

C# Dersleri

İlker Güzelcik

Temel .Net C# Eğitimi

- .Net C#'a Giriş
- İlk C # Program Örnekleri
- C # 'da Veri Türleri
- Operatörler ve İfadeler
- Kontrol Yapıları , Döngüler
- Nesne yönelimli programlama
- Sınıflar
- Operatörler
- Karakterler ve Dizeler
- Inheritance
- Sanal Yöntemler ve Polimorfizm 1
- Biçimlendirme ve Dönüştürme
- İstisnalar
- Arayüzler
- .NET Arayüzleri ve Koleksiyon
- Temsilciler ve Etkinlikler

.Net C# a Giriş: C# Tarihi

- C# zengin bir programlama mirasına sahiptir. C# direkt olarak dünyanın en başarılı programlama dillerinin ikisinden türetilmiştir: C ve C++ . Ayrıca bir başkasıyla da yakından ilişkilidir, Java. Bu ilişkilerin doğasını anlamak, C#'ı anlamak için kritik öneme sahiptir.
- C, 1960'ların *yapısal programlama (structured programming)* devrimi ile ortaya çıktı.
- C, Dennis Ritchie tarafından 1970'te, UNIX işletim sistemi kullanan DEC PDP-11 üzerinde icat edildi.
- C, 1980'lerin en yaygın olarak kullanılan yapısal programlama dili halini aldı.

.Net C# a Giriş: C# Tarihi

En zahmetli yönlerinden biri büyük programları ele almadaki yetersizliği idi. Proje belirli bir boyuta ulaştığı zaman C dili adeta bir engele çarpar ve bu noktadan sonra C programları anlaşılması ve sürdürülmesi zor bir hal alır. Bu limite tam olarak ne zaman ulaşılacağı programa, programcıya ve eldeki araçlara bağlı olmakla birlikte 5000 satır gibi kısa bir kodda bile bu durumla karşılaşılabilirdi.

.Net C# a Giriş: C# Tarihi

C++, Bjarne Stroustrup tarafından Murray Hill, New Jersey'deki Bell Laboratuvarlarında 1979'da icat edilmeye başlandı. Stroustrup, bu yeni dili ilk önce "C with Classes" ("Sınıflı C") olarak adlandırdı. Ancak, 1983'te dilin ismi C++ olarak değiştirildi. C++ , C dilinin bütününe içermektedir. Bu nedenle C, C++'ın üzerine inşa edildiği bir temeldir. Stroustrup'un C üzerine yaptığı değişikliklerin pek çoğu, nesne yönelimli programlamayı desteklemek amacıyla tasarlanmıştır. Yeni dili C temeli üzerine inşa ederek Stroustrup, nesne yönelimli programlamaya yumuşak bir geçiş sağlamıştır.

.Net C# a Giriş: C# Nedir?

- C# programlama dili Microsoft'un son zamanlarda geliştirdiği .NET platformunun bir ögesidir.
- C/C++ ve Java dillerinden türetilmiş, bu dillerin dezavantajlarının elenip iyi yönlerinin alındığı .Net platformu için sıfırdan geliştirilmiş nesne yönelimli bir dildir. Java dilinden farklı olarak C# dilinde işaretçiler (pointer) kullanılabilmektedir.
- .NET uyumlu dillerin hepsi aynı değişkenleri ve benzer nesne yönelimli özellikleri taşır.

.Net C# a Giriş: CLR (Common Language Runtime) - Ortak Dil Çalışma Platformu

- .NET altyapısında programların çalışmasını kontrol eden ve işletim sistemi ile programımız arasında yer alan arabirimdir.
- Platformdan bağımsız bir ortam istiyorsak, ihtiyaç duyulan şey CLR dir hangi platformda iseniz (Linux,Mac, Windows) CLR bu noktada devreye girer ve
- .NET programlarının farklı platformlarda işletim sistemine göre çalıştırır.

CTS (Common Type System) - Ortak Tip Sistemi

- *Bütün veri tiplerinin tanımlı olduğu bir sistem olarak düşünebiliriz C# dilindeki veri türleri aslında CTS'deki veri türlerine karşılık gelen arayüzlerdir.*
- ***CTS sayesinde .NET platformu için geliştirilen bütün diller aynı veri tiplerini kullanırlar, tek değişen türlerin tanımlama yöntemi ve sözdizimidir.***

