablo A.1'de gösterilen XML belgeleme imlerini destekler. XML açıklama imlerinin çoğunluğu kolayca anlaşılabilir niteliktedir. Bunlar, çoğu programcının zaten bildiği diğer XML imleri gibi işlev görürler. Ancak <list> imi diğerlerinden daha karmaşıktır. Bir listede iki bileşen bulunur: Liste başlığı ve liste elemanları. Bir liste başlığının genel şekli şöyledir:

```
<listheader>
    <term> isim </term>
    <description> metin </description>
</listheader>
```

Burada *metin*, listenin *isim*'ini tanımlar. Tablo için *metin* kullanılmaz. Bir liste elemanın genel şekli aşağıda gösterilmiştir:

```
<item>
    <term> eleman-ismi </term>
    <description> metin </description>
</item>
```

Burada *metin*, söz konusu *eleman-ismi*'ni belirtir. Noktalı ya da numaralı listelerde veya tablolarda *eleman-ismi* kullanılmaz. Bir listede birden fazla <item> ögesi bulunabilir.

**TABLO A.1: XML Açıklama İmleri**

**Etiket**

<c> *code* </c>

<code> *code* </code>

<example> *explanation* </example>

<exception cref = "name">  
*explanation*  
</exception>

**Açıklama**

*code* ile belirtilen metni program kodu olarak belirtir.

*code* ile belirtilen birden fazla satırdan oluşan metni program kodu olarak belirtir.

*explanation* ile ilintili metin, bir kod örneği tarif eder.

Kural dışı bir durumu tarif eder. Kural dışı durum *name* ile belirtilir.

<code>&lt;include file = 'fname' path = 'path [@tagName = "tagID"]' /&gt;</code>	Mevcut dosya için XML açıklamalarını içeren bir dosya belirtir. Dosya <b>fname</b> ile belirtilir. İm için dosya yolu, im ismi ve im kodu sırasıyla <b>path</b> , <b>tagName</b> ve <b>tagID</b> ile belirtilir.
<code>&lt;list type = "type"&gt;  &lt;/list&gt;</code>	<b>list-header</b> <b>list-items</b> Bir liste belirtir. Listenin tipi <b>type</b> ile belirtilir. <b>type</b> ya noktalı, numaralı ya da bir tablo olmalıdır.
<code>&lt;para&gt; text &lt;/para&gt;</code>	Bir başka im içinde bir paragraf uzunlığında metni belirtir.
<code>&lt;para name = 'param-name'&gt;  &lt;/param&gt;</code>	<b>explanation</b>  <b>paramname</b> ile belirtilen parametreyi belgeler. <b>explanation</b> ile ilintili metin parametreyi tarif eder.
<code>&lt;paramref name = "param-name" /&gt; &lt;permission cref = "identifier"&gt;  &lt;/permission&gt;</code>	<b>param-name</b> 'in bir parametre ismi olduğunu belirtir.  <b>explanation</b>  <b>identifier</b> ile belirtilen sınıf üyeleri ile ilintili izin ayarlarını tarif eder. <b>explanation</b> ile ilintili metin, izin ayarlarını gösterir.
<code>remarks&gt; explanation&lt; /remarks&gt;</code>	<b>explanation</b> ile belirtilen metin genel bir açıklamadır. Genellikle bir sınıf ya da yapı gibi bir tipi tarif etmek için kullanılır.
<code>returns&gt; explanation&lt; /returns&gt;</code>	<b>explanation</b> ile belirtilen metin, bir metodun dönüş değerini belgeler.
<code>&lt;see cref = "identifier" /&gt;</code>	<b>identifier</b> ile belirtilen bir başka elemana bir bağlantı deklare eder.
<code>&lt;seealso cref = "identifier" /&gt;</code>	<b>identifier</b> için “ayrıca bakınız” bağlantısı deklare eder.
<code>&lt;summary&gt; explanation&lt; /summary&gt;</code>	<b>explanation</b> ile belirtilen metin genel bir açıklamadır. Genellikle bir metot ya da bir başka sınıf üyesini tarif etmek için kullanılır.
<code>&lt;value&gt; explanation&lt; /value&gt;</code>	<b>explanation</b> ile belirtilen metin bir özelliği tarif eder.

## XML Belgelemesinin Derlenmesi

Belgeleme açıklamaları içeren bir XML dosyası üretmek için derleyicinin **/doc** seçeneğini belirtin. Örneğin, XML açıklamaları içeren ve adı **DocTest.cs** olan bir dosyayı derlemek için şu komut satırını kullanın:

```
csc DocTest.cs /doc:DocTest.xml
```

Visual Studio IDE kullanırken bir XML çıktı dosyası kullanmak için Property Pages iletişim kutusundan yararlanmanız gereklidir. Bu iletişim kutusuna View | Property Pages komutu

üzerinden ulaşabilirsiniz. Söz konusu menüye gittikten sonra Configuration Properties | Build seçeneğini işaretleyin. Ardından, XML'in adını, XML Documentation File özelliğinde belirtin.

