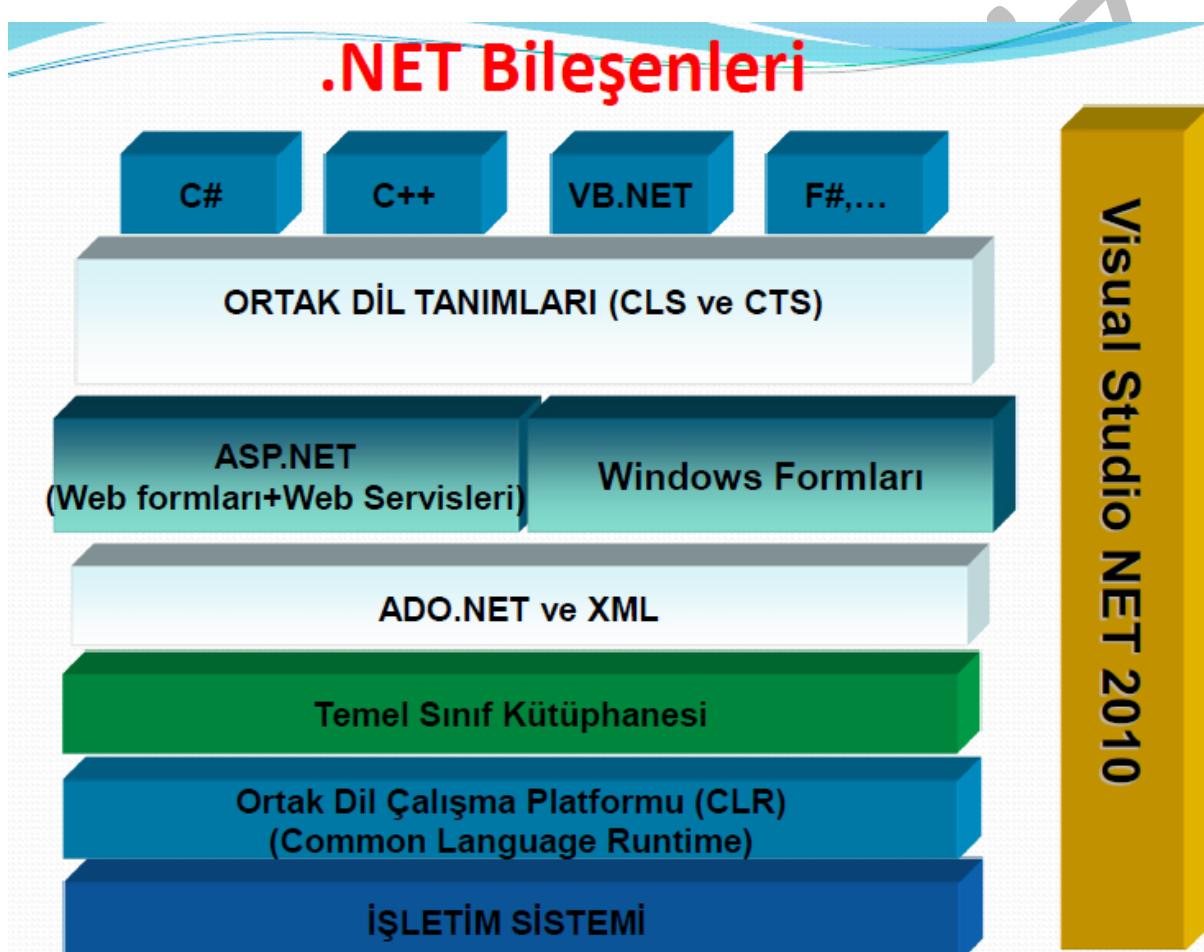


C#'da neler var

- Java dilinden farklı olarak C# dilinde işaretçiler (**pointer**) kullanılabilmektedir.



B'a'

Stack: Çalışma zamanındaki atamalar ram'in bu bölgesine yapılır.

Not: Derleyici hangi değişkene ne kadar yer tahsis edilceğini bilmelir. [bu yüzden değişken tipleri vardır. Buna göre stack'da yer ayrılır.]

Heap alanı: Tüm C# nesneleri bu bölgededir.

Not: Bu bölgede bir şey tanımlamak için new yapılmalıdır. Fakat çalışma zamanında tanımlanır ve stack bölgesine yerleştirmeye oranla daha yavaştır.

Not: Değer türündekiler stack bölgesine yerleşir. Referans türündekiler heap bölgesine yerleşir.

Register Bölgesi:

Mikroişlemcideki özel bölgedir diğer hafıza birimlerine oranla çok daha hızlı çalışırlar.

Static Bölge:

Belleğin herhangi bir bölgesinde saklanabilir. Fakat static alandaki veriler programın çalışma zamanı boyunca saklanır.

Sabit Bölge:

Programın içinde hiç değişmeyen sabitlerin saklandığı bölümdür.

Ram olmayan bölge:

Bellek bölgesinin dışındaki disk alanlarıdır.

TEMELLER

- C#’da değişken isimlendirmede büyük ve küçük harf duyarlılığı vardır.
- Değişken isimleri nümerik bir karakter ile başlayamaz.
- Değişken isimlerinde boşluk karakteri olamaz.

57

SABİTLER

- **const double pi=3.14**

sabitler const ile tanımlanır.

sabitler kendilerinden static oldukları için static yazılmaz.

sabitler sabitlere atanırlar yani

`const x=4`

`const y=7+x`

fakat eğer x const olmasaydı hata olurdu

veya çalışma zamanında değer atanmaya çalışılsaydı hata oluşurdu.

Yani sabitlere program yazılrken değer atanmalı ve sonra değiştirilmemelidir. Aksi takdirde hatayla karşılaşılır.

DEĞER VE REFERANS TİPLERİ

Değer:

Değişken değeri özel olarak bellek bölgesinden alınır

Referans:

Başka bir nesnenin erişilen özelliklerini taşıyan kopyasıdır. New ile referans tipinde değişken oluşturulabilir.

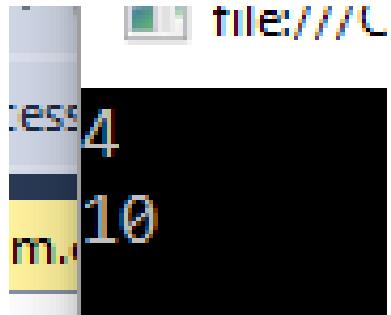
Not:

- Değerler stack bölgesinde edilir.
 - Referanslar heap bölgesinde edilir
-
- İki değer tipi eşitlenince saklanan değerler kopyalanır.
 - Fakat iki referans tipi eşitlendiğinde adres eşitlenir.

diziler int nesnesi olarak referans tipinde new ile tanımlanımlanır.

yani mesela 2 dizi eşitlenince dizilerin adresi eşitlenir bir dizinin bir elemanı değişince değerinin de o elemanı değişir.

```
static void Main(string[] args)
{
    int[] dizi1=new int[10];
    int[] dizi2;
    dizi1[5] = 4;
    dizi2 = dizi1;
    Console.WriteLine(dizi2[5]);
    Console.WriteLine(dizi2.Length);
    Console.ReadLine();
}
```



dizi2'nin ne boyutu ne de elemanları tanımlanmadığı halde dizi1'e eşitlenince aynı adresde bulundukları için dizi1 ile ilgili her değişiklik dizi2'e yapılır. Bu durum tam tersi içinde geçerlidir.

Nesneler içinde bu durum tam olarak geçerlidir.

NOT:

```
class nesne
{
    public int a;
    public string b;
}
```

nesnelerin değişkenlerine **referans ile erişilmek isteniyorsa değişkenler public olmalıdır**

Bir nesne tanımladığımızda new ile tanımlamasak içindeki değişkenlere default değer atanmaz o yüzden

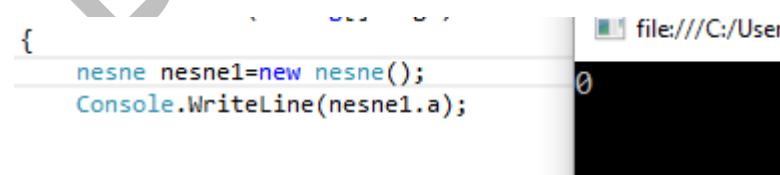
```
nesne nesne1;
Console.WriteLine(nesne1.a);
```

burada hata vardır nesne.a'ya bir değer atanılmamıştır.nesne sınıfında a=new int() olsa bile hata alınır hatta nesne sınıfında a=3 gibi bir değer atansa da yine hata alınır

```
static void Main(string[] args)
{
    nesne nesne1;
    nesne1.a = 0;
    Console.WriteLine(nesne1.a);
```

burada da nesne1.a'ya değer atandığı halde yine hata alınır.Yani ne olursa olsun nesneler eğer kendi tipinden başka bir nesneye atanmayıacaklarsa new ile tanımlanmalıdırlar

fakat eğer nesne1=new nesne() yapılsaydı içindeki değişkenlerle beraber default değer atanırdı nesneye

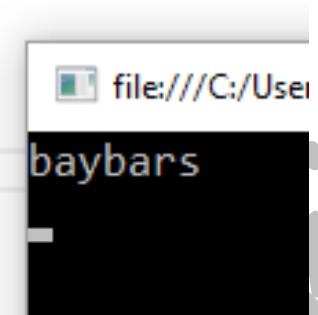


burada herhangi bir hataya karşılaşılmaz.

burada hiçbir şey yapılmamasına rağmen sadece nesne new ile yazılarak nesne'nin erişilen(public olan) tüm değişkenlerine default değer atanmış olur.

Referansların eşitlenmesi

```
using System;
static void Main(string[] args)
{
    nesne nesne1=new nesne();
    nesne nesne2;
    nesne2 = nesne1;
    nesne1.b = "baybars";
    Console.WriteLine(nesne2.b);
    Console.ReadLine();
}
```

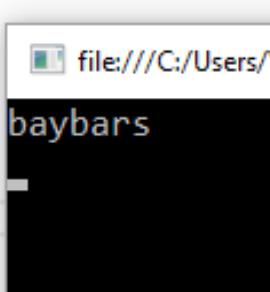


Burada nesne2 new ile tanımlanmadığı halde bir sorun oluşmuyor.

Bunun dışında nesne2=nesne1 yapıldığında nesnelerin adresleri eşitlendiği için artık içindeki değişkenlerde aynı adresdedir bu yüzden artık ne nesne1'den değiştirilen b değişkeni nesne2'ninde b'sinin olduğu adresde bulunur.

Tam tersinde de bu kural geçerliliğini koruyor. Çünkü artık ne olursa olsun iki nesne bellek de aynı adrese sahiptir birinde yapılan değişiklik diğeride geçerlidir.

```
using System;
static void Main(string[] args)
{
    nesne nesne1=new nesne();
    nesne nesne2;
    nesne2 = nesne1;
    nesne2.b = "baybars";
    Console.WriteLine(nesne1.b);
    Console.ReadLine();
}
```



DİKKAT:

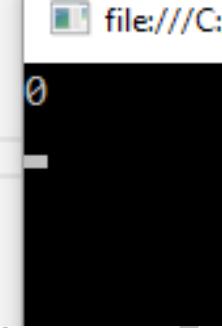
Bu gibi bir durumda eğer eşitliğin sağ tarafındaki new ile tanımlı değilse aşağıdaki gibi hata alınır.

```
nesne nesne1=new nesne();
nesne nesne2;
nesne1 = nesne2;
nesne2.b = "baybars";
Console.WriteLine(nesne1.b);
Console.ReadLine();
```

Sebebi:Nesne2'nin new ile tanımlı olmamasıdır.

New ile oluşturulursalar bile temel veri tipleri birbirlerine eşitlendikleri durumda adresleri de eşitlenmez.

[tamamen new ile eşitlenme durumu içindir.Temel veri türlerinde adreslemenin yapılabildiği durumlar C#'da mevcut olabilir.Fakat new ile tanımlı olmak bu kapsamda girmez.]



```
int a = new int();
int b = new int();
a = b;
b = 4;
Console.WriteLine(a);
Console.ReadLine();
```

Burada da new ile tanımlı olmayan bir temel veri tipi new ile tanımlı olan veri tipine atanmaya çalışılırsa hata ile karşılaşılır.

```
int a = new int();
int b;
a = b;
b = 4;
Console.WriteLine(a);
Console.ReadLine();
```

burada b new ile tanımlı olmadığı halde,new ile tanımlı olan a'ya atanmaya çalışıldığı için hatayla karşılaşılmıştır.

Fakat burada referans türündeki nesnelerde olduğu gibi herhangi bir sorunla karşılaşılmaz.

```
int a = new int();
int b;
b= a;
```

Aslında new default değer atama için denilebilir.

Değeri olmayan değişkenler birbirine atanamaz bu yüzden hata alınır.

Son görselde a default değeri olan 0'a tanımlandığı için b=a durumunda b=0 olmuş olur.

Not:

```
do
{
    i++;
} while (i<7);
```

while'de bu noktalı virgülü unutma

```
string a;
do
{
    a=Console.ReadLine();
} while (a!="baybars"&&a!="ekiz");
Console.WriteLine(a);
Console.ReadLine();
```

```
fsdaf
dfçd
fef
fvf
ekiz
ekiz
```

a baybars'a ve ekiz'e eşit olmadığı şartı sağlanıyorsa dön demektir.

Değer Tipleri

Tüm değer tipleri object nesnesinden türemiştir.

int a=3,b;

tanımlamaların bu şekilde olması mümkündür.

| C# Tipi | .NET Framework | Tanım | Değer Aralığı |
|---------|----------------|----------------------------------|--|
| object | System.Object | Tüm CTS türleri için temel sınıf | - |
| bool | System.Boolean | Mantıksal Doğru/Yanlış | true ya da false |
| byte | System.Byte | 8 bit işaretsiz tamsayı | 0 ~ 255 |
| sbyte | System.SByte | 8 bit işaretli tamsayı | 128 ~ 127 |
| char | System.Char | Karakterleri temsil eder | 16 Unicode karakterleri |
| decimal | System.Decimal | 128 bit ondalıklı sayı | $\pm 1,5 \cdot 10^{-28} \sim \pm 7,9 \cdot 10^{28}$ |
| double | System.Double | 64 bit çift kayan sayı | $\pm 5 \cdot 10^{-324} \sim \pm 1,7 \cdot 10^{308}$ |
| float | System.Single | 32 bit tek kayan sayı | $\pm 1,5 \cdot 10^{-45} \sim \pm 3,4 \cdot 10^{38}$ |
| int | System.Int32 | 32 bit işaretli tamsayı | -2.147.483.648 ~ 2.147.483.647 |
| uint | System.UInt32 | 32 bit işaretsiz tamsayı | 0 ~ 4.294.967.295 |
| long | System.Int64 | 64 bit işaretli tamsayı | 9.223.372.036.854.775.808 ~ -9.223.372.036.854.775.807 |
| ulong | System.UInt64 | 64 bit işaretsiz tamsayı | 0 ~ 18.446.744.073.709.551.615 |
| short | System.Int16 | 16 bit işaretli tamsayı | -32.768 ~ 32.767 |
| ushort | System.UInt16 | 16 bit işaretsiz tamsayı | 0 ~ 65.535 |
| string | System.String | Karakter Dizisi | Unicode Karakter Dizisi |

Tüm değer tipleri bunlardır ve hepsi object nesnesinden türemiştir.

```
int aint = new int(); 0
double adoub= new double(); 0
char acar = new char(); 0
decimal adec = new decimal(); 0
byte abayt = new byte(); 0
bool abul = new bool(); False
Console.WriteLine(aint);
Console.WriteLine(adoub);
Console.WriteLine(acar);
Console.WriteLine(deadec);
Console.WriteLine(abayt);
Console.WriteLine(abul);
Console.ReadLine();
```

yapılardırıcılı olarak tanımlanınca bazı değişkenlerin ilk değerleri bu şekilde dir.

Referans Tipleri

Önceden tanımlı 2 tane referans tipi vardır.

Object:

Tüm veri tiplerinin türedeği sınıfıdır.

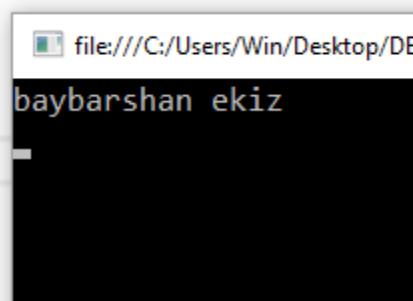
Tüm değişkenler object türüne atanabilir.

String:

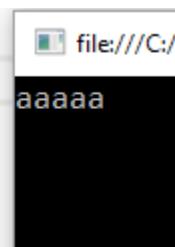
Unicode karakterlerden oluşan dizi denilebilir

Stringleri arka arkaya eklemek için '+' kullanılır

```
string a="baybarshan";
string b="ekiz";
string c = a + " " + b;
Console.WriteLine(c);
Console.ReadLine();
```



```
string a=new string('a',5);
Console.WriteLine(a);
Console.ReadLine();
```



Stringlerle yol belirtmek için iki yol vardır

1. \\ kullanmak

• **String path="C:\\WINDOWS\\assembly"**

2. string ifadenin tırnaklarından önce @ kullanmak

• **String path=@“C:\\WINDOWS\\assembly”**

string Değişkenler stack'de oluşturulur heap'de saklanır stack'da heap adresleri bulunur.

Aşağıda görüldüğü gibi bir object değişkene her türlü tip atanabiliyor

```
using System;
public class varsayılan_değerler
{
    static void Main()
    {
        object x;
        x=10;
        Console.WriteLine(x.GetType());
        x="B";
        Console.WriteLine(x.GetType());
        x=8.78F;
        Console.WriteLine(x.GetType());
        x=false;
        Console.WriteLine(x.GetType());
        x=5.489M;
        Console.WriteLine(x.GetType());
    }
}
```

Not:

x.GetType() ➔ x'in tipini öğrenir

```
object o;
o = false ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.Boolean

```
object o;
o = 32.807 ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.Double

```
object o;
o = 32 ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.Int32

```
object o;
o = "aadfgss" ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.String

```
object o;
o = 'a' ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.Char

```
object o;
o = "a" ;
Console.WriteLine(o.GetType());
Console.ReadLine();
```

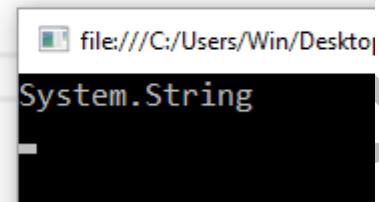
file:///C:/Users/Win/Desktop/ConsoleApp1.cs(4,17): System.String

not:char ifadeler 'd' şeklinde gösterilmelidir

Var değişken yapısı:

Object tipindeki bir değişkene program boyunca her türlü tipteki değer rahatlıkla atanabilir.

```
object o;  
o = 21;  
o = false;  
o = "fadsf";  
Console.WriteLine(o.GetType());  
Console.ReadLine();
```



Var tipinde ise değişkenin tipi gelen değere göre şekillenir ve artık değiştirilemez. Eğer bir kere değer tipi atandıktan sonra başka tipte bir değer atanılınca çalışmayıorsa hata ile karşılaşılır.

bu hatadır

```
var o;  
o = 21;  
Console.WriteLine(o.GetType());  
Console.ReadLine();  
  
var o= 21;  
o=false ;  
Console.WriteLine(o.GetType());  
Console.ReadLine();
```

buda hatadır

bir kere değer direk olarak o anda atanılmalıdır.

```
var o=new var(); buda hatadır
```

```
var vr_Degisken = "string değer"; Çalışır
```

Erkan hoca tavsiye notu:

- Not: var değişken tipi, diller arası, databasesler arası entegrasyonu sağlarken veri tipleri uyumazlığını gidermek için oluşturulmuş bir tiptir. Yani C#’ta int ile tanımlanan bir değişken Delphi ’de başka türlü tanımlanabilir. var değişken tipide bütün dillerde evrensellik özelliği taşımaktadır. Sizlere tavsiyem normal kodlamada verinizin tipine göre normal değişken tanımlayınız.

81

Dynamic değişken yapısı:

Objecte çok benzer fakat static değildir çalışma zamanında değiştirir veri tipini object tüm tiplerin içinde bulunduğu bir tip olduğu için asıl tipe çevirmek için unboxing yapılması gereklidir.

Fakat dynamic tiplerde böyle bir şeye gerek yoktur.

```
object Rakam = 10;  
Rakam = Rakam + 10; hata vardır
```

Aşağıdaki gibi unboxing işlemi yapılmalıdır.

Çünkü tipi belirsizdir. Object belli bir tip değildir her tipi içinde bulundurabilir.

Uyumsuzluğun giderilmesi için unboxing işlemiyle int tipinde olduğu belirtilmelidir.

```
Rakam = (int)Rakam + 10;
```

Fakat dynamic'de böyle bir durum yoktur bu tip bit uyum sorunu olsa bile kendi giderir.

```
dynamic sayı = 5;  
sayı =sayı+ 5;
```

SORUN YOK

Çünkü sayı artık int tipindedir 5'e atanırken sayı tipi int olarak ayarlanmıştır.

Sonradan Sayı farklı bir tipten degere atansa bile sayı değişkeni o tipe dönüşür.

Object tipiyse başlı başına bir tiptir.Tüm tipleri kapsadığı için çok yer kaplar ve 20 kat daha yavaş çalışabilir.

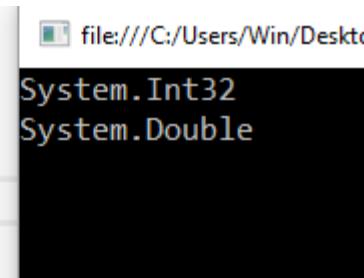
Dynamic'in işiyse sadece çalışma zamanında değişkene doğru tipi vermektedir.

Tür dönüşümleri:

Bilinçsiz(implicit) tür dönüşümleri:

Bir değerin farklı tipte bir değere sorunsuz bir şekilde çevrilmesidir.

```
int a = 43;
double b = a;
Console.WriteLine(a.GetType());
Console.WriteLine(b.GetType());
Console.ReadLine();
```



Sorun çıkmamıştır fakat bu bilinçsiz(implicit) tür dönüşümüdür.

Küçük türün büyük türü dönüştürülmesi:

- Herhangi bir sorun yaşanmadan gerçekleşir.
- Veri kaybı yaşanmaz.
- Yeni bitler 0 olur

```
int a = 89;
double b = a;
Console.WriteLine(b);
Console.ReadLine();
```



| Tür | Bilinçsiz Dönüşüme İzin verilen Türler |
|----------------|---|
| sbyte | byte, ushort, uint, ulong, veya char |
| byte | sbyte veya char |
| short | sbyte, byte, ushort, uint, ulong, veya char |
| ushort | sbyte, byte, short, veya char |
| int | sbyte, byte, short, ushort, uint, ulong, veya char |
| uint | sbyte, byte, short, ushort, int, veya char |
| long | sbyte, byte, short, ushort, int, uint, ulong, veya char |
| ulong | sbyte, byte, short, ushort, int, uint, long, veya char |
| char | sbyte, byte, veya short |
| float | sbyte, byte, short, ushort, int, uint, long, ulong, char, veya decimal |
| double | sbyte, byte, short, ushort, int, uint, long, ulong, char, float, veya decimal |
| decimal | sbyte, byte, short, ushort, int, uint, long, ulong, char, float, veya double |

Not:

float tipinde bir değişken tanımlarken değerin sonuna f yazılmmalıdır. Aksi takdirde hata ile karşılaşılır.

`float b =86.75f;`

Değer dönüşümlerinin şu şekilde bir kısıtı vardır. Bu şekildeki tip dönüşümlerine müsaade edilmez

Bool, decimal ve double türünden herhangi bir türü

Herhangi bir türden char türüne

Float ve decimal türünden herhangi bir türü
(float türünden double türüne dönüşüm hariç)

Bilinçli(explicit) Tür Dönüşümleri

(Büyük türün küçük türüne çevrilmesi):

C# bu tip tür dönüşümlerinin yapılmasını yasaklamıştır.

Eğer yapılmak isteniyorsa cast yani (değer tipi) işlemi yapılmalıdır.

Aslında cast yapılarak hem küçük tür büyük türü hem büyük tür küçük türü çevrilebilir. Fakat küçük türün büyük türü dönüştürülmesinde herhangi bir sorun ile karşılaşılmaz

Sorun genelde izin verilmeye dönüşümlerde ve büyük türün küçük türü dönüştürülmesi durumunda gerçekleşir.

Bu sorun ise cast operatörü yardımıyla giderilir.

Mantığı şu şekildedir:

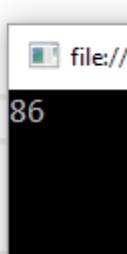
(dönüştürülecek tür) değişken_yada_sabit

```
double b =86;  
int a = b;  
Console.WriteLine(a);  
Console.ReadLine();
```

hata vardır.

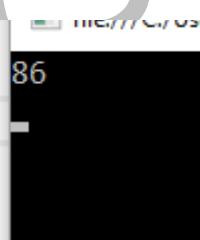
Hata aşağıdaki şekilde giderilir.

```
double b =86;  
int a = (int)b;  
Console.WriteLine(a);  
Console.ReadLine();
```



not:Aşağıdaki 3 örnekdede küçük tür büyük tür'e dönüştürüldüğü için sorunla karşılaşılmaz.

```
int b =86;  
double a = b;  
Console.WriteLine(a);  
Console.ReadLine();
```



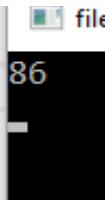
SORUN YOK

```
int b =86;  
double a = (double)b;  
Console.WriteLine(a);  
Console.ReadLine();
```



SORUN YOK

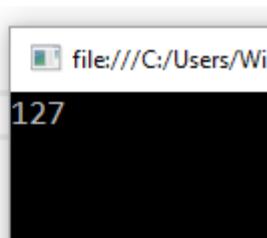
```
int b =86;  
double a = (int)b;  
Console.WriteLine(a);  
Console.ReadLine();
```



SORUN YOK

Aşağıdaki durumda ise hata yoktur fakat veri kaybıyla karşılaşılır.

```
int b = 895;  
byte a = (byte)b;  
Console.WriteLine(a);  
Console.ReadLine();
```



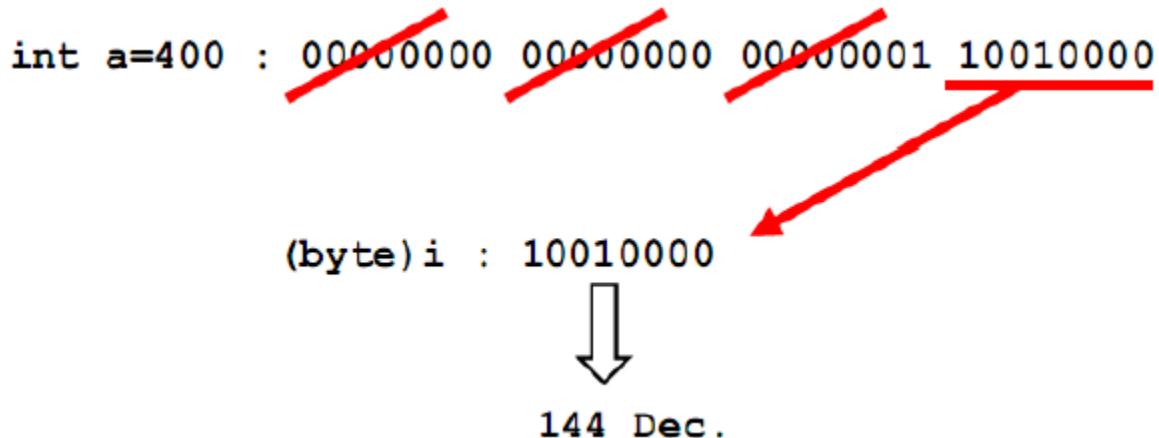
```
file:///C:/Users/Wi...  
127
```

Veri kaybını hesaplamak için tiplerin kaç bit olduğu bilinmelidir.

int:32 bit

byte:8 bit

Dönüşüm ise şu şekildedir:

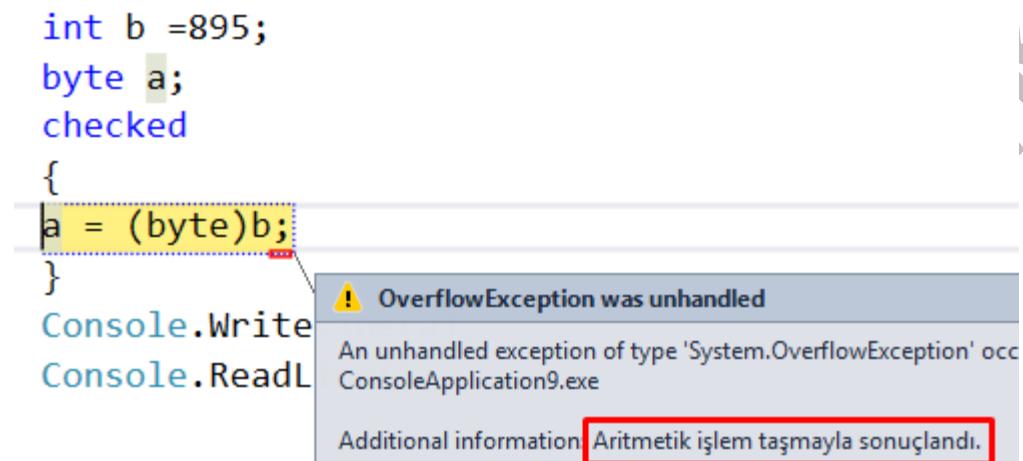


Bu durum taşıma olarakda adlandırılır.

Checked ve Unchecked

Checked:

Checked bloğunun içindeki atamalarda herhangi bir veri kaybının yaşanıp yaşanmadığının kontrolünün yapılmasını sağlar. Eğer yaşanıyorsa çalışma zamanında hata ile karşılaşılır.



A screenshot of a C# code editor showing a checked conversion error. The code is as follows:

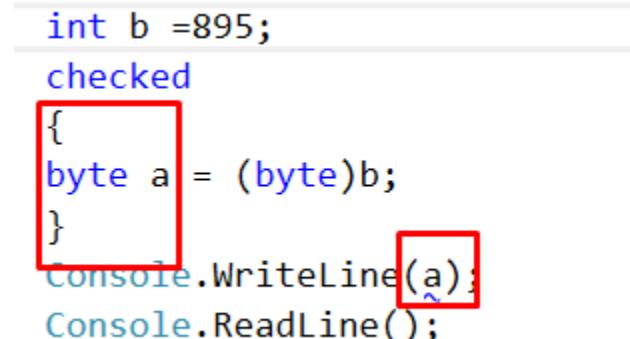
```
int b = 895;
byte a;
checked
{
    a = (byte)b;
}
Console.WriteLine(a);
Console.ReadLine();
```

The line `a = (byte)b;` is highlighted with a yellow selection bar. A tooltip window appears with the following details:

- ⚠ OverflowException was unhandled
- An unhandled exception of type 'System.OverflowException' occurred in ConsoleApplication9.exe
- Additional information: Aritmetik işlem taşmayla sonuçlandı.

Görüldüğü gibi veri kaybından(taşmadan) kaynaklı çalışma zamanı hatasıyla karşılaşılmıştır.

Dikkat edilmelidir ki: checkedin blok aralığında tanımlama yapılmamalıdır aksi takdirde bu durum derleme zamanı hatası ile sonuçlanır.



A screenshot of a C# code editor showing a checked conversion error with red boxes highlighting specific parts of the code. The code is identical to the one above, but the first two lines are enclosed in a red box, and the line `Console.WriteLine(a);` is also enclosed in a red box.

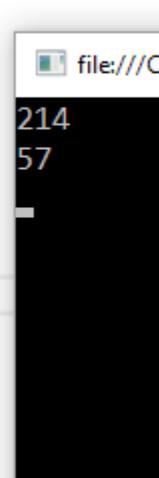
```
int b = 895;
checked
{
    byte a = (byte)b;
}
Console.WriteLine(a);
Console.ReadLine();
```

Unchecked:

Checkedin içindeki kontrolü istenmeyen kısımlar unchecked bloğuna alınır. Aslında çok mantıksız gibi gözükebilir fakat uzun checked bloklarında faydalı olabilmektedir.

```
int a = 214, b=825;
byte x, y;

checked
{
    x = (byte)a;
}
unchecked
{
    y = (byte)b;
}
Console.WriteLine(x);
Console.WriteLine(y);
```



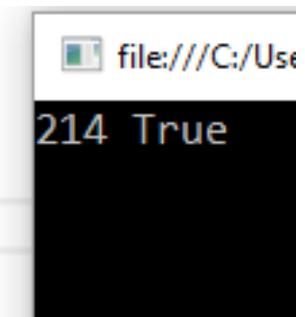
```
file:///C
214
57
```

REFERANS VE DEĞER TÜRLERİ ARASINDAKİ DÖNÜŞÜM

ToString() Kullanımı:

Herhangi bir tip string türüne dönüştürülmek isteniyorsa [stringde bir referans türüdür] ToString() kullanılır.

```
int a = 214;
bool b = true;
string x = a.ToString()+" "+b.ToString();
Console.WriteLine(x);
Console.ReadLine();
```



BOXİNG Kavramı

Referans ve değer tipleri arasındaki dönüşümüdür

- Referans tipleri bizim sınıflarımız, object ve string olarak düşünülebilir
- Değer tipleride int, bool, char, double v.s.'dir
- Bu ikisi arasında yapılan tür dönüşümleri boxing'dir

```
int a = 214;
object i = a;
```

Bu örnek en basit örneklerdendir. Kutulama gibi düşünülebilir. Kutulanan bir şeyin ne olduğu artık açılana kadar(unboxing) bilinemez.

Aşağıdada boxing işleminin bilinçli bir şekilde yapıldığını görüyoruz

```
int i=50; //değer tipi  
object o=(object) i; //boxing [bilinçli boxing]
```

UNBOXİNG Kavramı

Kutudaki bir şeyi çıkartmak denilebilir. Artık kutudan çıkan şeyin ne olduğu bilinir.

Yani boxing'in tam tersidir.

Referans türler bu işlem sayesinde doğru türe dönüşmüş olur.

```
int a = 214; → BOXİNG  
object i = a;
```

```
int k = (int)i; → UNBOXİNG
```

YANI TEMEL MANTIK:

BOXİNG: KÜÇÜĞÜ BÜYÜĞE ATMADIR.

UNBOXİNG: BÜYÜĞÜ KÜÇÜĞE KOYMADIR [CAST GEREKİR]

.Convert Sınıfı

String değerlerini ve temel veri türlerini birbirlerine çevirmek için kullanılır her veri sınıfının kendine özgü çevrim fonksiyonu mevcuttur.

Aşağıda veri str adlı string türünün temel veri türlerine çevrimi yapılıyor.

```
Convert.ToBoolean(str)  
Convert.ToByte(str)  
Convert.ToInt32(str)  
Convert.ToChar(str)
```

DİKKAT:

```
byte b=Convert.ToInt32(a); //Yanlış tür dönüşümü
```

Float'lar Parse edilirler

```
string c="12.34";  
a=float.Parse(c);
```

Aşağıdaki gibi durumlarda hata sebebidir

```
char a = 'a';  
bool x=Convert.ToBoolean(a);
```

InvalidCastException was unhandled

An unhandled exception of type 'System.InvalidCastException' occurred in mscorelib.dll

Additional information: 'Char' ile 'Boolean' arasında geçersiz atama.

```
int a=0; int d = (int) 6.0; //float -> integer dönüşüm
object k="merhaba"+15; //object türü, hem karakter hem sayısal
float b=10.5f; //float tanımı
double c=20.1; //double tanımı
Double dd = new double(); //referans olarak double tanımı
const double pi = 3.14; //sabit tanımı
string[] isimler ={ "Ozlem","Nesrin", "Ozge", "Fulya" }; //string dizi tanımı
object[] isim ={ "Ozlem","Nesrin", "Ozge", "Fulya" }; //object dizi tanımı
string s = "true"; //string tanımı
string dd="12.45f";
b= float.Parse(dd); //string tip float'a çevriliyor
b=Convert.ToSingle(dd); //String float'a çevriliyor
a =Convert.ToInt32(b + c); //float -> integer
bool cevap = (Convert.ToBoolean(s)); //boolean tanımı
Console.WriteLine((float)a/d+"\n"); // () operatörü ile float dönüşümü
Console.WriteLine("cevap=" + cevap); // cevap = true yazar
Console.WriteLine(k.GetType()); //bulunduğu sınıf,alanadını verir.
a = Convert.ToSingle(Console.ReadLine()); //girilen değer float'a çevriliyor
Console.WriteLine("a={0} b= {1} c={2} d={3} ", a, b, c,d);
if (isimler[0].Equals("Ozlem")==true) //eğer dizinin ilk elemanı Ozlem ise yazar
    Console.WriteLine("birinci isim Ozlem");
foreach (string ss in isimler) // string dizi içindeki her bir eleman yazdırılıyor
{ Console.WriteLine(ss); }
```

Baybars

OPERATÖRLER

- $x=10 \% 3 \rightarrow x=1$

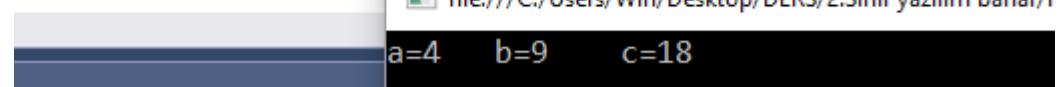
$b=a++ \rightarrow a'$ yı önce ata sonra arttır

$b=++a \rightarrow a'$ yı önce arttır sonra ata

'-' içinde aynısı geçerlidir

NOT: Console.WriteLine'ın kullanımıyla ilgili:

```
Console.WriteLine("a={0}    b={1}    c={2}",4,9,18);
Console.ReadLine();
```



Karşılaştırma Operatörleri:

İki değeri birbirleriyle karşılaştırıp bool bir değere ulaşırlar:

| | | | |
|-----|----------|----|---------------|
| = = | Eşittir | != | Eşit Değildir |
| < | Küçüktür | >= | Büyük Eşittir |
| > | Büyüktür | <= | Küçük Eşittir |

as operatörü:

Bir türü referans tipindeki bir tipe dönüştürür.

```
object i="50";
string s=i as string;
```

is operatörü:

Bir değerin türünün kontrolünü yapar.Bool değeri döndürür.

- int i=50;
- bool b1=i is int

Çalışma sonucu “true” değerini verir.

~ Bitsel değil operatörü

Tek operand alan ~ operatörü bir değer içindeki bitlerin teker teker tersini alır.

0000 1111 sayısının bitsel değili 1111 0000 olur.

```
using System;
class Operatorler
{
    static void Main()
    {
        byte b1 = 254;
        byte b2 = (byte)~b1;
        Console.WriteLine(b2);
    }
}
```

Bitsel Sola ve Sağa Kaydırma

Bit tabanlı operatörler byte tipindeki değişkenler için kullanılır genelde ve kısa devre analizi yapılmaz yani 2 tarafada bakılır. Bit tabanlı operatörler genelde 2 deðilde tek defada kullanılır mesela || olan or operatörü | şeklindedir.

ifade << öteleme sayısı

```
byte b = 0xFF          // 1111 1111 (255)
```

```
byte c = (byte)(b<<4) // 1111 0000
```

burada b=255'dir ve b değeri 4 bit sola kaydırılmıştır.

Özel Amaçlı Operatörler:

- ▶ ? : (**Ternary**) Operatörü:
 - ▶ Bir koşulu karşılaştırıp doğru ya da yanlış olduğu durumlar için farklı değerler üretmeyi sağlayan operatördür. Üç tane operanda sahiptir. Kullanımı:
 - ▶ koşul ? doğru değeri : yanlış değeri
 - ▶ sayı < 10 ? "10dan küçük" : "10 a eşit ya da büyük"

() Operatörü (Tür Dönüşümü):

- İfade önüne yazıldığında tür dönüştürme operatörü olarak çalışır

[] Operatörü (İndeks):

- Dizilerde elemanın indeksini belirtmek için kullanılır

+ , - (İşaret Operatörleri):

- Bir değişkenin içeriğindeki değerin negatif ya da pozitif olarak işlem görmesini sağlar. + aynı zamanda stringler de ekleme için de kullanılan bir operatördür.

& , * , -> ve sizeof Operatörleri:

- C/C++ dilindeki işaretçileri (pointer) C# dilinde de kullanmak mümkündür. Güvensiz (unsafe) kod yazarken bu operatörler de kullanılabilir.



• new Operatörü:

- Yeni bir nesne oluşturmak için kullanılan operatördür.
- Değer ve referans tiplerden yeni bir nesne oluşturmak için kullanılabilir
- Örnek : Sinif s = new Sinif();

• typeof Operatörü:

- Çalışma zamanında bir tür için System.Type sınıfı türünden bir nesne üretmesini sağlar. Nesneler için kullanılan **GetType()** ifadesinin benzer işlemini tür ve ya sınıf tanımlamaları için gerçekleştirir

```
using System;

class Operatorler13
{
    static void Main()
    {
        Type t;
        int i = 123;
        float f = 23.45F;

        t = typeof(int);
        Console.WriteLine(t.ToString() + "   ");
        Console.WriteLine(i.GetType());

        t = typeof(float);
        Console.WriteLine(t.ToString() + "   ");
        Console.WriteLine(f.GetType());
    }
}
```

Burada type tipindeki değişken typeof(değişkentipi) ile sonradan bir daha bir daha tanımlanabiliyor

KOŞUL İFADELERİ:

bu bölümde bilinen ama unutulabilir olanlar var

switch Deyimi

- ▶ Her case ifadesinin bir atlama (**break, goto, return, continue**) ifadesi ile sonlandırılmalıdır.
- ▶ **continue** deyimi switch ifadesi bir döngünün gövdesinde ise kullanılabilir.
- ▶ Bir case içinde hiçbir ifade verilmemezse atlama deyimi kullanılması gerekmek.
 - ▶ case 2: // ifade yok
 - ▶ case 3:
 - ▶ case 4: ifade....;break;
 - ▶ case 5: ifade...; goto case 4;

switch ifadesi kullanırken dikkat edilecek bazı kurallar vardır:

- case sözcüğünden sonra gelen ifadeler sabit olmak zorundadır.
- case ifadeleri tamsayı, karakter ya da string sabitler olabilir.
- default ve case ifadeleri istenilen sırada yazılabilir.
- Aynı switch bloğu içerisinde birden fazla aynı case ifadesi bulunamaz.
- default kullanmak zorunlu değildir.
- Akışı bir case ifadesinden bir başka case ifadesine yönlendirmek için mutlaka **goto** anahtar sözcüğü kullanılır.

DÖNGÜLER:

- for
- while
- do while
- foreach

foreach hariç diğerleri yüzeysel yapılacaktır.

NOT: Her döngü türünde koşul doğru olduğu sürece devam edilir [koşul true sonucu verdiği sürece]

Foreach Yapısı:

Nesnelerin elemanlarına tek tek ulaşılması sağlanır.

Fakat unutulmamalıdır ki ulaşılan elemanlar sadece okunabildirler.

- string[] isimler ={ "Ozlem","Nesrin", "Ozge", "Fulya" };
//string dizi tanımı

foreach (string ss in isimler) // elemanlar yazdırılıyor

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

- foreach (int i in dizi) // i sadece okunur(readonly)
{ Console.WriteLine(i);
/ i++; */ //yasak kullanılamaz* } → dikkat
}

NOT:

Jump (atlama) Deyimleri)

- **break:** Çalışan bir döngüden “break” sözcüğü kullanılarak çıkışılabilir. Program akışı döngüden sonraki satırlardan devam eder. break sözcüğü sadece döngü ve switch ifadelerinde kullanılabilir.
- **continue:** Döngünün bir sonraki tekrarına geçilmesini sağlar.
- **goto:** Programın etiket ile belirlenmiş herhangi bir kısmına atlamak için kullanılır. Nesneye yönelik programlamaya uygun bir yapı değildir. Switch ifadesindeki kullanımı dışında mümkün olduğunca kullanılmaktan kaçınılmalıdır.
- **return :** Metotların herhangi bir anda sonlandırılması için kullanılır. Metot sonlandırıldıktan sonra programın akışı metodu çağrıran fonksiyondan devam eder.

Baybarsı

.Random sınıfı:

Aşağıdaki gibi bir referans yoluyla kullanılır.

Random rnd=new Random();

Aşağıdaki gibi kullanımları vardır[kırmızı kutu içindekilere dikkat]

- **int rastgelesayı =rnd.Next(10,20);**
 - 10 ile 20 arasında bir sayı üretir. (10 dahil 20 hariç)
- **int rastgelesayı =rnd.Next(50);**
 - 0 ile 50 arasında bir sayı üretir.
- **int rastgelesayı =rnd.Next();**
 - Herhangi bir rasgele sayı üretir.
- **double rastgelesayı =rnd.NextDouble();**
 - 0.0 ile 1 arasında bir sayı üretir. (1 hariç)

Diziler(arrays)

Kullanımları:

```
int[] dizi = new int[19];
```

veya

```
int[] dizi;  
dizi = new int[19];
```

- **new** anahtar sözcüğü ile, dizinin her elemanına temel veri türleri için varsayılan değer, ilk değer olarak verilmektedir.
- Bu ilk değer; referans türleri için null, nümerik türler için 0, bool türü için ise false'tır.