İlk C # Program Örnekleri: Hello World

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace HelloWorld
8  {
9      internal class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.WriteLine("Hello World");
14             Console.ReadLine();
15         }
16     }
17 }
```

Namespaces and .NET Class Library

(İsim Alanları ve .NET Sınıf Kütüphanesi)

*C# dilinde .NET Framework sınıf kütüphanesi içerisindeki **veri türleri** ve **sınıflar** “using” anahtar sözcüğü ile kullanılır.*

.NET sınıf kütüphanesinde bulunan ve en sık kullanılan sınıf kütüphaneleri şunlardır :

- **System** : temel sınıfları içerir. Tüm sınıf kütüphaneleri bu isim alanı içinde kümelenmiştir.
- **System.Data** : veritabanı işlemlerinin tamamı için hazır gelen sınıf kütüphanesine bu isim alanı ile erişilir.
- **System.Xml** : veri biçimlendirme ve internetten veri paylaşımı için en çok kullanılan teknolojilerden biri olan XML ile çalışmak için gerekli sınırları içerir.
- **System.Net** : http ve ağ protokollerini içeren isim alanıdır.
- **System.IO** : dosyaları çalıştırmak (okuma - yazma) için kullanılır.
- **System.Windows.Forms** : Windos uygulamalarda kullanılan görsel kontrolleri barındıran isim alanıdır.

Namespaces ve Main()

C# dilinde her şey sınıflarla temsil edildiği için “Main()” işlevi de bizim belirlediğimiz bir sınıfın işlevi olmak zorundadır.

- Main() işlevi bizim için programımızın başlangıç noktasıdır.
- Sınıflar isim alanı (namespace) dediğimiz kavramla erişilmesi kolay bir hale gelmiştir.

Temel Veri Türleri

C#'da veri tipleri temel olarak 2 'ye ayrılırlar. Bunlar önceden tanımlanmış veri türleri ve kullanıcı tarafından tanımlanmış veri türleridir.

Önceden tanımlanmış olan veri türleri de kendi arasında değer tipi (value type) ve referans tipi (reference type) olarak 2'ye ayrılır.

Temel Veri Türleri

Verinin bellekte tutulması 6 bölgeden biri ile olmaktadır. Bunlar :

Stack Bölgesi : bir tamsayı türünden nesnenin çalışma zamanında yüklendiği yer RAM' in stack bölgesidir. Tanımlı değişkenlerin tutulduğu bellek alanıdır.

Heap Bölgesi : bütün C# nesneleri bu bölgede oluşturulur. Stack'ten farklı olarak bu bölgede tahsisatı yapılacak nesnenin derleyici tarafından bilinmesi zorunlu değildir. Bu bölgede bir nesneye alan ayırmak için **new anahtar sözcüğü** kullanılır.

new ile tahsis edilen alanlar dinamiktir. Çalışma zamanında tahsisat yapılır, derleme zamanında bir yer ayrılmaz Stack 'e göre daha yavaştır.

Temel Veri Türleri

- **Register Bölgesi** : Registerlar mikroişlemci üzerinde bulunan özel yapılardır. Diğer bölgelere göre **veri transferi daha hızlıdır**.
Static Bölge : Bellekteki herhangi bir bölgeyi temsil eder. Static alanlarda tutulan **veriler programın bütün çalışma süresince saklanır**.
Sabit Bölge : Program içerisinde, **değerlerin değişmeden sürekli olarak aynı kaldığı bölümdür**.
RAM Olayan Bölge : Bellek bölgesini temsil etmeyen **disk alanlarını temsil eder**.
- C#'da bir değişkene herhangi bir değer atamadan onu kullanmak yasaktır. Eğer **bir değişkeni kullanmak istiyorsak değişkenlere bir değer verilmesi zorunludur**. Bu kural değer ve referans tipleri için de geçerlidir.

Temel Veri Türleri

Değer veri türleri “**Stack**”, **Referans** veri türleri “**Heap**” te tutulurlar.

*C#’da bir değişkene herhangi bir değer atamadan onu kullanmak yasaktır. Eğer **bir değişkeni kullanmak istiyorsak değişkenlere bir değer verilmesi zorunludur.** Bu kural değer ve referans tipleri için de geçerlidir.*

Değer ve Referans Tipleri

Değer tipleri değişkenin değerini direkt bellek bölgesinden alırlar. Referans tipleri ise başka bir nesneye referans olarak kullanılırlar. Diğer bir ifade ile referans tipleri, heap alanında yaratılan nesnelerin adreslerini saklarlar.

Değer tipleri yaratıldıklarında stack bölgesinde oluşturulurlar. Referans tipleri ise kullanımı biraz daha sınırlı olan heap bellek bölgesinde saklanırlar.

Temel veri tipleri (int, double, float, ...) değer tipi, herhangi bir sınıf türü ise referans tipidir.

İki değer tipi nesnesi birbirine eşitlenirken değişkenlerde saklanan değerler kopyalanarak eşitlenir ve bu durumda iki yeni bağımsız nesne elde edilmiş olur. Birinin değerini değiştirmek diğerini etkilemez.

İki referans tipi birbirine eşitlendiğinde bu nesnelerde tutulan veriler kopyalanmaz, işlem yapılan nesnelerin heap bölgesindeki adresleridir.

Değer Tipleri (Value Types)

Değer tiplerinin tamamı Object denilen bir nesneden türemiştir. C#'da her nesne ya da veri tipi aslında Object tipidir.

C# da herşey bir nesnedir.

Referans Tipleri (Reference Types)

C#'ta önceden tanımlı iki referans tipi vardır **Object ve String**.

Object türü C#'ta bütün türlerin türediği sınıftır. Diğer bir deyişle Object türünden bir nesneye herhangi bir veri türünden nesneyi atayabiliriz.

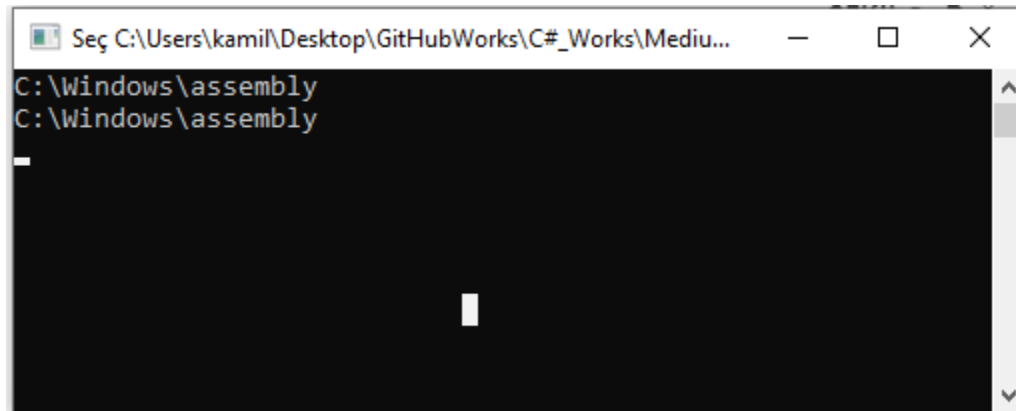
Object'e eşleştirme (Boxing) işlemi ve tersi, Object'i dönüştürme (Unboxing).

Değer ve Referans Tipi Ayrımı

Type	Default Value	Reference/Value
All numbers	0	Value Type
Boolean	False	Value Type
String	null	Reference Type
Char	'\0'	Value Type
Struct		Value Type
Enum	E(0)	Value Type
Nullable	null	Value Type
Class	null	Reference Type
Interface		Reference Type
Array		Reference Type
Delegate		Reference Type

String Türü

- Özel anlamlar içeren karakterleri ifade etmek için \ ifadesini kullanırız.
- String içinde görülen ifadenin aynısını belirtmek için string ifadesinin önüne @ işareti konulur.



```
Seç C:\Users\kamil\Desktop\GitHubWorks\C#_Works\Mediu...
C:\Windows\assembly
C:\Windows\assembly
-
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace MediumCSharpLearning
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             String path = "C:\\Windows\\assembly";
14             Console.WriteLine(path);
15
16             String path2 = @"C:\Windows\assembly";
17             Console.WriteLine(path2);
18
19             Console.ReadLine();
20         }
21     }
22 }
```

Veri Tipleri

- **byte** : Uzunluğu 1 byte'tır, 0 ile 255 arasında değer alır.
- **sbyte** : Uzunluğu 1 byte'tır, -128 ile 127 arasında değer alır.
- **short** : Uzunluğu 2 byte'tır, -32768 ile 32767 arasında değer alır.
- **ushort** : Uzunluğu 2 byte'tır, 0 ile 65535 arasında değer alır.
- **int** : Uzunluğu 4 byte'tır, -2.147.483.648 ile 2.147.483.648 arasında değer alır.
- **uint** : Uzunluğu 4 byte'tır, 0 ile 4.294.967.295 arasında değer alır.
- **long** : Uzunluğu 8 byte'tır, -10^{20} ile 10^{20} arasında değer alır.
- **ulong** : Uzunluğu 8 byte'tır, 0 ile 2×10^{20} arasında değer alır.
- **float** : Uzunluğu 4 byte'tır, 1.5×10^{-45} ile 3.4×10^{38} arasında değer alır.
- **double** : Uzunluğu 8 byte'tır, 5.0×10^{-324} ile 1.7×10^{308} arasında değer alır.
- **decimal** : Uzunluğu 12 byte'tır, $\pm 1.0 \times 10^{-28}$ ile $\pm 7.9 \times 10^{28}$ arasında değer alır.
- **char** : Uzunluğu 2 byte'tır, Bütün unicode karakterleri kapsar.
- **string** : Tek bir karakter, sözcük veya cümle gibi değerlerin saklanmasında kullanılır.
- **bool** : True – false değer tutan tiptir.

Data Types	
bool	Boolean value
byte	8-bit unsigned integer
char	16-bit Unicode character
decimal	128-bit precise decimal values with 28-29 significant digits
double	64-bit double-precision floating point
float	32-bit single-precision floating point
int	32-bit signed integer
long	64-bit signed integer
object	Base type for all other types
sbyte	8-bit signed integer
short	16-bit signed integer
string	String value
uint	32-bit unsigned integer
ulong	64-bit unsigned integer
ushort	16-bit unsigned integer

Uygulama: İki sayının Toplamı

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Uygulama2_Topla
8  {
9      internal class Program
10     {
11         static void Main(string[] args)
12         {
13             int a, b, c;
14             Console.Write("1. Sayıyı Giriniz:");
15             b = Convert.ToInt32(Console.ReadLine());
16             Console.Write("2. Sayıyı Giriniz:");
17             c = Convert.ToInt32(Console.ReadLine());
18             a = b + c;
19             Console.WriteLine("Sonuç:{0}", a.ToString());
20             Console.ReadKey();
21         }
22     }
23 }
```