## XML Belgeleme Örneği

Aşağıdaki örnek çeşitli XML açıklamalarının kullanımını göstermektedir:

```
// XML belgeleme ornegi.

using system;
/// <remark>
/// Bu bir XML belgeleme ornegidir.
/// Test sınıfı cesitli im orneklerini gosterir.
/// <remark>

class Test {
    /// <Summary>
    /// Programın calismasi Main'de baslar.
    /// </summary>
    public static void Main() {
        int sum;

        sum = Summation(5);
        Console.WriteLine("Summation of " + 5 + " is " + sum);
    }

    /// <summary>
    /// Summation, 1'den argumana kadar olan sayilarin
    /// toplamini verir.
    /// <param> name = "val"
    /// Toplanacak sayı, val paramteresinde gecilir.
    /// </param>
    /// <see> cref="int" </see>
    /// <returns>
    /// Toplam, bir int degeri olarak dondurulur.
    /// </returns>
    /// </summary>
    static int Summation(int val) {
        int result = 0;

        for(int i = 1; i <= val; i++)
            result += i;

        return result;
    }
}
```

Yukarıdaki programın **XmlTest.cs** olarak adlandırıldığını varsayırsak, aşağıdaki satırda yer alan ifade, bu programı derleyecek ve açıklamaların bulunduğu **XmlTest.xml** adlı bir dosya üretecektir:

```
csc XmlTest.cs /doc:XmlTest.xml
```