```
string[] dizi1={"Bilgisayar","Mühendisliği","Bölümü"};  
int [] dizi2= {5,9,12,56,23};  
float [] dizi3={8f,39f,324f,23f,2f};
```

yer ayırmaktadır. Ancak C# 'ta ise diziler referans tipi olduğu için dizi boyutları çalışma zamanında belirlenebilir.

Bir dizinin boyutu bir kez belirlendikten sonra artık değiştirilemez. Yani dizinin boyutunu dinamik olarak değiştirmemiz mümkün değildir.

```

static void Main()
{
    int k = 0;
    int[] dizi = new int[10];
    //diziye ait nesne oluşturuluyor
    k = dizi.Length;
    Console.WriteLine("Dizinin Uzunluğu : " + k);
}

```



```

foreach(int i in dizi)
{
    Console.WriteLine(i);
}

```

not:aslında foreach'da verilen veri tipinden o referansta kaç tane bulunuyorsa tek tek çekiliyor mantığı bu şekildedir

foreach'da referansın içinde o yapıdan bulunması yeterlidir bu bizim oluşturduğumuz nesneler içinde geçerlidir.Fakat şimdilik char'lardan oluşan string nesnesi için inceliycek olursak

```

string a = "huloo";
foreach (char x in a)
{
    Console.WriteLine(x);
}
Console.ReadLine();

```

h
u
l
o
o

bu şekilde bir çıktı elde etmiş oluruz.

ÇOK BOYUTLU DİZİLER

DÜZENLİÇOK BOYUTLU DİZİLER[MATRİSLER]

- Çok boyutlu dizi tanımı için tanımlama esnasında [] içine verilecek boyut kadar ',' değeri eklenir. "[, ,]" şeklinde yazılır.



- int [,] dizi = {{1,2},{3,4},{5,6}};
- Şeklinde tanımlanan bir 3x2'lik dizinin elemanları:

dizi[0,0]=1
dizi[0,1]=2
dizi[1,0]=3
dizi[1,1]=4
dizi[2,0]=5
dizi[2,1]=6

| dizi | |
|-------|-------|
| [0,0] | [0,1] |
| [1,0] | [1,1] |
| [2,0] | [2,1] |

- int[,] dizi1=new int[2,3];

dizi1[0,0]=1;

dizi1[2,1]=7;

```
int[,] dizi = new int[4, 4];
```

Gibi uygulamaları vardır.

DÜZENSİZÇOK BOYUTLU DİZİLER[JAGGED ARRAYS]

| | | | | |
|------------|------------|------------|------------|------------|
| dizi[0][0] | dizi[0][1] | dizi[0][2] | dizi[0][3] | dizi[0][4] |
| dizi[1][0] | dizi[1][1] | dizi[1][2] | | |
| dizi[2][0] | dizi[2][1] | dizi[2][2] | dizi[2][3] | |

Satırlar düzenli olarak belli bir sayı şeklinde verilir

Fakat sütünler verilmez düzensiz bir şekilde girilen değere göre satırların sütunları birbirinden farklı boyutlardadır.

Aşağıdaki yapı sayesinde bu sağlanır.

3 satırı olan ve sütün değerleri sonradan belirlenen bir dizi tanımlaması şu şekildedir.

```
int [][] dizi= new int[3][];  
dizi[0]=new int[5];  
dizi[1]=new int[3];  
dizi[2]=new int[4];
```

Length özelliği bir dizideki eleman sayısını verir.

Dizinin satır ve sütun sayıları ise şu şekilde alınır

```
Console.WriteLine(dizi.GetLength(0)); // dizinin satır sayısı  
Console.WriteLine(dizi[2].GetLength(0));//2.satırdaki sütun  
sayısı
```

| | |
|--------------------|--|
| IsFixedSize | Dizinin eleman sayısının sabit olup olmadığını verir. (Boolean) |
| IsReadOnly | Dizideki elemanların sadece okunabilir olup olmadığını verir. (Boolean) |
| Length | Dizideki eleman sayısını verir. |
| Rank | Dizinin boyutunu verir. |

| | |
|-----------------------|--|
| BinarySearch | Tek boyutlu dizide binary search algoritmasına göre arama yapar. |
| Clear | Dizinin elemanlarını varsayılan değere çeker. |
| Clone | Dizinin bit bit kopyasını çıkarır. |
| Copy | Dizinin bir bölümünü başka bir diziye kopyalar. |
| CopyTo | Bir dizinin belirlenen bir kısmını başka bir diziye kopyalar. |
| Length | Dizideki eleman sayısını verir. |
| GetValue | Dizideki ilgili eleman değerini verir. |
| IndexOf | Dizi içindeki bir değerin ilk görüldüğü indeksi verir. |
| Reverse | Diziyi tersine çevirir. |
| SetValue | Bir dizinin bir elemanına değer atar. |
| Sort | Bir boyutlu dizilerde sıralama yapar. |
| CreateInstance | Yeni bir dizi nesnesi oluşturur. |
| GetLength | Dizinin i. boyuttundaki değerinin verir. |
| GetUpperBound | Dizinin i. boyuttaki son indis değerini verir. |
| Resize | Mevcut dizinin elaman sayısını değiştirme. |

73

- **GetLength:** Dizinin i. boyuttaki değerini verir.
- **GetUpperBound:** Dizinin i. boyuttaki son indis değerini verir.
- - `int[,] dizi = new int[5,4,3];`
 - `int x = dizi.GetLength(0); → x=5`
 - `int y = dizi.GetLength(1); → y=4`
 - `int z = dizi.GetLength(2); → z=3`

Aşağıda **hatalı yazım var Get length**
bölümünde GetUpperBound olucak
yani en son değerin indisini kaç demektir

```
int X = dizi.GetLength(0); → X=4  
int Y = dizi.GetLength(1); → Y=3  
int Z = dizi.GetLength(2); → Z=2
```



- **GetValue:** Metodu belirtilen indekslerdeki elemanın değerini verir.
- **int[] dizi = { 1, 22, 34 };**

```
Console.WriteLine(dizi.GetValue(2)); // 34 değerini verir
```

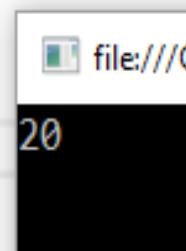
```
Console.WriteLine(dizi[2]); // 34 değerini verir
```

- **SetValue:** Metodu ilgili dizinin belirtilen indekslerindeki elemanın değerini değiştirmek için kullanılır.

- **dizi.SetValue(20,0); // (değer, 1.index)**
- **dizi.SetValue(20,0,0); // (değer, 1.index, 2.index)**



```
object a;  
int[,] dz = new int[4,5];  
dz[2, 3] = 25;  
dz.SetValue(20, 2, 3);  
Console.WriteLine(dz.GetValue(2,3)); 20  
Console.ReadLine();
```



- Çalışma zamanında dizinin tipinin belirlenmesi için kullanılır. CreateInstance metodu ile aşağıdaki gibi dizi nesnesi oluşturulabilir:
 - `Array dizi=Array.CreateInstance(typeof(int),5);`
 - 5 elemanlı int türünden bir dizi tanımlanır.
- İlk parametre her zaman Type türünden olmalıdır. Yukarıdaki gibi bir kullanımda, `typeof()` operatörü yardımıyla int türünün type sınıfındaki karşılığı elde edilmiş olmaktadır. Örn:
 - `CreateInstance(Type,int,int,int);`
 - 3 boyutlu bir dizi oluşturulmakta.
 - `CreateInstance(Type,int[]);`
Çok boyutlu diziler oluşturmak için kullanılır. Çok boyutlu dizilerin her birinin kaç elemanlı olduğunu belirlemek için int türden bir dizi metoda parametre olarak gönderilir.
 - `Array dizi=Array.CreateInstance(typeof(int),3,5);`
 - int tipinde 2 boyutlu bir dizi tanımlanır.

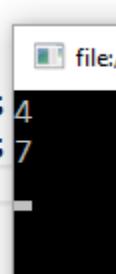
```

static void Main()
{
    Array dizi = Array.CreateInstance(typeof(int), 5, 4, 3);
    Random r = new Random();

    for (int i=0; i< dizi.GetLength(0); i++)
        for (int j=0; j< dizi.GetLength(1); j++)
            for (int k = 0; k < dizi.GetLength(2); k++)
            {
                dizi.SetValue(r.Next(10, 100), i, j, k);
                Console.WriteLine("dizi[{0},{1},{2}]={3,3}",
                                  i,j,k,dizi.GetValue(i,j,k));
            }
}
  
```

burada 3 boyutlu bir dizi oluşturulup tipide `typeof(tip)` ile belirtilmiştir.

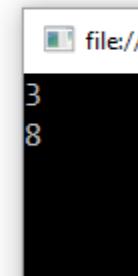
```
object a;  
int[,] dz = new int[4, 7];  
Console.WriteLine(dz.GetLength(0));  
Console.WriteLine(dz.GetLength(1));  
Console.ReadLine();
```



RESİZE

Aşağıda dizinin boyutu değiştiriliyor ve dizinin olan elemanları zarara uğramıyor. Not Resize sanırım sadece tek boyutlu dizilerde geçerli.

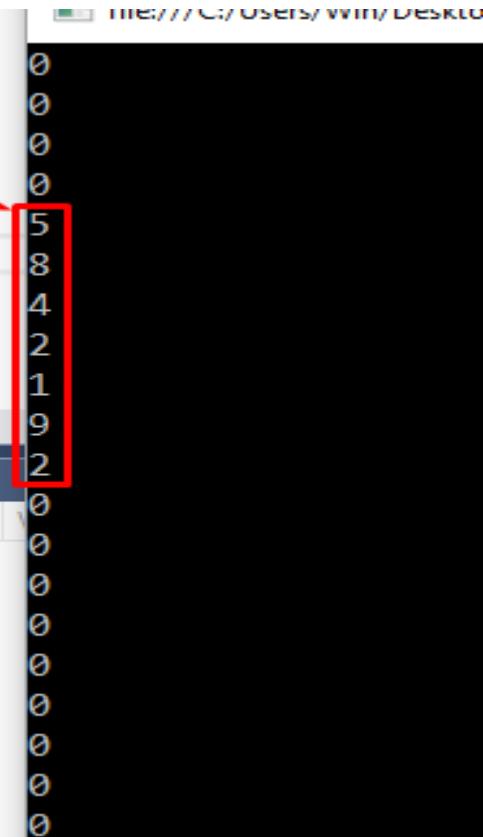
```
object a;  
int[] dz = new int[4];  
dz[3] = 3;  
Array.Resize(ref dz, 5);  
dz[4] = 8;  
Console.WriteLine(dz.GetValue(3));  
Console.WriteLine(dz.GetValue(4));  
Console.ReadLine();
```



CopyTo

Dizinin belirli bir yerinden sonrasında başka bir dizinin elemanlarını kopyalar

```
int[] dz = {5,8,4,2,1,9,2};  
int[] dz2 = new int[20];  
dz.CopyTo(dz2, 4);  
foreach (int i in dz2)  
{  
    Console.WriteLine(i);  
}  
  
Console.ReadLine();
```



- CopyTo metodu ile bir dizinin tamamı, başka bir dizinin istenilen yerine kopyalanabilir. Örn:
 - **int[] dizi1= {1,2,3,4,5,6,7};**
 - **int dizi2=new int[10];**
 - **dizi1. CopyTo(dizi2,3);**
 - dizi1'in elemanlarını dizi2'nin 3. indisinden itibaren kopyalanır.

Array.Copy

CopyTo'ya göre çok daha pratik ve mantıklıdır. iki dizidende belirtilen indexlerden itibaren kaç adedinin kopyalanacağı belirtilir.

```
int[] dz = {5,8,4,2,1,9,2};  
int[] dz2 = new int[20];  
Array.Copy(dz, 2, dz2, 8, 4);  
foreach (int i in dz2)  
{  
    Console.WriteLine(i);  
}  
  
Console.ReadLine();
```

4
2
1
9

**NOT:DİKKAT EDİLMELİDİR Kİ DEĞERLER ELDE BULUNAN DEĞERLERİ
GECMEMELİ.**

```
int[] dz = {5,8,4,2,1,9,2};  
int[] dz2 = new int[20];  
Array.Copy(dz, 4, dz2, 8, 4);  
foreach (int i in dz2)  
{  
    Console.WriteLine(i);  
}  
  
Console.ReadLine();
```

ArgumentException was unhandled

An unhandled exception of type 'System.ArgumentException' occurred in mscorlib.dll

Additional information: Kaynak dizi yeterince uzun değildi. srclIndex ve uzunluk ile dizinin alt sınırlarını denetleyin.

Troubleshooting tips:

[Get general help for this exception.](#)

Burada dz'nin 4. indis 1'dir ondan sonra sonra 4 indis daha bulunmadığı için çalışma zamanı hatasıyla karşılaşılmıştır.

Sort

Dizilerin küçükten büyüğe sıralamaya yarar. Not: String dizisiyle alfabetik olarak a öncelikli sıralar.

```
int[] dz = {5,8,4,2,1,9,2};  
Array.Sort(dz);  
foreach (int i in dz)  
{  
    Console.WriteLine(i);  
}
```

| |
|---|
| 1 |
| 2 |
| 2 |
| 4 |
| 5 |
| 8 |
| 9 |

Binary Search

Dizi içinde nesneyi arar varsa o nesnenin bulunduğu indis değerini geri döndürür yoksa negatif sayı döndürür [döndürülen negatif sayı herhangi bir negatif sayı olabiliyor] [bu deneme türü pek de sağlamlı değil eleman dizide olduğu halde negatif değer dönebiliyor].

```
int[] dz = {5,8,4,2,1,9,2};  
Console.WriteLine(Array.BinarySearch(dz,2));  
Console.ReadLine();
```

| |
|----------------------------|
| file:///C:/Users/Win/[...] |
| 3 |

BinarySearch aracılığıyla belli bir indis'den itibaren belirli bir aralık aranıla bilir.

BinarySearch(Array dizi, int baslangic, int uzunluk, object nesne);

```
int[] dz = {5, 8, 6, 4, 1, 9, 2};  
Console.WriteLine(Array.BinarySearch(dz, 2, 3));  
Console.ReadLine();  
}  
3
```

başlangıç
indisi
3 eleman
kadar ara
4 elemanını
ara

Clear

Belirtilen elemandan itibaren belirtilen uzunluk kadar eleman 0 olur.

```
int[] dz = {5, 8, 6, 4, 1, 9, 2};  
Array.Clear(dz, 1, 3);  
foreach (int i in dz)  
{  
    Console.WriteLine(i);  
}  
5  
0  
0  
0  
1  
9  
2
```

- `Array.Clear(dizi,1,3);`// 1. elemandan itibaren 3 elemanı sıfır yapar.
- `Array.Clear(dizi,0,2);`// Sıfırıncı elemandan itibaren 2 elemanı sıfır yapar.

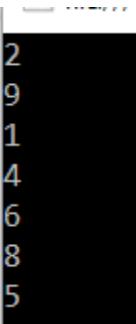
tümünü sıfırlamakda bu şekilde mümkündür

```
int[] dz = {5, 8, 6, 4, 1, 9, 2};  
Array.Clear(dz,0,dz.Length);  
foreach (int i in dz)  
{  
    Console.WriteLine(i);  
}  
0  
0  
0  
0  
0  
0  
0
```

Reverse

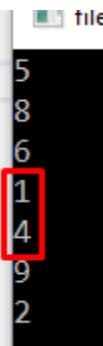
Tüm diziyi veya belirtilen aralığı ters çevirir.

```
int[] dz = {5,8,6,4,1,9,2};  
Array.Reverse(dz);  
foreach (int i in dz)  
{  
    Console.WriteLine(i);  
}  
Console.ReadLine();
```



burada tümü çevrildi

```
int[] dz = {5,8,6,4,1,9,2};  
Array.Reverse(dz,3,2);  
foreach (int i in dz)  
{  
    Console.WriteLine(i);  
}  
Console.ReadLine();
```



burada da belirtilen aralık ters çevrildi

NOT:

Unutulmamalıdır ki diziler birbirlerine eşitlendiğinde aslında adresleri eşitleniyor ve bu demek oluyor ki artık dizilerin birinde yapılan tüm işlemler diğerine de yansıyacaktır.

var ve dynamic değişkenleri ile dizi tanımlamak.

- **var**, gelen verinin değerine göre türü belirlenir ve bellekte ona göre yer ayrıılır. Aynı değişkene farklı türden değer atanamaz.
- **dynamic**, derleme zamanında gerçekleştirilen tip kontrollerinin pas geçilerek, bu kontrollerin tamamen çalışma zamanında gerçekleştirilmemesidir.
 - `var dizi_x = new[] { 2005, 2010, 2015, 2020 }; //Int32`
 - `var[] dizi_y = new var[3]; //Hatalı tanımlama Derleme zamanı hatası`
 - `var dizi_z = new[,] { { "x","y" }, { "m","n" } }; //string`
- `dynamic [] d_x = new dynamic[3];`
- `d_x[0] = 1; //Int32`
- `d_x[1] = "x"; //string`
- `d_x[2] = true; //bool`

ArrayList Sınıfı ve Özellikleri

- ArrayList, dinamik olarak büyüp küçülen, farklı değişken türlerini ve nesneleri depolayabilen koleksiyon tabanlı bir sınıfır.
 - İlk başlangıçta eleman sayısı belirtilmediği zaman (`ArrayList dizi = new ArrayList()`) boyut 2 ile başlar elemanlar eklendikçe ikinin katları şeklinde kendini artırır (`2,4,8,16,32...` gibi). Başlangıçta boyut belirtilirse `ArrayList dizi = new ArrayList(3);` boyut aşıldığında 2'nin katları şeklinde artar.
- ArrayList, dizilerin eksiklerini gidermek ve kısıtlamalarını ortadan kaldırmak için oluşturulmuş bir sınıfır.
- **Dizi ve ArrayList Arasındaki Farklar;**
 - Diziler sabit uzunlukta tanımlanırlar, oluşturulduktan sonra büyüp, küçültülemezler. ArrayList'de ise böyle bir kısıtlama yoktur. ArrayList'e yeni elemanlar eklendikçe boyutu otomatik olarak artırılır, elemanlar silindikçe ise boyutu azaltılır.
 - **Diziler tanımlanırken tutacağı elemanların türünün belirtilmesi gereklidir.** Bu tür dışındaki elemanların dizi içerisine eklenmesi mümkün olmaz. **ArrayList'te ise farklı türden değişkenler ve nesneler aynı koleksiyon içerisinde saklanabilir.**

ArrayList'deki metodlar

Add:

```
ArrayList baglilikst=new ArrayList();
int a=3;
float b=3.8f;
string c="ram";
bool d=true;
baglilikst.Add(4);
baglilikst.Add(false);
baglilikst.Add(a);
baglilikst.Add(b);
baglilikst.Add(c);
baglilikst.Add(d);
```

herhangi bir sorunla karşılaşılmamıştır. Burada baglilikst için 6 eleman olmasına rağmen 8 elemanlık yer ayrılmıştır hafızada bunun sebebi alanın her defasında 2 katına çıkmasıdır.

Remove:

Belirtilen elemanı listeden çıkartır.

```
ArrayList baglilikst=new ArrayList();
int a=3;
float b=3.8f;
string c="ram";
bool d=true;
baglilikst.Add(4);
baglilikst.Add(false);
baglilikst.Add(a);
baglilikst.Add(b);
baglilikst.Add(c);
baglilikst.Add(d);
baglilikst.Remove(a);
```

burada a baglilikstden çıkarılmıştır.

ArrayList İçerisindeki Verilere Erişmek

Yukarıdaki örnekte farklı veri tiplerini bir ArrayList içeresine eklenebilir. Pratikte böyle bir kullanım hem performans hem de algoritmik bakımdan bir çok problem yaratacağı için, genellikle benzer veri türlerinin ArrayList içerisinde saklanması daha uygundur. ArrayList içerisindeki verilere, aynı dizilerdeki gibi indis numarasıyla erişim sağlanabilir. Yada tüm ArrayList bir for-each döngüsü yardımıyla listelenebilir.

```
• class Program
• { static void Main(string[] args)
• {   ArrayList liste = new ArrayList();
•     liste.Add("Murat");
•     liste.Add("Yeşim");
•     liste.Add("Deniz");
•     Console.WriteLine(liste[2]);
•     foreach (String eleman in liste)
•     {   Console.WriteLine(eleman); }
•   }
• }
```

ayrıca farklı elemanları içeren bir arraylist'de foreach int'ler için yapılrsa hata ile karşılaşılır çünkü o anda int dışındada elemanlar içerebilmektedir

Aşağıda kırmızı kutu içerisinde olanlar dizerler içinde geçerlidir.

ArrayList Sınıfı ve Özellikleri

- **ArrayList Sınıfı İçerisinde Kullanılan Diğer Önemli Metodlar**
- **Count:** ArrayList içerisindeki toplam eleman sayısını int türünde döndürür.
- **AddRange:** Diğer koleksiyon tabanlı nesneleri ya da dizileri ArrayList içeresine aktarır.
- **Sort:** Listedeki elemanları sıralar. İstenirse sadece belirli indeks numarasından sonraki elemanlarda sıralanabilir.
- **Reverse:** Tüm listeyi tersine çevirir. Yani listedeki birinci eleman sonuncu, sonuncu eleman birinci olacak şekilde tüm liste tersine döndürülür.
- **Clear:** Tüm ArrayList içerisindeki elemanları siler.
- **BinarySearch:** Liste içerisinde parametre olarak verilen değerin bulunup bulunmadığını arar. Bulunursa indis numarasını döndürür, bulamazsa negatif bir değer döndürür. BinarySearch ile arama yapmak için öncelikle listenin sıralanması gereklidir.
- **CopyTo:** Tüm liste ya da listenin bir bölümünün başka bir ArrayList'e ya da diziye kopyalanmasını sağlar.

METOTLAR

Metotların genel yapısı:

```
[erişim] <dönüş değeri> metot ismi (parametre listesi)
{
    metot gövdesi
}
```

Örn:

```
int MetotAdı(int a,int b)
{ return a+b; }
```

Metodun erişimi belirtilmediği takdirde private olarak kabul edilir ki bu da sadece bulunduğu sınıfta kullanılacak demektir.

Metoda ulaşmak için bazı durumlar dışında '.' operatörü kullanılır .

nesnereferansı.metot gibi bir kullanımı vardır.

```
Metotlar nesne = new Metotlar();
int a = nesne.Topla(2, 5);
```

Static olan bir metod eğer kendi sınıfında çağrılıyorsa direk olarak referansa gerek kalmadan çağrılabılır.

```
0 references
class Mavi
{
    1 reference
    int metodum()
    {
        int a = 4;
        return a;
    }
0 references
    static void Main(string[] args)
    {
        int x = metodum();
```

görüldüğü gibi normal şartlarda hata ile karşılaşılmakta fakat static yazıp incelersek:

```
1 reference
class Mavi
{
    1 reference
    static int metodum()
    {
        int a = 4;
        return a;
    }
    0 references
    static void Main(string[] args)
    {

        int x = metodum();
    }
}
```

Hiçbir hatayla karşılaşılmadı.

Mesela aynı sınıf içinde **Mavi.metodum()**; şeklinde de erişilebilir ama bu sınıf üzerinden metoda erişim şekli aynı sınıfın içinde fazlalık gibi olacağı için bu şekildeki bir kullanım, metodun başka sınıfından çağrılmaması için kullanılabilir.

Fakat burada da unutulmamalıdır ki eğer başka sınıfın metoda buradaki gibi referanssız direkt sınıf üzerinden erişmek istiyorsak erişmek istediğimiz metodun sadece **static** olması yeterli olmuyacaktır aynı zamanda **public** olmalıdır.

Aşağıdaki örnekte görüldüğü gibi beyaz'daki *metodum*'a **static** olmasına rağmen erişim yoktur .

```
1 reference
class beyaz
{
    0 references
    static int metodum()
    {
        int a = 4;
        return a;
    }
}
0 references
class Mavi
{
    0 references
    static void Main(string[] args)
    {
        beyaz.
    }
}
```

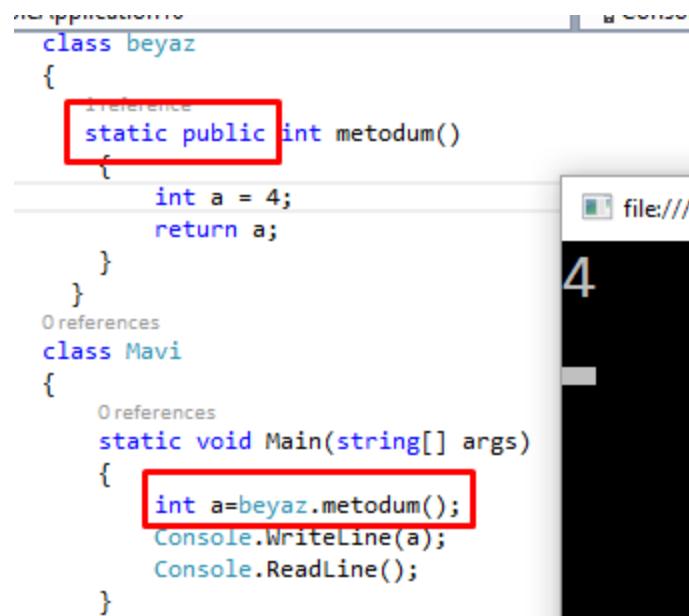
The screenshot shows a code editor with the following code:

```
1 reference
class beyaz
{
    0 references
    static int metodum()
    {
        int a = 4;
        return a;
    }
}
0 references
class Mavi
{
    0 references
    static void Main(string[] args)
    {
        beyaz.
    }
}
```

The cursor is at the end of the word "beyaz." in the last line. A tooltip is displayed with two options: "Equals" and "ReferenceEquals".

Bu yüzdede başka bir class'dan referanssız bir şekilde erişilecek metodun sadece static olması yetmez aynı zamanda public'de olmalıdır.

(Kalıtım v.b. durumlar için bu durum geçerli olmayabilir. Erişimlerin nasıl olduğu kalıtım konusunda ayrıca işlenilcektir.)



```
class beyaz
{
    static public int metodum()
    {
        int a = 4;
        return a;
    }
}

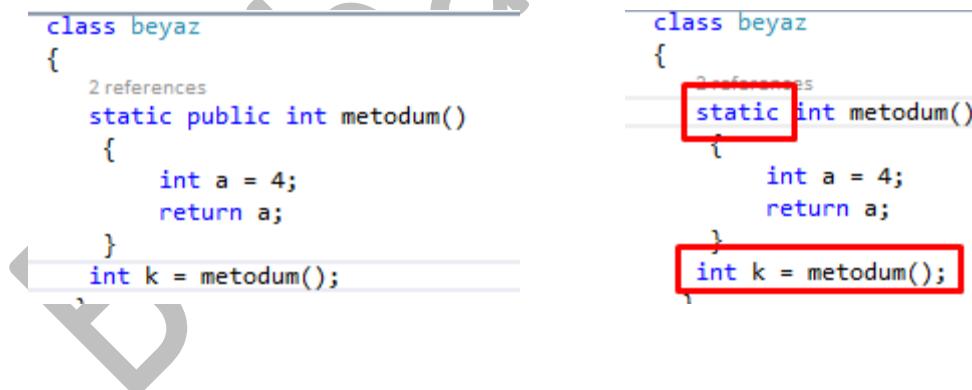
class Mavi
{
    static void Main(string[] args)
    {
        int a=beyaz.metodum();
        Console.WriteLine(a);
        Console.ReadLine();
    }
}
```



Görüldüğü gibi hiçbir sorunlar karşılaşılmamıştır.

Aynı metot içinde de **public** olmasa da direk olarak erişilir.

Aşağıdaki iki türlüde de hiçbir sorunla karşılaşılmaz



```
class beyaz
{
    static public int metodum()
    {
        int a = 4;
        return a;
    }
}

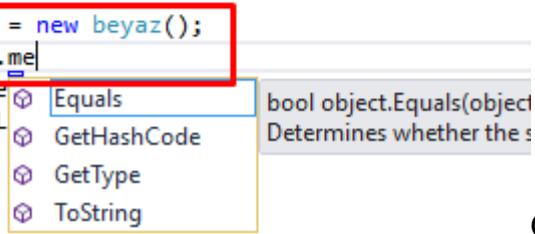
class Mavi
{
    static void Main(string[] args)
    {
        int k = metodum();
    }
}
```

Metot eğer public değilse başka bir sınıfta kullanılamaz ki bu demektir ki eğer public erişim belirleyicisini yazmazsak metot direk olarak private olacağı için diğer sınıflardan erişim sağlanamamış olacaktır.

(Yine kalıtım v.b. durumlarda bu konuya ilgili istisnadan bahsedilcektir.)

Aşağıda görüldüğü gibi referansla bile metoda erişilemiyor

```
class beyaz
{
    1 reference
    □ int metodum()
    {
        int a = 4;
        return a;
    }
    int k = metodum();
}
0 references
class Mavi
{
    0 references
    static void Main(string[] args)
    {
        beyaz refer1 = new beyaz();
        int a=refer1.me|  
        console.WriteLine(a);
        Console.ReadLine();
    }
}
```



Görüldüğü gibi erişim

yoktur. Fakat aşağıda erişim belirleyicisi public olduğundan dolayı herhangi bir sorunla karşılaşılmamıştır.

```
class beyaz
{
    1 reference
    □ public int metodum()
    {
        int a = 4;
        return a;
    }
}
0 references
class Mavi
{
    0 references
    static void Main(string[] args)
    {
        beyaz refer1 = new beyaz();
        int a = refer1.metodum();
        Console.WriteLine(a);
        Console.ReadLine();
    }
}
```



UNUTULMAMALIDIR Kİ:Eğer metot static değilse kendi sınıfında bile referansla çağrılmalıdır.

Not: Main'de bir metottur fakat dikkat edilmelidir ki tüm programda bir tane Main vardır bir tane daha olamaz. Main şu şekilde tanımlanır.

0 references
static void Main(string[] args)
{

Return Anahtar Sözcüğü:

Metodlar belirli bir tipte geri dönüş tipiyle tanımlanır ve tüm işlemler tamamlandığında return ile tanımlanan tipte geri dönüş yapılır.

```
beyaz refer1 = new beyaz();
int x = refer1.metodum();
```

metot çağrııldı

```
public int metodum()
{
    int a = 4;
    return a;
}
```

metoda gelindi ve geriye a değeri döndürüldü.

geri dönüş illa temel veri tipi olmak zorunda değildir. Bizim nesnelerimizden biride olabilir.

```
class Beyaz
{
    public int k; //DİKKAT:public ile tanımlandı diğer sınıflarda başka türlü kullanılamaz(bazı durumlar dışında)
}
class Mavi
{
    reference beyaz metot2()
    {
        beyaz referans=new beyaz();
        referans.k=9;
        return referans;
    }
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        beyaz refer2 = refer1.metot2();
        Console.WriteLine(refer2.k);
    }
}
```

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlar

9

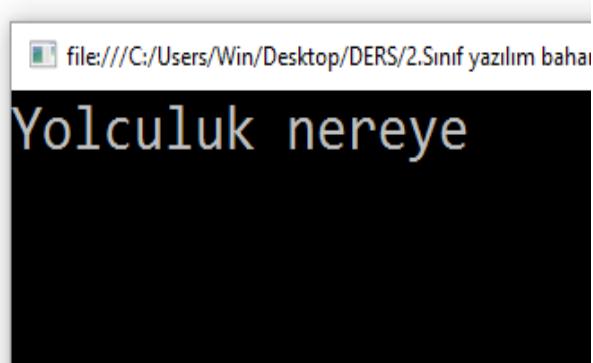
Fakat eğer geri dönüş yapılmayacak ise metot void şeklinde tanımlanılmadır.

Eğer void dışında bir tiple tanımlandığı halde geri dönüş yapılmıyorsa hata ile karşılaşılır. Tabi bu da metodun çağrıldığı yerde hiçbir şeye atanılmayacağı anlamına gelir.

```
2 references
class Mavi
{
    1 reference
    int metot2()
    {
        Console.WriteLine("Yolculuk nereye");
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        refer1.metot2();
        Console.ReadLine();
    }
}
```

Göründüğü gibi hata ile karşılaşılmıştır birde int yerine void ile tanımlayalım.

```
2 references
class Mavi
{
    1 reference
    void metot2()
    {
        Console.WriteLine("Yolculuk nereye");
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        refer1.metot2();
        Console.ReadLine();
    }
}
```



Göründüğü gibi bir sorun oluşmamıştır.

Yine aynı şekilde eğer void kullanılacaksa geri dönüş değeri yoktur aksi takdirde hata ile karşılaşılır.

Dikkat edilmelidir ki atanılan tiple metodun tipi uyumlu olmalıdır aksi takdirde cast işlemi uygulanmalıdır([taşma olabilir]).

Eğer atanılan tip daha büyük bir tipse bu durumda sorun oluşmaz.

```
1 reference
int metot2()
{
    return 9;
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    byte c=refer1.metot2();
    Console.WriteLine(c);
    Console.ReadLine();
}
```

burada sorun oluştı boxing yapılsa bile veri kaybı yaşanabilir.

```
2 references
class Mavi
{
    1 reference
    int metot2()
    {
        return 9;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        byte c=(byte)refer1.metot2();
        Console.WriteLine(c);
        Console.ReadLine();
    }
}
```

Aşağıdaki kullanımda hiçbir sorun oluşmaz.

```
2 references
class Mavi
{
    1 reference
    int metot2()
    {
        return 9;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        double c=refer1.metot2();
        Console.WriteLine(c);
        Console.ReadLine();
    }
}
```

Metot Parametrelerinin önceden atanması:

Metodun parametreleri eğer önceden atanmışsa default değer gibidir.

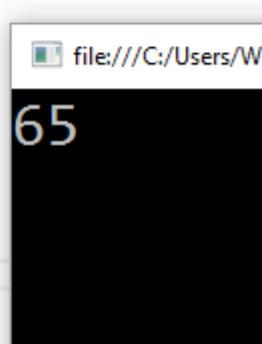
Girilmese de sorun oluşmaz atanın değer üzerinden işlem yapılır.

Aşağıdaki durumda a değeri kesin olarak girilmelidir fakat girilmediği için hatayla karşılaşılmış.

```
class Mavi
{
    1 reference
    int metot2(int a,int b=8,int c=48)
    {
        int x=a+b+c;
        return x;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        int c=refer1.metot2();
    }
}
```

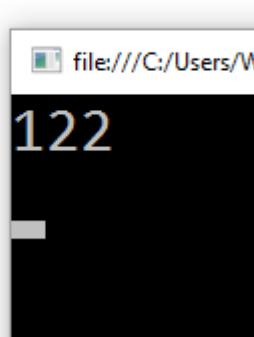
a değerini girdiğimizde hiçbir sorun oluşmaz.

```
2 references
class Mavi
{
    1 reference
    int metot2(int a,int b=8,int c=48)
    {
        int x=a+b+c;
        return x;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        int c=refer1.metot2(9);
        Console.WriteLine(c);
        Console.ReadLine();
    }
}
```



Eğer değer metotda atanmışsa ve ona rağmen o değer için değer girilirse girilen değer üzerinden işlem yapılır.

```
1 reference
int metot2(int a,int b=8,int c=48)
{
    int x=a+b+c;
    return x;
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    int c=refer1.metot2(9,65);
    Console.WriteLine(c);
    Console.ReadLine();
}
```



Eğer önceden atanmamışsa ve o değer girilmemişse hata ile karşılaşılır.

Aşağıda b değeride önceden atanmamış olduğu için girilmelidir.

```
class Mavi
{
    1 reference
    int metot2(int a,int b,int c=48)
    {
        int x=a+b+c;
        return x;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        int c=refer1.metot2(9);
        Console.WriteLine(c);
        Console.ReadLine();
    }
}
```

Eğer metodun parametrelerinden birine değer atanmışsa ondan sonraki tüm değerlere de değer atanmalıdır. Aksi takdirde hata ile karşılaşılır.

Aşağıdaki durumda hata ile karşılaşılmıştır. Çünkü *c* *b*'den sonra gelmesine rağmen *c*'ye değer atanmamıştır.

```
0 references
static int metot2(int a,int b=48,int c)
{
    int x=a+b+c;
    return x;
}
0 references
```

Not:

- Metot içinde metot olmaz.
- Parametredeki değişken tipleri tek tek belirlenmelidir ortak virgülle yapılamaz.

~~static int islem(int a, b)~~

burada hata alınır.

9-Metot parametrelerinin metot içerisinde tanımlanması geçersizdir.

```
static int islem(int a, b)
{
    int b;
    return a + b;
```

- 10- Metotlar içinde tanımlanan değişkenler, programın akışı metoda geldiğinde tanımlanır. Metodun işlevi bittiğinde ise bellekten silinirler.
- Programın akışı tekrar metoda geldiğinde değişkenler yeniden tanımlanır ve sonunda yine bellekten silinir. Bu tür değişkenlere **otomatik ömürlü nesneler** denir.

Metotlara aşağıdaki gibi dizide yollandabilir.

```
using System;
class Metotlar
{ static void Yaz(int[] dizi)
    { foreach(int i in dizi)
        Console.WriteLine(i);
    }
    static void Main()
    { int[] dizi={1,2,4,7,9};
        Yaz(dizi);
    }
}
```

Array şeklinde gönderimde mümkündür çoğu durumda kolaylıkda sağlar.

- Eğer sadece **int []** türündeki değil, bütün türlerdeki dizileri ekrana yazan genel bir metot yazmak istiyorsak dizi tipi **Array** tanımlanır:

```
using System;
class Metotlar1
{
    static void Yaz (Array dizi)
    { foreach(object i in dizi)
        Console.WriteLine(i);
    }
    static void Main() {
        int [] dizi={1,2,4,7,9}; Yaz(dizi); }
}
```

/object yerine var veya dynamic kullanılabilir

Metot parametreleri *object* veya *var* tipinde olamaz.Hata ile karşılaşılır.

Fakat metot parametreleri *dynamic* tipinde olabilir.

```
using System;
class Metotlar1
{
    static void Yaz (object [] dizi)
    { foreach(object i in dizi)
        Console.WriteLine(i);
    }
    static void Main()
    {
        int[] dizi={1,2,4,7,9}; Yaz(dizi); }
}
```

```
using System;
class Metotlar1
{
    static void Yaz (dynamic [] dizi)
    // veya static void Yaz (dynamic dizi)
    { foreach(dynamic i in dizi)
        Console.WriteLine(i);
    }
    static void Main()
    {
        int[] dizi={1,2,4,7,9}; Yaz(dizi); }
```

19

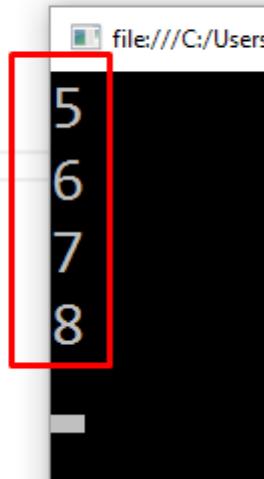
Metot parametrelerinin değer ve referans tipinde olması

Değer tipindekilerde hiçbir sorun oluşmaz yollanan değerin kopyası başka bir adres'e yazılır ve o adres üzerinden işlem yapılır.

Referans tipinde adresleme yapıldığı için metoda yollandığı için adresdeki değerler kopyalanmaz direkt olarak adresi gider ve o adresdeki değerler üzerinde işlem yapılır.

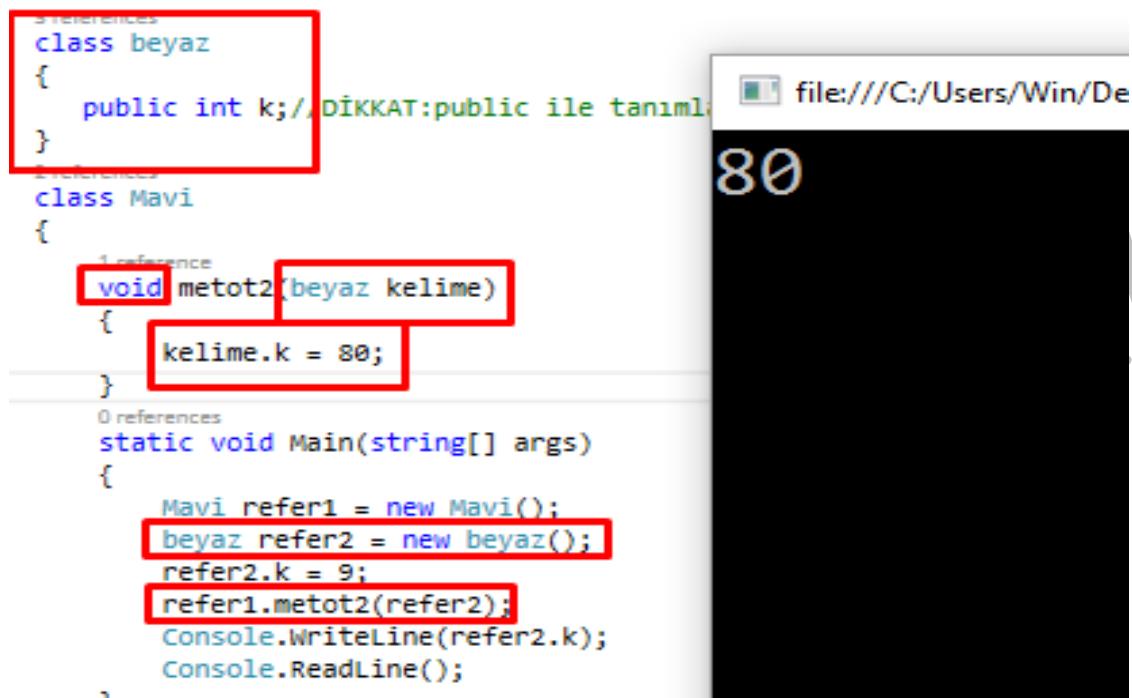
Mesela dizi referans tipinde olduğu için ana programdan metoda yollanması halinde metodda yapılan işlemler anaprogramdaki diziyede yansır.