Atama Operatörleri

Assignment Operators	
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
&=	AND assignment
=	OR assignment
^	XOR assignment
<<=	Left-shift assignment
>>=	Right-shift assignment

Mantıksal Operatörler

Logical and Bitwise Operators

&&	Logical AND
----	-------------

	Logical OR
--	------------

!	Logical NOT
---	-------------

&	Binary AND
---	------------

	Binary OR
--	-----------

^	Binary XOR
---	------------

~	Binary Ones Complement
---	------------------------

<<	Binary Left Shift
----	-------------------

>>	Binary Right Shift
----	--------------------

Karşılaştırma Operatörleri

Comparison Operators	
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Aritmetik Operatörler

Arithmetic Operators

+	Add numbers
-	Subtract numbers
*	Multiply numbers
/	Divide numbers
%	Compute remainder of division of numbers
++	Increases integer value by 1
--	Decreases integer value by 1

Diğer Operatörler

Other Operators

<code>sizeof()</code>	Returns the size of a data type
-----------------------	---------------------------------

<code>typeof()</code>	Returns the type of a class
-----------------------	-----------------------------

<code>&</code>	Returns the address of a variable
--------------------	-----------------------------------

<code>*</code>	Pointer to a variable
----------------	-----------------------

<code>? :</code>	Conditional expression
------------------	------------------------

<code>is</code>	Determines whether an object is of a specific type
-----------------	--

<code>as</code>	Cast without raising an exception if the cast fails
-----------------	---

Kontrol Yapıları

- C#'ta uygulama akışlarını kontroletmek için, genel olarak tercih edilen ikiadet deyim vardır. Bunlardan biri **if**, diğeri ise **switch** ifadesidir.
- **if**, uygulamada bir koşulun kontrolünü sağlayan deyimdir. Genel kullanımı aşağıdaki gibidir:

```
if (koşul)
{
    Koşul doğru ise çalıştırılacak komutlar;
}
else
{
    Koşul yanlış ise çalıştırılacak komutlar;
}
```

Kontrol Yapıları Örnek1

```
string parola = Console.ReadLine();  
if (parola == "1234")  
{  
    Console.WriteLine("Parola doğru");  
}  
else  
{  
    Console.WriteLine("Girilen parola doğru değil");  
}
```

Çok koşullu karar yapısı

Çok Koşullu if Yapısı

- if deyimi birden fazla durumlu koşulların sınanması amacıyla da kullanılabilir. Bu amaçla **else if** (koşul) yapısı kullanılır. Bu duruma ait kullanılacak yapı ise aşağıdaki gibidir:

```
if (koşul1)
{
    Koşul1 doğru ise çalıştırılacak komutlar;
}
else if (koşul2)
{
    Koşul1 yanlış ve Koşul2 doğru ise çalıştırılacak komutlar;
}
else
{
    Koşul1 ve Koşul2 yanlış ise çalıştırılacak komutlar;
}
```

Kontrol Yapıları

İç İçe if Yapısı

- if deyimi iç içe olarak da kullanılabilir. Bu yapıda parantezlerin doğru bir şekilde kullanılmasına dikkat edilmelidir. Bu duruma ait kullanılacak yapı aşağıdaki gibidir:

```
if (koşul1)
{
    Koşul1 doğru ise çalıştırılacak komutlar;
    if (koşul2)
    {
        Koşul1 ve Koşul2 doğru ise çalıştırılacak komutlar;
    }
    else if (koşul3)
    {
        Koşul1 ve Koşul3 doğru, Koşul2 yanlış ise çalıştırılacak komutlar;
    }
}
else if (koşul4)
{
    Koşul1 yanlış ve Koşul4 doğru ise çalıştırılacak komutlar;
}
else
{
    Koşul1 ve Koşul4 yanlış ise çalıştırılacak komutlar;
}
```

Kontrol Yapıları

```
string kullanıcıAdi = Console.ReadLine();  
string parola = Console.ReadLine();  
  
if (kullanıcıAdi == "admin" && parola == "1234")  
{  
    Console.WriteLine("Kullanıcı Adı ve Parola Doğru");  
}  
else  
{  
    Console.WriteLine("Kullanıcı Adı ve Parola Yanlış!");  
}
```


Kontrol Yapıları

```
string kullanıcıAdi = Console.ReadLine();
string parola = Console.ReadLine();

if (kullanıcıAdi == "admin")
{
    if (parola == "123456")
    {
        Console.WriteLine("Kullanıcı Adı ve Parola Doğru");
    }
    else
    {
        Console.WriteLine("Girilen Parola Doğru Değil!");
    }
}
else
{
    Console.WriteLine("Girilen Kullanıcı Adı Doğru Değil!");
}
```

Switch deyimi

```
switch (değişken)
{
    case 1:
        //değişken değeri 1'e eşitse yapılacak işler
        break;
    case 2:
        // değişken değeri 2'ye eşitse yapılacak işler
        break;
    case 3:
        // değişken değeri 3'e eşitse yapılacak işler
        break;
    default:
        // değişken değeri yukarıdakilerin hiç birine eşit değilse yapılacak işler;
        break;
}
```

Döngüler

- Döngüler, programlama dillerinde en çok ihtiyaç duyulan ifadelerin arasında yer alır. Program akışında tekrar tekrar gerçekleştirilmesi gereken iş süreçleri varsa, bu iş süreçleri döngüler yardımıyla gerçekleştirilir. C# dilinde 4 çeşit döngü vardır:
 - **for** döngüsü
 - **while** döngüsü
 - **do while** döngüsü
 - **foreach** döngüsü

BREAK komutu içinde bulunduğu döngüyü kırar, program kırılan döngüden sonra kaldığı yerden çalışmaya devam eder.

CONTINUE komutu BREAK komutuna benzer. Ancak break komutundan farklı olarak program CONTINUE komutunu gördüğünde döngüden çıkmaz, sadece döngünün o anki iterasyonu sonlanır ve döngü bir sonraki iterasyonu yapmak üzere tekrar başlatılır.

For Döngüsü

```
for (int i = 0; i < 10; i++)  
{  
    //döngü çalıştığı sürece çalıştırılacak komutlar  
    Console.WriteLine(i);  
}
```

```
int toplam = 0;  
for (int i = 0; i < 100; i++)  
{  
    toplam += i;  
}  
  
Console.WriteLine(toplam);
```

While döngüsü

C#'ta yaygın olarak kullanılan döngülerden biri de **while** döngüsüdür. *for* döngüsünde olduğu gibi, bir koşul sağlandığı sürece dönmeye devam eder. Koşul yanlış (**false**) sonucunu verdiği zaman ise sonlandırılır. Genel yazım şekli şöyledir:

```
while (koşul)
{
    koşul sağlandığı sürece çalıştırılacak komutlar
}
```

for döngüsünde yaptığımız toplama örneğini, bir de **while** döngüsüyle yapalım:

```
int toplam = 0;
int i = 0;
while (i < 100)
{
    toplam += i;
    i++;
}
Console.WriteLine(toplam);
```

Çıktı: 4950

While döngüsü örnek

```
int sayi = Convert.ToInt32(Console.ReadLine());
int basamak = 0;
while (sayi > 0)
{
    basamak++;
    sayi = sayi / 10;
}
Console.WriteLine("Girdiğiniz sayı " + basamak.ToString() + "basamaklıdır.");
```

Do while döngüsü

- *for* ve *while* döngülerinde koşul, döngü başlamadan önce kontrol edilir.
- **Do while** döngüsünde ise, bu kontrol her döngüden sonra gerçekleştirilir. Operasyon mantığında **do while** döngüsü, koşul ne olursa olsun en az bir kere çalıştırılır.
- Bu döngünün genel yazım şekli aşağıdaki gibidir:

```
do
{
    //döngü çalıştığı sürece çalıştırılacak komutlar
} while (koşul)
```

Do While Örnek

```
int toplam = 0;
int sayac = 0;
do
{
    sayac++;
    toplam += sayac;
}
while (sayac < 100);
Console.WriteLine("Toplam: " + toplam.ToString()
    + ", Sayaç: " + sayac.ToString());
```

Çıktı: 5050

For each döngüsü

for döngüsü gibi yaygın kullanılan bir diğer döngü de **foreach** döngüsüdür. **foreach**, dizi (**array**) ve koleksiyon (**collection**) tabanlı nesnelerin elemanları üzerinden ilerleyen, iterasyon gerçekleştirerek bu elemanlara erişip iş katmanınızı oluşturabileceğiniz bir döngüdür.