Derleme bittikten sonra aşağıdaki XML dosyası üretilir:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>t</name>
    </assembly>
    <members>
        <member name="T:Test">
            <remark>
                Bu bir XML belgeleme ornegidir.
                Test sinifi cesitli im orneklerini gosterir.
            </remark>
        </member>
        <member name="M:Test.Main">
            <summary>
                Programin calismasi Main' de baslar.
            </summary>
        </member>
        <member name="M:Test.Summation(System.Int32)">
            <summary>
                Summation, 1'den argumana kadar olan sayilarin
                toplamini verir.
            </summary>
            <param name="val">
                Toplanacak sayı, val paramteresinde gecilir.
            </param>
            <see cref="T:System.Int32" />
            <returns>
                Toplam, bir int degeri olarak dondurulur.
            </returns>
        </summary>
        </member>
    </members>
</doc>
```

Dikkat ederseniz, belgelenen her elemana benzersiz bir tanımlayıcı verilmiştir. Bu tanımlayıcılar, XML belgelemesi kullanan diğer programların kullanımına da açıktır.

E K B

B

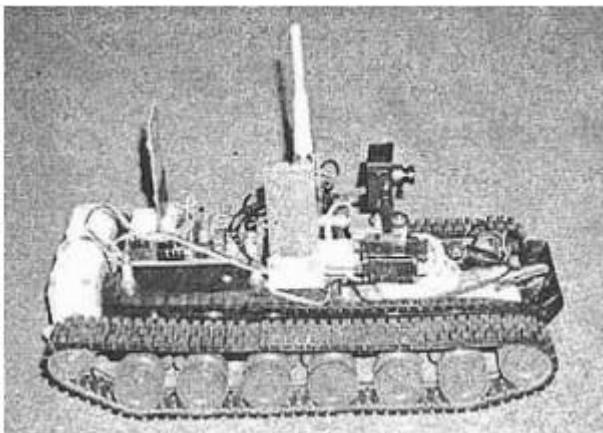
---

# C# VE ROBOTİK

Hakkında pek fazla yazmamış olsam da, robotik benim hobimdir. Aslında robotik uzun yıllar önce aynı zamanda benim işimdi. Endüstriyel robot kontrol dilleri tasarlar ve uygulardım. Robotlar heyecan vericidir, çünkü yazdığımız programların mantığını canlandırırlar. Robotlar aynı zamanda gerçek dünya ile etkileşirler. Uzun zamandır robotik ile profesyonel düzeyde aktif olarak ilgilenmem olmama rağmen, robotik her zaman ilgi alanına girmiştir. Robotik konusunu burada gündeme getirmemnin nedeni, C#'ın robotik programcısına bazı avantajlar sağladır.

Normal olarak robotik kontrol kodu dendiğinde C++'ta yazılan yüksek performanslı rutinler akla gelir. Ancak C# bu varsayımin sorgulanmasına neden olabilir. Çünkü robot kontrol programları gayet büyük olabilir; yönlendirme, görme, desen tanıma, motor kontrolü gibi pek çok alt sistem içerebilir. Bu alt sistemler C# bileşenleri olarak organize edilebilirler (ve uygulanabilirler). Robotik kodunu bileşen yaklaşımı ile yazmak, robot kontrolünde mevcut olan karmaşıklığı yönetmeye katkı sağlar. Bu sayede, ayrıca alt sistemlerin kolayca değiştirilmesi ve terfi ettirilmesi de mümkün olur.

Robotik konusu ile ilgileniyorsanız - özellikle de deney amaçlı olarak kendi robotunuzu yapmaya ilgi duyuyorsanız - Şekil B.I'deki robot ilginizi çekebilir. Bu, benim test robotumdur. Bu robottu ilginç kıلان birkaç özellik mevcuttur. Birincisi, robotta temel motor kontrolü ve algılayıcı geri beslemesi sağlayan dahili bir mikroişlemci yer almaktadır. İkinci olarak, ana bilgisayardan veri almak ve sonuçları ana bilgisayara geri iletmek üzere bir RS 232 alıcı/verici bulunmaktadır. Söz konusu yaklaşım sayesinde robotik için ihtiyaç duyulan işlem gücü, robottu ağırlaştırmadan uzaktaki bir bilgisayar tarafından sağlanabilmektedir. Üçüncü olarak, robotta kablosuz bir görüntü vericisine bağlı bir video kamera yer almaktadır.



**ŞEKİL B.1:** Basit ama aynı zamanda etkili deneysel bir robot.

Robot, bir Hobbico MI Abrams R/C tank şasisi üzerine inşa edilmiştir. (R/C model tankların ve otomobillerin şasisinin robot tabanı olarak çok iyi işe yaradığını gördüm.) Tankın iç sistemlerinin çoğunu söktüm. Alıcı ve hız kontrol devreleri de buna dahil. Ancak motorları bırakmadım. Hobico tank oldukça güçlü bir yapıya sahip olduğu için bir robotik platformu olarak oldukça uygundur. Motorlar iyi, oldukça fazla miktarda yük taşıyabilmekte ve paletleri

yerinden çıkmamaktadır. Paletli bir araç olduğu için olduğu yerde manevra yapabilmekte ve bozuk yüzeylerde de hareket edebilmektedir. Şasi 18 inç uzunluğunda, 8 inç enindedir.

Şasiyi boşalttıktan sonra yeni bileşenler ekledim. Dahili kontrol sistemi için bir BASIC Stamp 2 kullandım. Bu, Parallax Inc. ([www.parallaxinc.com](http://www.parallaxinc.com)) tarafından üretilen basit ama güçlü bir mikroişlemcidir. RS-232 alıcı/verici ile görüntü kamerası ve vericisi de yine Parallax'a ait. Kablosuz RS-232 alıcı/vericisi ve görüntü vericisinin menzili yaklaşık 300 ayak (100 m). Tank motorlarına ayrıca elektronik hız kontrolörleri ekledim. Yüksek performanslı R/C arabaları tarafından kullanılan türden olan bu kontrolörler BASIC Stamp mikroişlemciye bağlı olarak çalışmaktadır.

Robot şöyle çalışır: Uzaktaki bilgisayar ana robotik kontrol programını çalıştırır. Bu program; görüntü, yönlendirme ve uzaysal konumlandırma gibi tüm “ağır işleri” halleder. Ayrıca bir dizi hareketi öğrenebilir ve gerçekleştirebilir. Uzaktan kontrol bilgisayarı, hareket kontrol komutlarını (telsiz RS-232 bağlantısı üzerinden) robota gönderir. BASIC Stamp, bu komutları alır ve uygular. Örneğin bir “ileri hareket et” komutu gelmişse, BASIC Stamp, gerekli sinyalleri, motorlara bağlı elektronik hız kontrolörlerine gönderir. Robot bir komutu tamamladığı zaman alındı kodunu döndürür. Böylece uzaktan kontrol bilgisayarı ile robot arasında iki yönlü iletişim sağlanır ve her komutun başarıyla tamamlanıp tamamlanmadığı takip edilebilir.

Robotla ilgili ana işlemler uzaktan kontrol bilgisayarında yapıldığı için bu sistemin işlem gücü konusunda ciddi bir sınırlama bulunmamaktadır. Örneğin, bu yazının hazırlandığı sırada robot, görme sistemini kullanarak bir nesneyi izleyemekteydi. Bu yetenek ciddi bir işlem gücü gerektirir ve böyle bir işlem gücünü robot üzerinde taşımak zordur.

Bu yazının hazırlandığı sırada robotik kontrol kodunun çoğunluğu hala C++ dilinde yazılmış olarak duruyordu. Ancak, kodun parçalarını ilk firsatta C#'a geçirmeyi planlamaktayım. İlk dönüştüreceğim alt sistem, bir seri komutu çalıştırın alt sistem olacak. Liste tabanlı olan bu görev, bir C# koleksiyonu tarafından kolaylıkla ele alınabilir.