```
void metot2(int [] dizi)
{
    for (int i = 0; i < dizi.Length; i++)
    {
        dizi[i]=dizi[i]+5;
    }
}
0 references
static void Main(string[] args)
{
    int[] dz = { 0, 1, 2, 3 };
    Mavi refer1 = new Mavi();
    refer1.metot2(dz);
    foreach (int k in dz)
    {
        Console.WriteLine(k);
    }
    Console.ReadLine();
}
```



String'de referans olmasına rağmen böyle bir sorun yaşanmaz

Bizim oluşturduğumuz bir nesne üzerinde denersek adresleme olucuğu için değer değişir.



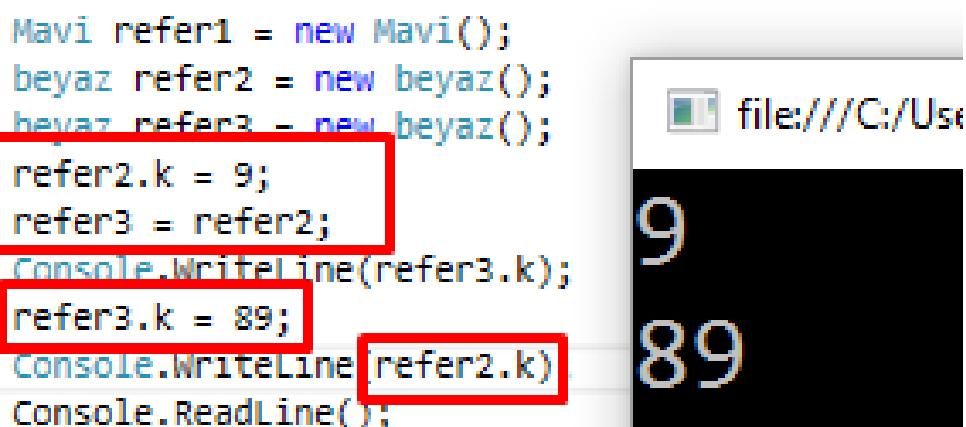
```
class beyaz
{
    public int k; // DİKKAT:public ile tanımlı
}

class Mavi
{
    void metot2(beyaz kelime)
    {
        kelime.k = 80;
    }
}

static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    beyaz refer2 = new beyaz();
    refer2.k = 9;
    refer1.metot2(refer2);
    Console.WriteLine(refer2.k);
    Console.ReadLine();
}
```

Görüldüğü gibi refer nesnesinin içeriği değişmiştir. Referans tipinde olduğu için adresi üzerinde işlem yapılmıştır.

İki referans birbirlerine eşitlendiğindede adresler eşitlenicegi için birinde yapılan işlemler diğerinede yapılmış gibi olacaktır.



```
Mavi refer1 = new Mavi();
beyaz refer2 = new beyaz();
beyaz refer3 = new beyaz();

refer2.k = 9;
refer3 = refer2;
Console.WriteLine(refer3.k);
refer3.k = 89;
Console.WriteLine(refer2.k)
Console.ReadLine();
```

Ref ve Out anahtar sözcükleri:

Ref anahtar sözcüğü:

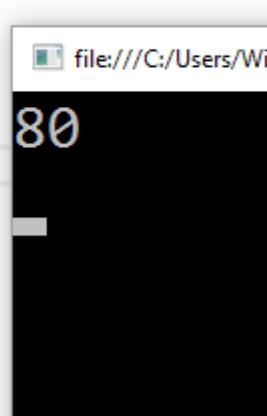
Bilindiği gibi değer tipleri metoda yollandığında referans gibi adresi yollanmadığı için asıl değerde değişim olmaz.

Fakat Eğer bir değer tipinde de referans gibi adres üzerinde işlem yapılmasını istiyorsak **ref** anahtar sözcüğünü kullanmalıyız.

(Referanslar içinde kullanılır herhangi bir sorunla karşılaşılmaz ama zaten referanslarda bu işlemler yapıldığı için gerek yoktur.)

ref anahtar sözcüğü çağrılan metodun kullanımı şu şekildedir:

```
1 reference
void metot2(ref int sayi)
{
    sayi = 80;
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    int a=4;
    refer1.metot2(ref a);
    Console.WriteLine(a);
    Console.ReadLine();
}
```



Yani metodun parametresinde referans tipinde kabul edilecek olan değişkenin başına ve anaprogramda metodun parametresine referans olarak gönderilecek değer yazılrken başlarına **ref** yazılır.

ref sözcüğü, hem metodu çağrıırken , hem de metodu yaratırken değişkenden önce yazılması gerekiyor.

Metotda:

```
static void Degistir (ref int sayı)
```

Anaprogramda:

```
int sayı=1;      Degistir(ref sayı);
```

Unutulmamalıdır ki bu şekilde kullanılan değerlere ana programında bir başlangıç değeri verilmelidir.

Çünkü metot ref ilemasına rağmen değer üzerinde değişiklik yapmadan işlemler yapıyorsa değere ait hiçbir değer olmadığı için hata ile karşılaşılır.a=a+5'de bile a'da değişiklik yapılmasına rağmen a'nın bir ilk değeri olmalıdır.

Bu yüzden kesinlikle a'yi metoda yollamadan önce a'ya bir ilk değer atamalıyız.

Aşağıdaki durumda hata ile karşılaşılacaktır.Cünkü a'ya ilk değer atanmamıştır.

```
1 reference
void metot2(ref int sayı)
{
    sayı = 80;
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    int a;
    refer1.metot2(ref a);
    Console.WriteLine(a);
```

Out Anahtar sözcüğü:

Gramer olarak kullanımı aynıdır.

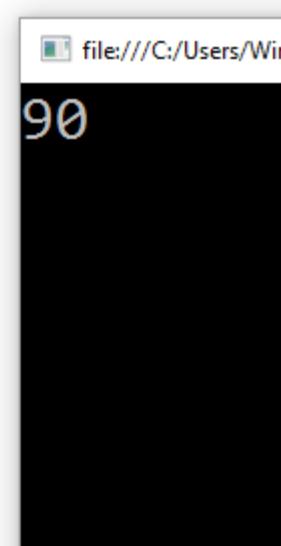
ref'den farkıysa ilk değer ataması olmasa da olur.

Fakat metot da out olarak çağrılan değişkene değer atamak zorunludur.

Eğer atanmadıysa nasıl ki ref'de çağrılan metot parametresine yazacağımız değişkene ilk değer atamayınca hata ile karşılaşıyoruz. out'da da metodun içinde ilk değeri atamadığımız takdirde hatayla karşılaşırız.

Aşağıdaki durumda hiçbir sorun yoktur.

```
↳ references
class Mavi
{
    1 reference
    void metot2(out int sayı)
    {
        sayı = 90;
    }
    0 references
    static void Main(string[] args)
    {
        Mavi refer1 = new Mavi();
        int a;
        refer1.metot2(out a);
        Console.WriteLine(a);
        Console.ReadLine();
    }
}
```



Fakat aşağıdaki durumda hata ile karşılaşılır. Çünkü metodun içinde ilk değer ataması yapılmamıştır. Yollandan önce yapılmasıının önemi yoktur.

```
1 reference
void metot2(out int sayi)
{
    int x=sayi+90;
}

0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    int a=5;
    refer1.metot2(out a);
    Console.WriteLine(a);
    Console.ReadLine();
}
```

Metotların Aşırı yüklenmesi (Method overloading)

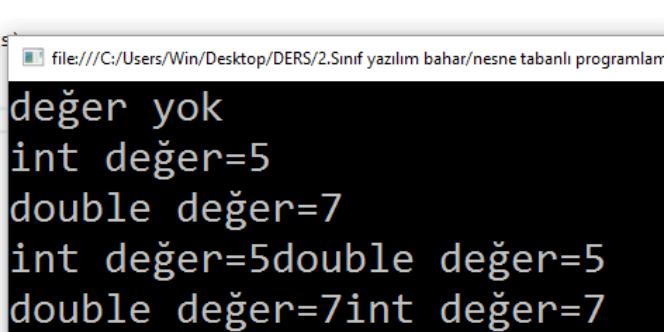
Aynı isimli metotların hangisinin çağırılacağına o metodun imzasına göre karar verilir.

Metodun parametresindeki değişken sayısı ve değişken türleri metodun imzasını belirler.

```
1 reference
void metot2()
{
    Console.WriteLine("değer yok");
}
1 reference
void metot2(int sayi)
{
    Console.WriteLine("int değer=" + sayi);
}

1 reference
void metot2(double sayi)
{
    Console.WriteLine("double değer=" + sayi);
}
1 reference
void metot2(double sayı,int sayı2)
{
    Console.WriteLine("double değer=" + sayı+"int değer=" + sayı);
}
1 reference
void metot2(int sayı, double sayı2)
{
    Console.WriteLine("int değer=" + sayı + "double değer=" + sayı);
}
```

```
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    int a=5;
    double b = 7;
    refer1.metot2();
    refer1.metot2(a);
    refer1.metot2(b);
    refer1.metot2(a,b);
    refer1.metot2(b,a);
    Console.ReadLine();
}
```



```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahr/nesne tabanlı programları
değer yok
int değer=5
double değer=7
int değer=5double değer=5
double değer=7int değer=7
```

Derleyici hangi metodu çağıracağına karar vermek için, metot bildirimini ile metot çağrımları arasındaki tam uyumu arar. Eğer tam uyumluluk bulunamaz ise parametreler arasında küçük türün büyük türü dönüşmesinin yasal olabileceği metot çağrıılır.

```
static void OrnekMetot(double x, double y)
{
    Console.WriteLine("1.Metot");
}
static void OrnekMetot(byte x, byte y)
{
    Console.WriteLine("4.Metot");
}

static void Main()
{
    OrnekMetot(1919,1923);
    OrnekMetot(12, 34);
    Console.ReadLine();
}
```

30

Fakat dikkat edilmelidir ki:

Metodun imzası metodun tipine göre belirlenmez aynı isimdeki aynı sayıdaki ve aynı tipteki parametrelere sahip olan 2 farklı tipte metot varsa hata ile karşılaşılır.

Aşağıdaki gibi bir durumda hata ile karşılaşılır.

```
1 reference
void metot2(int sayı)
{
    Console.WriteLine("int değer=" +sayı);
}
1 reference
int metot2(int sayı)
{
    Console.WriteLine("int değer=" + sayı);
}
```

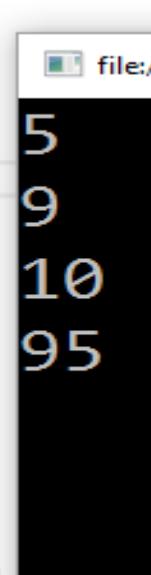
Params Anahtar Sözcüğü

Eğer kaç tane değer girilceği belirsizse params sözcüğüyle dinamik bir dizi tanımlanır.

Kullanımı şu şekildedir:

```
1 reference
void metot2(params int [] x)
{
    foreach (int i in x)
    {
        Console.WriteLine(i);
    }
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();

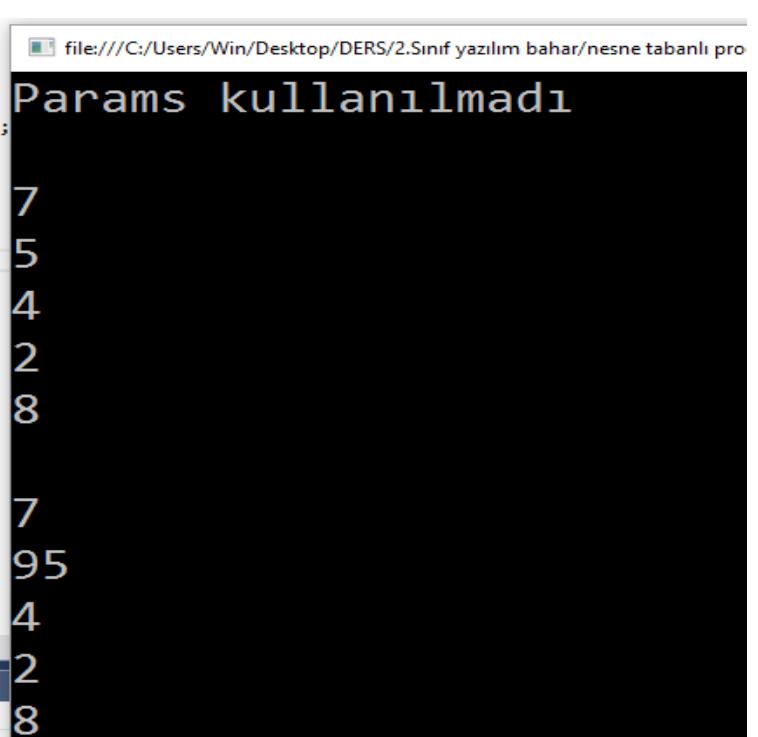
    refer1.metot2(5,9,10,95);
    Console.ReadLine();
}
```



```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı pro
5
9
10
95
```

Params ile kullanılan seçenek alternatif bulunduğu takdirde en son seçenektedir.

```
1 reference
void metot1(double z, int x)
{
    Console.WriteLine("Params kullanılmadı");
}
2 references
void metot2(double z,params int [] x)
{
    Console.WriteLine(z);
    foreach (int i in x)
    {
        Console.WriteLine(i);
    }
}
0 references
static void Main(string[] args)
{
    Mavi refer1 = new Mavi();
    double a = 7;
    int b = 5;
    refer1.metot2(a,b);
    Console.WriteLine();
    refer1.metot2(a,b,4,2,8);
    Console.WriteLine();
    refer1.metot2(a, 95, 4, 2, 8);
    Console.ReadLine();
}
```



```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı pro
Params kullanılmadı
7
5
4
2
8
7
95
4
2
8
```

Unutulmamalıdır ki:

Params parametrenin en son değişkeni için kullanılmalıdır.

Aşağıdaki kullanımların tümünde hatayla karşılaşılır.

```
!(params double[] z, string k)
```

2 references

```
void metot2(params double[] z, params int [] x)
```

```
{
```

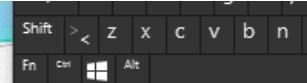
```
(params double[] z, string k, params int [] x)
```

Params'la ilgili Örnek:

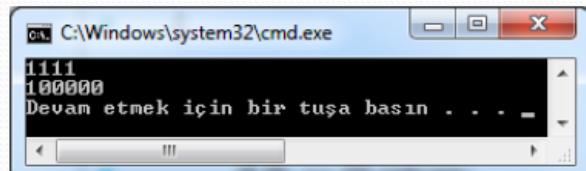
```
using System;
class Metotlar
{
    static void Metot1(int x, int y)
    { Console.WriteLine("1. metot çağrıldı."); }
    static void Metot1(int x, params int[] y)
    { Console.WriteLine("2. metot çağrıldı."); }
    static void Main()
    { Metot1(3, 6, 8); }
}
```

//Burada 2. metot çağrılr.

Rekürsif metotlar



- Kendi kendini çağıran metotlara **özyineli(recursive)** metot denilir. Bu metotlarda, çağrımı sonlandıran bir kontrol yapısı da olmalıdır.
 - Örnek: 10 luk sayıyı 2 lik sayıya dönüştüren program
- ```
using System;
class Program
{
 static void BitYaz(int b)
 {
 if (b == 0) return;
 BitYaz(b >> 1);
 Console.Write(b & 1); // console.WriteLine(b); ikiye bölümünden kalan sonucu yazar
 }
 static void Main()
 {
 BitYaz(15); Console.WriteLine(); // 1 3 7 15
 BitYaz(32); Console.WriteLine(); // 1 2 4 8 16 32
 }
}
```



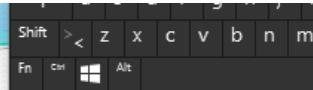
35

## Main Metodu



- Main metodunu diğer metodlardan temel olarak farkı yoktur, tek farkı programın çalışmaya başladığı nokta olmasıdır.
- Main metodunun **int** türünden bir geri dönüş değeri de olabilir. Bu değer işletim sistemine programın çalışması hakkında bilgi vermek amacıyla kullanılır. Orneğin Main metodu 0 değerini döndürürse işletim sistemi programın düzgün şekilde sonlandırıldığını, 1 değerini döndürürse de hatalı sonlandırıldığını anlayabilecektir.
- Main metodu parametre de alabilir. Komut satırından çalıştırıldığında verilen parametreleri bir **string** dizisi içinde alarak işleyebilir.
- static int Main (**string** [] args)

# Main Metodu



- static int Main (**string [] args**)
- Burada komut satırından girilen argümanlar string[] türündeki args dizisine aktarılıyor. Bu diziyi programımızda gönlümüzce kullanabiliriz. Örneğin programımız deneme.exe olsun. Komut satırından;
- **deneme ayşe bekir rabia**
- girilirse ilk sözcük olan deneme ile programımız çalıştırılır ve ayşe, bekir ve rabia sözcükleri de string[] türündeki args dizisine aktarılır. komut satırında program adından sonra girilen ilk sözcük args dizisinin 0. indeksine, ikinci sözcük 1. indeksine, üçüncü sözcük 2. indeksine vs. aktarılır. Örnek bir program:
  - **using System;**
  - **class Metotlar**
  - { **static void Main(string[] args) {**
  - **Console.WriteLine("Komut satırından şunları girdiniz: ");**
  - **foreach(string i in args) Console.WriteLine(i);**
  - } }

37

# System.Math Sınıfı()



- .Net sınıf kütüphanesinde belirli matematiksel işlemleri yapan birçok metod ve iki tane de özellik vardır. **System.Math** sınıfındaki metodlar static oldukları için bu metodları kullanabilmek için içinde bulundukları sınıf türünden bir nesne yaratmaya gerek yoktur. System.Math sınıfındaki iki özellik matematikteki pi ve e sayılarıdır. Pi ve E özellikleri double türünden değer tutarlar.

```
using System;
class Metotlar
{
 static void Main()
 {
 double e=Math.E;
 double pi=Math.PI;
 Console.WriteLine("e->" + e + " pi->" + pi);
 }
}
```

38

# System.Math Sınıfı()



- System.Math sınıfındaki bütün metodlar

|                   |                                                     |
|-------------------|-----------------------------------------------------|
| <b>Abs(x)</b>     | Bir sayının mutlak değerini tutar.                  |
| <b>Cos(x)</b>     | Sayının kosinüsünü tutar.                           |
| <b>Sin(x)</b>     | Sayının sinüsünü tutar.                             |
| <b>Tan(x)</b>     | Sayının tanjantını tutar.                           |
| <b>Ceiling(x)</b> | x sayısını x ten büyük en küçük tamsayıya yuvarlar. |
| <b>Floor(x)</b>   | x sayısını x ten küçük en büyük tamsayıya yuvarlar. |
| <b>Max(x,y)</b>   | x ve y sayılarından en büyüğünü bulur.              |
| <b>Min(x,y)</b>   | x ve y sayılarından en küçüğünü bulur.              |
| <b>Pow(x,y)</b>   | x üzeri y yi hesaplar.                              |
| <b>Round(x)</b>   | x i ne yakın tam sayıya yuvarlar.                   |
| <b>Sqrt(x)</b>    | x in karekökünü bulur.                              |
| <b>Log(x)</b>     | x sayısının e tabanında logaritmasını alır.         |
| <b>Exp(x)</b>     | e üzeri x'i hesaplar.                               |
| <b>Log10(x)</b>   | x sayısının 10 tabanında logaritmasını hesaplar.    |

39

Baybars

# SINIFLAR

- Sınıfların üye elemanları değişkenler(özellik) ve metodlardır.
  - Metotlar bir veya daha fazla komutun bir araya getirilmiş halidir; parametre alabilirler, geriye değer döndürebilirler.
  - Özellikler ise bellek gözeneklerinin programlamadaki karşılıklarıdır. Bu bakımından özellikler değişkenlere benzerler. Aradaki en temel fark değişkenlerin bir metod içinde tanımlanıp yalnızca tanımlandığı metod içinde etkinlik gösterebilmesine rağmen özelliklerin tíki metodlar gibi bir üye elemanı olmasıdır. Bu bakımından özelliklerin tuttuğu değerlere daha fazla yerden erişilebilir.

42

<erişim belirleyici> <veri türü> **özellik1**;

<erişim belirleyici> <geri dönüş tipi> **metot2(parametreler)**

## Erişim belirleyiciler

**public:** erişim belirteci ile tanımlanan sınıfın bir özelliğine ya da metoduna istenilen yerden erişime izin verilmiş olur. public üye elemanlar genelde sınıfın dışa sunduğu ara yüzü temsil eder.

**private:** erişim belirteci ile tanımlanan üyelerle ancak tanımlandıkları sınıfın içindeki diğer üye elemanlar erişebilir, dışarıdan erişmek mümkün değildir. Erişim belirteci verilmeyen tanımlamaların varsayılan değeri **private** olur.

**protected:** erişim belirteci sınıf türemesi yoksa private ile aynıdır. Fakat bir sınıfın başka bir sınıf türetiliyorsa private üyeleri diğer sınıfa aktarılmaz, sadece public ve protected elemanlar aktarılırlar.

internal ve protected internal o kadar da gerekli değil.

Nesnelerin bildirilmesi ve tanımlanması şu şekilde olur:

- **Ogrenci ogr1; (Bildirim)**

Bildirim ile nesneye erişim için bir değişken tanımlanmış olur. Fakat nesne bellekte oluşturulmamıştır. Yer tahsisatı yapabilmek için **new** kullanılır. Bunun için de;

- **Ogrenci ogr1 = new Ogrenci();**

kullanılır.

- **ogrenci ogr1; // nesneyi gösteren referans tanımlandı**
- **ogr1 = new Ogrenci(); // Nesne için bellek tahsil edildi**

Sınıflar, **new** anahtar sözcüğü ile tanımlandığında sınıfın içerisindeki bütün üyeleri varsayılan değere atanır.

Sınıftan tanımlanan bir nesnenin üyelerine **"** operatörü ile ulaşılır. **public** olarak tanımlanmış üye elemanlarının değerleri değiştirilebilir.

## Sınıf içinde başka bir sınıf tanımlama

Sınıflar ayrı ayrı tanımlanıldığı gibi sınıf içinde başka bir sınıfta oluşturulabilir. İçteki sınıfın eleman veya metodlarına erişmek için kurallar aynıdır.

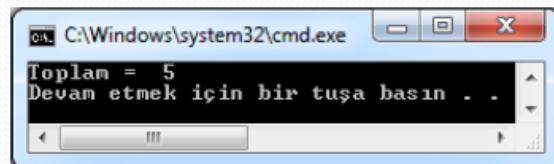
```
using System;
class Sinif
{ int x=10; static int y = 0;
 class Sinif2
 { public int x = 0; static public int y=10;
 }
 static void Main()
 { Sinif a = new Sinif();
 Sinif2 b = new Sinif2();
 //veya Sinif.Sinif2 b = new Sinif2();//
 a.x = 7; Sinif.y = 15;
 b.x = 12; Sinif2.y = 1;
 }
}
```

# this Anahtar sözcüğü

- Birçok durumda sorun yaşanmamakla birlikte bazen nesnenin üyeleri ile metot içindeki tanımlamalar aynı olabilir. Bu durumda metot değişkenleri ile nesnenin üyelerini birbirinden ayırmak için **this** anahtar sözcüğü kullanılır.
- **this** anahtar sözcüğü ilgili nesnenin referansını belirtir.
- **this** anahtar sözcüğü, içinde bulunulan nesneye ait bir referans döndürür
- Bunun sayesinde nesnelere ait global alanlara erişme fırsatı bulunur;

```
using System;
class Program
{
 int topla; // Global alandaki değişken
 public void sayıTopla(int topla) // yerel alandaki değişken
 {
 this.topla += topla;
 }
 public void sayiyiEkranaYaz() {
 Console.WriteLine("Toplam = " + topla);
 }
 static void Main(string[] args)
 { Program th = new Program();
 th.sayiTopla(2);
 th.sayiTopla(3);
 th.sayiyiEkranaYaz();
 }
}
```

this olmasaydı sonuç 0 olurdu. 



# get ve set Anahtar sözcükleri:

temel mantık 2 tane değişken yapıp gizli olarak birini diğerini üzerinden yönetmektir.

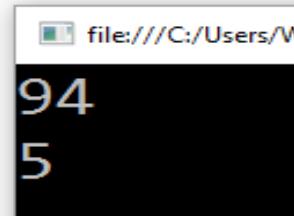
- Bir nesnenin üye değişkenlerine değer atarken ya da içerisindeki değeri kullanırken belli kontrollerin yapılması gerekiyorsa ya da kod bloklarının çalıştırılması isteniyorsa **set** ve **get** ifadeleri kullanılır.
- **set** sözcüğü nesnenin özelliklerine değer atandığında çalışır.
- **get** sözcüğü ise özellik değeri okunduğunda ya da farklı bir ifadeye aktarılmaya çalışıldığında çalışır.

değişken tipi ve değer girildikten sonra get ve set metotları ayarlanır

get değer çağrırlığında çalışır.[return değeri olabilir.]

set değer üzerinde işlem yapıldığında çalışır.

```
2 references
class Mavi
{
 4 references
 int a
 {
 get
 {
 return 5;
 }
 set
 {
 if(a>50)a = 51;
 }
 }
}
0 references
static void Main(string[] args)
{
 Mavi refer1 = new Mavi();
 int x = refer1.a + 89;
 Console.WriteLine(x);
 Console.WriteLine(refer1.a);
```



2. Örnekte **SahteOzellik** adlı bir özellik oluşturduk. Bu özellik gerçekten de sahtedir. Özelliğe bir değer atanmaya çalışıldığında **set** bloğundaki, özellik kullanılmaya çalışıldığında da **get** bloğundaki komutlar çalıştırılır. Aslında C#

```
• class Set_Get
• {
• int Sayi;
• public int SahteOzellik
• {
• set
• {
• if (value < 0) Sayi = 0;
• else Sayi = value;
• }
• get
• {
• if (Sayi>100) return Sayi/100;
• else return Sayi;
• }
• }
• }
```

59

Not: burada sayıya atıယacağımız değer sahteözelleiже gitmiş oluyor  
eğer dışarıdan bir değişken değilde o anda atanın değer  
kullanılmaya çalışılıyorsa value kullanılır fakat sorun yaratıyor.

Programdaki **value** sözcüğü özelliğe girilen değeri tutar. Özelliğe girilen  
değer hangi türdeyse o türde tutar. Ayrıca bu oluşturulan **SahteOzellik**  
ozelliğinin metotlara oldukça benzerdir. Görüldüğü gibi, değişkenin değerini  
değiştirdiğimizde çalışmasını istediğimiz kodları set blokları arasına yazarız.

Herhangi bir sahte özelliğin **set** veya **get** bloklarından yalnızca  
birini yazarak o özelliği salt okunur veya salt yazılır hâle  
getirebiliriz.

Örneğin **Array** sınıfının **Length** özelliği salt okunur bir özelliktir.

**NOT:** C# 2.0'da ya da daha üst versiyonlarda bir sahte özelliğin **set** ve **get**  
bloklarını ayrı ayrı **private** veya **public** anahtar sözcükleriyle belirtebiliyoruz.  
Yani bir özelliğin bulunduğu sınıfın dışında salt okunur ya da salt yazılır  
olmasını sağlayabiliyoruz.

```
private int m_uzunluk;
public int Uzunluk
{
 get
 {return m_uzunluk;
 }

 set
 { m_uzunluk = value;
 }
}
```

get ve set blokları  
public olarak tanımlıdır.

```
private int m_uzunluk;
public int Uzunluk
{
 get
 {return m_uzunluk;
 }

 private set
 { m_uzunluk = value;
 }
}
```

get ve set blokları;  
C#2.0'dan itibaren  
farklı erişim  
belirleyicileri ile  
belirlenebilmektedir.

- Değişken için bildirilen erişim belirleyicisi; **get** veya **set** bloğunda tanımlanan erişim belirleyicisinden daha yüksek erişim yetkisine sahip olmalıdır.

**NOT:** get ve set anahtar sözcükleriyle erişim belirleyiciler kullanırken uymamız gereken bazı özellikler vardır:

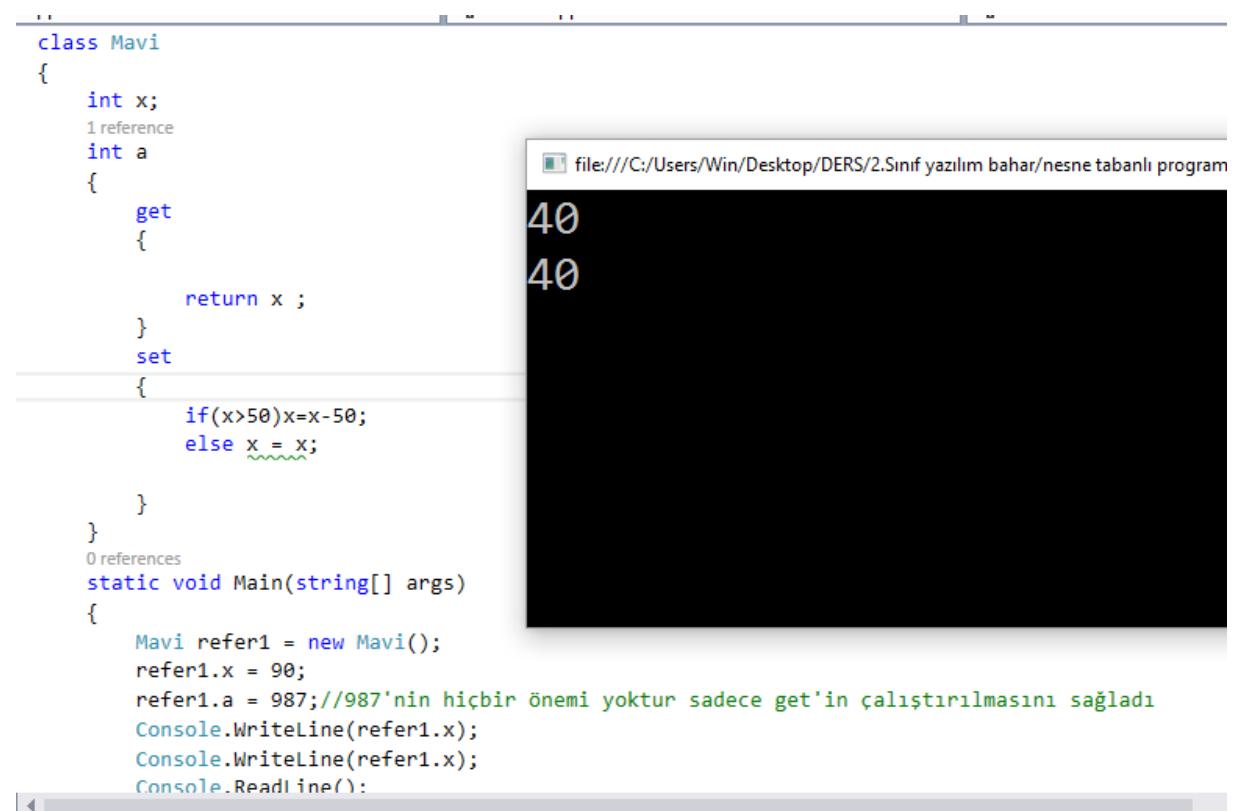
Daima özellik bildiriminde kullanılan erişim belirleyicisi **get** veya **set** satırında kullanılan erişim belirleyicisinden daha yüksek seviyeli olmalıdır. Örneğin özellik bildiriminde kullanılan erişim belirleyici **private** ise **get** veya **set** satırında kullanılan erişim belirleyici **public** olamaz.

**get** veya **set** satırında kullanılan erişim belirleyici özellik bildiriminde kullanılan erişim belirleyiciyle aynı olmamalıdır. (Zaten gereksizdir.)

Yani işin özü **set** ve **get** satırlarındaki erişim belirleyicileri yalnızca, özelliği **public** olarak belirtmiş ancak özelliğin **set** veya **get** bloklarının herhangi birisini **private** yapmak istediğimizde kullanılabilir.

**get** veya **set** için erişim belirleyicisi kullanacaksak sahte özelliğin bloğu içinde hem **get** hem de **set** bloğunun olması gereklidir.

## YANI TÜM DURUM ÖZETLENİRSE



```
class Mavi
{
 int x;
 1 reference
 int a
 {
 get
 {
 return x ;
 }
 set
 {
 if(x>50)x=x-50;
 else x = x;
 }
 }
 0 references
 static void Main(string[] args)
 {
 Mavi refer1 = new Mavi();
 refer1.x = 90;
 refer1.a = 987;//987'nin hiçbir önemi yoktur sadece get'in çalıştırılmasını sağladı
 Console.WriteLine(refer1.x);
 Console.WriteLine(refer1.x);
 Console.ReadLine();
 }
}
```

Bu programda x'e değerler atanıyor atanan değerler doğrultusunda a değişkeninin okunması veya işlem yapılması halinde get veya set tetikleniyor. Ve x'in değeri değişiyor.

**NOT: get ve set'i tekrar et mantığını tam olarak anla**

# DİZİLERİN METOT VE ÖZELLİKLERİİNİN KULLANILMASI

```
• using System;
• class YardimciSinif
• { public int[] Dizi = { 7, 4, 3 };
• public int[] Metot()
• {
• int[] a = { 23, 45, 67 };
• return a;
• }
• }
• class AnaSinif
• { static void Main()
• { YardimciSinif nesne = new YardimciSinif();
• Console.WriteLine(nesne.Dizi[0]);
• Console.WriteLine(nesne.Metot()[2]);
• }
• }
```

**HATIRLATMA:** Metotların parametresindeki dizi, Array DiziAdı yöntemiyle oluşturulduğu için bu dizinin elemanlarına klasik indeksleme yöntemiyle erişemeyiz. Bu yüzden bu dizinin elemanlarına ulaşmak için Array sınıfının GetValue() metodu kullanıldı. (Diğer yol foreach-object, dynamic, var kullanma)

66

```
• using System;
• class Donustur
• { public static int[] Inte(Array dizi)
• {
• int[] gecici=new int[dizi.Length];
• for(int i=0;i<dizi.Length;i++)
• {
• gecici[i]=Convert.ToInt32(dizi.GetValue(i));
• }
• return gecici;
• }
• }
```

```
using System;
class Donustur
{ public static int[] Inte(Array dizi)
{
 int[] gecici=new int[dizi.Length];
 int j = 0;
 foreach (object i in dizi)
 {
 gecici[j] = Convert.ToInt32(i);
 }
 Console.WriteLine(gecici[j]);
 j++;
}
```

## Yapıcı Metotlar(Constructors)

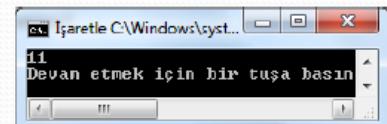
- Bir nesnenin dinamik olarak yaratıldığı anda otomatik olarak çalıştırılan metotlar vardır. Bu metotlar sayesinde bir nesnenin üye elemanlarına ilk değerler verilebilir ya da gerekli ilk düzenlemeler yapılabilir. Bu metotlara **yapıcı metotlar (constructors)** denir.
- Yapıçı metot tanımlanmasa dahi her sınıfın varsayılan bir yapıçı metodu (**default constructor**) mutlaka bulunur.
- Daha önceden bahsettiğimiz gibi bir nesne oluşturulduğunda **sayısal değerler için 0, referanslar için null, bool türü için false atanır**, bu atama bir yapıçı metot ile gerçekleştirilir.

## Varsayılan Yapıçı Metot (Default Constructor)

- Genel Özellikleri:
- Yapıcların dönüş değeri yoktur, isimleri sınıf ile aynı isimdir. Varsayılan yapıçı metodun parametresi yoktur. Nesnenin üye elemanlarına varsayılan değerlerini atar.
- Yapıçı metotlar bir değer tutamaz ve normal metodlardan farklı olarak void anahtar sözcüğü de kullanılmaz.
- Yapıçı metotlar dışarıdan çağrıldığı için genelde public olarak tanımlanırlar. private olursa dışarıdan erişilemez
- Eğer kendi yapıçı metodumuzu tanımlarsak varsayılan yapıçı metotlar çalıştırılmaz.
- Yapıçı metotlar da aşırı yüklenebilir. Aşırı yükleme durumunda boş yapıçı metot mutlaka bulunmalıdır.
  - Örneğin public Deneme();, public Deneme(int a);, public Deneme(int a, int b);

## Yapıcı Metot (Default Constructor)

- Aşırı yükleme **this** anahtar sözcüğü kullanılarak da oluşturulabilir. Bu örnekte this anahtar sözcüğü sayesinde ikinci, üçüncü ve dördüncü yapıçı metotlar içeriğini aynı isimli ve üç parametre alan metottan alıyor. this anahtar sözcüğüyle kullanılan ikinci, üçüncü ve dördüncü yapıçı metotların yaptığı tek iş, birinci yapıçı metoda belirli parametreleri göndermek oluyor.
- **using System;**
- **class Deneme**
- { **Deneme(int a,int b,int c)** { **Console.WriteLine(a+b+c);** }  
**Deneme():this(0,0,0) { }** }
- **Deneme(int a):this(a,0,0) { }**
- **Deneme(int a,int b):this(a,b,0) { }**
- **static void Main()** { **Deneme a=new Deneme(5,6);** }
- }
- NOT: this anahtar sözcüğü bu şekilde yalnızca yapıçı metotlarla kullanılabilir.



Eğer bir sınıfta parametre alan bir veya daha fazla yapıçı metot varsa ve parametre almayan yapıçı metot yoksa -bu durumda varsayılan yapıçı metot oluşturulmayacağı için- Sinif nesne=new Sinif(); gibi bir satırda parametre vermeden nesne oluşturmaya çalışmak hatalıdır. Çünkü Sinif nesne=new Sinif(); satırı Sinif sınıfında parametre almayan bir yapıçı metot arar, bulamaz ve hata verir. Ancak boş yapıçı metot kullanılmayacaksız hata vermez. Örnek:

```
class A
{
 // public A() {} tanımlanması gereklidir yoksa hata verir
 public A(int a){}
}

class B
{
 A a=new A();
}
```

Derleme anında hata verir.

75

## Kopyalayıcı Yapıçı Metot (Copy Constructors)

- Yapıcı metot parametre olarak kendi sınıfından bir nesne alıyorsa kopyalayıcı metot adı verilir.

```
using System;
class Toplama
{
 public int X;
 public int Y;

 public Toplama(int x, int y)
 {
 X = x;
 Y = y;
 }

 public Toplama(Toplama T)
 {
 X = T.X;
 Y = T.Y;
 }

 public int Islem()
 {
 return X + Y;
 }

 public void Yaz()
 {
 Console.WriteLine("X = {0}", X);
 Console.WriteLine("Y = {0}", Y);
 }
}

class Program
{
 static void Main()
 {
 Toplama t = new Toplama(15, 22);
 t.Yaz();

 Toplama t2 = new Toplama(t);
 t2.Yaz();
 }
}
```

## Yıkıcı (Destructors) Metotlar

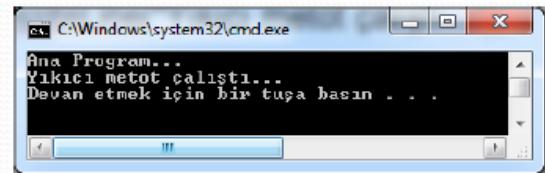
- Nesnelere erişim mümkün olmadığı durumlarda bu nesnelerin heap'ten silinmesi gereklidir, çünkü bu nesneler bellek bölgesi işgaline sebep olur. C++'ta bu işlemi kullanıcı kendi yapmak zorundadır.
- C# ta bu işlem için Gereksiz Nesne Toplayıcısı (Garbage Collector) mekanizması mevcuttur. Bu mekanizma gereksiz nesnelerin tuttuğu referans bölgelerini iade eder, ancak bu nesnelerin faaliyet alanlarının ne zaman iade edileceği kesin olarak bilinmez.
- Bu durumda yıkıcı metotlar (destructors) bellek iade işleminden hemen önce çalışarak bu belirsizliğin çözümünde yardımcı olurlar.

## Yıkıcı (Destructors) Metotlar

- Yıkıcı metotlar bildirilirken sınıf isminin önüne “~” (tilda) işaretinin eklenir.
- Herhangi bir dönüş değeri ve parametresi yoktur.
- Erişim belirteci (public, private) de kullanılmaz.
- Bir sınıfın sadece bir tane yıkıcı metodu olabilir.

## Yıkıcı (Destructors) Metotlar

```
using System;
class Yikici
{
 ~Yikici()
 {
 Console.WriteLine("Yıkıcı metot çalıştı...");
 }
}
class Program
{
 static void Main()
 {
 Yikici y = new Yikici();
 Console.WriteLine("Ana Program...");
 }
}
```

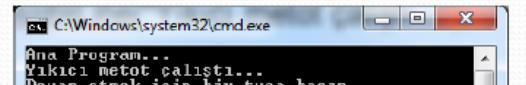


## Yıkıcı (Destructors) Metotlar

- Örnek programda `y` nesnesi faaliyet alanını doldurmasına rağmen hemen yok edilmemiştir.
- Bir alt satırda bulunan `Console.WriteLine()` metodu çalışmış ardından gereksiz nesne toplayıcısı işleyerek nesneyi silmiştir.
- Nesne yok edilirken de yıkıcı metodu çalıştırılmıştır.
- Gereksiz nesne toplayıcı (Garbage Collector) programının isteği dışında çalışmaktadır.
- `System.GC sınıfı` kullanılarak istenilen anda bu mekanizma çalıştırılabilir. Fakat bu tavsiye edilen bir kullanım değildir.

## Yıkıcı (Destructors) Metotlar

```
• using System;
• class Yikici
• {
• ~Yikici()
• { Console.WriteLine("Yıkıcı metot çalıştı..."); }
• }
• class Program
• {
• static void Main()
• {
• Yikici y = new Yikici();
• GC.Collect();
• Console.WriteLine("Ana Program...");
• }
• }
```



Yani özetle yıkıcı metod yapıçı metod gibi sınıfla aynı ismi taşır ve tüm program bittiğinde çalışır GC.Collect'de bu işlev sahipdir bu mekanizmayı çalıştırır. Fakat bu tavsiye edilen bir kullanım değildir.

# Dispose anahtar kelimesi

- Nesnelerden türetilen referanslarla ilgili işler bittikten sonra hafızadan kaldırılmasını sağlar
- Bu kullanılırsa yıkıcı metod çalışmaz
- Bunun kullanılmak istediği nesne IDisposable arayüzünden türetilir
- Ve dispose() metodu overread edilir
- Overload işleminde GC.SuppressFinalize(this) kodu yazılır.
- Artık istenildiği zaman nesnenin tüm bilgileri hafızadan silinebilir.