Bu döngünün genel kullanım şekli aşağıdaki gibidir:

```
foreach (tip değişken in koleksiyon)
{
    //döngü çalıştığı sürece çalıştırılacak komutlar
}
```

For each döngüsü örnek

```
int[] sayilar = { 1, 2, 3, 4, 5, 6 };  
int carpim = 1;  
foreach (int x in sayilar){  
    carpim = carpim*x;  
}
```

Diziler (Array)

Değişkenler, içlerinde tekil veriler tutan yapılardır. Ancak çok sayıda veri üzerinde çalışmamız gereken durumlarda aynı tipteki değişkenleri bir arada tutmamız gerekebilir. Bu noktada, yazılım dillerinin genelinde kullanılan **dizi (array)** devreye girer.

Bir dizi tanımlamanın genel yazım şekli şöyledir:

```
tip dizi1 = new tip[elemansayisi];
```

tip → Dizide tutulacak verinin tipi
dizi1 → Diziye verilen isim

elemansayisi → Dizide tutulacak eleman sayısı

Diziler (Array)

Örneğin; aşağıda 5 elemanlı bir tamsayılar dizisi tanımlanmıştır:

```
int[] sayilar = new int[5];
```

Dizilere veri atamak için, aşağıdaki söz dizimi kullanılır:

```
sayilar[indeks] = 1;
```

indeks: Dizideki verilere verilen sıra numarasıdır. [C#ta indeksler sıfırdan başlar.](#)

Dizilerden değer okumak içinse, aşağıdaki söz dizimi kullanılır:

```
int sayi = sayilar[indeks];
```

Diziler (Array)

Örneğin, 11 kişilik bir futbol takımındaki futbolcuların forma numaralarını ayrı ayrı değişkenlerde tutmak yerine, futbolTakimi adlı bir dizi içinde tutabiliriz.

```
short[] futbolTakimi = new short[11];
```

Bu ifadeyle, futbolTakimi isminde, içinde 10 tane short değeri tutabilecek bir dizi tanımlanır. Şimdi, forma numaralarını diziye aktaralım:

```
futbolTakimi[0] = 1;  
futbolTakimi[1] = 4;  
futbolTakimi[2] = 5;  
futbolTakimi[3] = 2;  
futbolTakimi[4] = 8;  
futbolTakimi[5] = 12;  
futbolTakimi[6] = 19;  
futbolTakimi[7] = 99;  
futbolTakimi[8] = 22;  
futbolTakimi[9] = 10;  
futbolTakimi[10] = 11;
```



Diziler (Array)

Bütün değerlere tek tek erişmek, eğer bu değerler birkaç tane ise pek sorun olmayacaktır. Fakat çok sayıda değer içeren bir dizideki değerlerin tümü listelenmek istenirse, bu yöntem pek uygun olmayacaktır. Bunun için **foreach** döngüsü kullanılabilir. Şimdi, foreach döngüsü ile tüm elemanları bir **Console.WriteLine()** ile ekrana yazdıralım:

```
foreach (short futbolcu in futbolTakimi)
{
    Console.WriteLine(futbolcu);
}
```

Sınıf (Class)

- **C# OOP** : Classlar bizim yapmak istediğimiz işlemleri gruplara ayırmak, o grup üzerinden işlemlerimizi yapmak ve rahatlıkla bu gruba ulaşmak için kullanabiliriz.
 - bir class'ı kullanabilmek için onun örneğini(**referansını**) oluşturmamız gerekmektedir.
 - bir class oluştururken kelimenin ilk harfi büyük oluşturulur. Ama örneği oluşturulduğunda ilk harfi küçük, sonraki kelimelerin ilk harfi büyük yazılır.
- Classlar bir gruplama tekniği olarak metodlar ile kullanılabilir.

Sınıf (Class)

- Sınıflar, Nesne Yönelimli Programlamanın(OOP) temel yapı taşlarıdır. Sınıflar birer veri yapısı olup programcıya bir veri modeli sunar. Bu veri modeli yardımıyla nesneler oluşturulur.
- Şimdi C# ile bir sınıf örneği yapalım. Personel ile ilgili bilgileri tutmasını istediğimiz bu sınıfın ismi personel.cs olsun.

Personel.cs

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

namespace _01_Siniflar
{
    public class Personel
    {
        public string Isim;
        public string Soyisim;
        public int Yas;
        public int Maas;
    }
}
```

Projemizin İsmi

Sınıfımızın İsmi

Sınıfı Çağırma Ve Bilgi Atama

Personel bilgilerini tutması için basit bir sınıfı oluşturduk. Şimdi ise oluşturduğumuz bu sınıfı main metodumuzda çağıralım ve içerisine bilgi atayalım.

Personel sınıfından yeni bir sınıf oluşturup içerisine verileri ekleyerek bunları ekrana yazdırdık. Fakat verileri yollarken herhangi bir koruma işlemi şu an için bulunmuyor. Örnek vermek gerekirse yaş kısmına 500 yazabiliyorum. Burada ise OOP'nin temel ilkelerinden biri olan encapsulation(kapsülleme) ortaya çıkıyor. Bir sonraki bölümde kapsüllemenin ne olduğundan bahsedelim.

```
class Program
{
    static void Main(string[] args)
    {
        Personel personel = new Personel();
        personel.Isim = "Berke";
        personel.Soyisim = "Kurnaz";
        personel.Yas = 20;
        personel.Maas = 3000;

        Console.WriteLine("Personel Ismi : " + personel.Isim);
        Console.WriteLine("Personel Soyismi : " + personel.Soyisim);
        Console.WriteLine("Personel Yasi : " + personel.Yas);
        Console.WriteLine("Personel Maasi : " + personel.Maas);

        Console.Read();
    }
}
```

Sınıf İçerisinde Metot Tanımlama

- Sınıflar içerisinde özellikler tanımladık. Gelin şimdide bir metot tanımlayalım. Bunun için ben yine personel sınıfını kullanıyor olacağım ve içerisine iki tane metot tanımlayacağım. Birince metot geriye herhangi bir değer döndürmeyen bilgileriYazdır metodu, ikincisi ise içerisine bir sayı alarak bu sayı miktarınca maaşa zaman yap geriye bir tam sayı döndüren maasaZamYap metodu.

```
public class Personel
{
    public string Isim;
    public string Soyisim;
    public int Maas;
    public int Yas;

    public void BilgileriYazdir()
    {
        Console.WriteLine(Isim);
        Console.WriteLine(Soyisim);
        Console.WriteLine(Maas);
        Console.WriteLine(Yas);
    }

    public int MaasaZamYap(int zamMiktari)
    {
        Maas = Maas + zamMiktari;
        return Maas;
    }
}
```

Sınıf İçerisinde Metot Tanımlama

- Metotlarımızı yazdık şimdi ise main metodu içerisinde bunları nasıl kullanacağız onu inceleyelim.
- Programı çalıştırdığımızda öncelikle ekrana bilgiler yazılacak. Ardından zam yapma metodu çalışıp düzenlenmiş maaş değeri ile tekrardan bilgiler ekrana yazılacak.

```
class Program
{
    static void Main(string[] args)
    {
        Personel personel = new Personel();
        personel.Isim = "Berke";
        personel.Soyisim = "Kurnaz";
        personel.Yas = 20;
        personel.Maas = 3000;

        personel.BilgileriYazdir();

        personel.MaasaZamYap(500);

        personel.BilgileriYazdir();

        Console.Read();
    }
}
```

Yapıcı Metodlar (Constructors)

- Bir nesne dinamik olarak oluşturulduğunda otomatik olarak çalışan metodlardır. Nesnenin elemanlarına ilk değeri vermek için ya da sınıf nesnesi için gerekli kaynak düzenlemeleri yapma da kullanılır. Personel sınıfı için bir yapıcı metod oluşturup burada ilk oluşturulma için değerler atayalım.

```
public class Personel
{
    public Personel() → Yapıcı Metod
    {
        Isim = "xxx";
        Soyisim = "xxx";
        Maas = 0;
        Yas = 0;
    }

    public string Isim;
    public string Soyisim;
    public int Maas;
    public int Yas;
```

Yapıcı Metotlarda Overloading (Aşırı Yüklenme)

Bu kısımda ise yapıcı metotlarda overloading(aşırı yüklenme) işlemi nasıl yapılıyor ona bakalım.

Örnekte yapıcı metotlardan ilki içerisine parametre almayacak ve ona göre bir atama yapılacaktır. İkincisi ise personel ismini, soyismini, yaşını ve maaşını parametre olarak alıp atamayı yapacaktır.

```
public class Personel
{
    public Personel()
    {
        Isim = "xxx";
        Soyisim = "xxx";
        Maas = 0;
        Yas = 0;
    }

    public Personel(string isim, string soyisim, int maas, int yas)
    {
        Isim = isim;
        Soyisim = soyisim;
        Maas = maas;
        Yas = yas;
    }
}
```

Yıkıcı Metotlar (Destructors)

- Yıkıcı metotlar, Garbage Collector bir nesne için bellekte ayrılan alanı kaynağa iade etmeden hemen önce çalışır. Bir sınıfın sadece bir tane yıkıcı metodu olabilir ve herhangi bir parametre almaz. Örnek olarak personel sınıfımız için bir yıkıcı metot oluşturalım.

```
~Personel()  
{  
    Console.WriteLine("Burası Yıkıcı Metot");  
}
```

Statik Metotlar

- Statik metotlar bir işlemin gerçekleştirilmesi için nesne oluşturulmasına gerek olmayan durumlarda kullanılır. Örneğin matematik adlı bir sınıfımız olsun ve içerisinde iki adet topla ve cikar isminde statik metot yazalım.

```
public class Matematik
{
    public static int Topla(params int[] dizi)
    {
        int toplam = 0;
        for (int i = 0; i < dizi.Length; i++)
        {
            toplam = toplam + dizi[i];
        }
        return toplam;
    }

    public static int Cikar(int s1, int s2)
    {
        return s1 - s2;
    }
}
```


Statik Metotlar

- Şimdi ise statik metotları main metodu içerisinde nasıl kullanıyoruz ona bakalım.

```
class Program
{
    static void Main(string[] args)
    {
        int toplam = Matematik.Topla(3, 5, 8);
        int fark = Matematik.Cikar(10,5);

        Console.WriteLine("Toplam : " + toplam);
        Console.WriteLine("Fark : " + fark);

        Console.Read();
    }
}
```

Ref ve Out

- C# programlama dilinde kullanılan veri tiplerini 2 başlıkta tanımlayabiliriz. Değer tipleri ve Referans tipleri.
- Kısaca tanımlamak gerekirse, değer tipler; ram de kaplayacağı alanı belli olan int,double,byte vs gibi struct olan tiplerdir ve stack'de bulunurlar. Referans tipler; string,object, kendi yazdığımız custom objeler bunlar ram'de kaplayacakları alan belli olmadığından heap'de tutulurlar.
- Gelelim konumuzun başlığında bulunan "ref" ve "out" parametrelerimize. C# da kullanıma sunulan ref ve out isminde iki tane parametre tanımlama çeşidi bulunmakta. Bu kullanımlarla birlikte ref ve out parametreleriyle yazmış olduğumuz metodlara değişkenleri göndermeden önceki değerleri ile metoddan çıktıktan sonraki değerleri kullanıma göre farklılıklar göstermektedir.

Ref ve Out

- Geliştirme yaparken tanımlamış olduğumuz parametre alan metodları kullanmak istediğimizde ilgili metoda parametre geçme işlemi 2 yolla yapılır "pass by value" ve "pass by reference". Kısaca tanımlamak gerekirse ;
- Pass by value fonksiyona parametre gönderirken o parametrenin bir kopyasının oluşturulup gönderilmesi ve metod içerisinde yapılan değişikliklerden metod içerisinde ki yeni değerin etkilenmesi,
- Pass by reference fonksiyona parametre gönderirken o parametrenin ram'de adresinin gönderilip metod içerisindeki yapılan değişikliklerden orjinal değerin etkilenmesi.

Ref ve Out farkları

- Metodu tanımlarken geçilmek istenen değişkenin önüne "ref" yazılmalıdır.
 - orjinalDeger değişkeni metoda parametre olarak geçilmeden önce bir başlangıç değeri almak zorundadır.
 - "i" değişkeni metod içerisinde herhangi bir değişiklik yapmadan da kullanılabilir.
- Metodu tanımlarken geçilmek istenen değişkenin önüne "out" yazılmalıdır.
 - orjinalDeger değişkeni metoda parametre olarak geçilmeden önce bir başlangıç değeri almak zorunda değildir.
 - "i" değişkeni metod içerisinde herhangi bir değişiklik yapmadan kullanılamaz.

Nesne Yönelimli Programlama

Nesne yönelimli programlamanın prensiplerini desteklemek amacıyla, C# da dâhil olmak üzere tüm nesne yönelimli diller üç ortak özelliğe sahiptir:

İlişkili verilerin paketlenmesi (encapsulation)

Çok biçimlilik (polymorphism)

Kalıtım (inheritance)

İlişkili Verilerin Paketlenmesi (*encapsulation*)

Verilerin paketlenmesi (encapsulation), kodun manipüle ettiği verilerle kodu birbirine bağlayan ve her ikisini dışarıdan gelebilecek istenmeyen etkilerden ve hatalı kullanımlardan koruyan bir programlama mekanizmasıdır.

Nesne içindeki kod, veri veya her ikisi de bu nesneye *özel (private)* veya *açık (public)* olabilirler. Özel kod veya veri, sadece bu nesnenin parçaları tarafından bilinir ve erişilebilir.

Encapsulation(Kapsülleme)

Sınıf içerisinde yer alan alanların sınıfın dışarından erişiminin kapatılarak kontrol altına alınmasına encapsulation(kapsülleme) diyoruz. Yukarıda verdiğimiz örneği yine ele alalım. Biz uygulamanın şimdiki haliyle yaş kısmına 500 veya -200 yazabiliriz. Fakat bu çok gerçekçi durmayacaktır. Yaş aralığını 0 ila 150 arasında kontrollü olmasını istersek aşağıdaki gibi bir kapsülleme işlemi yapmalıyız.

GET SET Kullanımı

C# da özellikleri, metotların ve sınıfların görünürlüklerini yönetmek için kullanırız. Kısaca örneklemek gerekirse bir sınıf içerisinde farklı bir sınıf içerisinde ki nesneye ulaşmak istiyorsak özellik metotlarını kullanmalıyız. Özellik metotları GET ve SET anahtar kelimesinden oluşan iki kod bloğundan oluşurlar. **GET metodu veri okunduğu zaman, SET metodu ise veri yazıldığı zaman (yani değer ataması yapıldığı zaman) yürütülür.** Özellik olarak bu iki anahtar kelimeyi aynı anda kullanabildiğimiz gibi, tek anahtar kelime ile de oluşturabiliriz. Örneğin sadece GET metodu ile oluşturduğumuz özellik sadece okunabilir, SET metodu ile oluşturduğumuz özellik ise sadece yazılabilir bir hal alır. Her iki anahtar kelimeyle oluşturduğumuz özellik ise hem okunabilir hem de yazılabilir özelliğe dönüşür.

Bir Class içerisinde bulunan bazı alanlara her zaman ulaşmak gerekmez. Çünkü bir nesneyi sürekli ulaşılabilir hale getirmek bilinçsiz kullanım, veri kaybı ve güvenliği gibi sorunları ortaya çıkartır. Zaten nesneleri tanımlarken varsayılan değer olarak “Private” erişim belirleyicisi olarak tanımlanması da bu tip gerekçelerden kaynaklanır. “Public” erişim belirleyicisi ise tamamen açık hale getirir. İşte tam bu noktada nesnelerimizin erişimini yönetmek için “Property” kavramı devreye girer. “Property” yani Özellik metotları nesnelerimiz üzerinde kontrollü kullanım sağlar.

Encapsulation(Kapsülleme)

Artık yaş değerini 0 ve 150 arasında kapsülleme sayesinde kontrol altına almış olduk. Bu gibi durumlar altında encapsulation(kapsülleme) kolaylık sağlamış olsa da temiz ve okunabilir bir kod açısından kendi fikrime göre eksikleri bulunuyor. Bu gibi validasyon işlemleri için farklı araçlar veya sınıflar kullanmamız kodumuzun okunabilirliği açısından daha iyi olacaktır. Fakat üst kısımda da belirttiğim gibi son birkaç cümle kendi görüşüme göre olan durumlar, öyle veya böyle yapmanız arasında sonuç olarak fark olmayacaktır.