```
namespace Samples
{ class Ogrenci: IDisposable
 { string _Adi, _SoyAdi, _DogumTarihi;

 public string Adi
 { get { return _Adi; } set { _Adi = value; } }

 public string SoyAdi
 { get { return _SoyAdi; } set { _SoyAdi = value; } }

 public DateTime DogumTarihi
 { get { return _DogumTarihi; } set { _DogumTarihi = value; } }

 public void Dispose()
 {
 GC.SuppressFinalize(this);
 }
 }

 public static void Main()
 {
 Ogrenci insOgrenci = new Ogrenci();
 insOgrenci.Adi = "onur";
 insOgrenci.SoyAdi = "yilmaz";

 insOgrenci.Dispose();
 }
}
```

Görüldüğü gibi hafızadaki yerler geri verilir insOgrenci referansı üzerinden Dispose metodu çağrılarak.(not:disposable:yok edilebilir)

## Yıkıcı (Destructors) Metotlar

- C#’da yıkıcı metotlar genelde nesnenin kullandığı bellek alanını iade etmek için kullanılmaz.
- Daha çok nesne yok edilirken bir takım işlemlerin yapılması, özellikle sınıfın global üye değişkenlerinin değerlerinin işlenmesi ya da değiştirilmesi için kullanılır.

## Statik Üye Elemanlar

- Daha önce de belirtildiği gibi metodlara sınıflar üzerinden erişilir. Bazı durumlarda da ise sınıf türünden nesneler oluşturulur ve bu nesneler üzerinden ilgili metoda erişebiliriz.
- Statik elemanlar bir sınıfın global düzeydeki elemanlarıdır. Yani statik üyeleri kullanmak için herhangi bir nesne tanımlamamıza gerek yoktur.
- Bir üye elemanın statik olduğunu bildirmek için bildirimden önce **static** anahtar sözcüğü eklenir.
- Bir sınıf nesnesi oluşturulduğunda static üye elemanlar için bellekte ayrı bir yer ayrılmaz. (derleme zamanında zaten static bölümünde yerleri ayrılmıştır)

Başlangıç

## Statik Metotlar

- Nesne ile doğrudan iş yapmayan metotlar statik olarak tanımlanabilir.
- Statik tanımlanan metotlara **SınıfAdı.Metot** şeklinde erişilir.
- Statik metotlara nesne üzerinden erişmek mümkün değildir.

```
using System;
class AritmetikIslem
{
 public static int Topla(params int[] dizi)
 {
 int toplam = 0;
 foreach (int eleman in dizi) toplam += eleman;
 return toplam;
 }
}
class Program
{
 static void Main()
 {
 Console.WriteLine(AritmetikIslem.Topla(1, 5, 9, 15));
 }
}
```

```
using System;
class AritmetikIslem
{
 public static int Topla(params int[] dizi)
 {
 int toplam = 0;
 foreach (int eleman in dizi) toplam += eleman;
 return toplam;
 }
}
class Program
{
 static void Main()
 {
 AritmetikIslem islem = new AritmetikIslem();
 Console.WriteLine(islem.Topla(1, 5, 9, 15));
 }
}
```

### DİKKAT:

REFERANS ÜZERİNDEN ERIŞİLMEZ NESNE YANI SINIF  
ÜZERİNDEN ERIŞİLİR STATIC METODLARA EĞER  
REFERANS ÜZERİNDEN ERIŞİM DENENİRSE BU  
HATADIR.BU DURUM STATİK DEĞİŞKENLER İÇİNDE  
GEÇERLİDİR REFERANS ÜZERİNDEN ÇAĞIRMAK  
YANLIŞDIR DİREK OLARAK CLASS İLE ÇAĞIRILMALIDIR.

- Main() metodu da herhangi bir nesneye ihtiyaç duymadan çalışabilmesi için static olarak tanımlanmaktadır. Eğer static olarak tanımlanmasaydı metodun çalışabilmesi için içinde bulunduğu sınıf türünden bir nesneye ihtiyaç olacaktı. Bu durumda da Main metodu işlevi ile çelişecekti. Çünkü Main metodu programımızın çalışmaya başladığı yerdir.

- static anahtar sözcüğü erişim belirleyiciden önce ya da sonra yazılabilir.
- static tanımlanmış metodlar içinden diğer static metodlar çağrılabılır, normal metodlar çağrılamaz. Normal metodlar nesne gerektirdiği için nesne adreslerine ihtiyaç duyarlar (bu adresler gizli şekilde ya da this ile aktarılır).
- static metodlar sınıfın global metodlarıdır ve referansları yoktur. Bu yüzden static bir metot içinden normal bir metot çağrırmak geçersizdir.

91

## Statik Metotlar

```
using System;

class StatikMetot
{
 public void SMetot1()
 {
 Console.WriteLine("Statik Metot 1");
 }

 public static void SMetot2()
 {
 // Statik Metot 2;
 Console.WriteLine("Statik Metot 2");
 }
}

class Program
{
 static void Main()
 {
 StatikMetot.SMetot2();
 }
}
```

- Örneklerden görüldüğü gibi static olmayan elemanlara ulaşabilmek için bir nesne olmalıdır. Eğer bir nesne var ise static olmayan elemanlara erişilebilir.

- ▶ Herhangi bir nesne ile statik değişkenlere ulaşamamız mümkün değildir. Statik olarak tanımlanmış olan değişkenlere ancak sınıfın ismi yardımıyla ulaşılabilir.
- ▶ **SınıfAdı.değişken** şeklinde erişilir.
- ▶ Statik değişkenler sınıf içerisinde global özellikler tanımlamak için kullanılırlar.
- ▶ Bu değişkenler tanımlandıklarında varsayılan değere atanırlar.
- ▶ Statik üyelere ancak statik bir metot içerisinde erişilebilir.

```

using System;
namespace ConsoleApplication2
{
 class Statik
 {
 public static int b = 10;
 public static int deneme()
 {
 int a = 5; b = a;
 return b;
 }
 }
 class Program
 {
 static void Main(string[] args)
 {
 Statik b = new Statik(); // gereksizdir ama hata vermez.
 Console.WriteLine(b.deneme()); // Hata verir, Statik metodlara nesne üzerinden
 // erişmek mümkün değildir.
 Console.WriteLine(Statik.deneme()); // deneme metoduna erişim var. 5 değerini yazar
 Console.WriteLine(Statik.b); // b değişkenine erişim var. 5 değerini yazar
 Console.WriteLine(Statik.a); // Hata verir, static veya statik olmayan metod
 // içerisindeki değerlere erişim yoktur.
 Console.ReadLine();
 }
 }
}

```

## Statik Yapıçı Metotlar

- Normal metotlar gibi yapıçı metotlar da static olabilirler
- **Statik yapıçı metotlar** bir sınıfın statik değişkenleri ile ilgili işlemler yapmada kullanılırlar.
- Bir nesne ilk defa yaratıldığında **statik üye** değişkenini değiştirmek için genellikle **statik yapıçı metotlar** kullanılır.
- **Statik yapıçı metotlar ilk nesne tanımlandığında çalıştırılır.**

## Statik Yapıçı Metotlar

```
• using System;
• class Deneme
• { static Deneme()
• { Console.WriteLine("Static metot çağrıldı."); }
• Deneme()
• { Console.WriteLine("Static olmayan metot çağrıldı."); }
• static void Main()
• { Deneme a=new Deneme();
• Deneme b=new Deneme();
• }
• }
• Bu program ekrana şunları yazacaktır:
• Static metot çağrıldı.
• Static olmayan metot çağrıldı.
• Static olmayan metot çağrıldı.
```

static yapıçı metod normal yapıçı metodla beraber bir kere çalıştı daha sonraki tanımlamalarda static metot artık çalıştığı için bir daha çalışmaz artık normal metot çalışır

a oluşturulduğunda static metod çalıştırıldığı için bir daha çalıştırılmaz artık normal metot çalışır.

Gördüğünüz gibi bir sınıf türünden bir nesne yaratıldığında önce static metot sonra (varsayıf) static olmayan metot çalıştırılıyor. Sonraki nesne yaratımlarında ise static metot çalıştırılmıyor.

## HATIRLATMA:

YAPICI VE YIKICI METODLAR GERİ DÖNÜŞ DEĞERİ ALMAZ.YANI INT,STRING V.S.[VOID'DE DAHİL]

```

6 references
class Mavi
{
 0 references
 static Mavi()
 {
 Console.WriteLine("statik yapıcı çalıştı");
 }
 2 references
 public Mavi()
 {
 Console.WriteLine("normal yapıcı çalıştı ");
 }

 0 references
 static void Main(string[] args)
 {
 Mavi refer1 = new Mavi();
 Mavi refer2 = new Mavi();
 Console.ReadLine();
 }
}

```

```

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlar
statik yapıcı çalıştı
normal yapıcı çalıştı
normal yapıcı çalıştı

```

Yıkıcı metod'da olsa GC.Collect'de olsa  
dispose'de olsa hiçbirsey fark etmez durum  
yine bu şekilde olur.

Aşağıdaki kodların çıktılarında yine ilginçtir statik yapıcı  
görüldüğü gibi parametresine uymasa da bir kereye mahsus  
olmak üzere çalışıyor.[Static metodların parametresi yoktur.]

```

class Mavi
{
 int a;
 0 references
 static Mavi()
 {
 Console.WriteLine("statik yapıcı çalıştı");
 }
 1 reference
 public Mavi(int x)
 {
 Console.WriteLine("normal yapıcı çalıştı ");
 this.a = x;
 }
 0 references
 static void Main(string[] args)
 {
 int b = Convert.ToInt32(Console.ReadLine());
 Mavi refer1 = new Mavi(b);
 Console.WriteLine(refer1.a);
 Console.ReadLine();
 }
}

```

```

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlar
statik yapıcı çalıştı
45
normal yapıcı çalıştı
45

```

# STATIC SINIFLAR:

Bir sınıf statik olarak bildirildiğinde o sınıf türünden bir nesne yaratılamaz. Statik metodlar C# 2.0'dan itibaren dile eklenmiştir.

Statik sınıflar içerisinde static olmayan metot veya özellik tanımlanamaz.

Static sınıfların yapıcı metotları olmaz. Dolayısıyla yapıcı metot tanımlamaya çalışırsak derleyici hata verir.

Daha sonra bahsedilecek olan nesne konusundaki türeme başlığında tekrar bahsedeceğimiz üzere; **static sınıflar türetmeyi desteklemez**.

```
class Urun
{
 public int Id;
 public string Ad;
 public double Fiyat;
 public static double KdvOran;
}
```

KdvOran static tanımlandığından bir kez kullanıldı. Eğer 1000 ürün olsaydı kullanımın ne kadar önemli olduğu daha net anlaşılabilir.

```
class Program
{
 static void Main(string[] args)
 {
 Urun urun1 = new Urun();
 urun1.Id = 1;
 urun1.Ad = "Monitör";
 urun1.Fiyat = 200;

 Urun urun2 = new Urun();
 urun2.Id = 2;
 urun2.Ad = "Klavye";
 urun2.Fiyat = 80;

 Urun.KdvOran = 0.18;
 }
}
```

## const ve readonly Elemanlar

- Değişkenler konusunda gördüğümüz **const** anahtar sözcüğü özelliklerle de kullanılabilir.
- const olarak tanımlanan özellikler aynı zamanda static'tir. Dolayısıyla const özellikleri **tekrar static anahtar sözcüğüyle belirtmek hatalıdır**, const özellikler static özelliklerin taşıdığı tüm özellikleri taşır.
- const** anahtar sözcüğü yalnızca object dışındaki özelliklerle (ve değişkenlerle) kullanılabilir. Dizilerle vs. kullanılamaz. **const** özellikler tanımlanır tanımlanmaz bir ilk değer verilmelidir.
- Sabit tanımlanmış üyeleri de tipki statik tanımlılar gibi sınıfın genel elemanlarıdır ve nesne üzerinden erişilemez.
- Sabitlere erişmek için **SınıfAdı.Sabit** şeklinde kullanılırlar.

Yani referans üzerinden erişilemez. Haliyle static olan bir yapı hafızada tek bir tane ise bundada durum aynıdır

Referans tiplerin (object, diziler, nesneler) değerlerinin çalışma zamanında ayarlanmasıdan dolayı, referans tipler sabit olamaz.

**string** veriler bu kuralın dışındadırlar. Yani **const** ile tanımlanabilirler.

Referans tipleri sabit olarak tanımlamak için **“readonly”** anahtar sözcüğü kullanılır.

```
using System;
class Sinif
{
 readonly int[] a={1,5,8};
 readonly object b=5;
 static void Main()
 {
 Sinif n=new Sinif();
 Console.WriteLine(n.a[0]);
 Console.WriteLine(n.b);
 }
}
```

Referans türlerinde sabitlik bu şekilde sağlanır  
Bunlar artık sadece okunabilirdir.

Statik ifadelere de **readonly** eklenirse referans tipleri **const** gibi tanımlanmış olur.

**Static readonly** ve **const** olarak tanımlanmış değişkenlerin tek farkı ilk değer verilme zamanında ortaya çıkar.

**const** değişkenlerine derleme zamanında **static readonly** değişkenlerine ise çalışma zamanında ilk değer verilir.

Referans tipinde olan ve derleme zamanında atanan çalışma zamanında değişmeyen nesneyi static readonly sağlar.

Şimdi bir de salt okunur nesne oluşturalım:

```
using System;
class Sinif
{
 int a=5;
 static readonly Sinif nesne=new Sinif();
 static void Main()
 { Console.WriteLine(nesne.a); }
}
```

//Şimdilik nesnelerin salt okunur olması saçma gelebilir. Çünkü salt okunur nesnelerle ulaştığımız özellikleri değiştirebiliriz. Salt okunur yaptığımız nesnenin kendisidir.

readonly ile ilgili önemli bilgiler:

readonly anahtar sözcüğü bir metot bloğunun içinde kullanılamaz.

readonly anahtar sözcüğü bir nesne için kullanılacaksa static sözcüğü de kullanılmalıdır.

readonly ve static sözcüklerinin sırası önemli değildir.

readonly bir dizi ya da object türü ayrıca static olarak belirtilebilir.

readonly anahtar sözcüğü const' un kullanılabilıldığı int gibi türlerle de kullanılabilir.

readonly'nın const'un aksine özellikleri staticleştirme özelliği yoktur.

static sözcüğü de readonly gibi bir metot içinde kullanılamaz.

## HATIRLATMA:

REFERANS TİPİ:object,diziler,string[string bazı koşullarda sorun yaratabilir mesela metoda adresi değil kopyası yollanır o yüzden tam olarak referans tipi tanımıyla ilgili istisnaları vardır.]

TEMEL TİP:int,double,bool v.s.

NESNE:Bizim referans ile tanımladığımız sınıflardır referans tipinden olarak kabul edilirler ve metoda adres yollama adres üzerinden işlem yapma gibi özellikleri referans tipinin özellikleriyle örtüşür.

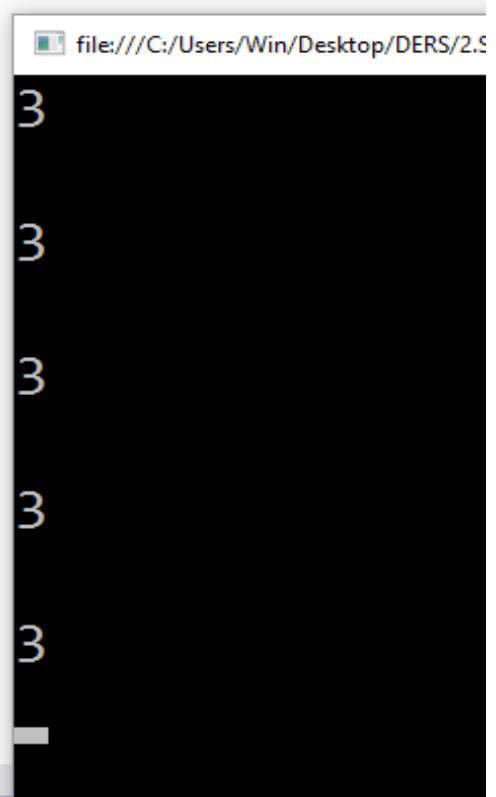
NOT:

Aşağıdaki durumda random hep aynı değer olur.

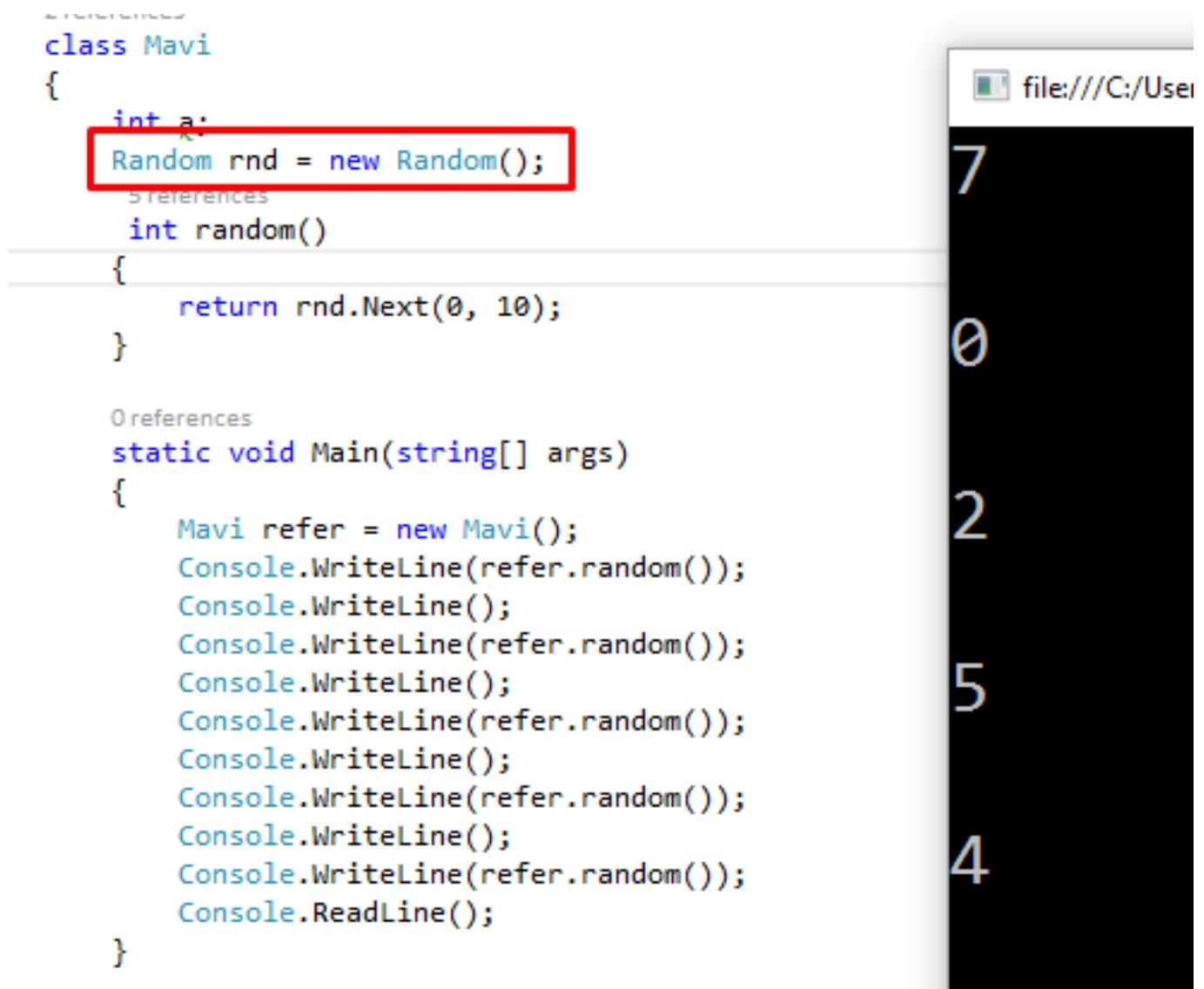
```
class Mavi
{
 int a;

 5 references
 int random()
 {
 Random rnd = new Random();
 return rnd.Next(0, 10);
 }

 0 references
 static void Main(string[] args)
 {
 Mavi refer = new Mavi();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.ReadLine();
 }
}
```



Bu durum şu şekilde düzelttilir:



```
class Mavi
{
 int a;
 Random rnd = new Random(); // Reference count: 5
 int random()
 {
 return rnd.Next(0, 10);
 }

 static void Main(string[] args)
 {
 Mavi refer = new Mavi();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.WriteLine();
 Console.WriteLine(refer.random());
 Console.ReadLine();
 }
}
```

The code editor shows a C# program. A red box highlights the line `Random rnd = new Random();` in the constructor. To the right of the editor, a terminal window displays five random integers generated by this code: 7, 0, 2, 5, and 4.

Bay

# Aşırı yüklenen operatörler

## [OPERATOR OVERLOADİNG]

### Operatör Metotları Tanımlanırken Uyulması Gereken Kurallar

- 1- Operatör metotları **static** olarak tanımlanmalıdır.
- 2- Operatör metotları isimlerinde “**operator**” anahtar sözcüğü kullanılmalıdır. (**operator+**, **operator\*** gibi)
- 3- Bütün operatör metotları tek ya da iki parametre almalıdır.
- 4- Klasik metotlar gibi, operatör metotları da aşırı yüklenebilir.
- 5- Operatör metotlarının da parametrelerden biri mutlaka sınıf türünden olmalıdır.
- 6- Operatör metodlarında **ref** ve **out** anahtar sözcükleri kullanılmamalıdır.

- public static dönüş-tipi **operator op** (parametre-tipi parametre)

**Dönüş tipi:** Herhangi bir tip olabileceği gibi operatörün aşırı yüklemekte olduğu sınıf ile aynı tipte olabilir. Genellikle tercih edilen sınıf ile aynı tipte olmasıdır.

**Parametre-tipi :** **ref** ve **out** anahtar kelimeleri hariç dönüş tipi kullanılabilir. Sınıf tipinde de olabilir.

**Op:** Aşırı yüklenecek operatör (+, /, \*, -).

```

using System;

class KarmasikSayi
{
 private double mGercek;
 private double mSanal;

 public double Gercek
 {
 get { return mGercek; }
 set { mGercek = value; }
 }

 public double Sanal
 {
 get { return mSanal; }
 set { mSanal = value; }
 }

 public KarmasikSayi(double x, double y)
 {
 mGercek = x;
 mSanal = y;
 }

 public KarmasikSayi()
 {
 mGercek = 0;
 mSanal = 0;
 }
}

public KarmasikSayi(KarmasikSayi k)
{
 mGercek = k.mGercek;
 mSanal = k.mSanal;
}

public void Yaz()
{
 if (mSanal > 0)
 Console.WriteLine("{0}+{1}i",
 mGercek,
 mSanal);
 else
 Console.WriteLine("{0}-{1}i",
 mGercek,
 -mSanal);
}
}

class Program
{
 public static void Main()
 {
 KarmasikSayi k = new KarmasikSayi(-5,-6);
 k.Yaz();
 }
}

```

```

public static KarmasikSayi operator +(KarmasikSayi a,
 KarmasikSayi b)
{
 double gt = a.Gercek + b.Gercek;
 double st = a.Sanal + b.Sanal;
 return new KarmasikSayi(gt, st);
}

```

```

class Program
{
 public static void Main()
 {
 KarmasikSayi k1 = new KarmasikSayi (-5,-6);
 KarmasikSayi k2 = new KarmasikSayi (4, 7);
 KarmasikSayi t = k1 + k2;
 t.Yaz();
 }
}

```



- Dört işlem operatörleri (+, -, \*, /) herhangi bir koşul olmaksızın aşırı yüklenebilirler.

## Aritmetiksel Operatörlerin Aşırı Yüklenmesi

```

public static KarmasikSayi operator +(KarmasikSayi a, double b)
{
 double gt = a.Gercek + b;
 return new KarmasikSayi(gt, a.Sanal);
}

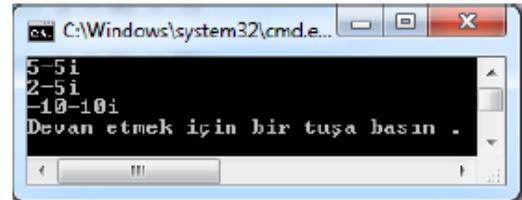
public static KarmasikSayi operator +(double b, KarmasikSayi a)
{
 return a + b; // tekrar üsteki metot çağrılmıyor
}

```

```

class Program
{
 public static void Main()
 {
 KarmasikSayi k1 = new KarmasikSayi(5,-5);
 KarmasikSayi t = 10 + k1;
 t.Yaz();
 KarmasikSayi y = k1 + 7;
 y.Yaz();
 KarmasikSayi z = k1 + k1;
 z.Yaz();
 }
}

```



### NOT:

HEP 2 PARAMETRE ALINIR AMA MESELA 3 TANE YOLLANDIĞINDA  
ŞU ŞEKİLDE OLUR:

$$A+B+C \rightarrow (A+B)+C$$

GİBİ OLUR YANI ÖNCE İLK 2 PARAMETRE YOLLANIR SONRA GERİ DÖNEN DEĞER 3. PARAMETREYLE BİR DAHA YOLLANIR.

```

using System;
class program
{ public int x, y;
 public program (int dd, int ff) { x = dd; y=ff; }

 public void yaz () { Console.WriteLine(x + " " + y); }

 public static program operator +(program a, program b)
 { int c = a.x + a.y; int z = b.x + b.y;
 return new program(c, z);
 }
}
class Sinif
{ static void Main()
{ program a = new program(5,8); program b = new program(7,9);
 program m = new program(6,2);
 program c = a + b + m; c.yaz();
}
}

```

13

## Aritmetiksel Operatörlerin Aşırı Yüklenmesi

```

public static KarmasikSayi operator -(KarmasikSayi a, double b)
{
 double gf = a.Gercek - b;
 return new KarmasikSayi(gf, a.Sanal);
}

public static KarmasikSayi operator -(double b, KarmasikSayi a)
{
 double gf = b- a.Gercek ;
 return new KarmasikSayi(gf, a.Sanal);
}

```

```

class Program
{
 public static void Main()
 {
 KarmasikSayi k1 = new KarmasikSayi(5,-5);
 KarmasikSayi t = 10 - k1;
 t.Yaz();
 KarmasikSayi y = k1 - 7;
 y.Yaz();
 KarmasikSayi z = k1 - k1;
 z.Yaz();
 }
}

```

## Aritmetiksel Operatörlerin Aşırı Yüklenmesi

- Çarpma ve bölme işlemleri için de aynı yaklaşımla operatör metodları oluşturalım.

```
public static KarmasikSayi operator *(KarmasikSayi a,
KarmasikSayi b)
{
 double sanal1 = a.Gercek * b.Sanal;
 double sanal2 = a.Sanal * b.Gercek;
 double sc = sanal1 + sanal2;

 double gercek1 = a.Gercek * b.Gercek;
 double gercek2 = a.Sanal * b.Sanal;
 double gc = gercek1 - gercek2;

 return new KarmasikSayi(gc, sc);
}
```

## DİKKAT EDİLMELİDİR Kİ:

- C# dilinde işlemli atama operatörleri (**`+=`** , **`&&`**, **`||`**, **`[]`**, **`()`**, **`=`**, **`?:`**, **`is`** , **`sizeof`**, **`new`**, **`typeof`** gibi operatörler aşırı yüklenemez.

# İLİŞKİSEL PARAMATRELERİN AŞIRI YÜKLENMESİ:

İlişkisel operatör metodları ( $\neq$ ,  $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ), **true** veya **false** değer ile geri dönerler.

Bu operatörlerin aşırı yüklenmesinde dikkat edilecek tek husus zıt anlamlı operatörlerin her ikisinin de aynı anda yüklenmiş olması gerektidir. Yani **operator ==** bildirilmişse **operator !=** metodu da bildirilmelidir.

```
public static bool operator ==(KarmasikSayi a,
 KarmasikSayi b)
{
 if (a.Sanal == b.Sanal && a.Gercek == b.Gercek)
 return true;
 else return false;
}
```

```
public static bool operator !=(KarmasikSayi a,
 KarmasikSayi b)
{
 return !(a==b);
}
```

## true ve false Operatörlerin Aşırı Yüklenmesi

```
public static bool operator true(KarmasikSayi a)
{
 if (a.Sanal != 0 || a.Gercek != 0)
 return true;
 else return false;
}

public static bool operator false(KarmasikSayi a)
{
 if (a.Sanal == 0 || a.Gercek == 0)
 return true;
 else return false;

}
```

Baybars'

## Mantıksal Operatörlerin Aşırı Yüklenmesi

- Mantıksal operatör metodlarının aşırı yüklenmesini 2 kısımda incelemek mümkündür:
  - **Birinci grup : &, |, ve !**
    - & ve |, ! operatörlerinin aşırı yüklenenebilmesi için herhangi bir şart yoktur
  - **İkinci grup : && ve || operatörleri**
    - && ve || operatörleri direkt olarak aşırı yüklenemezler.
    - Bazı ön şartları vardır.



## Mantıksal Operatörlerin Aşırı Yüklenmesi

- **& ve | operatörleri aşırı yüklenmiş olmalıdır**
- **true ve false operatörlerinin aşırı yüklenmiş olması gereklidir.**
- **operator & ve operator | metodlarının parametreleri ve geri dönüş değeri ilgili sınıfın türünden olmalıdır.**
- Yukarıdaki şartlar sağlandığı takdirde bizim ayrıca && ve || operatörlerini aşırı yüklememize gerek kalmaz. Yukarıdaki şartlar sağlandığı takdirde sanki && ve || operatörleri aşırı yüklenmiş olur.

```

class Sinif
{
 public int Sayi;
 public Sinif(int sayi) { Sayi = sayi; }
 public static bool operator true (Sinif a) { return true; }
 public static bool operator false (Sinif a){ return false; }
 public static Sinif operator &(Sinif a, Sinif b) { return new Sinif(20); }
 public static Sinif operator ||(Sinif a, Sinif b) { return new Sinif(30); }
}
class AnaProgram
{
 static void Main()
 {
 Sinif a=new Sinif(50); Sinif b=new Sinif(10);
 Console.WriteLine((a||b).Sayi);
 Console.WriteLine((a&&b).Sayi);
 }
}

```

DİKKAT

**NOT: GERİ DÖNERKEN**

SINIF X=NEW SINIF()

X İLE İŞLEMLER

RETURN X

YERİNE

RETURN (NEW SINIF())..İŞLEMLER

**(TABİ BU KULLANIM DURUMA GÖRE DEĞİŞİR FAKAT X ÜZERİNDE AZ  
İŞLEM YAPILIYORSA BU DAHA MANTIKLIDIR)**

## Dönüşüm Operatörlerin Aşırı Yüklenmesi

Burada cast işlemini nesneler üzerinde yaptığımızda ortaya çıkan sonuç belirlenir.

Fakat unutulmamalıdır ki operator'un önüne explicit veya implicit yazılmalıdır

```
public static implicit operator HedefTur(DonusturulecekTur)
{
 return HedefTur
}
```

veya

```
public static explicit operator HedefTur(DonusturulecekTur)
{
 return HedefTur
}
```

## implicit'le explicit'in farkı

• KarmasikSayi k = new KarmasikSayi();

• int a = k; → BİLİNÇSİZ TÜR DÖNÜŞÜMÜ [IMPLICIT'E GİDER]

• int b = (int) k; → BİLİNÇLİ TÜR DÖNÜŞÜMÜ [EXPLICIT'E GİDER]

```
int a = k ; /* atamasının geçerli olabilmesi için */
public static implicit operator int(KarmasikSayi k)
{
 return 10; // return k.Gercek ifadesi de olabilirdi.
}
```

Eğer k KarmasikSayi türünden bir nesne ise;

```
int a = (int) k ; /* atamasının geçerli olabilmesi için */
public static explicit operator int(KarmasikSayi k)
{
 return 10; // return k.Gercek ifadesi de olabilirdi.
}
```

// Her iki örnekte de  
k.Gercek double ise  
**return (int) k.Gercek** ile  
bilinçli tip dönüşümü  
yapılmalı. Tip dönüşüm  
kuralları bunun içinde  
geçerli

# OPERATÖRLERİN AŞIRI YÜKLENMESİNİ ÖZETLİYCEK OLURSAK:

- Operatörlerin aşırı yüklenmesi için parametrede en az bir tane nesnemizin olması gereklidir.
- static olmalıdır.[public static'de oluyo olabilir.]
- +,-,\*,/ işlemsel OPERATÖRLERİ DİREK OLARAK AŞIRI YÜKLENEBİLİR
- == ,!= ,<=,<,true,false gibi ilişkisel olan ve bool değer döndüren operatörlerin terside aşırı yüklenmelidir.[mesela == aşırı yüklenmişse !='de aşırı yüklenmelidir]
  - C# dilinde **işlemli atama operatörleri** (**+=** , **&&** , **||** , **[]** , **()** , **=** , **?:** , **is** , **sizeof**, **new**, **typeof** gibi **operatörler aşırı yüklenemez**.
- && , ||,! gibi Mantıksal operatörlerin aşırı yüklenmesinde bazı durumlar vardır:
  - !, &, | operatörleri hiçbir sorun olmadan direkt olarak aşırı yüklenebilir.
  - fakat && ve || aşırı yüklenemez olarak geçiyo olabilir fakat şöyledir bir durum vardır ki:operatörlerinin aşırı yüklenmesi için
  - Aşadaki operatörler aşırı yüklenmelidir
  - true ve false operatörü(dikkat:geri dönüş değerini işlem yapılan nesnenin[sınıfın] türünden olmalıdır)
  - / operatörü
  - & operatörü
  - zaten bu işlemler yapıldığında && ve || operatörlerini aşırı yüklemeye gerek kalmaz
- Temel veri türlerinin cast veya başka türlü aşırı yüklenmesi ise eğer cast yapılıyorsa operator'un öncesine explicit eğer cast yapılmıyorsa implicit yazılır ve ona yollanır.İçerdeki işlemler genelde aynıdır

# İNDEKSLEYİCİLER[INDEXERS]

## Genel Tanımlaması:

ElemanTipi **this** [IndeksTipi indeks]

```
{
 get { return değer; }
 set { işlemler; ...; }
}
```



İndeksleyicilerde indeks değeri dizilerde olduğu gibi tamsayı olmak zorunda değildir. Fakat ideal kullanımda tamsayı olarak seçilmelidirler.

Çok ilginç bir yapısı vardır. Yazan kişinin hayal gücü doğrultusunda çok farklı işler çıkartılabilir bir çeşit metottur denilebilir bunda [] içinde parametre vardı aşırı yüklenme olabilir ve

**sınıf\_referansı[indeks]** şeklinde bir kullanımı vardır içindeki indekslere işlem yaptırılabilir gelen value değerinde işlem yaptırılabilir, denildiği gibi aşırı yükleme mevcuttur yani 2 index için ayrı işlem yapılabilir indexin tipine göre ayrı işlem yapılabilir ve index tipi veya geri dönüş tipi temel tip olmak zorunda değildir nesnelerimizden biride olabilir.

Kullanımı:

Aşağıdaki gibi bir sınıfımız vardır.

```
class beyaz
{
 private int a;
 6 references
 public int this[int indeks]
 {
 get
 {
 return a;
 }
 set
 {
 if (indeks > 0)
 {
 a = value + a;
 }
 else
 {
 a = value;
 }
 }
 }
}
```

BURADA YAPILAN EĞER İNDEKS DEĞERİ POZİTİFSE A'YA ANIN DEĞERİYLE  
ATANAN DEĞERİN TOPLAMINI ATA TOPLA POZİTİF DEĞİLSE DİREK OLARAK A'YA  
ATANAN DEĞERİ[VALUE] ATA

```
0 references
static void Main(string[] args)
{
 beyaz refer1 = new beyaz();
 refer1[89]=14; → POZİTİF
 Console.WriteLine(refer1[54]);
 refer1[65] = 96; → POZİTİF
 Console.WriteLine(refer1[89]);
 refer1[-21] = 23; → NEGATİF
 Console.WriteLine(refer1[75]);
 Console.ReadLine();
}
```

```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım b.
14 a=a+value=0+14
110 a=a+value=14+110
23 a=23
```

Çok boyutlu'da indexler vardır yani

```
public int this[int indeks1,int indeks2....]{get... set...}
```

ve overload'da edilebilirler unutulmamalıdır ki bu  
nesne[indeks1..] şeklinde kullanılır yani sınıfların kendine özgü bir  
özellikini tanımlamak gibidir.

not:this'e dikkat edilmelidir

```
class Sinif
{
 private int Sayi;
 public int this[int i, int j]
 {
 get { return i + j + Sayi; }
 set { Sayi=i * j + value; }
 }
}
class AnaProgram
{
 static void Main()
 {
 Sinif a=new Sinif(); a[5,4]=45; Console.WriteLine(a[-6,12]);
 }
}
```

Görüldüğü gibi kullanımı tamamen hayal gücüne kalmıştır. Kullanım  
mantığı metotların ve get set'lerim kullanım mantığıyla kesişiyor  
denilebilir.

İndeksleyiciler daha çok sınıfın üye dizilerinden birine direkt erişmek için  
kullanılırlar. Diziler dinamik olarak oluşturulabilir. Nesne ismine indeks verilerek üye  
dizinin elemanının değeri elde edilir.

#### NOT:

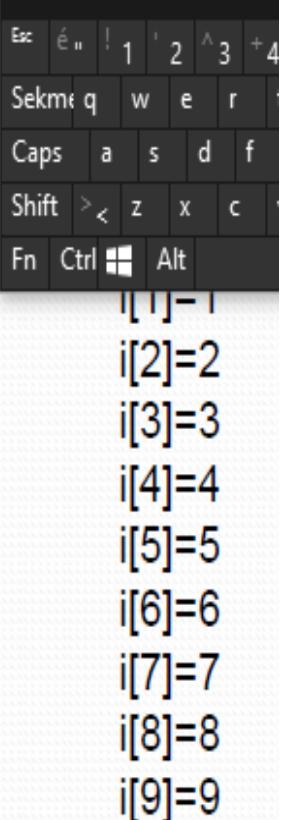
İndeksleyiciler de aşırı yüklenebilir. Bunu ise parametre türünü farklı yaparak  
sağlıyoruz. Fakat bu çok kullanılan bir yöntem değildir.

İndeksleyicilerin parametre ve geri dönüş tipi herhangi bir tip olabilir. int olması şart  
değildir.

İndeksleyicilerin set ve get blokları sahte özelliklerdeki set ve get bloklarının tüm  
özelliklerini taşırlar. Örneğin set bloğunu koymayarak nesnenin indekslerine değer  
atanmasını engelleyebiliriz.

İndeksleyicilerin en yaygın kullanımı sınıfımız içindeki bir dizinin elemanlarına direkt  
nesne ve [] operatörünü kullanarak erişmektir. Bunu mantığınızı kullanarak  
yapabilirsiniz.

```
using System;
class indeksleyici
{ private int[] dizi;
 public indeksleyici(int dizi_uzunluk)
 { dizi = new int[dizi_uzunluk]; }
 public int DiziBoyutu
 { get { return dizi.Length; } }
 public int this[int indeks]
 { get { return dizi[indeks]; }
 set { dizi[indeks] = value; }
 }
}
class AnaProgram
{ static void Main()
 { indeksleyici i = new indeksleyici(10);
 for (int k = 0; k < i.DiziBoyutu; k++)
 Console.WriteLine("i[{0}]={1}", k, i[k] = k);
 }
}
```



Hatırlatma:

get ve set metotları sahte bir değişkene gelen değer üzerinden sınıfındaki değişkenlere gelen değer doğrultusunda işlem yapar veya geri dönüş değerinin kontrollü olmasını sağlar. Biz aslında sınıfındaki değişkenlerle işlem yaparız fakat get ve set metodu sayesinde başka bir değişken buna aracı olur ve işlemler kontrollü bir biçimde gerçekleşir. Fakat kullanıcının direk olarak erişimine imkan verilmeyecekse asıl değişken private yapılır

The screenshot shows a C# code editor with the following code:

```
0 references
static void Main(string[] args)
{
 beyaz refer1 = new beyaz();
 refer1.sahte = -45;
 Console.WriteLine(refer1.sahte);
 Console.ReadLine();
}

references
class beyaz
{
 private int b; //direk olarak erişim engellendi çalışır sorun yok
 2 references
 public int sahte
 {
 get
 {
 return b;
 }
 set
 {
 if (value < 0)
 {
 b = -1 * value;
 }
 else
 {
 b = value;
 }
 }
 }
}
```

The code editor has a red box around the declaration of the private variable 'b'. Another red box surrounds the assignment statement 'refer1.sahte = -45;' and the conditional logic within the 'set' accessor of the 'sahte' property.

file:///  
45

# Yapılar

## (Structs)

C# dilinde yapılar değer tipindedir. Sınıf bildirimine çok benzer şekilde tanımlanırlar. **struct** anahtar sözcüğü kullanılır.

Bazı durumlarda belli bir grup verinin bir arada tanımlanması için sınıfların kullanılması verimsiz olur. Sınıf kullanıldığında stack alanında referans tipte değişken oluşturulur ve üyeleri içinde heap alanında ayrıca bellek ayrıılır. Verilere ulaşmak için referans kullanmak istenmeyebilir.

Bu gibi durumlarda az sayıda ve veri tipindeki değişkenleri yapı şeklinde tanımlamak hız ve verimlilik sağlayabilir.

Oluşturulması:

```
struct Ogrenci
{
 public int Numara;
 public string Ad;
 public string Soyad;
}
```

**Yapılar daha çok birbiri ile ilişkili değerleri bir araya toplamak için kullanılır.**

**Yapılar değer tipidir ve yapı türündeki nesneler stack alanında saklanır.**

**Yapılar da diğer tüm nesneler gibi object sınıfından türetilmiştir.**

**Yapılar kalıtımı desteklemez, türetme yapılamaz.**

Yani özetle nasıl ki sınıflar kendi referans tipinde değişkenimizi oluşturmamızı sağlıyorsa

Yapılarda kendi temel tipteki değişkenimizi oluşturmamızı sağlıyor.

Tabi bunun dışında class'dan ayrılan bazı önemli noktaları da vardır ki bunlara dikkat edilmelidir. Mesela türemelerin olmaması yani kalıtımın yapılarda desteklenmemesi gibi...

**Yapılar tanımlandıktan sonra bir yapı nesnesi oluşturmak için yine new operatörü kullanılır.**

**Varsayılan yapıçı metot ya da bizim belirlediğimiz yapıçı metot çalışarak ilk değer ataması yapar.**

**Sınıflardan farklı olarak new kullanılmadan da yapılar tanımlanabilir.**

Bu şekilde tanımlanan yapı nesnelerinin üye elemanlarına ilk değer elle verilmelidir.

Yapı nesnesinin elemanlarına sınıflarda olduğu gibi ":" ile erişilir.

## Kullanım örneği:

```
using System;

class Yapilar
{
 public struct Ogrenci
 {
 public int Numara;
 public string Ad;
 public string Soyad;
 }

 public static void Metot(Ogrenci o)
 {
 o.Numara = 999;
 }

 public static void Main()
 {
 Ogrenci ogr=new Ogrenci();

 ogr.Numara = 123;
 ogr.Ad = "Ali";
 ogr.Soyad = "Türk";

 Metot(ogr);

 Console.WriteLine("{0} {1} {2}", ogr.Numara, ogr.Ad, ogr.Soyad);
 }
}
```

Baybú

# Dikkat edilmelidir ki:

Yapıların Yapıçı metod'larında varsayılan yani `yapıcı(){...}//hata parametresi boş` gibi parametresi boş olan bir yapıçı metod tanımlanamaz bu zaten vardır ve yapıdaki değişenlere varsayılan değer atar biz bunu yapılarda ayrıca yapamayız bu bir hatadır.Dikkat edilmelidir.Yapıcı metodun görevi değişkenlere varsayılan değer atamaktadır ve eğer tüm değişkenlere varsayılan değer atamıyorsa buda bir hatadır yani parametre(boş parametre değil) alınarak yapılandırıcı oluşturulduysa bu yapılandırıcı içinde yapı'daki tüm değişkenlere değer atanmalıdır.Aksi takdir'de hata ile karşılaşılır.

Ayrıca yıkıcı metot'da bulunmaz.

Sınıflarda olduğu gibi yapılar içinde değişkenler için set-get metotları ile indeksleyiciler kullanılabilir.