```
public class Personel
{
    public string Isim;
    public string Soyisim;
    public int Maas;

    private int mYas;
    public int Yas
    {
        get
        {
            return mYas;
        }
        set
        {
            if(value < 0 || value > 150)
            {
                value = 0;
            }
            else
            {
                mYas = value;
            }
        }
    }
}
```

Erişim

- **Private:** Sadece tanımlandığı sınıf içerisinde erişilebilir. (*Kalıtım ile aktarılmaz.*)
- **Public:** Her yerden erişilebilir. (*Kalıtım ile aktarılır.*)
- **Internal:** Sadece bulunduğu projede erişilebilir. (*Kalıtım ile aktarılır.*)
- **Protected:** Tanımlandığı sınıfta ve o sınıfı miras (*kalıtım*) alan sınıflardan erişilebilir. (*Kalıtım ile aktarılır.*)

Kalıtım (inheritance)

Kalıtım (*inheritance*), bir nesnenin diğer bir nesnenin özelliklerini miras olarak almasını sağlayan bir yöntemdir. Kalıtım, hiyerarşik sınıflandırma kavramını destekler. Bilgiler hiyerarşik (yani, yukarıdan aşağıya doğru) sınıflandırma sayesinde yönetilebilir kılınır.

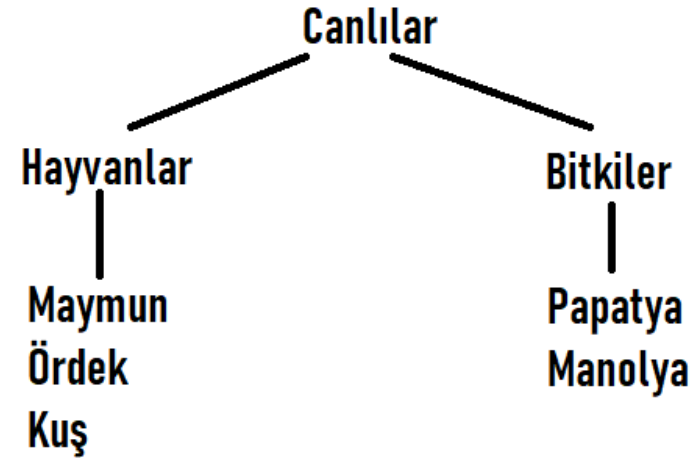
Örneğin, kırmızı tatlı elma, *elma* sınıflandırmasının bir parçasıdır. Elma, *meyve* sınıfının bir parçasıdır. Meyve ise daha büyük bir sınıf olan *yiyecek* sınıfının altında yer alır. Yani, *yiyecek* sınıfı, aynı zamanda mantıksal olarak kendisinin bir alt sınıfı olan meyve sınıfına da uygulanabilecek belirli özelliklere (yenilebilir, besleyici vs.) sahiptir. Bu özelliklere ek olarak, *meyve* sınıfının da kendisini diğer sınıflardan ayırt eden, kendisine özgü özellikleri (sulu, tatlı vs.) vardır. *Elma* sınıfı bu tür özelliklerden elmaya özgü olanları (ağaçta yetişir, tropik meyve değildir vs.) tanımlar. Kırmızı Tatlı elma bu durumda, kendisinden önce gelen tüm sınıfların tüm özelliklerini kalıtım yoluyla devralacaktır ve bu özelliklerden sadece kendisini eşsiz kılanları tanımlayacaktır.

Kalıtım (inheritance)

Kalıtım bir sınıfın bir üst sınıftan miras almasına denir. Adından da anlaşılacağı üzere üst sınıfın bütün özellikleri miras alan sınıfa aktarılır.

Günümüzde birçok programlama dili kafa karışıklığı olmaması için sadece tekli kalıtıma izin vermektedir. c#'da tekli kalıtıma izin veren dillerden birisidir.

Öncelikle canlılar adlı üst bir sınıf oluşturup altına bitkiler ve hayvanlar adlı iki tane daha sınıf oluşturalım. Daha sonra ise hayvanlardan miras alan maymun, ördek ve kus sınıfını; bitkilerden miras alan papatya ve manolya sınıfını oluşturalım. Öncelikle hiyerarşimizi ben burada paylaşıyorum.



Kalıtım (inheritance)

İlk adımda canlılar sınıfını oluşturalım. Canlılar sınıfında beslenme, hareket ve boşaltım gibi ortak özellikler bulunsun.

```
public class Canlilar
{
    public void Beslenme()
    {
        Console.WriteLine("Canli Beslendi");
    }
    public void Hareket()
    {
        Console.WriteLine("Canli Hareket Etti");
    }
    public void Bosaltim()
    {
        Console.WriteLine("Canli Bosaltim Yapti");
    }
}
```

Kalıtım (inharitance)

- Şimdi ise hayvanlar ve bitkiler sınıflarını oluşturup bu iki sınıfı da canlılar sınıfından miras aldıralım. Ayrıca hayvanlar sınıfına ek bir metot olarak koşma, bitkiler sınıfına ise güneşeYonelme isimli metotlar ekleyelim.