```
using System;

class Yapilar
{
 public struct Ogrenci
 {
 public int Numara;
 public string Ad;
 public string Soyad;

 public Ogrenci(int no, string ad, string soyad)
 {
 Numara = no;
 Ad = ad;
 Soyad = soyad;
 }
 }

 public static void Main()
 {
 Ogrenci ogr = new Ogrenci(123, "Ali", "Türk");

 Console.WriteLine("{0} {1} {2}", ogr.Numara, ogr.Ad,
 ogr.Soyad);
 }
}
```

## Notlar:

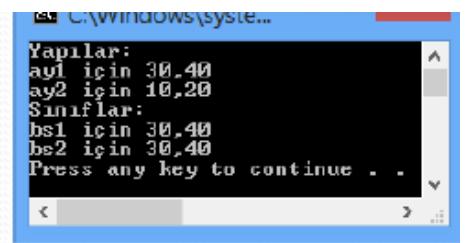
Yapılarda yıkıcı metot olmamasına rağmen faaliyet alanı bitiminde bellekten otomatik silinirler.

## Aşadaki özelliğe dikkat:

Yapıları, **atama operatörü** ile kopyalayabiliriz. Yapılarda, atama işlemi sonucunda değerler kopyalanır, fakat sınıf nesnelerinde sınıfın içeriği değil referanslar kopyalanır.

```
using System;
struct ayapi { public int cx; public int cy;}
class bsinif { public int cx; public int cy;}
class anasınıf
{
 static public void Main()
 { ayapi ay1, ay2;
 ay1.cx = 10; ay1.cy = 20;
 ay2 = ay1;
 ay1.cx = 30; ay1.cy = 40;
 Console.WriteLine("Yapılar:");
 Console.WriteLine("ay1 için {0},{1}", ay1.cx, ay1.cy);
 Console.WriteLine("ay2 için {0},{1}", ay2.cx, ay2.cy);

 bsinif bs1, bs2;
 bs1 = new bsinif();
 bs1.cx = 10; bs1.cy = 20;
 bs2 = bs1;
 bs1.cx = 30; bs1.cy = 40;
 Console.WriteLine("Sınıflar:");
 Console.WriteLine("bs1 için {0},{1}", bs1.cx, bs1.cy);
 Console.WriteLine("bs2 için {0},{1}", bs2.cx, bs2.cy);
 }
}
```



C:\Windows\system...  
Yapılar:  
ay1 için 30,40  
ay2 için 10,20  
Sınıflar:  
bs1 için 30,40  
bs2 için 30,40  
Press any key to continue . . .

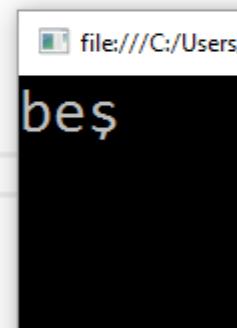
# Numaralandırmalar

## (Enumaration)

Enum diye bir anahtar ile değişken tipi oluştururlar gelecek değere göre değişkene nasıl yansıyacağı belirtilir.

Kullanımı:

```
0 references
class Mavi
{
 2 references
 enum not:byte{sıfır,bir,iki,üç,dört,beş,altı};
 0 references
 static void Main(string[] args)
 {
 not x = (not)5;
 Console.WriteLine(x);
 Console.ReadLine();
 }
}
```



```
• using System;
• class Program
• { enum notu : byte { basarisiz, gecmez, gecer, orta, iyi, pekiyi }
• static void Main()
• { Console.Write("Lütfen notunuzu giriniz: ");
• notu a = (notu)Convert.ToByte(Console.ReadLine());
• Console.WriteLine(a);
• }
• }
```

programımızda byte türünü notu türüne (enum sabitine) bilinçli olarak dönüştürdük. Bilinçsiz olarak dönüştüremezdik. Ayrıca direkt olarak stringten nota dönüşüm yapamazdık. Yalnızca enum sınıfı ile string ve object dışındaki temel veri türleri arasında dönüşüm yapabiliriz.

| Sözcük    | Temsil ettiği sayı |
|-----------|--------------------|
| basarisiz | 0                  |
| gecmez    | 1                  |
| gecer     | 2                  |
| orta      | 3                  |
| iyi       | 4                  |
| pekiyi    | 5                  |

Yani enum'da tanımlanan bir veri tipi ile değişken varsa bu değişkende sayı yazılıyorsa temsil ettiği kelime,kelime yazılıyorsa temsil ettiği sayı alınır.

```
using System;
class Üniversite
{
 enum bolumler
 {
 Yazılım_Müh_,
 Elektrik_Elektronik_Müh_,
 Mektronik_Müh_,
 Bilgisayar_Müh_,
 Makina_Müh_,
 }
 public static void Main()
 {
 bolumler b;
 for (b = bolumler.Yazılım_Müh_; b <= bolumler.Makina_Müh_; b++)
 Console.WriteLine("{0} 'nın etiketi ={1,4:d}", b, (int)b);
 }
}
```

```
C:\Windows\system32\cmd.exe
Yazılım_Müh_ 'nın etiketi = 0
Elektrik_Elektronik_Müh_ 'nın etiketi = 1
Mektronik_Müh_ 'nın etiketi = 2
Bilgisayar_Müh_ 'nın etiketi = 3
Makina_Müh_ 'nın etiketi = 4
```

Bu örnekte `b=bolumler.yazılım_muh` 0'ı temsil eder

Aşağıdaki örnekte de farklı bir gösterim şekli vardır.

```
0 references
class Mavi
{
 1 reference
 enum not:byte{sıfır,bir,iki,üç,dört,beş,altı};
 0 references
 static void Main(string[] args)
 {
 int x;
 x = (byte)not.bir;
 Console.WriteLine(x);
 Console.ReadLine();
 }
}
```

```
file:///C:/Users/Win/Des
1
```

Enumla ilgili ilginç bir özellik vardır ki bu da kelimelerin temsil ettiği sayılar elle yazılabilmektedir:

```
enum not:byte{basarisiz=6,basarili=10}
```

Bu örnekte basarisiz 6'yı, basarili 10'u temsil eder.

Fakat eğer başarılı=10 yerine hiçbirşeye eşitlenmeden bırakılsaydı başarılı otomatik olarak 7'yi temsil ederdi.

```
enum not:byte{basarisiz=6,basarili}
```

Bu örnekte basarisiz 6'yı, basarili 7'yi temsil eder.

ilk parametre birşeye eşit değilse 0'dan başlar ve ne zaman müdahale ediliyorsa o değerden itibaren artarak devam edilir

```
enum not:byte{basarisiz=-21, basarili}
```

Bu örnekte basarisiz -21'i basarili -20'yi temsil eder.

```
enum not:byte{basarisiz, gecmez=5,gecer,orta,iyi=2,pekiyi}
```

Bu örnekte ise şu temsiller söz konusudur

**Atamalar tek ve ilk değerler mantıklı verilmelidir.**

**Bir birini örten değerler veya aynı sabit verilmez.**

| Sözcük    | Temsil ettiği sayı |
|-----------|--------------------|
| basarisiz | 0                  |
| gecmez    | 5                  |
| gecer     | 6                  |
| orta      | 7                  |
| iyi       | 2                  |
| pekiyi    | 3                  |

54

**Çakışmaların olmamasına çok dikkat edilmelidir.**Aksi takdirde hata ile karşılaşılır.

- › enum sabitlerine enumAdı.Sabit şeklinde ulaşılır.
- › enum sabitleri de birer veri tipi oldukları için fonksiyona parametre olarak gönderilebilir.
- › enum static olduğu için başka sınıfın erişildiğiinde nesne oluşturulmaz. Ör: sınıfAdı.enumdeğişkeni şeklinde erişim yapılır. (başka sınıfın erişebilmek için erişim kuralları bunun içinde geçerli (public, private...))
- › System.Enum sınıfından türetildikleri için bazı metodlar ile sabit sembollerin isimleri alınabilir.
- › enum yapısı bir sınıfı ifade ettiği için bir sınıf içinde kullanılma zorunluluğu yoktur. Sınıf içinde olmadığı zaman erişim belirleyici public durumundadır ve sadece enum adı ile erişilir.

```

class Program
{
 public enum Renkler : byte { Siyah, Beyaz, Kırmızı, Mavi, Sarı}
}
class Sinif
{
 enum Gunler : byte { Pazartesi, Salı, Çarşamba, Perşembe, Cuma,
Cumartesi, Pazar }

 static void Main()
 {
 string[] a = Gunler.GetNames(typeof(Gunler));
 Console.WriteLine(a[0]); //veya
 Console.WriteLine(Gunler.GetNames(typeof(Gunler))[3]);

 Program.Renkler b = (Program.Renkler)1;
 Console.WriteLine(b + "," + (byte)b);
 }
}

```



```
using System;
enum Renkler : byte { Siyah, Beyaz, Kırmızı, Mavi, Sarı}
class Sınıf
{
 static void Main()
 {
 Renkler b = (Program.Renkler)1;
 Console.WriteLine(b + "," + (byte)b);
 }
}
Sınıf dışında kullanımında enum varsayılan erişim belirleyicisi public tir.
```

Baybarshan

# İSİM ALANLARI VE SYSTEM

## İSİM ALANI

Java'daki package yapısı burada namespace yani isim alanı olarak karşımıza çıkıyor

C#'da package'lerin altında sınıflar v.b. yapılar tanımlanarak aynı klasör içinde .cs uzantılı olarak kaydedilir.

Daha sonra istenildiği takdirde tüm aynı klasördeki .cs uzantılı namespace çok karmaşık ve uzun kod satırlarından kurtarır bizi ve ayrıca alt klasörlerde başka bir isimdeki namespace oluşturulup çağrılabılır Aşadaki örnekleri inceleyelim

**İlk olarak baybars namespace adı altında bir dosya oluşturduk.**

```
Program.cs
C# baybarspace baybarspace.Pro Main(string[] args)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

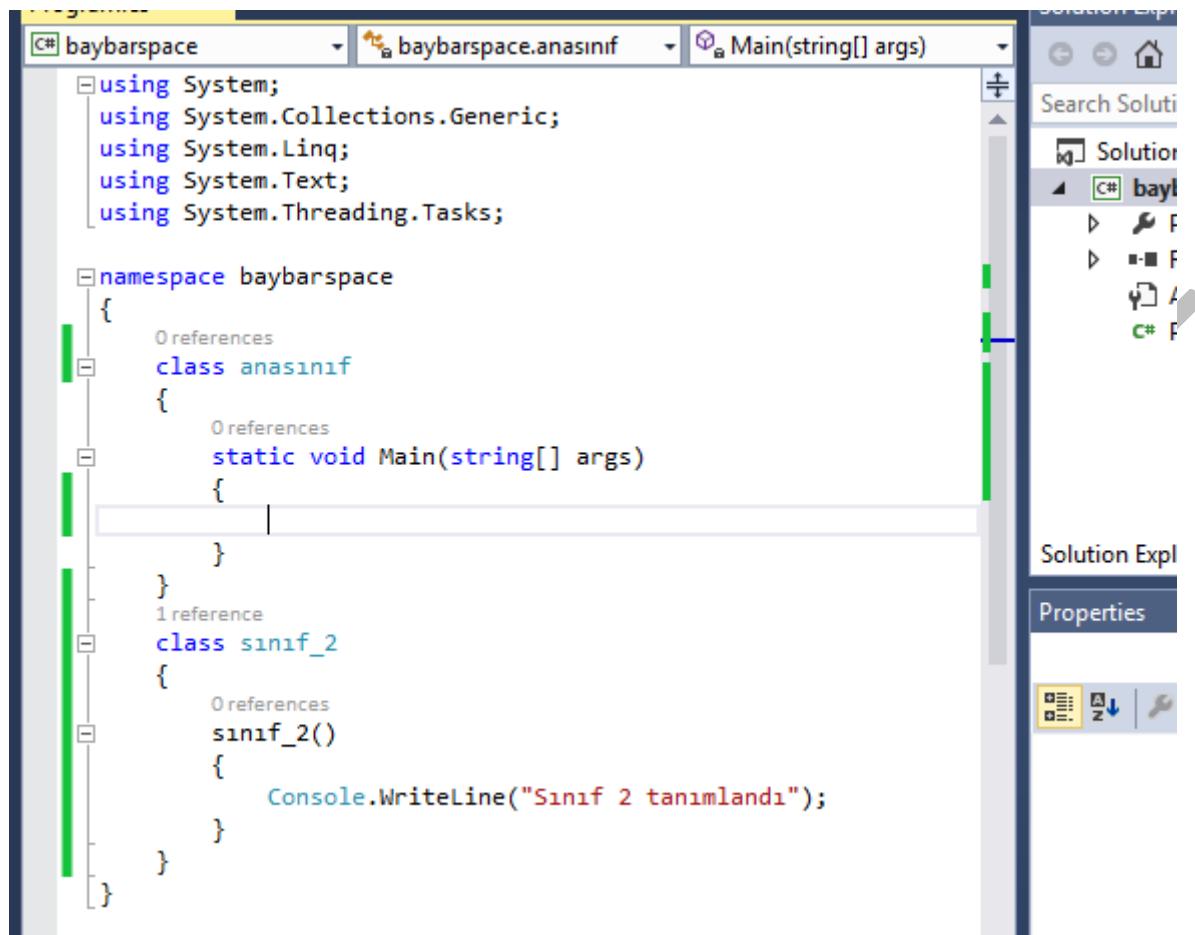
namespace baybarspace
{
 class Program
 {
 static void Main(string[] args)
 }
}
```

Solution Explorer

- Search Solution Explorer (Ctrl+Shift+F)
- Solution 'baybarspace' (1 project)
  - baybarspace
    - Properties
    - References
    - App.config
    - Program.cs

Solution Explorer Team Explorer Class View

Bu baybarspace namespace'ının direkt olarak altına class.v.s. eklenebilir



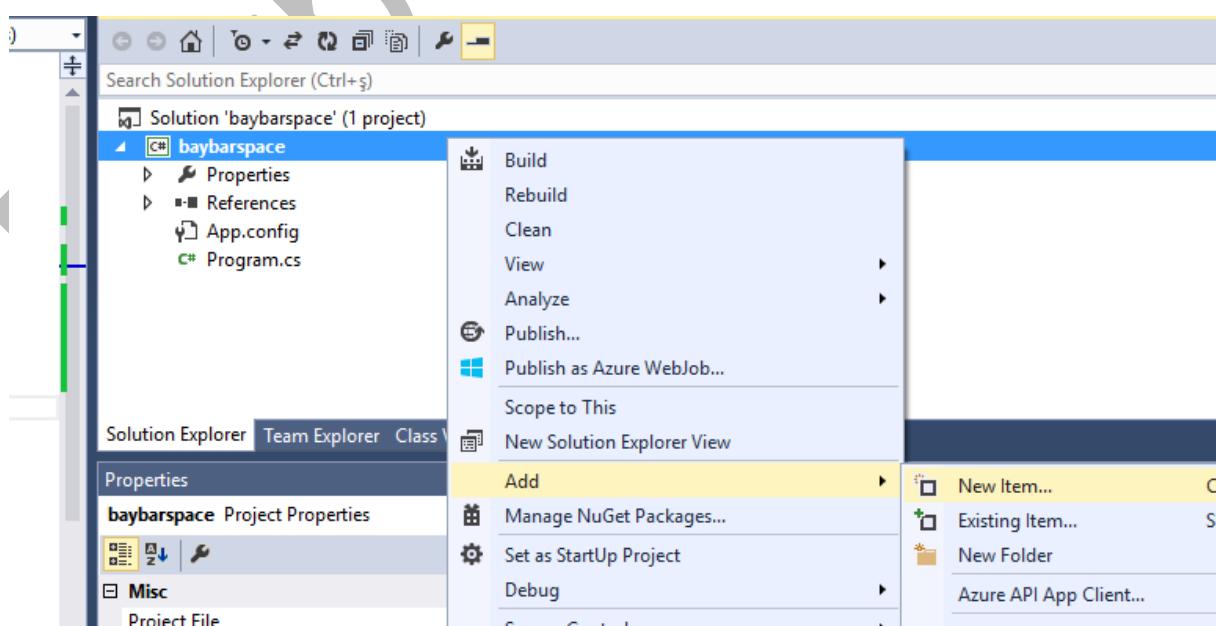
```
C# baybarspace
C# baybarspace.anasınıf
Main(string[] args)

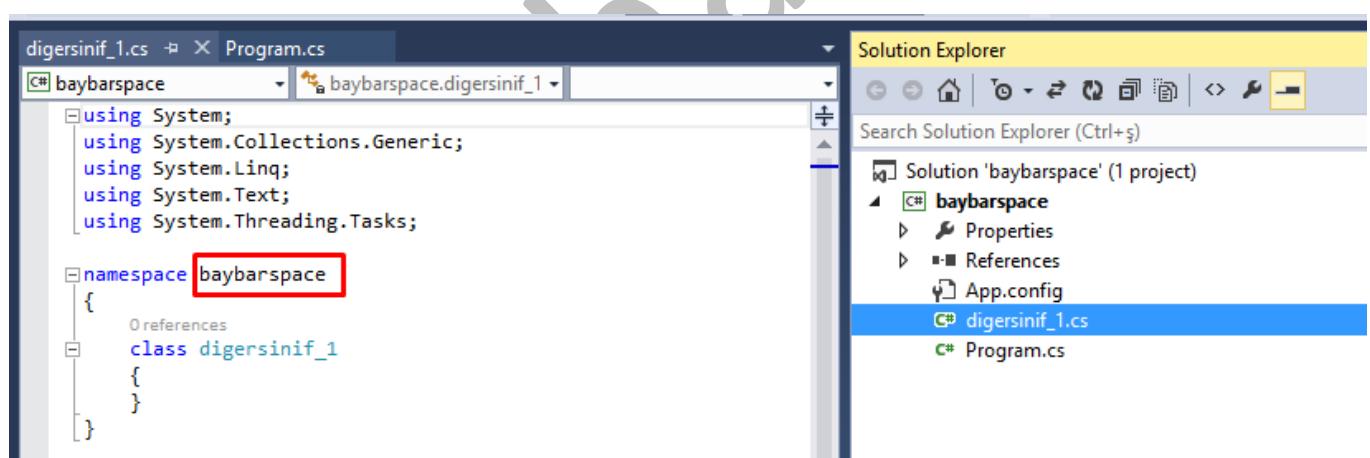
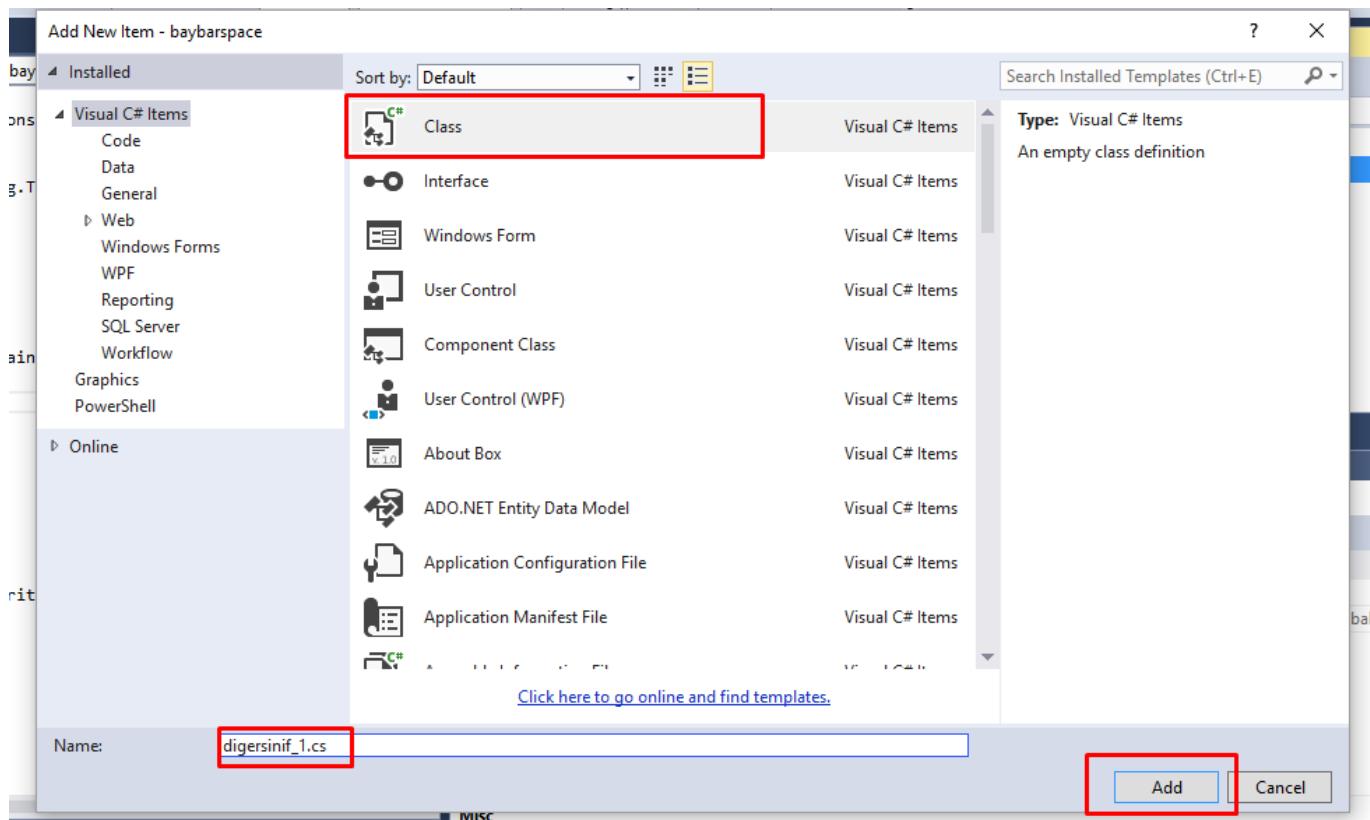
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 }
 }
}

class sınıf_2
{
 sınıf_2()
 {
 Console.WriteLine("Sınıf 2 tanımlandı");
 }
}
```

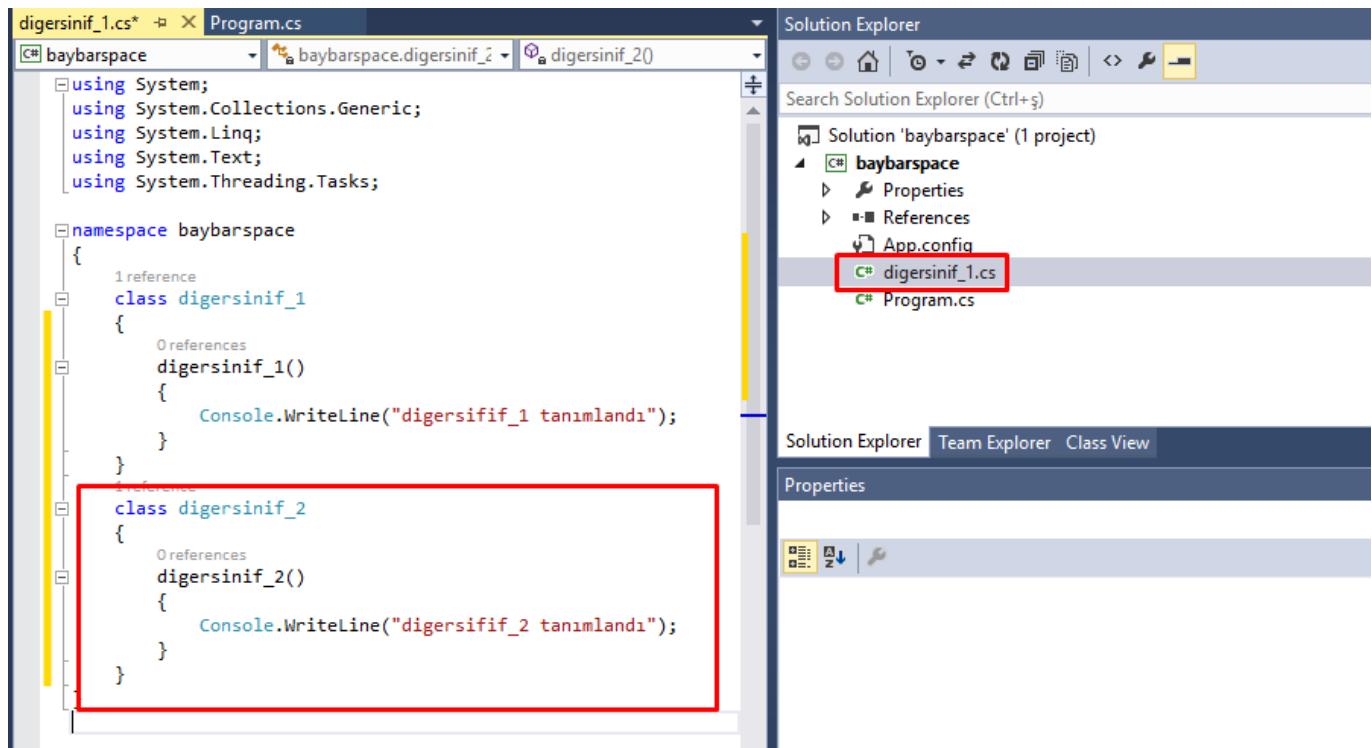
Yada daha farklı bir şekilde şöyle yapılır





görüldüğü gibi baybarspace namespace'inin altında digersinif\_1.cs adı altındabir sınıf daha tanımlandı .

artık istediği gibi burdada sınıflar oluşturulabilir tüm bunlar düzen sağlamak içindir sağ tarafındaki .cs uzantılılarının adları da mesela soyut\_siniflar.cs, arayuzler.cs v.s. şeklinde olabilir. İlk oluşturulduğunda ilk class'ın adıyla oluşturulurlar sağ taraftaki .cs uzantılıları.



Dikkat edilmelidir ki diğer sınıflardan metodlara erişimin olabilmesi için bu metodların public olmaları gereklidir.

```
namespace baybarspace
{
 class digersinif_1
 {
 public digersinif_1()
 {
 Console.WriteLine("digersinif_1 tanımlandı");
 }
 }

 class digersinif_2
 {
 public digersinif_2()
 {
 Console.WriteLine("digersinif_2 tanımlandı");
 }
 }
}
```

Aşağıda görüldüğü gibi hiçbir sorun yaşanmadan tanımlama yapıldı.

The screenshot shows a C# code editor with the following code:

```
namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 Console.ReadLine();
 }
 }
}
```

To the right of the code editor is a terminal window displaying the output of the program:

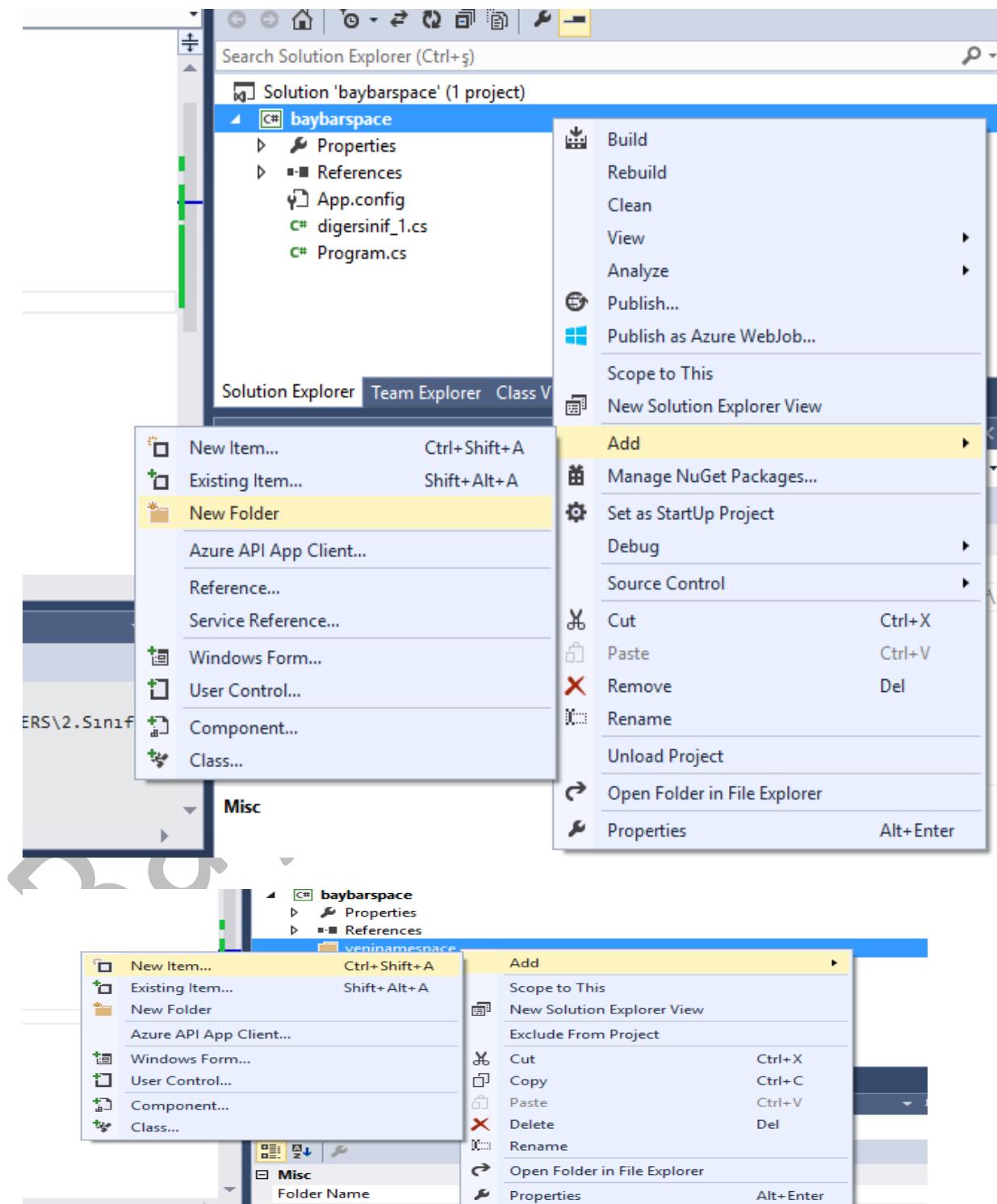
```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf
digersifif_2 tanımlandı
```

Klasör'deki görünümse bu şekildedir

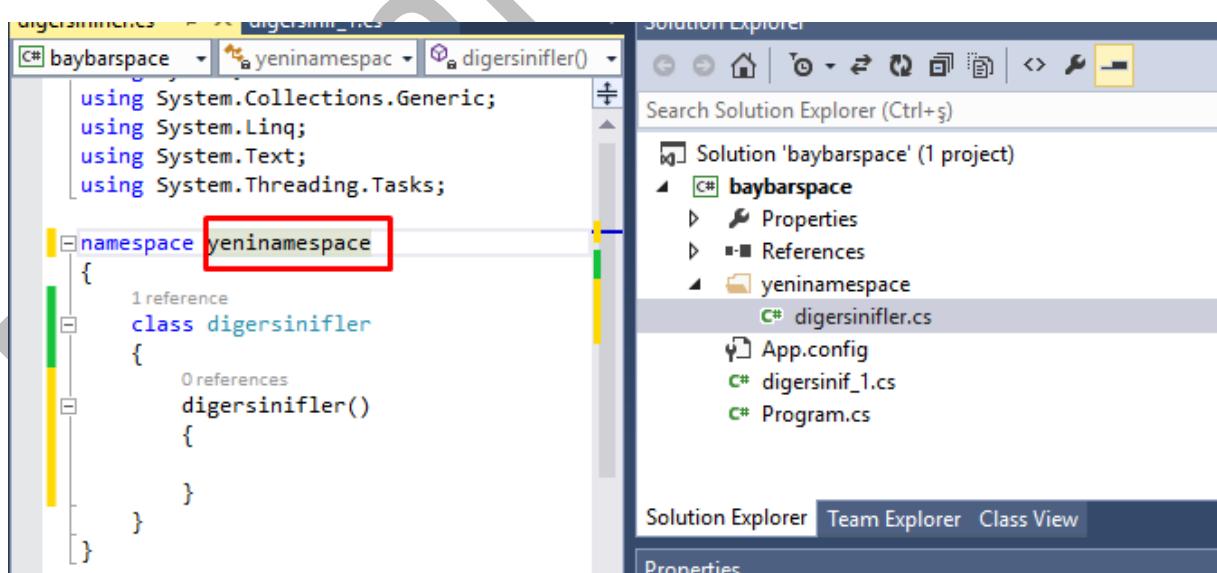
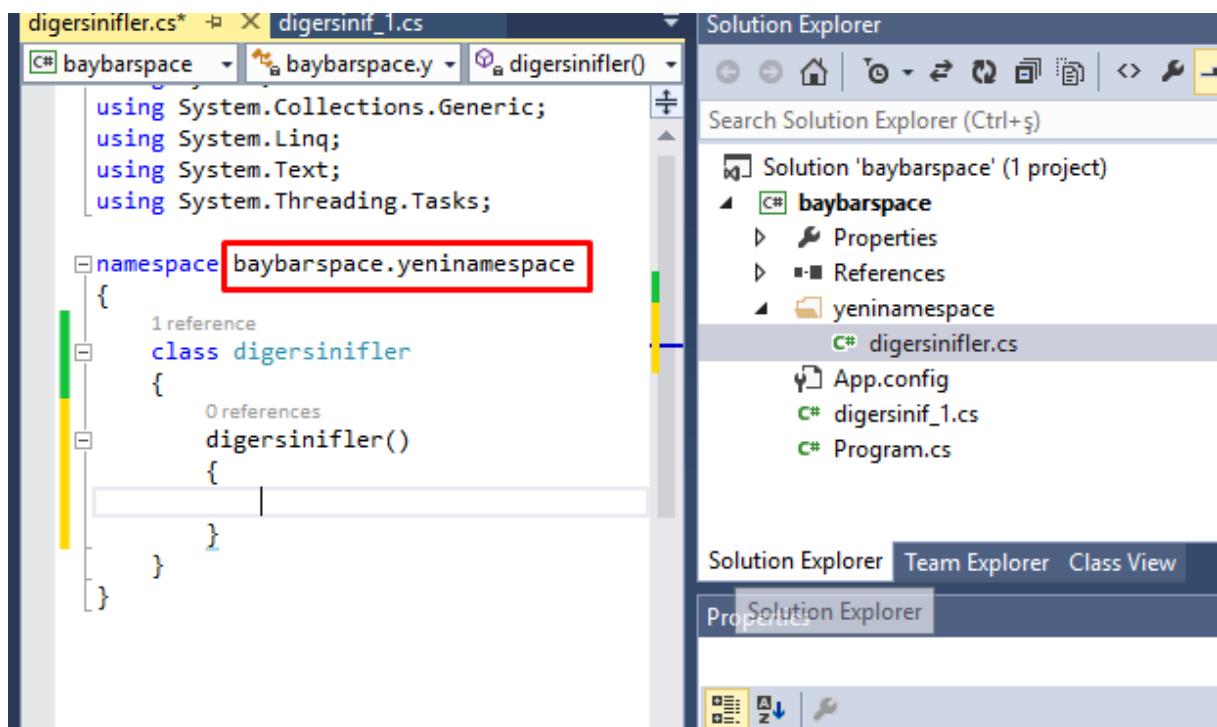
|                    |                  |                        |
|--------------------|------------------|------------------------|
| bin                | 29.03.2016 01:03 | Dosya klasörü          |
| obj                | 29.03.2016 01:03 | Dosya klasörü          |
| Properties         | 29.03.2016 01:03 | Dosya klasörü          |
| App.config         | 29.03.2016 01:03 | XML Configuratio...    |
| baybarspace.csproj | 29.03.2016 01:16 | Visual C# Project f... |
| digersinif_1.cs    | 29.03.2016 01:18 | Visual C# Source f...  |
| Program.cs         | 29.03.2016 01:19 | Visual C# Source f...  |

dikkat edildiyse aynı namespace altındakiler aynı klasördedirler buna dikkat edilmeli.

# Şimdi daha farklı bir namespace tanımlayıp birbirleriyle using kullanarak bağlantı kurulacak



Oluşturulan namespace bir önceki namespacenin alt dizini gibi olsun istemiyorsak şu düzenlemeyi yapabiliriz:



The screenshot shows a Visual Studio interface with two tabs open: 'digersinifler.cs' and 'digersinif\_1.cs'. The code editor displays a class definition within a namespace 'yeninamespace'. The Solution Explorer on the right shows a project named 'baybarspace' containing files 'App.config', 'digersinif\_1.cs', and 'Program.cs'. A red box highlights the 'yeninamespace' declaration in the code editor, and another red box highlights the 'digersinif\_1.cs' file in the Solution Explorer.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace yeninamespace
{
 class digersinifler
 {
 digersinifler()
 {
 Console.WriteLine("diğer name space tanımlandı");
 }
 }
}
```

aynı namespace altında başka bir .cs  
tanımlayıp orda class oluşturalım

**DİKKAT:ERİŞİMLER UNUTULMAMALIDIR.**

The screenshot shows a Visual Studio interface with three tabs open: 'baska.cs', 'digersinif\_1.cs', and 'Program.cs'. The code editor displays a class 'baska' within the 'yeninamespace'. The Solution Explorer on the right shows a project named 'baybarspace' containing files 'App.config', 'baska.cs', 'digersinif\_1.cs', and 'Program.cs'. A red box highlights the 'yeninamespace' declaration in the code editor, and another red box highlights the 'baska.cs' file in the Solution Explorer. A red arrow points from the word 'DİKKAT' in the code editor to the 'baska.cs' file in the Solution Explorer.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace yeninamespace
{
 class baska
 {
 baska()
 {
 Console.WriteLine("baska sınıfı tanımlandı");
 }
 }
}
```

**using ile yukarıda tanımlanarak artık aynı namespacedeymiş gibi izin verilenlere erişilebilir.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using yeninamespace;
namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 baska x = new baska();
 Console.ReadLine();
 }
 }
 class sınıf_2
 {
 }
}
```

**hiçbir sorun çıkmadı:**

```
using yeninamespace;
namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 baska x = new baska();
 Console.ReadLine();
 }
 }
 class sınıf_2
 {
 }
}
```

```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım baha
digersinif_2 tanımlandı
baska sınıfı tanımlandı
```

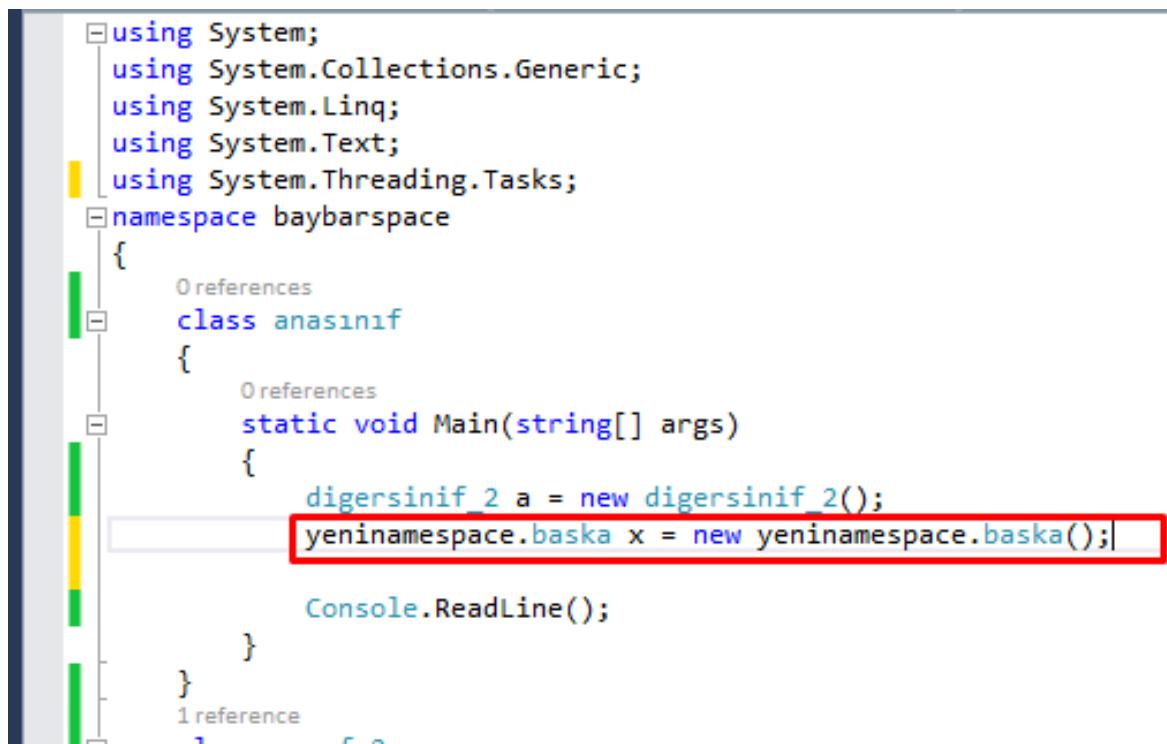
# Klasördeki durumlarda bu şekildedir

| Ad                 | Değiştirme tarihi | Tür                    | Boyut |
|--------------------|-------------------|------------------------|-------|
| bin                | 29.03.2016 01:24  | Dosya klasörü          |       |
| obj                | 29.03.2016 01:03  | Dosya klasörü          |       |
| Properties         | 29.03.2016 01:03  | Dosya klasörü          |       |
| yeninamespace      | 29.03.2016 01:33  | Dosya klasörü          |       |
| App.config         | 29.03.2016 01:03  | XML Configuration...   | 1 KB  |
| baybarspace.csproj | 29.03.2016 01:30  | Visual C# Project f... | 3 KB  |
| digersinif_1.cs    | 29.03.2016 01:18  | Visual C# Source f...  | 1 KB  |
| Program.cs         | 29.03.2016 01:30  | Visual C# Source f...  | 1 KB  |

## yeninamespace'nin içi

| Ad               | Değiştirme tarihi | Tür                   | Boyut |
|------------------|-------------------|-----------------------|-------|
| baska.cs         | 29.03.2016 01:33  | Visual C# Source f... | 1 KB  |
| digersinifler.cs | 29.03.2016 01:30  | Visual C# Source f... | 1 KB  |

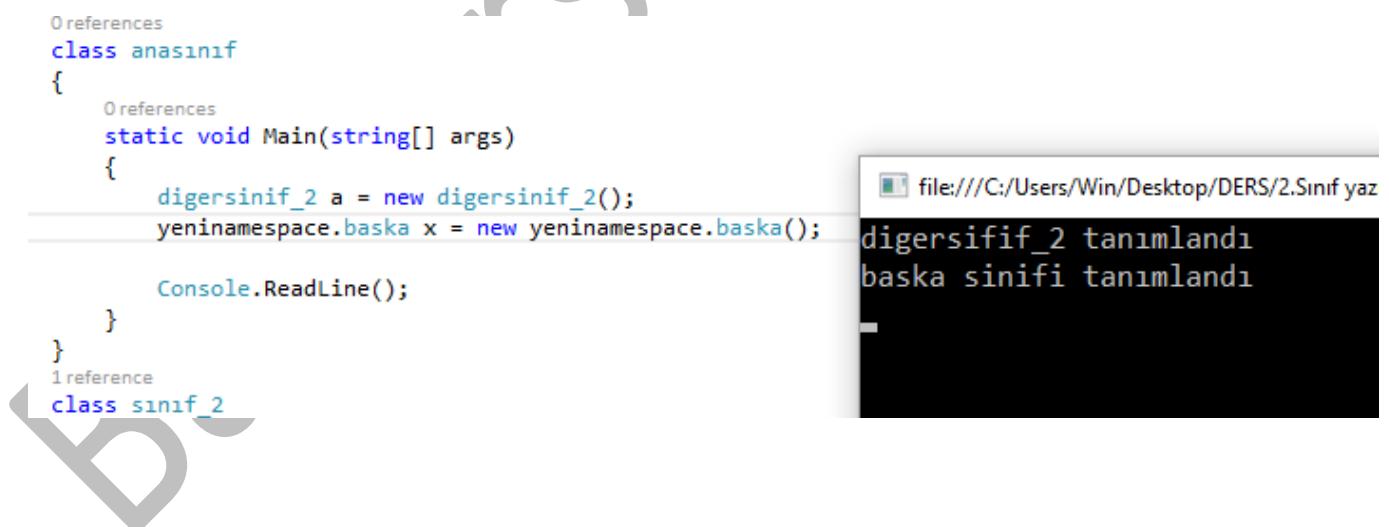
# USİNG KULLANMADAN BU ŞEKLİDE OLURDU



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 yeninamespace.baska x = new yeninamespace.baska();

 Console.ReadLine();
 }
 }
}
```

Hiçbir sorun çıkmadan çalışır:



```
class anasınıf
{
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 yeninamespace.baska x = new yeninamespace.baska();

 Console.ReadLine();
 }
}
```

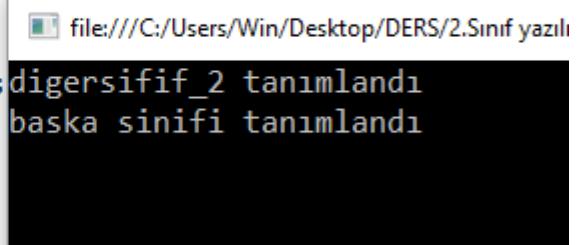
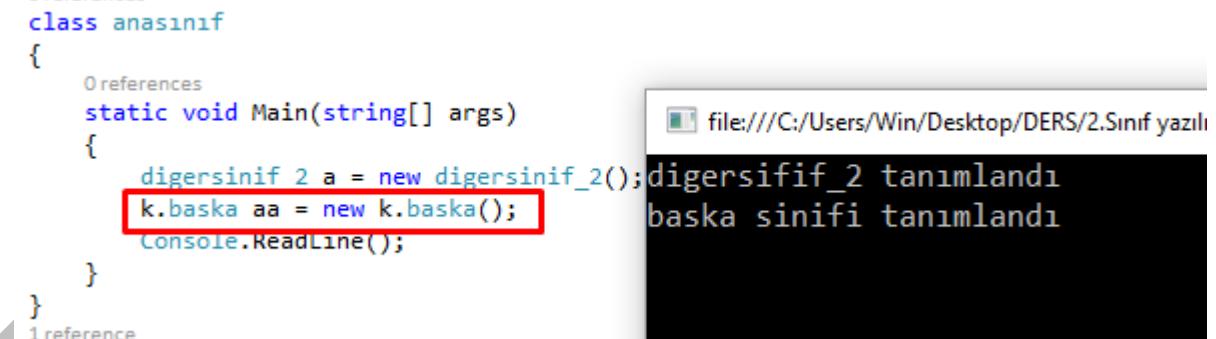
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yaz  
digersifif\_2 tanımlandı  
baska sınıfı tanımlandı

# Using kullanılarak takma isimde(alias) verilebilir isimizi kolaylaştırır.

```
using System.Text;
using System.Threading.Tasks;
using k = yeninamespace;
namespace baybarspace
{
 class anasınıf
 {
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 k.baska aa = new k.baska();
 Console.ReadLine();
 }
 }
 class sınıf_2
 {
```

Hiçbir sorun olmadan çalışır:

```
class anasınıf
{
 static void Main(string[] args)
 {
 digersinif_2 a = new digersinif_2();
 k.baska aa = new k.baska();
 Console.ReadLine();
 }
}
```



## Using'de alias kullanımıyla ilgili başka örnek

```
using System;
using yaz = System.Console;
class AnaSınıf
{
 static void Main()
 {
 yaz.WriteLine("Deneme");
 }
}

// yaz=Console; ifadesi tek başına yazıldığında hata verirdi mutlaka isim alanı ile birlikte yazılmalı.
```

burada system'le ilgililere dikkat edilmelidir aslında using size belirli bir dizinden sonrasıń açıyor diye biliriz eğer a klasör(namespace) içinde b1 b2 b3 varsa using a; dendığında b1 b2 b3'e direkt erişilir eğer using a.b1; denseydi b1'in içindeki fonksiyonlara değişkenlere direkt olarak ulaşılabilir.

## iç İçe Geçmiş İsim Alanları Tanımlamak (Nested NameSpaces)

```
using System;

namespace Alan
{
 class Sinif1
 {
 public Sinif1() { }
 }

 namespace AltAlan
 {
 class Sinif2
 {
 public Sinif2() { }
 }
 }
}

class Program
{
 static void Main()
 {
 Alan.Sinif1 s1 = new Alan.Sinif1();
 Alan.AltAlan.Sinif2 s2 = new Alan.AltAlan.Sinif2();
 }
}
```

yani buradan anlaşılacağı gibi iç içe class olduğu gibi iç içe namespace'de olabiliyormuş.

# KÜTÜPHANE DOSYASI

## OLUŞTURMA

Sağdaki solution explorer'den add→Class library eklenir ve oraya gerekli işlemler yapılır bir önceki bölümde anlatıldı[classlibrary içindeki işlemler namespace ile aynı mantıkda].Daha sonra classlibrary build edilip eklenilceği yerde add→ references bölümünden aşadaki browse'ye tıklanır eklenilcek class bulunur bin→debug→dsasa.dll gibi bir dosya vardır ona tıklanarak dll dosyası alınır.

Alındıktan sonra zaten bir önceki namespace using v.s. işlemlerle fonksiyonlara sınıflara değişkenlere v.s. erişilebilir.

## Harici Takma İsimler (External Alias)

- Daha önce bahsedildiği gibi, farklı isim alanları altında aynı isimli sınıflar tanımlandığında ve her iki isim alanı da using deyimi ile programa eklendiğinde kod bloğu içersinden sınıflara erişmek istediğimizde takma isimler tanımlamak gerekiyordu.
- Boylelikle referans edilmiş her bir sınıfa takma ismi yardımıyla erişebilmekteydi.
- Bu tarz bir çözüm büyük program parçalarında anlam bütünlüğünün bozulmasına neden olabilir.
- Bunun için C# 2.0'dan itibaren C# dilinde oluşturulmuş olan sınıf kütüphanelerine (dll'ler) harici takma isimler verebilmekteyiz.

## Harici Takma İsimler (External Alias)

- İsim alanları ve sınıflar aynı olduğundan herhangi bir sınıfı kullanmaya kalktığımızda çakışma olacaktır.
- Şimdi dosya1.cs ve dosya2.cs dosyalarını oluşturun ve içine şu kodları yazın:

|           |           |
|-----------|-----------|
| dosya1.cs | dosya2.cs |
|-----------|-----------|

```
namespace IsimAlani
{
 public class bir {}
 public class iki {}
}
```

```
namespace IsimAlani
{
 public class bir {}
 public class iki {}
}
```

- Bu iki dosyanın içeriği tamamen aynı. Ancak siz sınıfların içeriği farklılaşmış gibi düşünün. Şimdi her iki dosyayı da kütüphane dosyası (DLL) hâline getiriniz. Konsol veya proje ekleyerek bunu gerçekleştiriniz.

class library'e dönüştürme

```
csc /r:Dosya1=dosya1.dll /r:Dosya2=dosya2.dll program.cs
```

Artık elimizde dosya1.dll ve dosya2.dll adlı iki sınıf kütüphanesi var. Bunlar haricî kütüphanelerdir. Ana programı yazarken öncelikle alias lar belirtilecek.

```
extern alias Dosya1;
extern alias Dosya2;
```

```
extern alias Dosya1;
extern alias Dosya2;// extern ifadeleri using ten önce olmalı
using System;

class Ana {
 static void Main()

 {Dosya1 :: isimalani.Class1 aa=new Dosya1 :: isimalani.Class1();
 Dosya2.isimalani.Class1 bb=new Dosya2.isimalani.Class1();

 }

}
```

**extern alias'lar tamamen aynı olan dll'leri veya namespace gruplarını birbirinden ayırt etmeye yarar.**

## global harici takma ismi

- .Net Framework kütüphanesi, kendi oluşturduğumuz kütüphaneler ve takma ad verilmemiş DLL kütüphanelerimiz otomatik olarak global takma adını alır. Yani programlar aşağıdaki gibi de yazılabilir.

```
• using System;
• namespace ia
• { class Yardimci
• { public Yardimci()
• { Console.WriteLine("Yardimci"); } }}
```

```
• namespace ana
• { class Sınıf
• { static void Main()
• { global::ia.Yardimci a=new global::ia.Yardimci(); } }
• }
```

tanımlı değilse artık tanımlıdır ve takma isim  
üzerinden tanımlanmıştır.  
Bu durum dll'ler için geçerlidir

yani eğer hiç oluşturulmamışsa usinde kolaylık sağlanır.

global ile en kökten gidilir.O namespace veya .dll ile ilgili hiçbirşeyin tanımlanmadığı durumlarda kullanmak mantıklıdır.

Başka bir örnek:

```
class prg
{
static void Main()
{ global::System.Console.WriteLine("Deneme"); }
}
```

# System İsim alanı

- .NET sınıf kütüphanesinde yer alan System isim alanı içerisinde oldukça kullanışlı bazı sınıflar bulunmaktadır.
- Bunlardan **System.Array**, **System.Random**, **System.Convert**, **System.Math** ve **System.GC** 'den daha önce bahsedilmiştir.
- System isim alanı içerisinde yer alan temel veri tiplerinin sahip olduğu bazı metodlar da zaman zaman kullanılmaktadır.

- **Tip.Parse();** : string biçimindeki verileri tip türüne çevirir.
- **Nesne.CompareTo(object o);** : metodu çağrılan nesne ile o nesnesini karşılaştırır. Değerler eşit ise 0, nesne değeri küçük ise negatif, büyük ise pozitif değer döndürür.
- **Nesne.Equals(object o);** : metodu çağrılan nesne ile o nesnesini karşılaştırır. Eşit ise true aksi halde false döndürür.
- **Nesne.ToString();** : nesne değerini string şeklinde geri döndürür.

- int a=2; int b=32;
  - b.**CompareTo(a)**
  - b<a ise -1, b>a ise 1, b==a ise 0
- 
- b. **Equals(a)**
  - b==a ise true, b!=a ise false

bunları ezberleme gereklirse bakarsın ve bu bölümler çok ezber sınavda çıkmaz büyük ihtimalle.[char]

|                        |                                         |
|------------------------|-----------------------------------------|
| <b>IsControl</b>       | Karakter kontrol karakteri ise          |
| <b>IsDigit</b>         | Karakter bir rakam ise                  |
| <b>IsLetter</b>        | Karakter bir harf ise                   |
| <b>IsLetterOrDigit</b> | Karakter bir harf ya da rakam ise       |
| <b>IsLower</b>         | Karakter küçük harf ise                 |
| <b>IsNumber</b>        | Karakter rakam ise                      |
| <b>IsPunctuation</b>   | Karakter noktalama işaretleri ise       |
| <b>IsSeparator</b>     | Karakter boşluk gibi ayırcı ise         |
| <b>IsSurrogate</b>     | Karakter Unicode yedek karakteri ise    |
| <b>IsSymbol</b>        | Karakter sembol ise                     |
| <b>IsUpper</b>         | Karakter büyük harf ise                 |
| <b>IsWhiteSpace</b>    | Karakter tab ya da boşluk karakteri ise |

Decimal değerler içinde bazı metodlar mevcuttur:

```
Decimal.Add(d1, d2); //d1 ve d2'nin toplamını decimal türünden tutar. (+)
Decimal.Divide(d1, d2); //d1'in d2'ye bölümünü decimal türünden tutar. (/)
Decimal.Multiply(d1, d2); //d1 ile d2'nin çarpımını decimal türünden tutar. (*)
Decimal.Subtract(d1, d2); //d1-d2'nin sonucunu decimal türünden tutar. (çıkarma işlemi) (-)
Decimal.Remainder(d1, d2); //d1'in d2'ye bölümünden kalanı decimal türünden tutar. (mod alma işlemi)
Decimal.Floor(d1); //d1'den büyük olmayan en büyük tam sayıyı decimal türünden tutar. (aşağı yuvarlama)
Decimal.GetBits(d1); /*d1 için decimal sayı tanımlarken kullandığımız yapıcı işlevdeki beş parametreyi int türündeki bir dizi olarak tutar.*/
Decimal.Negate(d1); //d1'in negatifini tutar.
Decimal.Round(d1, sayı); /*sayı int türünde olmalıdır. Bu metot ile d1'in ondalık kısmındaki hane sayısı sayı kadar kalır. Yani d1 12.53666 ve sayı 3 ise 12.537 tutulur. Son kaybedilen hane 5 ya da 5'ten büyükse son kalan hane 1 artırılır. sayı 0 olursa d1 tam sayıya yuvarlanmış olur.*/
Decimal.Truncate(d1); //d1'in tam sayı kısmını tutar. Herhangi bir yuvarlama yapılmaz.
```

```
string a = "25";
int b = Int32.Parse(a);
```

## Char Tipinde Yapılar

|                          |                                         |
|--------------------------|-----------------------------------------|
| <b>ToLower(char str)</b> | str büyük harf ise küçük harfe çevirir. |
| <b>ToUpper(char str)</b> | str küçük harf ise büyük harfe çevirir. |

## **DateTime ve TimeSpan**

C# dilinde tarih ve saat işlemleri System isim alanında bulunan **DateTime** ve **TimeSpan** yapıları ile gerçekleştirilir.

**DateTime** yıl, ay, gün, saat, dakika, saniye gibi bilgileri tutan bir yapıdır.

**TimeSpan** ise iki zaman bilgisi arasındaki farkı temsil etmek için kullanılır.

### **DateTime Özellikleri**

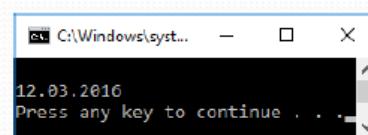
**DateTime.Now:** Sistem saatinin hem tarih hem de zaman bilgisini öğrenmek

- **DateTime.Now.Today:** Sistem tarihini öğrenmek
- **DateTime.Now.DayOfWeek:** Haftanın gününü öğrenmek
- **DateTime.Now.DayOfYear:** Yılın kaçinci günü olduğunu öğrenmek
- **DateTime.Now.Day, Month ve Year:** Sistem tarihinin gün ay, yıl bilgisini öğrenmek
- **DateTime.Now.Hour, Minute, Second ve Millisecond :** Sistem zamanının saat, dakika, saniye ve milisaniye bilgisini öğrenmek
- **DateTime.Now.Ticks:** DateTime yapısının bu özelliğinden yararlanılarak bir işlemin ne kadar sürdüğünü öğrenebilirsiniz.
  - Ticksözellik 1 Ocak 0001'den içinde bulunan ana kadar geçen süreyi tick sayısı cinsinden içermektedir. Bu özelliğin içeriği her 100 nano saniyede 1 artmaktadır.
  - *Saniye = Tick\_sayı / 10.000.000* değerini verir
  - İşlemin başında ve sonunda Ticks özelliği kullanılarak ikisi arasındaki farkı alarak programın çalışma süresini bulabilirsiniz.

## DateTime Metotları

**ToShortDateString()**: Sistemin sadece tarih kısmını verir.

- `DateTime Tarih = DateTime.Today;`
- `Console.WriteLine(Tarih.ToShortDateString());`



**ToLongDateString()**: Sistemin sadece tarih bilgisinin uzun halini verir.

- `Console.WriteLine(Tarih.ToString("yyyy/MM/dd"));`

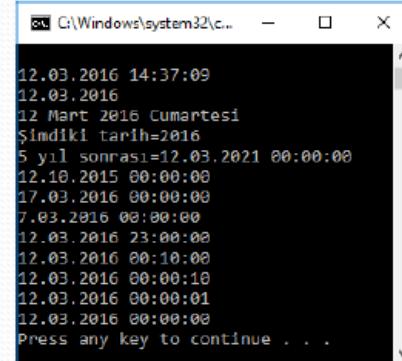
**Add(TimeSpan value)**: Geçerli tarihe belli sayıda gün, tarih veya yıl eklenmek

**AddDays(), AddMonths(), AddYears(), AddHours(), AddMinutes() AddSeconds()**

**Metotları:** Değişik şekilde elde edilmiş geçerli tarih bilgisine gün, ay, yıl, zaman bilgisine saat, dakika, saniye gibi değerleri ekleyip veya çıkarıp başka bir tarih ve zaman bilgisi elde etme.

- `Console.WriteLine(Tarih.AddDays(5));` Mevcut tarihin gününden 5 gün olmasını verir.
- `Console.WriteLine(Tarih.AddDays(-5));` Mevcut tarihin gününden 5 gün öncesini verir.

```
using System;
class zaman
{ public static void Main()
 { DateTime Tarih = DateTime.Today;
 Console.WriteLine("Kısa Tarih=" + Tarih.ToShortDateString());
 Console.WriteLine("Uzun Tarih=" + Tarih.ToString("yyyy/MM/dd"));
 Console.WriteLine("Şimdiki Yıl=" + Tarih.Year);
 Console.WriteLine("5 yıl sonrası=" + Tarih.AddYears(5));
 Console.WriteLine(Tarih.AddMonths(-5));
 Console.WriteLine(Tarih.AddDays(5));
 Console.WriteLine(Tarih.AddDays(-5));
 Console.WriteLine(Tarih.AddHours(23));
 Console.WriteLine(Tarih.AddMinutes(10));
 Console.WriteLine(Tarih.AddSeconds(10));
 Console.WriteLine(Tarih.AddMilliseconds(1000));
 Console.WriteLine(Tarih.AddTicks(10));
 }
}
```



**TimeSpan** ve **DateTime** yapıları ile tanımlanmış bazı operatörler bulunur.

İki tarih arasındaki farkı bulmak için çıkarma (-), ileriki bir tarihi hesaplamak için toplama (+), iki tarih arasında büyükük küçükük karşılaştırması yapmak için de “<” ve “>” operatörleri aşırı yüklenmiştir.

```
using System;

class Program
{
 static void Main()
 {
 int yil, ay, gun;
 Console.Write("Doğum Yılıınız:");
 yil = Convert.ToInt32(Console.ReadLine());
 Console.Write("Doğum Ayınız:");
 ay = Convert.ToInt32(Console.ReadLine());
 Console.Write("Doğum Gününüz:");
 gun = Convert.ToInt32(Console.ReadLine());

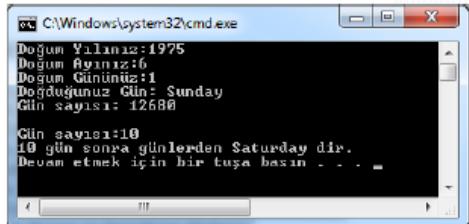
 DateTime Bugun = DateTime.Today;
 DateTime DogumGunu = new DateTime(yil, ay, gun);

 TimeSpan fark = Bugun - DogumGunu;

 Console.WriteLine("Doğduğunuz Gün: {0}", DogumGunu.DayOfWeek);
 Console.WriteLine("Gün sayısı: {0}", fark.Days);

 Console.WriteLine();
 Console.Write("Gün sayısı:");
 gun = Convert.ToInt32(Console.ReadLine());

 TimeSpan GunSayisi = new TimeSpan(gun, 0, 0, 0);
 DateTime Gelecek = DateTime.Today + GunSayisi;
 Console.WriteLine("{0} gün sonra günlerden {1} dir.", gun, Gelecek.DayOfWeek);
 }
}
```



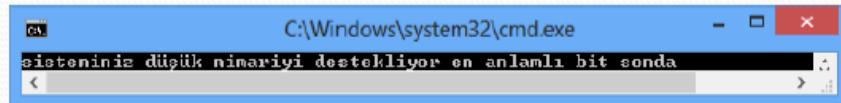
# Bit işlemleri

En önemli metodu da **GetBytes**'tir. Amacı da farklı sayı türlerini byte dizisine çevirmektir.

## ilginç bir uygulama

Mimarınızı öğrenmek için;

```
using System;
class bitconverter
{
 public static void Main()
 {
 if (BitConverter.IsLittleEndian)
 Console.WriteLine("sisteminiz düşük mimariyi destekliyor
en anlamlı bit sonda");
 else
 Console.WriteLine("sisteminiz yüksek mimariyi destekliyor
en anlamlı bit başta");
 }
}
```



BAYK

## AŞADAKİ ÖRNEĞE DİKKAT

```
int a=258; // 258/256=1 => 258-2561 *1 = 2
//00000000 00000000 00000001 00000010
2563 2562 2561 2560
byte[] dizi=BitConverter.GetBytes(a);
foreach(byte b in dizi)
{ Console.WriteLine(b); }
2
1
0
0
byte[] dizi={2,1,0,0}; → en anlamlı bit en sonda
Console.WriteLine(BitConverter.ToInt32(dizi,0));
258
Bilinçli tür dönüşümünde kayıplar en anlamlı bitlerde
olur.
```

## System.Buffer

- byte[] kaynak={1,2,3,1};  
• 00000001 , 00000010 , 00000011 , 00000001
- short[] hedef=new short[4];  
• 00000000000000, 00000000000000 , 0000000000000000 , 0000000000000000
- Buffer.BlockCopy(kaynak,0,hedef,0,4);
- hedef dizisinin yeni hali:  
• 0000001000000001 , 0000000100000011 , 0000000000000000 ,  
0000000000000000
- foreach(short a in hedef)      Console.Write(" "+a);
- Bu program, mimarımız Little Endian ise ekranı: 513 259 0 0 yazar.
  - Burada derleyici her elemanı bellekte 1 bayt yer kaplayan kaynak dizisinin elemanlarını her elemanı bellekte 2 bayt yer kaplayan hedef dizisine olduğu gibi kopyaladı. Derleyici burada karşılaştığı ilk baytı düşük anlamlı bayta, ikinci baytı da yüksek anlamlı bayta kopyaladı.
- Bilinçsiz tür dönüşümü Buffer.BlockCopy mantığı ile gerçekleşir.

```
short[] dizi=new short[4];
Console.WriteLine(Buffer.ByteLength(dizi)); // 8 yazar
```

**static byte** GetByte(Array dizi, **int** a)  
dizinin a. baytını verir.

**static void** SetByte(Array dizi, **int** a, **byte** deger)  
a. baytı deger olarak değiştirir.

## AŞADAKİ ÖRNEĞE DİKKAT

**byte[]** dizi={0,3,2,1,4}; Console.WriteLine(Buffer.GetByte(dizi,3)); // Ekrana 1 yazar  
• Bu kod 1 değerini döndürür. Eğer dizinin tipi byte değil de short olsaydı işler değişirdi. Çünkü o zaman hem sıfırla beslenen baytlar da hesaba katılırdı ve hem de bilgisayarımızın mimarisini sonucu etkilerdi. Bu durumu örneklendirelim:

```
short[] dizi={0,3,2,1,4};
0-> 0000000000000000 , 2-> 000000000000000011 , 4-> 0000000000000010 ,
1 0 3 2 5 4
6-> 0000000000000001 , 8-> 00000000000000100
7 6 9 8
```

```
Console.WriteLine(Buffer.GetByte(dizi, 4)); // Ekrana 2 yazar
Console.WriteLine(Buffer.GetByte(dizi,8)); // Ekrana 4 yazar
```

convertı geçiyorum

# STRING İŞLEMLERİ

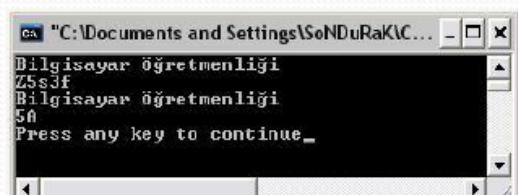
## string.Concat()

- `static string Concat(params Array stringler)`
- String verilerin ardarda eklenmesini sağlar. + operatörü ile eşdeğerdir.

YANI DEĞERLERİ  
BİRLEŞTİRİYOR

## String.Concat()

```
using System;
class Concat
{
 public static void Main()
 {
 String str1 = String.Concat("Bilgisayar", " Öğretmenliği");
 Console.WriteLine(str1);
 String str2 = String.Concat("Z", "5", "s", "3", "f");
 Console.WriteLine(str2);
 String str3 = "Bilgisayar" + " Öğretmenliği";
 Console.WriteLine(str3);
 String str4 = String.Concat(5, "A");
 Console.WriteLine(str4);
 }
}
```



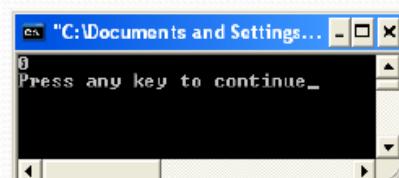
### string.Compare()

- İki string değeri karşılaştırır. **==** ve **!=** operatörleri ile benzer işlem gerçekleştirir. Fakat Compare metodunun bazıası yüklemeleri ile daha gelişmiş (büyük küçük harf duyarlılığı gibi.) karşılaştırmalar yapılabilir.
- static int Compare(string a, string b)**
- static int Compare(string a, string b, bool c)**
- static int Compare(string a, int indeks1, string b, int indeks2)**
- static int Compare(string a, int indeks1, string b, int indeks2, bool c)**
- kıyaslama ilk elemanların a[indeks1] ve b[indeks2] sayılmasıdır. Yani stringlerdeki bu elemanlardan önceki elemanlar yok sayılır.

FK'

### String.Compare()

```
using System;
class class1
{ public static void Main()
{ string a="Aa";
 bool sa=true;//false büyük küçük duyarsız
 string v="aa";
 int c=String.Compare(a,v,sa);
 Console.WriteLine(c);
}
```



TRUE [BUYUK KUCUK]  
DUYARLI

Ba'

`stringnesne.CompareTo(string str)` : İki stringi karşılaştırır. Değerler eşit ise 0, nesne değeri küçük ise negatif, büyük ise pozitif değer döndürür.

#### `stringnesne.IndexOf()`

- string içerisinde alt stringlerin aranmasını sağlar. Geriye aranan alt stringin bulunduğu konumu ya da bulunamadı (-1) bilgisini döndürür.

- `int IndexOf(string a)`

- `int IndexOf(char b)`

#### `stringnesne.LastIndexOf()`

- `IndexOf` ile aynı çalışır Farkı ise aranan karakterin en son nerede görüldüğünün indeksini geri döndürür.

- `int LastIndexOf(string a)`

- `int LastIndexOf(char b)`

#### `IndexOf`

```
using System;
class arama
{ public static void Main()
{ string yazı="firat üniversitesi";
 Console.WriteLine(yazı.IndexOf("ver"));
 Console.WriteLine(yazı.IndexOf('t'));
 Console.WriteLine(yazı.IndexOf('c'));
}
```

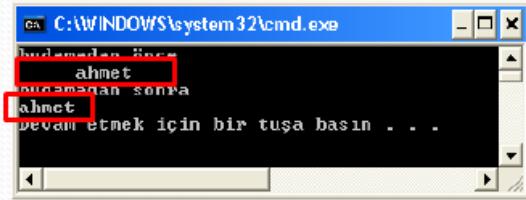
ÇOK HARF VARSA İLK HARFE GÖRE



### stringnesne.Trim()

- Bir string ifadenin başındaki ve sonundaki boşlukları ya da belirlenmiş karakterleri atar.

```
using System;
class tarih {
 public static void Main()
 { string a = " ahmet ";
 Console.WriteLine("budamadan önce");
 Console.WriteLine("'" + a);
 a = a.Trim ();
 Console.WriteLine("budamadan sonra");
 Console.Write("'" + a);
 }
}
```

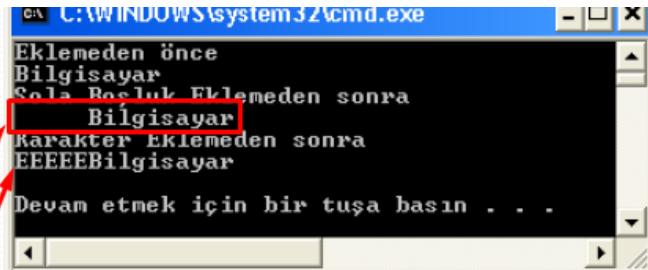


### PadRight, PadLeft

- Bir stringin sağına ya da soluna yeni karakterler ilave etmek için kullanılır.

- `string PadRight(int boyut);`
  - stringin uzunluğu boyuta eşit olana kadar stringin sağına boşluk ekler
- `string PadRight(int boyut,char a);`
  - stringin uzunluğu boyuta eşit olana kadar stringin sağına 'a' karakterini ekler
- `string PadLeft(int boyut);`
  - stringin uzunluğu boyuta eşit olana kadar stringin soluna boşluk ekler
- `string PadLeft(int boyut, char a);`
  - stringin uzunluğu boyuta eşit olana kadar stringin soluna 'a' karakterini ekler

```
using System;
class tarih
{
 public static void Main()
 {
 string a = "Bilgisayar";
 Console.WriteLine("Eklemeden önce");
 Console.WriteLine("'" + a);
 a = a.PadLeft(15);
 Console.WriteLine("Sola Boşluk Eklemeden sonra");
 Console.WriteLine("'" + a);
 Console.WriteLine("Karakter Eklemeden sonra");
 a = "Bilgisayar";
 a = a.PadLeft(15,'E');
 Console.WriteLine("'" + a);
 Console.WriteLine("");
 }
}
```



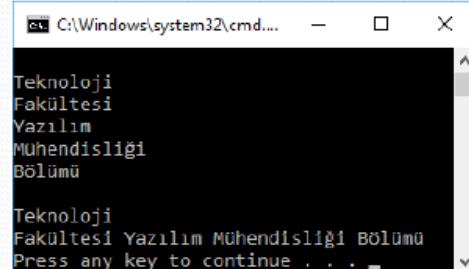
**bool string.StartsWith(string a)** : Metodu çağrıran "string a" ile başlıyorsa **true**, diğer durumlarda **false** değeri üretir.

**bool string.EndsWith(string b)** : Metodu çağrıran "string b" ile bitiyorsa **true**, diğer durumlarda **false** değeri üretir.

## ► stringnesne.Split()

- Belli bir biçimde sahip olan toplu string verileri, belirtilen ayırcı karakterlerden ayırip ayrı bir string dizisi üretir.
  - `string [] Split(params char[] ayırcı)`
  - `string [] Split(params char[] ayırcı, int toplam)`
- İkinci metotta ise bu parçalama işlemi en fazla toplam sayısı kadar yapılır.

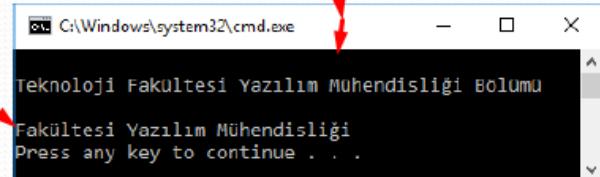
```
using System;
class Ayirmaşlemi
{ public static void Main()
{
 string str="Teknoloji Fakültesi Yazılım Mühendisliği
 Bölümü";
 char[] ayırcı={' '};
 string[] ayır=str.Split(ayırcı);
 foreach(string i in ayır)
 Console.WriteLine(i);
 Console.WriteLine();
 ayır = str.Split(ayırcı,2);
 foreach (string i in ayır)
 Console.WriteLine(i);
}
}
```



## string.Join()

- Split metodunun tersi gibi çalışır. Ayrı stringleri belli bir ayırcı karakter ile birleştirip tek bir string üretir.
  - static string join(string ayırcı, string[] yazılar)
  - static string join(string ayırcı, string[] yazı, int başla, int toplam)
  - İkinci metotta ise [basla] 'dan itibaren toplam kadar yazı elemanı birleştirilir.

```
using System;
class bireştirmeyislemi
{ public static void Main()
{
 string [] str={"Teknoloji", "Fakültesi", "Yazılım",
 "Mühendisliği", "Bölümü"};
 string birles=String.Join(" ",str);
 Console.WriteLine(birles);
 Console.WriteLine();
 birles = String.Join(" ", str,1,3);
 Console.WriteLine(birles);
}
}
```



# AŞADAKİLERE SADECE BİR GÖZ AT

## stringnesne.ToUpper()

- Stringin karakterlerinin hepsini büyük harfe çevirip geri döndürür.

## stringnesne.ToLower()

- Stringin karakterlerinin hepsini küçük harfe çevirip geri döndürür.

## stringnesne.Remove(int indeks, int adet)

- String içinden belli sayıda karakterin atılmasını sağlar.

## stringnesne.Insert(int indeks, string str)

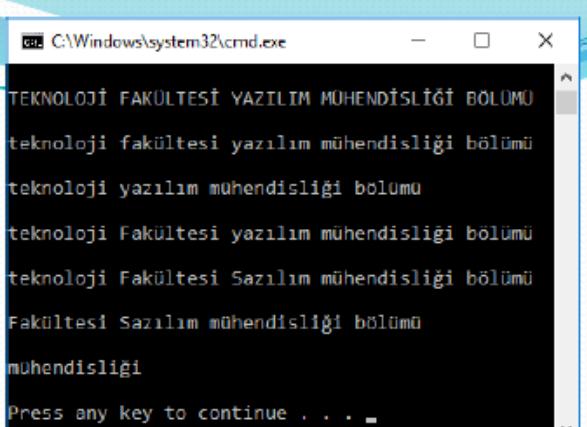
- String içine belirli indeksten itibaren yeni bir string eklemek için kullanılır.

## stringnesne.Replace(char | string c1,char|string c2)

- String içindeki belirlenen karakter ya da karakterleri başkalarıyla değiştirir.

## stringnesne.SubString(int indeks |int indeks,int toplam)

- String ifadenin belli bir kısmının elde edilmesini sağlar.



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window displays the output of a C# program. The program code is as follows:

```
using System;
class digermetotlar
{
 public static void Main()
 {
 string str="Teknoloji Fakültesi Yazılım Mühendisliği Bölümü";
 str=str.ToUpper();
 str=str.ToLower();
 str=str.Remove(9,10);
 str=str.Insert(10,"Fakültesi ");
 str=str.Replace('y','S');
 str=str.Substring(10);
 str = str.Substring(18,12);
 }
}
```

The output of the program is:

```
TEKNOLOJİ FAKÜLTESİ YAZILIM MUHENDİSLİĞİ BÖLÜMÜ
teknoloji fakültesi yazılım mühendisliği bölümü
teknoloji yazılım mühendisliği bölümü
teknoloji Fakültesi yazılım mühendisliği bölümü
teknoloji Fakültesi Sazılım mühendisliği bölümü
Fakültesi Sazılım mühendisliği bölümü
mühendisliği
Press any key to continue . . .
```

The code uses several string manipulation methods: `ToUpper`, `ToLower`, `Remove`, `Insert`, `Replace`, and `Substring`. The output shows the transformation of the initial string 'Teknoloji Fakültesi Yazılım Mühendisliği Bölümü' through these operations.

# BİÇİMLENDİRME

Bu metodlarda kullanılan `{ }`  parantezleri arasındaki ifadeler belli değişkenlerin belli bir düzende biçim metnine aktarılmasını sağlıyor:

- `Console.WriteLine("Merhaba {0}!",isim)`



En genel kullanımı

- `{ değişken_no, genişlik : format}`
- `int a=54;`

Sadece değişken verildiğinde değişkenin türüne göre varsayılan ayarlar kullanılır.

**Genişlik**, yazılacak olan verinin diğer verilerle olan mesafesini ve hangi yöne hizalanması gerektiğini belirler.

- `Console.WriteLine("{0,10} numara",a);`
- `Console.WriteLine("{0,-10} numara",a)`
- 54 numara
- 54 numara

**Format**, ise verinin türüne göre değişik biçimlendirilmesini sağlar.

Bayb

Bu formatlar ezberlenemez

| Belirleyici | Açıklama                                           | Duyarlılık Anlamı                                                                  |
|-------------|----------------------------------------------------|------------------------------------------------------------------------------------|
| C/c         | Para birimi                                        | Ondalık basamakların sayısını verir.                                               |
| D/d         | Tam sayı verisi                                    | En az basamak sayısını belirtir, gereğinde boş olan basamaklar sıfır ile beslenir. |
| E/e         | Bilimsel notasyon                                  | Ondalık basamak sayısını verir.                                                    |
| F/f         | Gerçek sayılar (float)                             | Ondalık basamak sayısını verir.                                                    |
| G/g         | E ve F biçimlerinden hangisi kısa ise o kullanılır | Ondalık basamak sayısını verir.                                                    |
| N/n         | Virgül kullanarak gerçek sayıları yazar            | Ondalık basamak sayısını verir.                                                    |
| P/p         | Yüzde                                              | Ondalık basamak sayısını verir.                                                    |
| X/x         | Onaltılık sayı sisteminde yazar.                   | En az basamak sayısını belirtir, gereğinde boş olan basamaklar sıfırla beslenir.   |

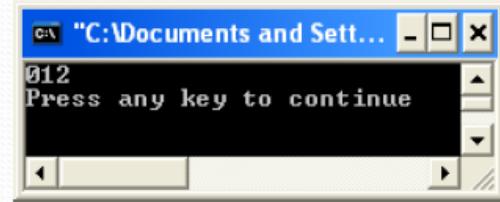
**using System;**

```
class Formatlama {
 static void Main() {
 float f=568.87f;
 int a=105;
 Console.WriteLine("{0:C3}",a);
 Console.WriteLine("{0:D5}",a);
 Console.WriteLine("{0:E3}",f);
 Console.WriteLine("{0:F4}",f);
 Console.WriteLine("{0:G5}",f);
 Console.WriteLine("{0:N1}",f);
 Console.WriteLine("{0:P}",a);
 Console.WriteLine("{0:P}",1);
 Console.WriteLine("{0:X5}",a);
 Console.WriteLine("{0:C3}",f); } }
```

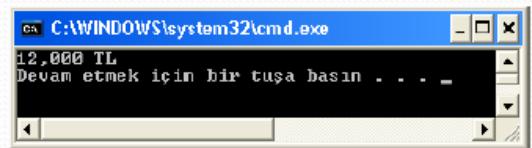
|            |    |
|------------|----|
| 105,000    | TL |
| 00105      |    |
| 5,689E+002 |    |
| 568,8700   |    |
| 568,87     |    |
| 568,9      |    |
| %10.500,00 |    |
| %100,00    |    |
| 00069      |    |
| 568,870    | TL |

## String.Format() ve ToString() Metotları İle Biçimlendirme

```
using System;
class StringFormat
{
 public static void Main()
 {
 int a=12;
 string str=String.Format("{0:d3}",a);
 Console.WriteLine(str);
 }
}
```



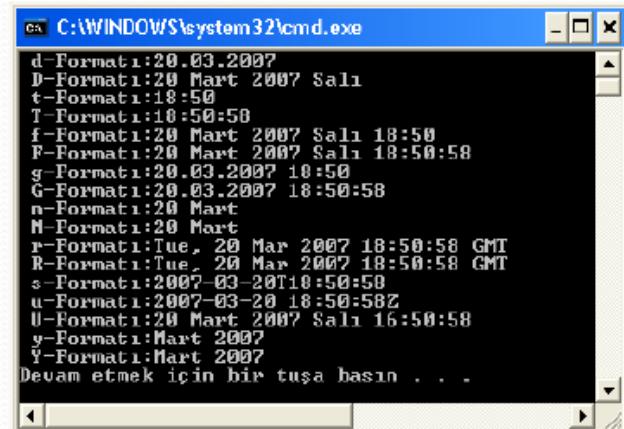
```
using System;
class Tostring
{
 public static void Main()
 {
 int a=12;
 string str=a.ToString("C3");// (para biçimi)
 Console.WriteLine(str);
 }
}
```



45

## Tarih ve Saat Biçimlendirme

- using System;
- class TarihveSaat {
- public static void Main() {
- DateTime d=DateTime.Now;
- Console.WriteLine(" d-Formatı:{0:d}",d);
- Console.WriteLine(" D-Formatı:{0:D}",d);
- Console.WriteLine(" t-Formatı:{0:t}",d);
- Console.WriteLine(" T-Formatı:{0:T}",d);
- Console.WriteLine(" f-Formatı:{0:f}",d);
- Console.WriteLine(" F-Formatı:{0:F}",d);
- Console.WriteLine(" g-Formatı:{0:g}",d);
- Console.WriteLine(" G-Formatı:{0:G}",d);
- Console.WriteLine(" m-Formatı:{0:m}",d);
- Console.WriteLine(" M-Formatı:{0:M}",d);
- Console.WriteLine(" r-Formatı:{0:r}",d);
- Console.WriteLine(" R-Formatı:{0:R}",d);
- Console.WriteLine(" s-Formatı:{0:s}",d);
- Console.WriteLine(" u-Formatı:{0:u}",d);
- Console.WriteLine(" U-Formatı:{0:U}",d);
- Console.WriteLine(" y-Formatı:{0:y}",d);
- Console.WriteLine(" Y-Formatı:{0:Y}",d);
- }



46

Standart biçimlendirmelerin dışında özel biçimlendirmeler de tasarılanabilir. "**#**" "**,**" "**.**" "**0**" "**E**" ve "**%**" karakterleri ile özel biçimlendirmeler oluşturulabilir.

- **#** → Rakam değerleri için kullanılır.
  - **,** → Büyük sayılarla binlikleri ayırmak için kullanılır.
  - **.** → Gerçek sayıarda ondalıklı kısımları ayırmak için kullanılır.
  - **0** → Yazılacak karakterin başına ya da sonuna 0 ekler.
  - **%** → Yüzde ifadelerini belirtmek için kullanılır.
  - **E0, e0, E+0, e+0, E-0, e-0** → Sayıları bilimsel notasyonda yazmak için kullanılır.
- 
- **{0:#,###.##}**
  - **{0:#%}**
  - **{0:#,###E+0}**

```
using System;
class özelbiçimlendirme
{
 public static void Main()
 { Console.WriteLine("{0:#,###}",1234567);
 Console.WriteLine("{0:#.##}",1234.5678);
 Console.WriteLine("{0:#,###E+0}",1234567);
 Console.WriteLine("{0:#%}",0.123);
 }
}
```



**SON BİR KAÇ  
BÖLÜME ÇOK  
YOĞUNLAŞMA  
YÜZEYSEL  
OLARAK  
ANLAYABİL  
YETER**

# **NESNE YÖNELİMLİ PROGRAMLAMA**

Sarmalama:Nesnenin veri fonksiyon v.b. bileşenleri içermesidir.

Bilgi saklama:Nesne dış dünyaya belirli hizmetler verir.Bu hizmetleride içeriği veri fonksiyon v.b. sayesinde yapar.Bu hizmetlerin alınabilmesi için ara yüzün dış dünya tarafından erişilen bölümünü kullanmak yeterlidir.Hizmetlerin hangi fonksiyon ve hizmetlerle karşılandığı dış dünya tarafından bilinmeyebilir.Bunada bilgi saklama(information hiding)denir.

Geç Bağlanma:Nesneler birbirlerinden genelde bağımsızdır.Fakat iletişim halinde olabilirler.Birbirleriyle veri fonksiyon v.b. alışverişler yapabilirler.Nesnelerin birbirleriyle iletişim halinin derleme halinde bilinememesi durumunda geç bağlanma(late binding) gerçekleşir.

Kalıtım:Nesneler birbirlerinden türeyebilirler.Bu türeme sonucu türeyen sınıf türetilen sınıfın erişebildiği tüm özelliklerine sahip olur.Kalıtım yoluyla türetilen sınıflar hiyerasyik sınıf organizasyonuyla gerçekleştirilebilir.

Çok biçimlilik(polymorphism):Temel sınıfın referansı üzerinden onun üzerinden türeyen sınıflara ait farklı özellikdeki metodlar çağrılabılır.Bu durumda nesneye yönelimdeki çok biçimlilik özelliği

# KALITIM(inheritance)

Sınıflardan nesne oluşturmak için new kullanılır.