```
public class Hayvanlar : Canlilar
{
    public void Kosma()
    {
        Console.WriteLine("Hayvan Kostu");
    }
}
```

```
public class Bitkiler : Canlilar
{
    public void GuneseYonelme()
    {
        Console.WriteLine("Bitki Gunese Yoneldi");
    }
}
```

Kalıtım (inheritance)

Şimdi ise hayvanlar sınıfından miras alacak maymun, ördek ve kuş sınıflarını oluşturalım. Hepsinin ek olarak kendi isimleri ve aynı zamanda + Olmak olan bir metodu olacak.

```
public class Kus : Hayvanlar
{
    public void KusOlmak()
    {
        Console.WriteLine("Kus Olmak");
    }
}
```

```
public class Ordek : Hayvanlar
{
    public void OrdekOlmak()
    {
        Console.WriteLine("Ordek Olmak");
    }
}
```

```
public class Maymun : Hayvanlar
{
    public void MaymunOlmak()
    {
        Console.WriteLine("Maymun Olmak");
    }
}
```

Kalıtım (inheritance)

- Aynı işlemi papatya ve manolya sınıfı içinde yapalım. Fakat bu iki sınıf bitkiler sınıfından miras alsın.

```
public class Papatya : Bitkiler
{
    public void PapatyaOlmak()
    {
        Console.WriteLine("Papatya Olmak");
    }
}
```

```
public class Manolya : Bitkiler
{
    public void ManolyaOlmak()
    {
        Console.WriteLine("Manolya Olmak");
    }
}
```


Kalıtım (inheritance)

- İlgili sınıfların tümünü yazdık. Artık main metodu içerisinde bunları çağırmak kaldı. Örneğin maymun sınıfını çağırdığımız zaman canlılar sınıfındaki tüm metotlara ulaşabiliyoruz, aynı şekilde hayvanlar sınıfındaki tüm metotlarada. Papatya nesnesi oluşturduğumuz zaman ise yine canlılar ile ilgili olanların tümüne ve bitkiler ile ilgili olanlara ulaşabiliyoruz.
- Bu klasik örnek kalıtımın ne olduğunu ve nasıl kullanıldığını anlamanızda size büyük yardımcı olacaktır. Fakat vermiş olduğumuz örnek programlamadaki gerçek hayat problemleri ile ilgili değil, gerçek hayatta kalıtımı nasıl nerede ve niçin kullanıyoruz bunu başka bir yazıda proje üzerinde anlatmak istiyorum ama bu örnekte kalıtım ile ilgili net fikirlerinizin oluşması amacıyla bu örneği seçtim.

Sealed Class (Mühürlü Sınıf)

- Nesneye dayalı programlama yaklaşımında kalıtım (inheritence) özelliği sayesinde bir sınıftan başka sınıflar türetilebilir ve bu sınıflara yeni özellikler eklenerek daha zengin ve kullanışlı sınıflar yaratılabilir. Ancak bazı durumlarda **sınıflardan türetme yapılması istenmez**. Bu durumda sınıf tanımlamasının başına mühürlü (sealed) kelimesi getirilir. **Bu sayede sabit özellikler ve metotlara sahip bir sınıf elde edilir**. Mühürlü sınıf (sealed class) hiçbir sınıfın kendisinden türetilmeyeceğini ifade eder.

Sealed Class (Mühürlü Sınıf)

```
namespace Sealed
{
    public sealed class Ogrenci
    {
        public Ogrenci() { }
        public int id { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
    }
}
```

Eğer Ogrenci sınıfından AsistanOgrenci isminde yeni bir sınıf türetilmesi istenirse aşağıdaki şekilde tanımlanır:

```
public class AsistanOgrenci : Ogrenci
{ }
```

Fakat Ogrenci sınıfı mühürlü sınıf (sealed class) olduğu için yeni bir sınıf türetilemez ve şu şekilde bir hata mesajı alınır:
'Project.AsistanOgrenci': cannot derive from sealed type 'Project.Ogrenci'

Interface (arayüz)

- Interface (Arayüz) Sınıflara (Class) ne yapacaklarını söyleyen bu işlem yapılırken hangi metot veya değişkenleri kullanabilecekleri konusunda yol gösteren yapılardır. Interface içinde sadece tanımlama yapılır kod blokları yer almaz

Interface (Arayüz)

Interface'ler kendisini uygulayan sınıfların kesin olarak içereceği metotları, özellikleri bildirirler. Interface'leri genellikle yazacağımız kodlara rehber olması amacıyla hazırlarız. **Özellikle birden fazla programcı tarafından geliştirilen uygulamalarda ekibe büyük fayda sağlarlar.**

Şimdi gerçek hayatta kullanabileceğimiz bir interface örneği hazırlamaya çalışalım. Senaryomuz gereği veritabanına **ekleme, silme, güncelleme ve listeleme** işlemleri yapan bir sınıfımız için interface yazalım.

Interface'lerin yazılı olarak olmasa da isimlendirirken başlarına I harfi konur.

Örnek interface olarak **IDatabaseOperations** adında bir interface oluşturuyorum. Interface içerisinde gerekli işlemlerim için **4 farklı metot yazalım.**

```
public interface IDatabaseOperations
{
    void add();
    void delete(int id);
    void update(int id);
    void getById(int id);
}
```

Interface (Arayüz)

Şimdi ise DatabaseOperations adlı bir sınıf oluşturup yazmış olduğumuz interface'i orada kullanalım.

Interface'i sınıfa ekledikten sonra implement interface seçeneğine tıklayarak hızlıca interface içerisinde yer alan metotlarımızı sınıf içerisine getirebiliyoruz. Ardından ise yapmak istediklerimizi metotların içerisine yazabiliriz.

```
public class DatabaseOperations : IDatabaseOperations
{
    public void add()
    {
        throw new NotImplementedException();
    }

    public void delete(int id)
    {
        throw new NotImplementedException();
    }

    public void getById(int id)
    {
        throw new NotImplementedException();
    }

    public void update(int id)
    {
        throw new NotImplementedException();
    }
}
```

Interface

1. Interface içerisinde sadece method ve property tanımlaması yapabiliriz.
2. Interface içerisinde değişken tanımlaması **yapılmaz!!!**
3. Interface isimleri tavsiye edildiği şekilde, 'I' harfi ile başlar..

Çok biçimlilik (polymorphism)

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

Sanal (Virtual) Metotlar

Sanal metotlar ana sınıf içerisinde bildirilmiş ve miras alınan sınıfta tekrar bildirilebilen sınıflardır. Sanal metotlar kullanılarak nesne yönelimli programlamanın ilkesi olan çok biçimlilik (polymorphism) uygulanmış olur. Sanal metotları bildirmek için virtual anahtar sözcüğünü kullanırız, miras alan sınıfta ise override anahtar sözcüğünü kullanırız.

Canlı adında bir sınıf oluşturup buna Konuş adlı sanal bir sınıf ekliyorum. Daha sonra ise İnsan adında bir sınıf oluşturup Konuş metodunu override ediyorum. Override ettiğim bu metotta eğer `base.Konuş()` satırını eklersem ana sınıftaki Konuş metodu içerisinde yer alan kodlarda çalışacak. Eğer `base.Konuş()` satırını üste eklemesem üstteki metot tamamen geçersiz kılınıp sadece İnsan sınıfı içerisindeki Konuş metodunu yaptıklarım çalışacak.

Sanal (Virtual) Metotlar

```
public class Canli
{
    public virtual void Konus()
    {
        Console.WriteLine("Canli Olarak Konustum");
    }
}
```