```
Sinif nesne = new Sinif();
```

Sınıflar nesnelerin şeklini belirlerler. Yani nesnenin türünü tanımlarlar. Kısaca sınıflar bir **tür** bilgisidir.

Türetme yapmak için sınıf tanımlaması şu şekilde yapılmalıdır:

```
class TüretilenSınıf : TemelSınıf
```

```
• class Hayvan //temel sınıf
• {
• public double boy;
• public double kilo;
• public void OzellikGoster()
• {
• Console.WriteLine("Boy="+boy);
• Console.WriteLine("Ağırlık="+kilo);
• }
• }
```

- Ayrıca Kedi sınıfında kedinin türünü yazacak bir de metot olsun. Türetme aşağıdaki şekilde gerçekleştirilir;

```
class Kedi : Hayvan//türetilmiş sınıf
{
 public string Turu;
 public void TurGoster()
 {Console.WriteLine("Tur="+Turu);}
}
```

- Burada Kedi türetilmiş(derived) sınıf, Hayvan da temel (based) sınıfıdır.

Temel sınıf:based

Türetilmiş sınıf:derived

Kalıtım yolu ile **public** ve **protected** elemanlar aktarılır. Diğer sınıfların kullanımına kapalı ancak türetme ile türemiş sınıfa geçebilen özellikler **protected** anahtar sözcüğü kullanılır.

Eğer türetme söz konusu değilse **protected** olarak bilinen elemanlarla **private** olanlar arasında bir fark olmayacağından.

**private** özelliklere türetilen sınıflardan erişilemez.

**protected** özellikler ise türeyen sınıfa **private** olarak geçer.

DİKKAT:ERİŞİM BELİRLEYİCİLERDE PROTECTED OLAN BİRŞEY SINIFIN TÜRETİLEN SINIFLARI TARAFINDAN ERİŞİLEBİLİR.

**FAKAT EĞER TÜRETİLEN SINIFTANDA BİR SINIF TÜRETİLİYORSA O SINIF ERİŞEMEZ.**

**YANI PROTECTED ÖZELLİKLERE TÜRETİLMEMEYLE ERİŞİLEBİLİR FAKAT TÜRETİLEN SINIFTA O ÖZELLİKLER PRIVATE OLURLAR.**

C#'ta yapıcı metodların türetimiyle ilgili şu kurallar geçerlidir:

- C#'ta yapıcı metodlar fiziksel olarak türetilmez.
- Türemiş sınıfından bir nesne yaratıldığında önce ana sınıfın parametre almayan yapıcı metodu, ardından türemiş sınıftaki imzaya uyan yapıcı metod çalıştırılır.
- Türemiş sınıfından nesne yaratımında daima türemiş sınıfın imzaya uyan bir yapıcı metodu olması gereklidir.
- Türemiş sınıfından nesne yaratımlarında, ana sınıfın parametre almayan yapıcı metodu yavru sınıfın üye elemanlarıyla işlem yapar.
- Türemiş sınıfından nesne yaratımında, Türemiş sınıftaki ilgili yapıcı metoda **base** takısı eklenmişse ana sınıfın parametre almayan yapıcı metodu çalıştırılmaz.

**Yani eğer anasınıfta bir yapılandıracı varsa bir tane de boş parametreli yapılandıracı olmalıdır.**

**Çalıştırılmada ilk olarak anasının boş parametreli yapılandıracısı çalışır ve hiyeraşik bir şekilde devam eder bu sıra...**

**Yani bir üst sınıfın boş yapılandıracısı varsa önce o çalışır üst sınıfında üstünde boş parametreli yapılandıracı varsa o daha da önce çalışır ve bu hiyeraşik yapıyla tüm yapılandıracılar tetiklenir ama hiyeraşik olarak tepede olan daha önce önce çalışır. Eğer aradaki sınıfta boş yapılandıracı yoksa o sınıf hata verdirtmez bilinen mantıkla devam edilir ve bir üst sınıfın yapılandıracısı çalışır.**

**Fakat eğer sınıfın yapılandıracı parametresi varsa o çalıştırılsa bile daha önce boş parametreli ana sınıfın yapılandıracıları çalışır. Dikkat edilmelidir ki: Üst sınıflarda parametreye uygun bir yapılandıracı olsa bile boş parametreli yapılandıracı çalışır. Eğer üst sınıflarda parametresi dolu olan yapılandıracı varsa o sınıflarda kesin olarak boş parametreli metotda bulunmalıdır. Çünkü her halükarda çalışacaktırlar.**

**Yani o anda çalıştırılan metotda parametreye göre yapılandıracı seçilir fakat ne olursa olsun üst sınıflarda boş parametreli yapılandıracı çalışır. Üst sınıflarda hiçbir yapılandıracı metodun olmaması hata değildir. Fakat varsa boş parametreli yapıçı metotda olmalıdır.**

**Dikkat edilmelidir ki çalışma sırası hiyeraşiktir ve en temel sınıfın yapılandıracısı en başta çalışır ve bu hiyeraşiyle devam edilir.**

**[eğer base kullanılıyorsa en temel sınıfta boş parametreli yapılandıracı yoksa ve base ile yapılandırcılardan biri çağrılıyorsa buda hatayı engeller.]**

## ÖRNEKLER:

```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama/nesneTabanliProgramlama/nesneTabanliProgramlama.cs
class A
{
 0 references
 public A()
 {
 Console.WriteLine("a'nın bi halta yaramayan yapılandıricısı çalıştı");
 }
}

2 references
class B:A
{
 0 references
 public B()
 {
 Console.WriteLine("b'nın bi halta yaramayan yapılandıricısı çalıştı");
 }
}

3 references
class C:B
{
 1 reference
 public C()
 {
 Console.WriteLine("c'nın bi halta yaramayan yapılandıricısı çalıştı");
 }
}

0 references
class Program
{
 0 references
 static void Main(string[] args)
 {
 c nesn = new c();
 Console.WriteLine();
 Console.ReadLine();
 }
}
```

a'nın bi halta yaramayan yapılandıricısı çalıştı  
b'nın bi halta yaramayan yapılandıricısı çalıştı  
c'nın bi halta yaramayan yapılandıricısı çalıştı

Baybal

```
2 references
class A
{
 0 references
 public A()
 {
 Console.WriteLine("a'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

4 references
class B:A
{
 1 reference
 public B()
 {
 Console.WriteLine("b'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

1 reference
class C:B
{
 0 references
 public C()
 {
 Console.WriteLine("c'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

0 references
class Program
{
 0 references
 static void Main(string[] args)
 {
 b nesn = new B();
 Console.WriteLine();
 Console.ReadLine();
 }
}
```

```
2 references
class A
{
 0 references
 public A()
 {
 Console.WriteLine("a'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

1 reference
class B:A
{
 /* public B()
 {
 Console.WriteLine("b'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
*/
}

3 references
class C:B
{
 1 reference
 public C()
 {
 Console.WriteLine("c'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

0 references
class Program
{
 0 references
 static void Main(string[] args)
 {
 c nesn = new C();
 Console.WriteLine();
 Console.ReadLine();
 }
}
```

The screenshot shows a C# code editor with three tabs: `ConsoleApplication2.cs`, `ConsoleApplication2.cs`, and `c()`. The code defines three classes: `a`, `b`, and `c`, and a `Program` class.

```
ConsoleApplication2.cs
public class a
{
 public a()
 {
 Console.WriteLine("a'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

class b:a
{
 public b()
 {
 Console.WriteLine("b'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

class c:b
{
 public c()
 {
 Console.WriteLine("c'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }

 public c(int asd)
 {
 Console.WriteLine("c'nın yapılandırıcısı çalıştı");
 }
}

class Program
{
 static void Main(string[] args)
 {
 c nesn = new c(98);
 }
}
```

A red box highlights the constructor `c(int asd)`. Another red box highlights the line `c nesn = new c(98);` in the `Program` class's `Main` method. A third red box highlights the parameter `98` in that line. Red arrows point from the highlighted code in the editor to the corresponding output in the terminal window.

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama/ken

a'nın bi halta yaramayan yapılandırıcısı çalıştı  
b'nın bi halta yaramayan yapılandırıcısı çalıştı  
c'nın yapılandırıcısı çalıştı

The screenshot shows a code editor with three classes: `b:a`, `c:b`, and `Program`. The `Program` class has a static `Main` method that creates an instance of `c` and prints "c'nin bi halta yaramayan yapılandırıcısı çalıştı". A red arrow points from the line of code to the output window, which displays the same text.

```
3 references
class b:a
{
 0 references
 public b()
 {
 Console.WriteLine("b'nin bi halta yaramayan yapılandırıcısı çalıştı");
 }
 0 references
 public b(int ed)
 {
 Console.WriteLine("b'nin yapılandırıcısı çalıştı");
 }
}

4 references
class c:b
{
 0 references
 public c()
 {
 Console.WriteLine("c'nin bi halta yaramayan yapılandırıcısı çalıştı");
 }
 1 reference
 public c(int asd)
 {
 Console.WriteLine("c'nin yapılandırıcısı çalıştı");
 }
}

0 references
class Program
{
 0 references
 static void Main(string[] args)
 {
 c nes0 = new c(98);
 Console.WriteLine();
 Console.ReadLine();
 }
}
```

a'nın bi halta yaramayan yapılandırıcısı çalıştı  
b'nin bi halta yaramayan yapılandırıcısı çalıştı  
c'nin yapılandırıcısı çalıştı

The screenshot shows the same code structure. A red box highlights the constructor `c()` in class `c:b`. A red callout box points to this line with the text: "B'DEN BİR SINIF TÜREDİĞİ İÇİN VE B'DE YAPILANDIRICI OLMASINA RAĞMEN BOŞ YAPILANDIRICI OLMADIĞI İÇİN HATA İLE KARŞILAŞILIR". Another red box highlights the line `c nes0 = new c(98);` in the `Main` method of `Program`.

```
onsoleApplication2
class a
{
 0 references
 public a()
 {
 Console.WriteLine("a'nın bi halta yaramayan yapılandırıcısı çalıştı");
 }
}

2 references
class b:a
{
 0 references
 public b(int ed)
 {
 Console.WriteLine("b'nin yapılandırıcısı çalıştı");
 }
}

4 references
class c:b
{
 0 references
 public c()
 {
 Console.WriteLine("c'nin bi halta yaramayan yapılandırıcısı çalıştı");
 }
 1 reference
 public c(int asd)
 {
 Console.WriteLine("c'nin yapılandırıcısı çalıştı");
 }
}
```

B'DEN BİR SINIF TÜREDİĞİ İÇİN VE B'DE YAPILANDIRICI OLMASINA RAĞMEN BOŞ YAPILANDIRICI OLMADIĞI İÇİN HATA İLE KARŞILAŞILIR

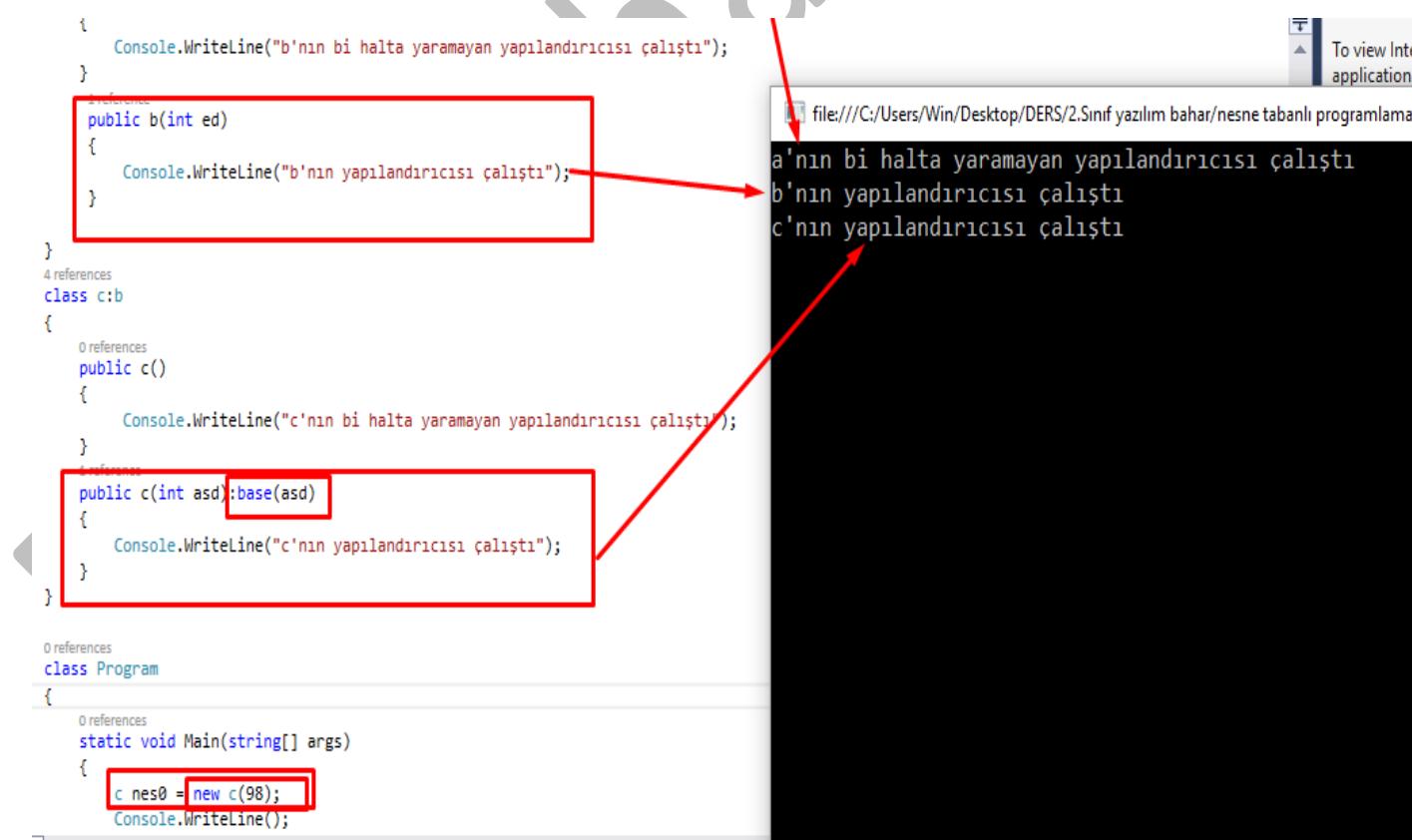
# BASE ANAHTAR SÖZCÜĞÜ:

## base anahtar sözcüğü

Yapıcı metotlar aşırı yüklenmişse türemiş sınıfın yapıcı metotları çağrırlarken belli değerlerle temel sınıfında yapıcı metodunun çağrılması mümkündür ve bu işlem **base** anahtar sözcüğü ile yapılır.

base anahtar sözcüğü ile parametre gönderimi yalnızca yapıcı metotlarla kullanılabilir. Yani base anahtar sözcüğünü yalnızca türemiş sınıftaki yapıcı metoda ekleyebiliriz ve base anahtar sözcüğünün ana sınıfta var olan bir yapıcı metodu belirtmesi gereklidir.

## KULLANIMI:



The screenshot shows a Microsoft Visual Studio interface with two main panes. On the left, the code editor displays three classes: 'a', 'b', and 'Program'. The 'a' class has a constructor that prints a message. The 'b' class has a constructor that prints a message. The 'Program' class has a main method that creates an instance of 'c' and prints its message. Red boxes highlight the constructor definitions in 'b' and 'c', and the instantiation line in 'Program'. Arrows point from these highlighted areas to the corresponding output in the 'Output' window on the right. The output window shows three lines of text: 'a'nın bi halta yaramayan yapılandırıcısı çalıştı', 'b'nın yapılandırıcısı çalıştı', and 'c'nın yapılandırıcısı çalıştı'.

```
class a
{
 Console.WriteLine("a'nın bi halta yaramayan yapılandırıcısı çalıştı");
}

class b(int ed)
{
 Console.WriteLine("b'nın yapılandırıcısı çalıştı");
}

class c()
{
 Console.WriteLine("c'nın bi halta yaramayan yapılandırıcısı çalıştı");
}

class Program
{
 static void Main(string[] args)
 {
 c nes0 = new c(98);
 Console.WriteLine();
 }
}
```

To view Intellisense

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama

a'nın bi halta yaramayan yapılandırıcısı çalıştı  
b'nın yapılandırıcısı çalıştı  
c'nın yapılandırıcısı çalıştı

**Üst yapılandırıcıya istenildiği kadar değer yollanabilir.Yeterki üst yapılandırıcıda bu değerleri karşılıyacak bir yapılandırıcı olsun.Üst yapılandırıcı base ile çağrıldığı zaman üst yapılandırıcıda boş parametreli olan çalışmaz.**

**base**, ana sınıfta hangi yapıçı metodun devreye gireceğini de belirtiyor.

**NOT:Şu ana kadar yapılan örnekler çoklu kalıtım üzerinden yapıldı normal kalıtımından farkı sadece bir sınıf başka bir sınıfdan türetiliyor çokluda türetilen sınıfında üretim yapılıyor.Farkları bu şekildedir.**

```
using System;
class A
{ public int OzellikA;
 public A(int a) { OzellikA = a; }
}
class B : A
{ public int OzellikB;
 public B(int b) { OzellikB = b; }
}
class C : B
{ public int OzellikC;
 public C(int c, int b) : base(b)
 { OzellikC = c; }
 static void Main()
 { C nesne = new C(12, 56);
 Console.WriteLine(nesne.OzellikA + " " +
nesne.OzellikB + " " + nesne.OzellikC);
 }
} // Hata verir
```

**burada hata 2 türlü çözülür:**

**ya A sınıfına boş yapılandırıcı metot  
koyulcak**

**yada B'de base ile A'nın parametreli  
yapılandırıcısı çağrılcak.**

## İsim Saklama (Name Hiding)

- Türemiş sınıfıta bazen temel sınıfıktaki üye elemanları aynı isimli bir eleman veya metot tanımlanmış olabilir. Bu durumda temel sınıfıktaki elemana veya metoda normal yollarda erişmek mümkün değildir çünkü türeyen sınıfıktaki eleman veya metot temel sınıfıktaki elemanı veya metodu gizlemiştir.
- Temel sınıfıktaki elemana ve metoda erişmek için yine **base** anahtar sözcüğünden faydalananır. **base** ile hem özelliklere hem de metodlara erişilebilir.
- base** anahtar sözcüğünün temel elamana ulaşma şeklindeki kullanımı **this** referansına benzemektedir.
- this** referansı kendisini çağrıran sınıfı temsil ederken **base** anahtar sözcüğü türetmenin yapıldığı temel sınıfı temsil eder.

```
using System;
class A
{
 public int a;
 public A() { a = 1; }
 public void yaz() {...}
}
class T : A
{
 public new int a;
 public int b;
 public T() { a= 2; b = base.a; }
 public void yaz() { base.yaz(); ...}
}
```

# SANAL METOTLAR (VIRTUAL METOTS)

Çok biçimlilik olayını öncelikle hatırlayalım

Elimizde bu şekilde sınıflarımız olduğunu düşünelim.

```
{
 3 references
 class A
 {
 0 references
 public ~A()
 {
 Console.WriteLine("A çalıştı");
 }
 }
 3 references
 class B:A
 {
 0 references
 public ~B()
 {
 Console.WriteLine("B çalıştı");
 }
 }
 5 references
 class C:B
 {
 3 references
 public ~C()
 {
 Console.WriteLine("C çalıştı");
 }
 }
 0 references
 class Program
```

Burada görüldüğü gibi

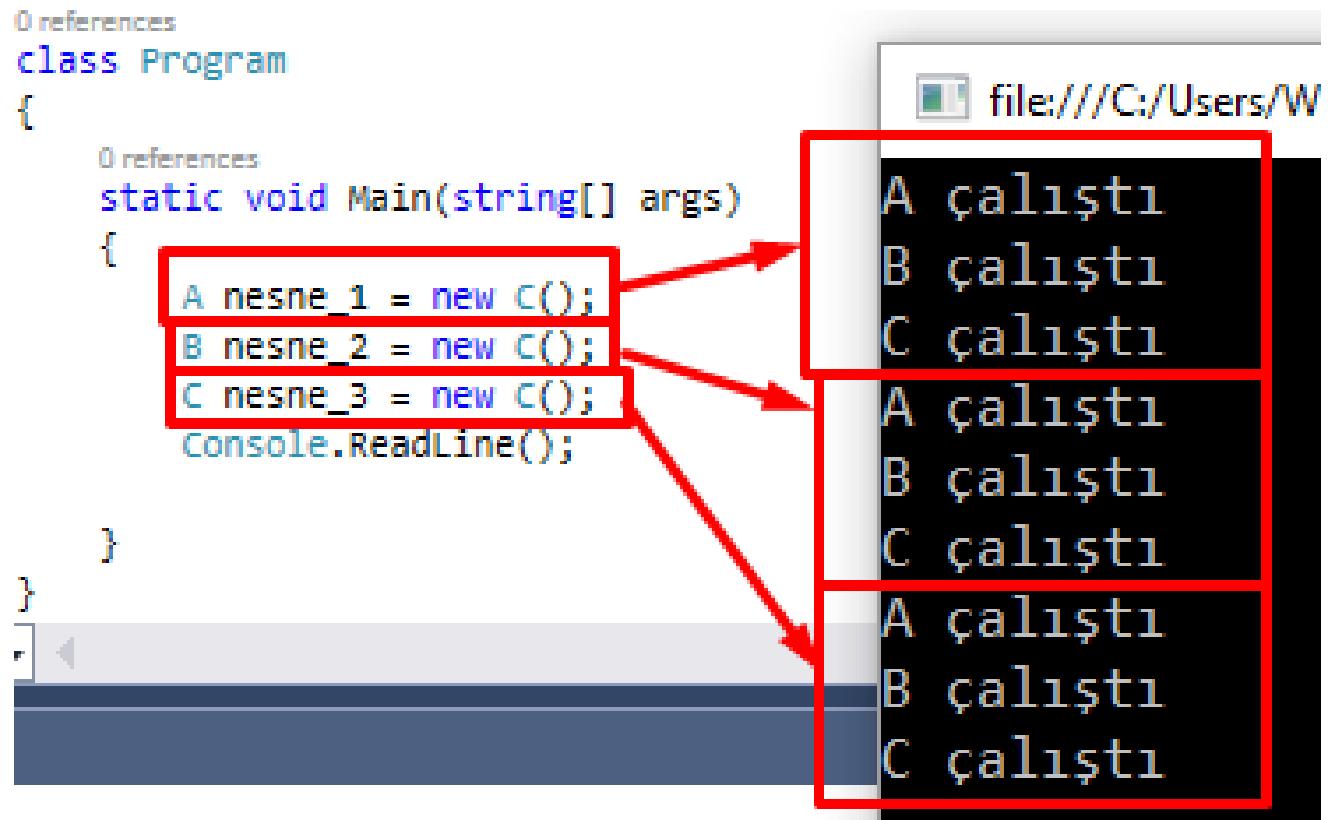
B sınıfı A sınıfından

C sınıfıysa B sınıfından türetilmiştir peki yapılandırıcılar

Mesela:

A nesne= new C();

şeklinde çağrıldığında sonuç ne olur A'nını mı C'nını yapılandıracı çalışır.Yoksa ikisi birdenmi? [C nesne1=new A() \\\HATADIR]



The screenshot shows a C# console application window. On the left, the code defines a class Program with a Main method. Inside Main, three instances of class C are created: A nesne\_1 = new C();, B nesne\_2 = new C();, and C nesne\_3 = new C();. After the object creation, Console.ReadLine() is called to pause the application. On the right, the application's output window displays six lines of text, each starting with the letter 'A' followed by the word 'çalıştı'. This indicates that the application is running three separate instances of class C simultaneously.

```
0 references
class Program
{
 0 references
 static void Main(string[] args)
 {
 A nesne_1 = new C();
 B nesne_2 = new C();
 C nesne_3 = new C();
 Console.ReadLine();
 }
}
```

Durum yapılandırıcılarında bu şekildedir.Fakat buna tam olarak çok biçimlilik yani polimorfizm demek doğru olmayabilir.

Polimorfizm'in tam tanımı tek bir referans üzerinden birbirile ilişkili sınıfların özelliklerine erişebilme özelliğidir.Yani tek bir nesnemiz var fakat biz o nesneye ilişkili nesnelerinde özelliklerine erişebiliyoruz.

Burda bir birleriyle ilişkili denilerek anlatılmak istenen birbirleri üzerinden türeyen nesneleri kast etmektir.

Mesela az önceki örnekte A B ve C birbirleriyle ilişkilidirler haliyle burada polimorfizm'de söz konusu olabilir.Birbirile ilişkili olmayan nesneler için polimorfizm'den bahsetmek yanlış olur.

Polimorfizminde tabi kendine göre kuralları vardır ki bu kurallara dikkat edilmelidir bu kurallar az önceki metod örneklerinde de geçerlidir:

- Polimorfizm çok biçimlilik demektir
- Bu çok biçimlilik üst sınıfıktaki nesnelerin alt sınıfıktaki nesnelere erişimi için geçerlidir.Tam tersi mümkün değildir hatadır.
- Sınıflar kendinden türeyen sınıflarda bulunan özelliklerden kendilerinde olan olan özellikleri kullanabilirler.
- Böylece yukarıdaki sınıflar kendinde bulunan özelliklerin farklı versiyonlarına erişmiş olurlar ki buda alt taraftaki sınıfların tepedeki sınıfın özelliklerini overload etmesiyle mümkün kündür.

(not:Override yapılan özelliklerin üst sınıflarındaki versiyonlarına çok biçimlilik ile ulaşmak mümkün değildir.Burada base anahtar sözcüğü hatırlanmalı.O sözcük bunu sağlar)

Aşağıda sınıf yapısını inceleyelim

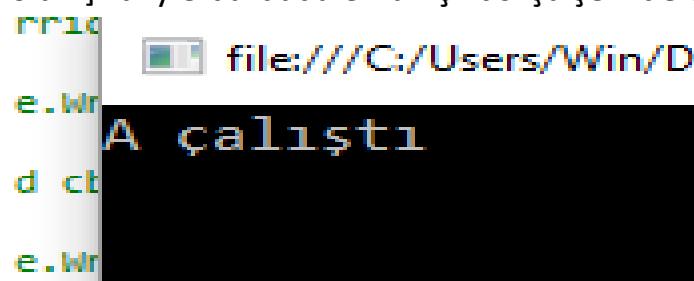
```
3 references
class A
{
 1 reference
 public void abc()
 {
 Console.WriteLine("A çalıştı");
 }
}
2 references
class B:A
{
 0 references
 public void abc()
 {
 Console.WriteLine("B çalıştı");
 }
}
/*class C:B
```

görüldüğü gibi A'dan türünen bir B sınıfı var. Polimorfizm'de baskınlık dikkate alınır yukarıdaki sınıflar daha baskındır bu tip bir durumda eğer bunu yazsaydık ne sonuç alırdık.

```
0 references
static void Main(string[] args)
{
 A nesne_a = new A();
 B nesne_b = new B();
 nesne_a = nesne_b;
 nesne_a.abc();
 Console.ReadLine();
```

**burada A nesne\_a=new B() yapılsa da sonuç değişmez.**

Polimorfizm'de baskınlığa bakılır demistik aslında yapılandırmacıların çalıştırılma sırasında baskınlık önemliydi genel olarak bu yapıda baskınlık önemlidir. Bakın olan sınıf dikkate alınır atamalar, new anahtarları istenildiği kadar kullanılın baskın sınıf nasıl istiyorsa öyle olur [baskın sınıf:hiyerarsik olarak daha üstte olan.] Haliyle burdada ekran çıktısı şu şekilde olucaktır.



```
file:///C:/Users/Win/D
A çalıştı
```

Fakat hatalardan biride B'yi A'ya atamak değilde A'yı B'ye atamaktır. Bu bir hatadır aşadakiler yukardakilere erişemezler.

Örnekde de görüldüğü gibi hata ile karşılaşılır.

```
static void Main(string[] args)
{
 A nesne_a = new A();
 B nesne_b = new B();
 nesne_b = nesne_a; // HATA
 nesne_a.a();
 Console.ReadLine();
```

Base anahtar sözcüğüyle türetilen sınıfta değiştirilen özelliği ana sınıftaki versiyonu ile kullanabildiğimizi biliyoruz. Polimorfizm ile de üstdeki sınıf nasıl isterse öyle özellik seçebilceğimizi biliyoruz yani şimdije kadar görmesek birazdan görüşeceğimiz override ve visual komutları içinde bulduğumuz sanal metod'lar konusunun merkezi niteliği taşır. Orada kısmen de olsa base'nin tam tersini yapıcıaz diye biliriz. Tam olarak tam tersi diyememizin sebebiyse burada alt sınıfında üst sınıfa o özelliği kullanabilmesi için izin vermesi ve üst sınıfın erişiceği metodu bu şekilde tanımlaması gerekmektedir.

Eğer türeyen sınıf sanal metodu devre dışı bırakmamışsa temel sınıftaki sanal metod çağrıılır. Çağrılan metodun hangi türde ait olduğu ise çalışma zamanında belirlenir. Hangi metodun çağrılacağının çalışma zamanında belirlenmesine geç bağlama (late binding) olarak isimlendirilir.

Bu şekilde aynı nesne referansı üzerinden bir çok sınıfa ait farklı versiyonlardaki metodların çağrılabilmesi çok biçimlilik (polymorphism) olarak adlandırılır.

# SANAL METOTLAR:

Metodun sanal olduğu **virtual** ile bildirilir.

Türeyen sınıfda sanal yani virtual olan metodu devre dışı bırakmak için **override** kullanılır

Statik metodlar sanal olarak bildirilemez.

Aşağısı önemlidir.

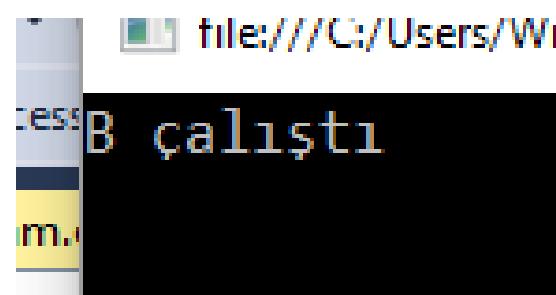
Türeyen sınıfta devre dışı bırakılan metodların temel sınıftaki sanal metodların ismi ve parametrik yapısı temel sınıftaki metodun parametrik yapısı ile aynı olmalıdır.

Aşadaki örneği inceleyelim:

```
3 references
class A
{
 2 references
 public virtual void abc()
 {
 Console.WriteLine("A çalıştı");
 }
}
3 references
class B:A
{
 2 references
 public override void abc()
 {
 Console.WriteLine("B çalıştı");
 }
}
0 references
```

```
static void Main(string[] args)
{
 A nesne_a = new A();
 B nesne_b = new B();
 nesne_a = nesne_b;
 nesne_a.abc();
 Console.ReadLine();
}
```

çıktı:



```
file:///C:/Users/Wi
ness
m.
B çalıştı
```

Aşağıdaki örnek ise çok data entresan bir örnektir:

```
using System;
class A
{ virtual public int ozellik
 { set{}
 get{return 12;}
 }
}
class B:A
{ override public int ozellik
 { get{return 100;}
 set{Console.WriteLine("Bu bir denemedir");}
 }
 static void Main()
 {
 B nesne=new B();
 A nesne2=new A();
 nesne2=nesne;
 Console.WriteLine(nesne2.ozellik);
 nesne2.ozellik=200;
 }
}
```

Ekran Çıktısı  
100  
Bu bir denemedir

Yani özetle virtual olmanın özelliği izinli olarak base'yi tersten kullanmadır diyebiliriz. Karşı taraf o metot için hem erişime izin vercek hem parametre sayısı ve tipi aynı olacak hemde override olarak tanımlıycak ki üst sınıf alt sınıfın bu özelliğini kullanabilisin.

# **Özet(Abstract )(Soyut) Sınıflar**

**Başka sınıfların kendinden türeyebilmesi için temel özelliklerini barındırırlar.**

**Gövdeleri yoktur**

**Aynı zamanda abstract(özet) metotlarda içerebilirler yani gördesiz metot içerirler**

**Bu metotlara virtual yazmaya gerek yoktur zaten virtual'dan daha özetdir**

**Özet sınıflarda özet metotlar olduğu gibi özet olmayan metot'larda bulunabilir.**

**Fakat tam tersi yani normal bir sınıfta özet metot kullanmak doğru değildir.Hatadır**

**Hiyerarşik yapının tepesindedirler.**

**Referans ile tanımlanamazlar.Diğer sınıflarda belirli temel niteliklerin kesin olarak barındırılması için kendileri vardır.Kendinden sınıf türemesi için tanımlanır**

**abstract class x yazmak yeterlidir**

**yine aynı şekilde abstract public int özellik{...}'de yeterlidir.**

**Örnek vermek gerekirse kesin olarak kullanılacak metot'ların yanı sıra get set ayarları yapılacak değişkenler varsa bunu zorunlu kılmak için yapılabilir.Yani bazı zorunlu durumların yerine getirilmesini sağlar.**

## Örnek:

```
using System;
abstract class A
{ abstract public int ozellik { set; get; } }
class B:A
{
 override public int ozellik
 { get{return 100;}
 set{Console.WriteLine("Bu bir denemedir");}
 }
 static void Main()
 { B nesne=new B();
 Console.WriteLine(nesne.ozellik); nesne.ozellik=200; }
}
```

arsı

```
using System;
abstract class A
{ abstract public int ozellik { set; } }
class B:A
{
 override public int ozellik
 { get{return 100;}
 set{Console.WriteLine("Bu bir denemedir");}
 }
 static void Main()
 { B nesne=new B();
 Console.WriteLine(nesne.ozellik); nesne.ozellik=200; }
 } // hata verir çünkü temel sınıfındaki özellik bloğunda get yoktur. Aynı
 durum set içinde geçerli olurdu.
```

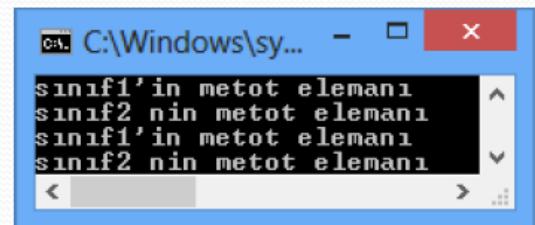
# Arayüzler(INTERFACES)

Abstract yapılara benzer fakat abstract'tan farkı metot'ların v.b. tamamen gövdesiz olmasıdır.

Arayüzler ile referanslar (nesneler) oluşturulabilir, ama new anahtar sözcüğü kullanılmaz . Arayüz referansları tek başına bir anlam ifade etmez fakat kendisini uygulayan bir sınıf nesnesinin referansı atanabilir. Bu durumda arayüz referansı ile arayüzde bulunan metot ya da özellikler hangi sınıf referansı tutuluyorsa oradan çağrılabılır.

```
using System;
interface arayüz { void metot1(); }

class sınıf1 : arayüz
{ public void metot1()
 { Console.WriteLine("sınıf1'in metot elemanı"); }
}
class sınıf2 : arayüz
{ public void metot1()
 { Console.WriteLine("sınıf2 nin metot elemanı"); }
}
public class start
{
 static void Main()
 { arayüz a;
 sınıf1 s1 = new sınıf1();
 sınıf2 s2 = new sınıf2();
 a = s1; a.metot1();
 a = s2; a.metot1();
 s1.metot1(); s2.metot1();
 }
}
```



B'a'

## DİKKAT:

### Arayüzler (Interfaces)

- Arayüz tanımlamalarında dikkat edilecek bazı kısıtlamalar vardır:

- Arayüz elemanları static olamaz.
- Arayüz elemanları public yapıdadır. Ayrıca erişim belirteci ile bildirilemez.
- Üye değişken içeremezler.
- Yapıcı ve yıkıcı metotlar tanımlanamaz ya da bildirilemez.

```
• interface IArayuz
 {
 • int Metot1();
 • int Metot2();
 • int sahteozellik { set; get; }
 • int this[int indeks] { get; }
 }
```

Şimdiye kadar sınıfların türetmeleriyle ilgili abstract bile olsa türetmenin tek sınıf üzerinde olduğunu gördük

Fakat burada bir istisna vardır:

- Sınıflar istenildiği kadar interface'den türetilabilir.
- Interface'lerde birbirlerinden türetilabilir

# sealed anahtar sözcüğü:

Bazı durumlarda sınıfların türetilmemesi istenebilir.

Bu durumda sınıfın türetilerek başka sınıflar oluşturulmasının önüne geçmek için türetilmesi istenmeyen sınıf sealed ile tanımlanmalıdır

Dikkat edilmelidir ki sealed olarak tanımlanan sınıflar:

abstract olamaz, static yapı içeremez...

## sealed class Sınıf

```
{
}
```

Partial (kısmi) Yapılar

Normalde class'larımız interfacelerimiz v.s. tek bir .dll veya .cs uzantılı klasörde bulunurdu. Fakat partial ile aynı isimdeki yapılar bölünerek farklı konumlarda aynı başlık altında tanımlanabiliyor:

```
• Ozellikler.cs
• partial class Sınıf
{
 public int x;
 public int y;
}
```

```
• Metotlar.cs
• partial class Sınıf
{
 public int Topla()
 {
 }

 public void EkranaYaz()
 {
 }
}
```

# Partial (Kısmi) Tipler

- Birleştirilmesini istediğimiz bütün aynı isimli türleri partial olarak bildirmeliyiz. Yalnızca birisini partial olarak bildirmek yeterli değildir.
- Sınıflar, yapılar ve arayüzler partial olarak bildirilebilir.
- Kısmi türlerden biri sealed ya da abstract anahtar sözcüğüyle belirtilmişse diğerinin de belirtilmesine gerek yoktur.
- Partial türler minimum üye eleman düzeyinde iş görürler. Yani aynı metodun gövdesinin bir kısmını bir dosyada, başka bir kısmını başka bir dosyada yazmak partial türler ile de mümkün değildir.

Baybars

# İSTİSNAİ DURUM YÖNETİMİ(EXCEPTION HANDLING)

Program ne kadar iyi yazılırsa yazılışın hataların oluşma olasılığı bir çok durumda mümkündür.

Özellikle çalışma alanında meydana gelebilecek hatalar kod yazılırken tespit edilemeyebilir.

Tüm hatalara karşı tek tek tedbir almak çok zor bir iştir.

Bu yüzden modern dillerde çalışma alanında meydana gelebilecek hataları yakalamak için bir hata yakalama mekanizması mevcuttur.

Çalışma zamanında beklenmeyen bir hata sonucunda oluşturulan nesnelere **istisnai durum sınıf nesneleri** denir. .NET sınıf kütüphanesinde çok sık oluşabilecek hatalara karşı istisnai durum sınıfları tasarlanmıştır. Bu sınıflar hata ile ilgili çeşitli bilgileri tutmaktadır.

.NET sınıf kitaplığında istisnai durum sınıfı hiyerarşisindeki en temel sınıf System isim alanı içerisinde yer alan **System.Exception**'dır. Exception(istisna) sınıfı oluşan hatalar için çok genel bilgiler verdiğinden türemiş diğer bilgiler sınıflar kullanılır:

- SystemException, ArgumentException, StackOverflowException, ArithmeticException, IOException, IndexOutOfRangeException...

İstisnai durumları yakalamak için dört tane anahtar sözcükten faydalanılır:

**try, catch, finally ve throw**

try, catch ve finally sözcükleri birer kod bloğunu temsil ederken, throw ise bir hatanın nesnesinin oluşturulmasını sağlar.

**try** : Hatanın kontrol edileceği kod bloğunun yazılacağı kısımdır. Yani bu blokta bir hata meydana gelirse ilgili hata sınıfı nesnesi oluşturulacaktır.

**catch** : try bloğunda yakalanan hataya göre işlemlerin yapılmasını sağlayan bloktur.

**finally** : try ve catch bloklarında açılan kaynakların kapatılması burada gerçekleşir. Kodlar hata oluştursa dahi çalıştırılır. Bu bloğu belirtme zorunluluğu yoktur.

**throw** : try bloğu içerisinde belirli bir hata olduğunda ilgili hata sınıfı nesnesini oluşturmak için kullanılır. Eğer try içindeki nesneler zaten hata oluşturuyorsa throw'un kullanılmasına gerek olmayabilir.

```
using System;
class Program
{
 static void Main()
 {
 int[] x = new int[5];
 try
 { x[6] = 25; //hatanın fırlatıldığı bölüm
 }
 catch
 { Console.WriteLine("Hata Oluştu...");
 //kural dışı tip için yönetici
 }
 finally
 { Console.WriteLine("Finally Bloğu...:");
 //kaynakların temizlenmesi
 }
 }
}
```

İlgili hata sınıfı nesnesi kullanıldığında hata hakkında daha ayrıntılı bilgilere sahip olunabilir.

Eğer catch tanımlamasında bir nesne bulunursa sadece o hata meydana geldiğinde catch bloğu çalıştırılacaktır.

try bloğu içersinde metodlar da çağrılabılır. Bu sayede metod içinde oluşan çalışma hataları da tespit edilebilir.

Bir try bloğu içersinde birden fazla hata oluşabilme ihtimali vardır. Bu durumda her bir hata durumu için birer catch bloğu tanımlanabilir.

İç içe geçmiş try blokları da kullanmak mümkündür.

```
try
{ //A
 try
 { //B }
 catch { //C //İçteki catch bloğu }
 finally { //D //İçteki finally bloğu }
}
catch { //Dıştaki catch bloğu }
finally { //Dıştaki finally bloğu }
```

B'a'

**ArrayTypeMismatchException:** Depolanmakta olan değerin tipi dizi tipi ile uyumsuz.

**DivideByZeroException:** Sıfıra bölümme hatasında fırlatılır.

**IndexOutOfRangeException:** Dizi indexi sınıf dışına taşılığında fırlatılır.

**InvalidOperationException:** Programın çalışması sırasında geçersiz tür dönüşümü yapıldığında fırlatılır.

**OverflowException:** Aritmetik taşıma meydana geldiğinde fırlatılır.

**NullReferenceException:** Null referans üzerinde işlem yapıldığında fırlatılır.

**StackOverflowException:** Yiğin taşmıştır.

**FormatException:** Metotlarda yanlış formatta parametre gönderildiğinde fırlatılır.

**ArithmeticException:** Bu sınıf ile DivideByZeroException ve OverflowException hataları yakalanabilir. Aritmetik işlemler sonucunda oluşan istisnai durumlarda fırlatılır.

**OutOfMemoryException:** Programın çalışması içi yeterli bellek miktarı olmadığında fırlatılır.

```
• using System;
• class deneme
• {
• static void Main()
• {
• try
• {
• int[] a=new int[2];
• Console.WriteLine(a[3]);
• }
• catch(IndexOutOfRangeException)
• {
• Console.WriteLine("Dizi sınırları aştı");
• }
• }
• }
```

```
• using System;
• class deneme
• {
• static void Main()
• {
• for(;;)
• {
• try
• {
• Console.WriteLine("Lütfen çıkmak için 0 ya da 1 girin:");
• int a=Int32.Parse(Console.ReadLine());
• int[] dizi=new int[2];
• Console.WriteLine(dizi[a]);
• break;
• }
• catch { continue; }
• }
• }
• }
```

Lütfen çıkmak için 0 ya da 1 girin: 2  
Lütfen çıkmak için 0 ya da 1 girin: 5  
Lütfen çıkmak için 0 ya da 1 girin: 1  
0

TRY → HATA OLUŞABİLCEK BÖLÜMÜ DENER

CATCH → HATA OLUŞTUĞUNDA VERİLCEK TEPKİYİ BELİRTİR

FINALLY →

THROW →

```

• using System;
• class hatayakalama1
• { public static void Main()
• {
• { try
• { int a=50,b=0; Console.WriteLine(a/b); }
• catch(DivideByZeroException e)
• { Console.WriteLine("Sıfıra bölünme hatası
yakalandı"); }
• finally
• { Console.WriteLine("\n program
sonlandırılıyor"); }
• }
• }
• Çıktı: Sıfıra bölünme hatası yakalandı
• program sonlandırılıyor

```

```

• using System;
• class deneme
• { static void Main()
• { try { Metot(); }
• catch(IndexOutOfRangeException nesne)
• { Console.WriteLine("Metodu kullananda
hata yakalandı"); }
• }
• static void Metot()
• { try { int[] a=new int[2];
Console.WriteLine(a[3]); }
• catch(IndexOutOfRangeException nesne)
• { Console.WriteLine("Metodun
kendisinde hata yakalandı."); }
• }
• Çıktı: Metodun kendisinde hata yakalandı.

```

## throw anahtar sözcüğü

- Hatalar otomatik olarak üretilmelerine rağmen bazı özel durumlarda **throw** ile kendi istisnai durum nesnelerimizi oluşturabiliriz.
- **throw** anahtar sözcüğü istisnai durum sınıf nesnesi türünden bir nesne gönderir.
- Bir hata nesnesi **throw** anahtar sözcüğü yardımıyla şöyle fırlatılabilir:
  - **throw new** IndexOutOfRangeException("Dizinin sınırları aşıldı");
- veya
  - **IndexOutOfRangeException nesne=new** IndexOutOfRangeException("Dizinin sınırları aşıldı");
  - **throw nesne;**

```

• using System;
• class Notlar KENDİ HATAMIZI OLUŞTURDUK THROW İLE
• { private int mNot;
• public int Not
• { get{return mNot;}
• set
• {
• if(value>100)
• throw new FazlaNotHatali();
• else if(value<0)
• throw new DusukNotHatali();
• else
• mNot=value;
• }
• }
• public class FazlaNotHatali:ApplicationException
• {
• override public string Message
• {
• get{ return "Not 100'den büyük olamaz.";}
• }
• }
• public class DusukNotHatali:ApplicationException
• {
• override public string Message
• {
• get{return "Not 0'dan küçük olamaz.";}
• }
• }
• class Ana
• { static void Main()
• { try
• { Notlar a=new Notlar();
• Console.Write("Not girin: ");
• int b=Int32.Parse(Console.ReadLine());
• a.Not=b;
• Console.WriteLine("Notu başarıyla girdiniz.");
• }
• catch (Exception nesne)
• { Console.WriteLine(nesne.Message); }
• }
• }
• }
```

**NOT:**

**FİNALLY VE THROW TAM OLARAK  
ANLAŞILAMADI TEKRAR ET**

# TEMSİLCİLER

## (DELEGATES)

Temsilciler metotlar temsil ederler. Temsilcilerin nesneleri oluşturulabilir ve bu nesneler metotları temsil ederler.

Delegate'ler metot'ların bellekdeki adresini tutarlar.

Genel kullanımları:

[Erişim belirleyicisi] **delegate** dönüşdeğeri **temsilci(parametreler)**  
**delegate int temsilci(int a,string b);**

Temsilcilerin kullanılmasının amacı derleme zamanında belli olmayan metotların çalışma zamanını belirlemektir.

Temsilciler kendi imzalarına uygun herhangi bir metodu temsil edebilirler. Bu metotlar statik de olabilir.

**delegate** bir veri tipi olduğundan erişim belirteci ile tanımlanabilir. Return ile değer gönderilir ise en son gönderilen belirli değişkene atanır.

```
using System;
class delegeler
{
 public delegate void temsilci();
 public static void Bilgisayar()
 {
 Console.WriteLine("Bilgisayar Öğretmenliği");
 }
 public static void Elektronik()
 {
 Console.WriteLine("Elektronik Öğretmenliği");
 }
 public static void Main()
 {
 temsilci nesne = new temsilci(Bilgisayar);
 nesne();
 nesne = new temsilci(Elektronik);
 nesne();
 }
}
```

Elektronik Öğretmenliği  
Press any key to

İMZALARI AYNI

KULLANIMI

```
public delegate void temsil();
public static void toplam_bul()
{
 Console.WriteLine("toplam bul çalıştı");
}
0 references
static void Main(string[] args)
{
 temsil baybars_temsili;
 baybars_temsili = new temsil(toplam_bul);
 Console.ReadLine(),
}
```

Bir temsilci birden fazla metodu temsil edebilir. + ve - operatörleri ile temsilciye metot ekleme ve çıkarma yapılabilir.

+ ve - metotları yerine daha pratik olması için += ve -= de kullanılabilir.

Çoklu temsilci çağrıldığında metotlar temsilciye eklenme sırasına göre çalıştırılır.

```
delegate void temsilci(int a);
class deneme
{
 public static void Metot1(int a)
 { Console.WriteLine("Metot1 çağrıldı." + a); }
 public static void Metot2(int a)
 { Console.WriteLine("Metot2 çağrıldı." + a); }
 public static void Metot3(int a)
 { Console.WriteLine("Metot3 çağrıldı." + a); }
}
class Program
{
 static void Main()
 {
 temsilci nesne = null;
 nesne += new temsilci(deneme.Metot2); nesne += new temsilci(deneme.Metot1);
 nesne += new temsilci(deneme.Metot3); nesne(1);
 Console.WriteLine("***"); nesne -= new temsilci(deneme.Metot1);
 nesne(2);
 }
}
```

Ekran Çıktısı:  
Metot2 Çağrıldı.1  
Metot1 Çağrıldı.1  
Metot3 Çağrıldı.1  
\*\*\*  
Metot2 Çağrıldı.2  
Metot3 Çağrıldı.2

önce new ile metot parametreye nesne üzerinden yazılıyo bu işlem temsilciye metot aşama nesne(2) şeklindekilerde referansa parametre atamadır

yani 2 aşaması vardır temsilcilerin:

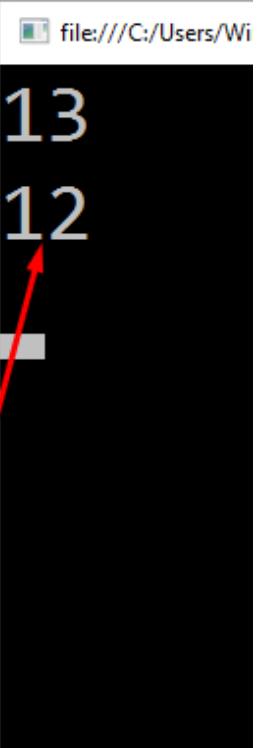
temsilciye nesne atama

```
nesne += new temsilci(deneme.Metot2);
```

temsilci referansına değer atama(değer parametreli bir delegate ise)

```
nesne(2);
```

```
class sinif
{
 public delegate int temsil(int x,int y);
 1 reference
 public int toplam_bul(int a,int b)
 {
 return a + b;
 }
 1 reference
 public int carpim_bul(int a,int b)
 {
 return a*b;
 }
 0 references
 static void Main(string[] args)
 {
 sinif snf=new sinif();
 temsil baybars_temsili;
 baybars_temsili = new temsil(snf.toplam_bul);
 Console.WriteLine(baybars_temsili(4, 9));
 baybars_temsili = new temsil(snf.carpim_bul);
 Console.WriteLine(baybars_temsili(3, 4));
 Console.ReadLine(),
 }
}
```



```
baybars_temsili = new temsil(snf.toplam_bul); →METOT ATAMA
```

```
(baybars_temsili(4, 9)); →PARAMETREYE DEĞER ATAMA
```

```
class Program
{
 public delegate void Temsilci(int a, int b);

 public static void Topla(int a, int b)
 {
 Console.WriteLine("Toplam:{0}", a + b);
 }

 public static void Cikar(int a, int b)
 {
 Console.WriteLine("Fark :{0}", a - b);
 }

 static void Main()
 {
 Temsilci t = new Temsilci(Topla);
 t += new Temsilci(Cikar);

 t(10, 8);
 }
}
```

Kendi yarattığımız temsilciler gizlice System isim alanındaki Delegate sınıfından türer. Dolayısıyla kendi yarattığımız temsilcilerin nesneleri üzerinden bu sınıfın static olmayan üye elemanlarına erişebiliriz.

```
using System;
class Ana
{
 delegate void temsilci();
 static void Metot1()
 {
 Console.WriteLine("Metot1");
 }
 static void Metot2()
 {
 Console.WriteLine("Metot2");
 }
 static void Main()
 {
 temsilci t = new temsilci(Metot1);
 t += new temsilci(Metot2);
 Delegate d = t; temsilci c = (temsilci)d; // tür dönüşümüne dikkat
 c(); // metodların sırasıyla çağrılması
 }
}
```

Ekran Çıktısı:  
Metot1  
Metot2

## **TEMSİLCİLERİN BİR BAŞKA KULLANIMI[İSİMSİZ METOTLAR]**

Metotlar olmadan temsilci oluşturulmasına isimsiz metotlar adı verilir.

Örnek:

```
using System;
class isimsiz
{
 delegate double temsilci(double a, double b);
 static void Main()
 {
 //temsilci nesnesine bir kod bloğu bağlanıyor.
 temsilci t = delegate(double a, double b)
 {
 return a + b;
 };
 //temsilci nesnesi ile kodlar çalıştırılıyor.
 double toplam = t(1d, 9d);
 Console.WriteLine(toplam);
 }
}//Gördüğünüz gibi bir temsilci nesnesine bir metot yerine direkt kodlar atandı.
```

## **DELAGATEİNİN SON BÖLÜMÜNE BİR GÖZ AT**

### **PEK ANLAŞILAMADI**

# OLAYLAR

## (EVENTS)

Temsilcilerin özel bir formudur olaylar.



[Erişim belirleyici] **event** [temsilci türü] [olay adı];

```
using System;
delegate void OlayYonetici(); //Olay yöneticisi bildirimi
class Buton //Olayın içinde bulunacağı sınıf bildirimi
{
 public event OlayYonetici Click; //Olay bildirimi
 public void Tiklandi() //Olay meydana getirecek metot
 {
 if(Click!=null) Click(); // Olaya yüklü metodların çağrılması
 }
}
class AnaProgram
{
 static void Main()
 {
 Buton buton1=new Buton();
 buton1.Click +=new OlayYonetici(Click); //Olay sonrası işletilecek metodların eklenmesi
 buton1.Tiklandi(); //Olayın meydana getirilmesi.
 }
 //Olay sonrası işletilecek metot
 static void Click() // Olaydan sonra çalıştırılacak metot ya da metodların static olma zorunluluğu yoktur.
 {
 Console.WriteLine("Butona tıklandı.");
 }
} // Bu program ekrana Butona tıklandı. yazacaktır.
```

Temsilcilerin isimleri tip gibidir yani nesnedirler. Temsilci tanımlandıktan sonra olaylarda herhangi bir temsilci tipinde event anahtarıyla tanımlanmalıdır

Şimdi programımızı derinlemesine inceleyelim. Programımızdaki en önemli satırlar:

**delegate void OlayYonetici();** ve Buton sınıfındaki **public event OlayYonetici Click;** satırıdır. Birinci satır bir temsilci, ikinci satır da bir olay bildirimidir. Bu iki satırdan anlamamız gereken Click olayı gerçekleştiğinde parametresiz ve geri dönüş tipi void olan metodların çalıştırılabilceğidir.

```
public void Tiklandi() { if(Click!=null) Click(); }
```

Burada bir kontrol gerçekleştiriliyor. Burada yapılmak istenen tam olarak şu: Eğer Click olayı gerçekleştiğinde çalışacak bir metot yoksa hiçbir şey yapma. Varsa olayı gerçekleştir.

```
buton1.Click+=new OlayYonetici(Click);
```

Main() metodundaki bu satırda da buton1'in Click olayına bir metot ekliyoruz. Yani buton1 nesnesinin Click olayı gerçekleştiğinde Click metodu çalışacak.

```
public delegate int temsil();
```

0 references

```
class tus
```

```
{
```

```
 public event temsil tiklama;
```

0 references

```
 public void tiklandi()
```

```
{
```

```
 if(tiklama!=null)
```

```
{
```

```
 tiklama();
```

```
}
```

```
}
```

```
}
```

TEMSİLCİMİZ TANIMLANDI

TEMSİLCİ TİPİNDE BİR  
EVENT TANIMLANDI

TIKLAMA EVENTİNE OLAYLAR YÜKLENİR  
FAKAT ÇALIŞICAĞI ZAMANIN  
TETİKLENMESİ TUŞ SINIFIYLA  
TANIMLANAN REFERANS ÜZERİNDEN  
[STATIC İÇİN DURUM FARKLI]  
TIKLANDININ ÇALIŞTIRILMASI GEREKİR.  
EĞER TIKLAMA BOŞ DEĞİLSE TIKLAMAYA  
ATANAN OLAYLAR ÇALIŞTIRIR  
tiklama(); sayesinde

Event'ları daha iyi tanıya bilmek için aşadaki örneği inceleyelim

```
public delegate void temsil();
2 references
class tus
{
 public event temsil tiklama_olayı;
 1 reference
 public void tiklandı()
 {
 if(tiklama_olayı!=null)
 {
 tiklama_olayı();
 }
 }
}
3 references
```

YUKARDAKİ BÖLÜMDE EVENT'LAR VE DELEGATE'LER TANIMLANMIŞTIR.

AŞADA DA METOTLARI OLAN STANDART BİR SINIF VAR EVENTLARIN KULLANIMINDA BU SINIFTAKİ METOTLAR EVENTA YÜKLENİCEKTİR VE TETİKLEME YAPILDIĞINDA TÜM METOTLAR AYNI ANDA ÇALIŞTIRILCAKTIR.

---

```
public class işlemler
{
 1 reference
 public void topla()
 {
 Console.WriteLine("topla çalıştırıldı");
 }
 0 references
 public void çıkar()
 {
 Console.WriteLine("çıkar çalıştırıldı");
 }
 0 references
 public void çarp()
 {
 Console.WriteLine("çarp çalıştırıldı");
 }
}
```

Şimdi tüm bunları başka bir sınıfın ana programında deneyelim

```
class sinif
{
 0 references
 static void Main(string[] args)
 {
 tus buton = new tus();
 islemler islem = new islemler();
 buton.tiklama_olayı += new temsil(islem.topla);
 buton.tiklama_olayı += new temsil(islem.cıkar);
 buton.tiklama_olayı += new temsil(islem.carp);
 buton.tiklandı();
 Console.ReadLine(); TÜM YÜKLENEN METOTLAR
 TETİKLENİR
 }
}
```

buton.tiklanma\_olayı adlı evente temsil tipi tüzerinden metot yükler

```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama/ödev/erkan hoca ödev/ye
topla çalıştırıldı
çıkar çalıştırıldı
çarp çalıştırıldı
```

EĞEĞR buton.tiklandı() şeklinde bir tetikleme yapılamamasıydı metotlar yüklenirdi fakat tetiklenmezdi haliyle ekranada bir şeye yazmadı.

```
class sinif
{
 0 references
 static void Main(string[] args)
 {
 tus buton = new tus();
 islemler islem = new islemler();
 buton.tiklama_olayı += new temsil(islem.topla);
 buton.tiklama_olayı += new temsil(islem.cıkar);
 buton.tiklama_olayı += new temsil(islem.carp);
 //buton.tiklandı();
 Console.ReadLine();
 }
}
```

tetiklenme açıklama satırı yapıldı yani artık kullanılmadığı için tetiklemede olmaz haliyle buton.tiklanma\_olayı adlı event dolu olmasına rağmen çalışmamış olur

```
file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama/ödev/erkan hoca ödev/ye
```

## EVENT'IN PARAMETRELİ KULLANIMIDA ŞU ŞEKİLDEDİR:

```
public delegate void temsil(int a ,int b);
2 references
class tus
{
 public event temsil tıklama_olayı;
1 reference
 public void tiklandı(int k, int z)
 {
 if(tıklama_olayı!=null)
 {
 tıklama_olayı(k,z);
 }
 }
}

4 references
public class islemler
{
 1 reference
 public void topla(int x,int y)
 {
 Console.WriteLine(x + y);
 }
 1 reference
 public void çıkar(int x, int y)
 {
 Console.WriteLine(x-y);
 }
 1 reference
 public void carp(int x, int y)
 {
 Console.WriteLine(x*y);
 }
}
0 references
class sınıf
{
 0 references
 static void Main(string[] args)
 {
 tus buton = new tus();
 islemler işlem = new islemler();
 buton.tıklama_olayı += new temsil(islem.topla);
 buton.tıklama_olayı += new temsil(islem.cıkar);
 buton.tıklama_olayı += new temsil(islem.carp);
 buton.tiklandı(9, 8);
 Console.ReadLine();
 }
}
```



EVENTLARDA TANIMLAMALARI ALIŞILANIN DIŞINDA OLDUĞU İÇİN  
DİKKAT EDİLCEK NOKTALAR DANDIR.

YÖNETİCİ OLARAK BİR DELEGATE TANIMLANMALIDIR

```
public delegate void temsil(int a ,int b);
2 references
```

YÖNETİCİ DELEGE TİPİNDE EVENT TANIMLANMALIDIR.

```
public event temsil tiklama_olayı;
1 reference
public void tiklandı(int k, int z)
{
 if(tiklama_olayı!=null)
 {
 tiklama_olayı(k,z);
 }
}
```

İŞLEM VE TETİKLEMEDE ŞU ŞEKİLDEDİR

```
tus buton = new tus();
buton.tiklama_olayı += new temsil(islem.topla);
buton.tiklandı(9, 8);
```

DİKKAT EDİLMELİDİRKİ EVENTE ULAŞILIP BU EVENTE BİR ŞEY ATAMA  
İŞLEMİNDE EŞİTLİĞİN SAĞ TARAFINDA

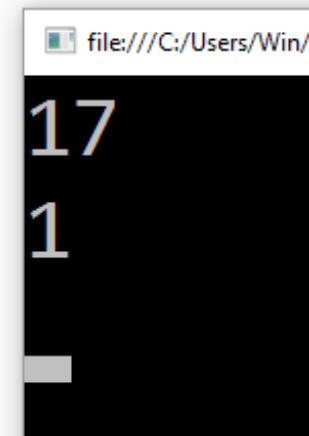
new temcilci\_adı(metot)

temsilci\_adı → [eventin tanımlandığı yönetici delegate ]

(metot) → (parametresi yazılmıycak bir şekilde imzası tutan metot)

Tabi temsilciler bu kadarla sınırlı değildir += kullanılarak event'e  
metot eklendiği gibi metot çıkarmak için -= kullanılır:

```
0 references
class sınıf
{
 0 references
 static void Main(string[] args)
 {
 islemler islem = new islemler();
 tus buton = new tus();
 buton.tiklama_olayı += new temsil(islem.topla);
 buton.tiklama_olayı += new temsil(islem.cıkar);
 buton.tiklama_olayı += new temsil(islem.carp);
 buton.tiklama_olayı -= new temsil(islem.carp);
 buton.tiklandı(9, 8);
 Console.ReadLine();
 }
}
```



Göründüğü gibi islem.carp metodu çıkartılmıştır ve artık tetikleme  
yapıldığında islem.carp çalışmaz.

Event'lar bu kadarla sınırlı değildir nasılıki bir değişken tanımlarken get set ayarları yapabiliyorsak burada da event tanımlanırken add ve remove ile -= ve += işlemlerinde event'ın yapılacakları belirlenir

add → += işleminde yapılacaklar

remove → -= işleminde yapılacaklar

ADD VE REMOVE TANIMLAMA GET VE SET'DE OLDUĞU İÇİN  
HERŞEYİN ELLE YAPILMASINI GEREKTİRİR YİNE GÜVENLİK SAĞLAR  
FAKAT ZAHMETLİDİR

```
using System;
delegate void OlayYonetici();
class Buton
{ public OlayYonetici[] olay = new OlayYonetici[3];
 public int i = 0; public int j = 0;
 public event OlayYonetici Olay
 { add { Console.WriteLine("Olaya metot eklendi =" +(i + 1) + ". Olay");
 if (olay[i] == null) { olay[i] = value; }
 i++;
 if (i == 3) { --i; Console.WriteLine("Olay listesi Dolu"); }
 }
 remove { Console.WriteLine("Olaydan metot çıkarıldı.");
 if (olay[i] == value) { olay[i] = null; }
 i--;
 if (i == -1) Console.WriteLine("Olay bulunamadı");
 }
 }
 public void Tiklandi()
 { for (int j = 0; j <= i; j++) if (olay[j] != null) olay[j](); }
}
class AnaProgram
{ static void Metot1() { Console.WriteLine("Metot1"); }
 static void Metot2() { Console.WriteLine("Metot2"); }
 static void Click() { Console.WriteLine("Butona tıklandı."); }
 static void Main(){ Buton button1 = new Buton();
 button1.Olay += new OlayYonetici(Click); button1.Olay += new OlayYonetici(Metot1);
 button1.Olay += new OlayYonetici(Metot2); button1.Tiklandi();
 button1.Olay -= new OlayYonetici(Metot2); button1.Tiklandi();
 }
}
```

**EKLЕНME OLAYINDA YAPILCAKLAR**  
Olaya metot eklendi =1. Olay  
Olaya metot eklendi =2. Olay  
Olaya metot eklendi =3. Olay  
Olay listesi Dolu  
Butona tıklandı.  
Metot1  
Metot2  
Olaydan metot çıkarıldı.  
Butona tıklandı.  
Metot1

**ÇIKARILMA OLAYINDA YAPILCAKLAR**  
Olaya metot eklendi =1. Olay  
Olaya metot eklendi =2. Olay  
Olaya metot eklendi =3. Olay  
Olay listesi Dolu  
Butona tıklandı.  
Metot1  
Metot2  
Olaydan metot çıkarıldı.  
Butona tıklandı.  
Metot1

**TETİKLENME DURUMDA YAPILCAKLAR**

**add** ve **remove** bloklarının her ikisi de mutlaka aynı anda tanımlanmalıdır. Delege olay sayısı kadar dizi tanımlanmalıdır.

Olaylar da bir tür olduğu için arayüzlerde bildirilebilir. Ayrıca sanal (virtual) ve özet (abstract) olarak da bildirilebilirler.

Yalnız add ve remove içeren olaylar özet olarak bildirilemez.

**Bu bilgilerden de anlaşılacağı üzere *add* ve *remove* eklemenin dizi tanımlanması yapma,o dizi üzerinden değer çekilip alınması,dizi değişkeninin event'ın adıyla aynı olması gibi zahmetleri vardır.**

# ŞABLON TIPLER

Şablon tipler temel olarak şu şekildedir:

```
class islem<sablon>
{
 public sablon degisken;
}
```

şablon olarak bildirilmiştir ve şablon tipinde değişkenler oluşturulmuştur

islem sınıfı tanımlanırken sablon belirlenicektir ve şablon olarak tanımlananlar o belirlenen tipte olacaktır.

Mesela islem sınıfı başka bir sınıfta tanımlandığında eğer int tipinde değişken alınmasını istiyorsak şablonun şu şekilde tanımlarız.

```
islem<int>sayi=new islem<int>();
```

artık degiskenimiz int tipinde olmuştur çünkü şablon tipiyle tanımlanmıştır ve burada sayı değişkeni oluşturulurken şablon'a int değeri verilmiştir artık tüm şablon tipindeki değişkenler int tipinde olur.

```
islem<int>sayi=new islem<int>();
islem<string> kelime = new islem<string>();
islem<double>ondalikli=new islem<double>();
```

Burada islem sınıfı üzerinden sayı,kelime,double olarak farklı tipler tanımlanmıştır.

Aşağıda şablon tipinde tanımlandığı için işlem sınıfındaki değişken şablon'a girilen tipler doğrultusunda oluşur. Görüldüğü gibi hem şablon tipteki değişkenler o tip sınıf tanımlanırken tipin belirtildiği zamana kadar belirsizdir. Tanımlama yapıldığında şablon belirlendiğinde ise şablon tipindekiler üzerinde artık o belirtilen tip üzerinden işlem yapılır.

```
static void Main(string[] args)
{
 islem<int>sayi=new islem<int>();
 islem<string> kelime = new islem<string>();
 islem<double>ondalikli=new islem<double>();
 sayi.degisen = 5;
 kelime.degisen = "baybars";
 ondalikli.degisen = 432.64;
 Console.WriteLine(sayi.degisen + " " + kelime.degisen + " " + ondalikli.degisen);
 Console.ReadLine();
}
```



Tabi şablon illa bir tane olmak zorunda değildir.

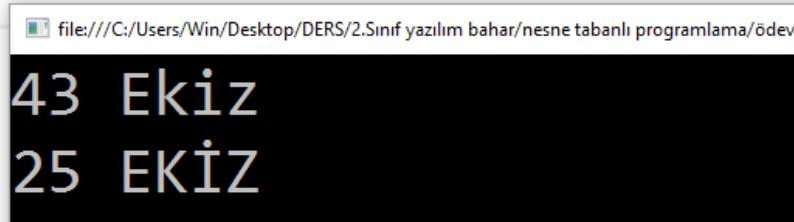
```
class islem<sablon_tip_1,sablon_tip_2>
{
 public sablon_tip_1 degisen_1;
 public sablon_tip_2 degisen_2;
}
```

artık islem sınıfı tanımlandığında 2 tane tip belirtilmelidir.

```

class sınıf
{
 0 references
 static void Main(string[] args)
 {
 islem<int, string> Baybarshan = new islem<int, string>();
 Baybarshan.degisken_1 = 43;
 Baybarshan.degisken_2 = "Ekiz";
 islem<int, string> Girayhan = new islem<int, string>();
 Girayhan.degisken_1 = 25;
 Girayhan.degisken_2 = "EKİZ";
 Console.WriteLine(Baybarshan.degisken_1 + " " + Baybarshan.degisken_2);
 Console.WriteLine(Girayhan.degisken_1 + " " + Girayhan.degisken_2);
 Console.ReadLine();
 }
}

```



Fakat şöyle bir durum vardır ki aşırı yüklenme olma ihtimali olabilir yani 2 tane aynı isimli metodumuzun olduğunu ve bu metodların parametrelerinden birinin normal tipte değişken aldığıını birinde sablon tipinden değişken alırsa aşırı yüklenme olur ve parametresi şablon tipinde değer alan metot çalışmaz önceden tipi tanımlanmış olan metot çalışır.

```

class islem<sablon_tip_1>
{
 2 references
 public void yazdir_metot(sablon_tip_1 x)
 {
 Console.WriteLine("ŞABLON tip parametreli olan metot çalıştı");
 }
 2 references
 public void yazdir_metot(int a)
 {
 Console.WriteLine("NORMAL tip parametreli olan metot çalıştı");
 }
}

```

0 references

```
static void Main(string[] args)
{
 islem<double> Girayhan = new islem<double>();
 islem<int> Baybarshan=new islem<int>();
 Baybarshan.yazdir_metot(89);
 Girayhan.yazdir_metot(85.78);
 Console.ReadLine();
}
```

AŞIRI YÜKLENME OLUR  
ÖNCEDEN TANIMLI  
PARAMETRELİYE GİDER

0 references

```
static void Main(string[] args)
{
 islem<double> Girayhan = new islem<double>();
 islem<int> Baybarshan=new islem<int>();
 Baybarshan.yazdir_metot(89);
 Girayhan.yazdir_metot(85.78);
 Console.ReadLine();
```

file:///C:/Users/Win/Desktop/DERS/2.Sınıf yazılım bahar/nesne tabanlı programlama/ödev/erkan hoc

NORMAL tip parametreli olan metot çalıştı  
ŞABLON tip parametreli olan metot çalıştı

- Şablon tipler herhangi bir tipi temsil ederler. Bu yüzden C#, yalnızca bazı türlere özgü olan operatörler (+, -, \*, ...) kullanmasına izin vermediği gibi şablon tip türünden bir nesne yaratılıp bu nesneye bir değer verilmesine engel olur.
  - Örnekler:
    - `class Sınıf<T> { public int Metot(T a) { return a+2; }}` → HATA
    - Bu sınıf derlenmez.** Çünkü T tipinin hangi tip olduğunu bilmiyoruz. + operatörü bu tip için aşırı yüklenmiş olmayıabilir. Bu yüzden C# bu gibi tehlikeli durumları önlemek için bu tür bir kullanımı engeller. Başka bir kullanım:
      - `class Sınıf<T> { public int Metot(T a) { T nesne=0; }}` → HATA
    - Yine burada da bir hata söz konusudur.** Çünkü her tipten nesneye 0 atanmayabilir. Benzer şekilde yapı nesnelerine de null değer atanamaz.

- default** operatörü bir şablon tipin varsayılan değerini elde etmek için kullanılır.  
Örnek:
  - `class Sınıf<T> { public void Metot( { T nesne=default(T); } }`
  - Varsayılan değer bazı türler için 0 (int, short, float vs.) bazı türler için null (tüm sınıflar) bool türü için de false'tur.

Şablonların hangi tipleri kabul ettiği belirtilebilir.

Bu belirtilme durumunda şablon ya belirtilen kısıta uyucaktır yani o tiplerden olucaktır yada o tiplerden türetilecektir.

Mesela kısıt `class_adi` şeklinde bir class ise tanımlama yapıılırken şablon'a ya `class_adi` tipinde bir şey atanmalıdır yada `class_adi`'dan türetilmiş bir şey atanılmalıdır.

Tekrar hatırlanmalıdır ki şablon tipinde tanımlanan değişkenler için `+` `*` - / operatörleri kullanılamaz ve atama yapılamaz fakat `default` ile atama durumuyle ilgili bir istisna vardır.

**struct** şablon tip yalnızca yapılar olabilir.

**class** şablon tip yalnızca sınıflar olabilir.

**new()** şablon tip yalnızca nesne yaratılabilen tiplerden olabilir. (tür abstract, static, vb. olamaz)

**inheritance (türetme)** şablon tip mutlaka belirtilen bir türden türemiş olmalıdır.

**interface** şablon tip mutlaka belirtilen bir arayüzden türemiş bir sınıf olmalıdır.

Kısıtlar **where** anahtar sözcüğüyle yapılmaktadır.

**where T : struct** T tipi değer (value) tipleri olmalıdır.

**where T : class** T tipi referans (reference) tipleri olmalıdır.

**where T : new()** T tipi yükleyicisi (constructor) parametresiz olan bir tip olmalıdır.

**where T : class\_name** T tipi oluşturduğunuz bir sınıf yada bu sınıfı ile genişletilmiş alt sınıflar olmalıdır.

**where T : interface\_name** T tipi belirtilen interface ile genişletilmiş bir obje olmalıdır.

Referans tipleri : Dynamic, Delegate, Interface, Strings, Object, Class.

Değer tipleri ise : int , float , double,char vb. tipleridir

Göründüğü gibi

where sablon\_1:struct

ile kısıtlama yaparak sablon\_1'e atanılacak tiplerin  
double,int,float,char gibi tipler olmasını sağlayabiliriz

kısıtlımız class olsaydı bu sablonumuza kesinlikle bir referans tipinde  
nesnenin atanmasının zorunlu olmasını  
sağlardı:string,object,delegate,interface,class,dynamic...

class ve class değişkeni farklıdır.

Aşağıda D class'ı için A class'ı kısıtı koyulmuştur yani T şablonuna ya A  
atancak yada A'dan türeyenler;B,C

Fakat class deseydi T şablonuna her class atanabilirdi

```
class A{}
class B:A{}
class C:A{}
class D<T> where T:A{} //A sınıfı veya A dan türeyen sınıf nesni olabilir.
```

Kısıt illa bir tane olmak zorunda değildir.

Tabii ki bir şablon tipe birden fazla kısıt eklenebilir. Bu durumda kısıtlar virgülle  
ayrılır. Örnek:

```
class Sinif<T> where T: class, IComparable, new() { }
```

Kısıtlarda diğerlerinin sırası önemli değildir. Ancak -varsayı **new()** kısıtı en sonda  
olmalıdır. Bir sınıfa eklenen birden fazla şablon tip varsa her biri için ayrı ayrı kısıtlar  
koyulabilir. Örnek:

```
class Sinif<T,S> where T:IComparable,IEnumerable where S:AnaSinif
```

DİKKAT:Şablonlar main içeren sınıf için tanımlanamazlar.

Şablonlar metot interface class v.b. için tanımlanabilirler şimdije kadar hep class'lar için şablon gördük diğerleri içinde durum farklı değildir mantık aynıdır.

```
static T EnBuyuk<T>(T p1,T p2) where T:IComparable
{ T geridonus=p2;
 if(p2.CompareTo(p1)<0) geridonus=p1;
 return geridonus; } }
```

new() ile kısıtlanmak demek şablona atanılacak tipin kesinlikle boş parametreli yapılandırıcıyı içeriyor olması demektir

Ancak halen new operatörüyle şablon tip türünden nesne oluşturamayız. Bunun için şablon tipe new() kısıtını eklemeliyiz. Yani şablon tipe yalnızca nesnesi oluşturulabilen türler koyulabilecek.

```
class A
{ public int a;
 public A() {a=5;} public A(int c) {a=c*5;}

 public virtual void yaz() { Console.WriteLine(a); }
}
class D<T> where T:new()//parametresiz yapıcı olacak
{ private T info;
 public T Bilgi //new ile nesne oluşturuldu.
 { get { return info; }
 set { info = value; }
 }
}
```

BU ÖRNEKDE DEĞİŞKEN D TANIMLANIRKEN NEW İLE OLAN BÖLÜMDE PARAMETRE BOŞ OLMAK ZORUNDA YANI ŞABLONA ATANAN TİP BOŞ PARAMETRELİ YAPILANDIRICIYI ÇALIŞTIRABİLİYO OLMALI.

```
class AnaProgram
{
 static void Main()
 {
 A x = new A();
 D<A> nesne1 = new D<A>();
 nesne1.Bilgi = x;
 Console.WriteLine(nesne1.Bilgi.a); // Çıktı → 5
```

SORUN YOK KİSITA UYUYOR  
BOŞ PARAMETRELİ METOT  
KULLANILDЫ

BURDA SORUN YOKTUR BİRDE AŞADAKİ ŞEKİLDE BAKALIM:

```
A x = new A(9); HATA
// D<A> nesne1 = new D<A>(); Hata verir new()
parametresiz yapıcı olacak
D<int> nesne1 = new D<int>();
nesne1.Bilgi=6;
```

Baybca

## HATA ALIRINIR ÇÜNKÜ KISITA UYULMADI

```
public interface IPerson
{
 string Ad { get; set; }
}
class H : IPerson
{
 string aa;
 public string Ad
 {
 get { return aa; }
 set { aa = value; }
 }
}
class E<T> where T : IPerson
{
 public T deger;
}
class AnaProgram
{
 static void Main()
 {
 H h = new H();
 E<IPerson> nesne2 = new E<IPerson>();
 //veya E<H> nesne2 = new E<H>();
 nesne2.deger = h; nesne2.deger.Ad = "Ahmet";
 Console.WriteLine(nesne2.deger.Ad);
 }
}
```

T şablonuna atanılacak tipin IPerson interfacesinden türetilmesi gereklidir veya IPerson'da olabilir

ikiside uygundur

GORULDUGU GIBI  
nesne2.değer nesnesi  
şablon tipinden olduğu için  
ve şablonla H türü atadığı  
için artık nesne2.değer  
H tipinde bir nesne olmuştur  
halile nesne2.değer'e  
H tipindeki bir değişkene  
yapılabilen tüm işlemler uygunlaşanabilir

Mesela şablonumuza atanılacak tipin karşılaştırma işlemini kabul eden yani yani CompareTo(..) metodunu içeren bir tip olmasını istiyorsak

şablonumuzu IComparable interfacesiyle kısıtlayabiliriz. Böylece IComparable'den türeyen tüm nesnelerin tip olarak atanması zorunlu hale gelir.

Not:IComparable'de CompareTo() metodu bulunur halile ondan türeyen tüm nesnelerde CompareTo() metodu bulunur. Bu da demektir ki o nesne karşılaştırmayı kabul ediyor. Aslında burada interface'lerin varoluş nedenlerinden birini daha görmüş oluyoruz.

```
class A <T> where T : IComparable<T>
{
 public static int Karsilastir(T a, T b) { return a.CompareTo(b); }
}
class Program
{
 static void Main()
 {
 int s1 = A<int>.Karsilastir(4, 5);
 int s2 = A<float>.Karsilastir(2.3f, 2.3f);
 int s3 = A<string>.Karsilastir("Ali", "Veli");
 int s4 = A<DateTime>.Karsilastir(DateTime.Now, DateTime.Now.AddDays(1));
 Console.WriteLine("{0} {1} {2} {3}", s1, s2, s3, s4);
 }
}
```

Bu program ekrana -1 0 -1 -1 çıktısını verecektir. .Net Framework kütüphanelerindeki **IComparable** arayüzünde **CompareTo()** metodu bulunmaktadır. Dolayısıyla bu arayüzü uygulayan her sınıfta da bu metod bulunur. Bu sayede de C# şablon tip nesnesinden ilgili metodun çağrılmasına izin vermiştir.

Baybarsha'

Örnek:

```
using System;
class karsilastirma {
 static void Main() {
 Console.WriteLine(EnBuyuk<int>(4,5));
 Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
 }
 static T EnBuyuk<T>(T p1,T p2) where T:IComparable
 { T geridonus=p2;
 if(p2.CompareTo(p1)<0) geridonus=p1;
 return geridonus; } }
```

Bu program alt alta 5 ve Veli yazacaktır.

```
static void Main()
{ Console.WriteLine(EnBuyuk<int>(4,5));
Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
}
```

Burada parametrelerin türleri belli olduğu hâlde ayrıca int ve string türlerini de belirttik. İstersek bunu şöyle de yazabilirdik:

```
static void Main()
{ Console.WriteLine(EnBuyuk(4,5));
Console.WriteLine(EnBuyuk("Ali","Veli")); }
```

```
using System;
class karsilastirma {
static void Main() {
Console.WriteLine(EnBuyuk<int>(4,5));
Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
}
static T EnBuyuk<T>(T p1,T p2) where T:IComparable
{ T geridonus=p2;
if(p2.CompareTo(p1)<0) geridonus=p1;
return geridonus; } }
```



Bu program alt alta 5 ve Veli yazacaktır.  
Az önceki örneğin sadece Main() metodunu alalım:

```
static void Main()
{ Console.WriteLine(EnBuyuk<int>(4,5));
Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
}
```

Burada parametrelerin türleri belli olduğu hâlde ayrıca int ve string türlerini de belirttik. İstersek bunu şöyle de yazabilirdik:

```
static void Main()
{ Console.WriteLine(EnBuyuk(4,5));
Console.WriteLine(EnBuyuk("Ali","Veli")); }
```

Bu programda metot şöyle düşünecektir: "Benim parametrelerim T türünden, o hâlde yalnızca parametreme bakarak T'nin hangi tür olduğunu bulabilirim."  
Gördüğünüz gibi metotların aşırı yükleyerek saatlerce uğraşarak yazabileceğimiz programları şablon tipli metotlar kullanarak birkaç dakikada yazabiliyoruz.

Temsilciler de şablon tip alabilirler. Bu sayede temsilcinin temsil edebileceği metot miktarını artırabiliriz. Örnek:

```
using System;
delegate T Temsilci<T>(T s1,T s2);
class deneme {
 static int Metot1(int a,int b) {return 0;}
 static string Metot2(string a,string b){return null;}
 static void Main() {
 Temsilci<int> nesne1=new Temsilci<int>(Metot1);
 Temsilci<string> nesne2=new Temsilci<string>(Metot2);
 Console.WriteLine(nesne1(1,2)); Console.WriteLine(nesne2("w","q"));
 }
}
```

Temsilci şablon tipleri de kısıt alabilirler. Örnek:

```
delegate T Temsilci<T>(T s1,T s2) where T:struct
```

Burada T yalnızca bir yapı olabilir. Yani bu temsilcinin temsil edeceği metodun parametreleri ve geri dönüş tipi yalnızca bir yapı olabilir.

Bildığınız gibi yapı nesneleri null değer alamaz. Örneğin şu kod hatalıdır:

```
int a=null;
```

Ancak System isim alanındaki `Nullable<T>` yapısı sayesinde yapı nesnelerinin de null değer alabilmesini sağlayabiliriz. `System.Nullable<T>` yapısı şu gibidir:

```
public struct Nullable<T> where T:struct
{
 private T value;
 private bool hasValue;
 public T Value{get {...}}
 public bool HasValue{get {...}}
 public T GetValueOrDefault(){...}
}
```

Bu yapıya göre null değer alabilen yapı nesneleri şöyle oluşturulur:

```
Nullable<int> a=new Nullable<int>();
a=5; a=null;
Nullable<double> b=new Nullable<double>(2.3);
Console.WriteLine("{0}{1}",a, b);
```

? takısı kısa yoldan nullable tipte nesne oluşturmak için kullanılır. Örneğin:

Nullable<double> d=10; ile double? d=5;

satırları birbirine denktir.

Nullable nesneleri normal nesnelere tür dönüştürme operatörünü kullanarak dönüştürebiliriz. Ancak nullable nesnenin değeri null ise çalışma zamanı hatası alırız. Örnek: int? a=5; int b=(int)a;

Benzer şekilde tür dönüşüm kurallarına uymak şartıyla farklı dönüşüm kombinasyonları da mümkündür:

int? a=5; double b=(double)a;

Normal ve nullable nesneler arasında ters dönüşüm de mümkün değildir. Normal nesneler nullable nesnelere bilinçsiz olarak dönüştürülebilir. Örnek:

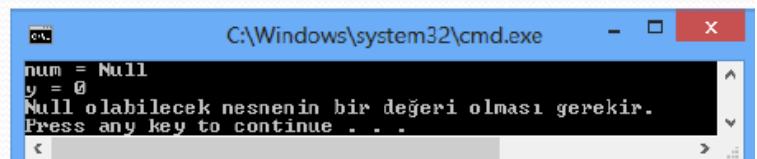
int a=5; int? b=a;

Nullable nesneler operatörler ile kullanılabilir. Örnek:

int? a=5; int? b=10; int c=(int)(a+b);

Burada a+b ifadesinin ürettiği değer yine int, ? türünden olduğu için tür dönüşüm operatörü kullanıldı.

```
class NullableExample
{
 static void Main()
 {
 int? num = null;
 if (num.HasValue == true)
 { Console.WriteLine("num = " + num.Value);}
 else
 { Console.WriteLine("num = Null");}
 //y değerine 0 atanıyor
 int y = num.GetValueOrDefault();
 // num.Value throws an InvalidOperationException if num.HasValue is false
 Console.WriteLine("y = " + y);
 try
 { y = num.Value; }
 catch (InvalidOperationException e)
 { Console.WriteLine(e.Message); }
 }
}
```



`??` operatörü `Nullable<T>` yapısındaki `GetValueOrDefault()` metoduna benzer şekilde çalışır. Örnek:

```
int? a=null; int b=a??10;
```

Burada eğer `a` null ise `??` operatörü 10 değerini döndürür.

`??` operatörünün döndürdüğü değer normal (nullable olmayan) tiptedir.

Eğer ilgili nullable nesne null değilse olduğu değeri döndürür. Başka bir örnek:

```
int? a=5; int b=a??50;
```

Burada ise 5 döndürülür.

`??` operatöründe `GetValueOrDefault()` metodundan farklı olarak ilgili nesne null olduğunda döndürülecek değeri belirleyebiliyoruz.

Baybars

Baybarshan Ekiز