```
public class Insan : Canli
{
    public override void Konus()
    {
        base.Konus();
        Console.WriteLine("Insan Olarak Konustum");
    }
}
```

Programın Çıktısı

```
Canli Olarak Konustum
Insan Olarak Konustum
```

Delegate (Temsilci)

Delege, bir olay (event) tetiklendiğinde hangi metodun çağrılacağını söylemenin bir yoludur.

delegeler metot tutuculardır diyebiliriz. Delegeler, metotların adreslerini dolayısıyla metotların kendilerini tutabilen yapılardır. Delegeler referans tipli yapıda oldukları için nesne alınabilir.

Delege kullanırken dikkat etmemiz gereken noktalar vardır. Bunlar;

- Geri dönüş tipi tuttuğu metodun geri dönüş tipiyle aynı olmalıdır.
- Tuttuğu metodun parametre tipleriyle aynı olmalıdır.
- Aynı sayıda parametre içermelidir.

Try catch Hata Yönetimi

```
try
{
    // Hatalı olabilecği düşünülen işlemler...
}
catch (Exception hata)
{
    // Hata alındığında yapılacak işlemler...
    throw;
}
finally
{
    // Hata olsa da almasa da yapılması gereken işlemler...
}
```

Sık Kullanılan İstisnai Durum Sınıfları

- `System.OutOfMemoryException`: Programın çalışması için yeterli bellek kalmadıysa oluşur.
- `System.StackOverflowException`: Stack (Yığın) bellek bölgesinin birden fazla metod için kullanılması durumunda oluşur. Genellikle kendini çağıran metodların hatalı kullanılmasıyla meydana gelir.
- `System.NullReferenceException`: Bellekte yer ayrılmamış bir nesne üzerinden sınıfın üye elemanlarına erişmeye çalışırken oluşur.
- `System.OverflowException`: Bir veri türüne kapasitesinden fazla veri yüklemeye çalışılırken oluşur.
- `System.InvalidCastException`: Tür dönüştürme operatörüyle geçersiz tür dönüşümü yapılmaya çalışıldığında oluşur.
- `System.IndexOutOfRangeException`: Bir dizinin olmayan elemanına erişilmeye çalışılırken fırlatılır.
- `System.ArrayTypeMismatchException`: Bir dizinin elemanına yanlış türde veri atanmaya çalışılırken oluşur.
- `System.DividedByZero`: Sıfıra bölme yapıldığı zaman oluşur.
- `System.ArithmeticException`: `DividedByZero` ve `OverflowException` bu sınıftan türemiştir. Hemen hemen matematikle ilgili tüm istisnaları yakalayabilir.
- `System.FormatException`: Metodlara yanlış biçimde parametre verildiğinde oluşur.

throw anahtar sözcüğü

- ```
static void Main(string[] args)
{
 try
 {
 throw new DivideByZeroException;
 }
 catch (Exception e)
 {
 Console.WriteLine(e.Message);
 }
}
```

- throw (fırlat) Anahtar Sözcüğü  
throw ifadesiyle istenilen istisna istenilen anda ortaya çıkarılır. Bu istisnanın kullanıldığı yerde program durur ve istenen istisnayı üretir. Aşağıda "throw" anahtar sözcüğünün kullanıldığı bir örnek yer almaktadır.

# finally

- ```
static void Main(string[] args)
{
    Console.WriteLine("Sayı giriniz");
    try
    {
        int i = int.Parse(Console.ReadLine());
    }
    catch (FormatException exp)
    {
        Console.WriteLine(exp.Message);
    }
    catch (OverflowException exp)
    {
        Console.WriteLine(exp.Message);
    }
    finally
    {
        Console.WriteLine("finally bloğu çalışıyor...");
    }
}
```

Struct

- Sınıflar gibi, struct yapıları veri ve fonksiyon üyeleri içeren veri yapılarıdır. Sınıflardan farklı olarak, struct yapısı değer tipidir ve heap bölge tahsisi gerektirmez. Bir struct değişkeni, direkt olarak struct'ın verisini tutar. Oysa sınıf tipinde bir değişken, dinamik olarak ayrılmış nesneye referans tutar. Struct yapısı, kullanıcı tanımlı kalıtımı desteklemez ve tüm struct yapıları dolaylı olarak object tipinden kalıtım alır.
- Bu yapılar özellikle küçük veri yapıları kullanımında kullanışlı olurlar. Bir yapıda sınıf yerine struct kullanımı, bellek tahsis etmede ve programınızın performansında büyük farklılıklar yaratabilir.
- **Önemli:** Struct yapısının performans açısından sağladığı faydayı “Her zaman struct kullanın” şeklinde algılamak yanlış olur. Tabi ki bazı senaryolarda struct yapısına bellek ayırmak ve bellekten almak daha az zaman alır fakat her struct ataması bilindiği gibi değer kopyalamasıdır (value copy). Bu her zaman referans kopyalamasından daha fazla zaman alır.
- Struct yapıcılar **new** operatörü ile çağrılırlar fakat bu demek değildir ki belirli bir bellek ayrılmıştır. Nesnenin veya referansının dinamik olarak bellek ayrımı yerine, bir struct yapıcısı basitçe struct değerinin kendisini döndürür (genellikle yığının geçici bölgesinde) ve bu değer gerekli olduğunda kopyalanır.

Abstraction(Soyutlama)

- Karmaşıklığı yönetmek için kullanılır. Nesnenin diğer tüm nesne türlerinden ayıran temel özelliklerini belirtir, böylece izleyicinin bakış açısından açıkça tanımlanmış bir kavramsal sınır sağlar. Nesne yönelimli programlamada, ayırma, ayrıntıları tanımlamaktan ziyade sınıflar veya yöntemler için temel görevleri tanımlamak anlamına gelir. Temel olarak, problemi çözmek için kullanılan yöntem öncelikle daha genel, daha basit ve soyut olmalıdır.
- **Abstract Class:** abstract olarak tanımlanan bir sınıf temel sınıftır. Bu sınıftan new anahtar kelimesi kullanılarak bir nesne oluşturulamaz.
- **Abstract Metot:** Sadece soyut sınıflar içerisinde kullanılabilirler. Mirasçı sınıflarda override edilmek zorundadırlar. Abstract metotlar sadece tanımlanır. Herhangi bir işlemi yerine getirmezler. Yapacakları işlemler override edildikleri sınıfta kodlanmalıdır.

Abstract Sınıf Özellikleri

- Abstract sınıfları genel olarak inheritance (kalıtım) uygularken kullanırız.
- **new** anahtar sözcüğü ile nesneleri **oluşturamaz**.
- İçerisinde değişken ve metod **bulundurulabilir**.
- Abstract sınıflardan türetilen sınıfların abstract metodları implement etmesi **zorunludur**. Diğer metodları override etmeden de kullanabilir.
- Constructors (yapıcı metodlar) ve destructors (yıkıcı metodlar) **bulundurulabilirler**.
- Static **tanımlanamazlar**. (Tanımlanmaya çalışılırsa compiler *“an abstract class cannot be sealed or static”* hatası verir)
- Bir sınıf yalnızca bir abstract sınıfı inheritance yoluyla implement edebilir. Çoklu kalıtım (multiple inheritance) **desteklenmez**.
- Abstract olmayan metodları da **bulundurulabilir**.
- Kendisinden inherit alacak sınıflar ile arasında **“is-a”** ilişkisi vardır. (Burası ilk başlarda çok önemsenmeyen ancak hangi senaryoda Abstract hangi senaryoda Interface kullanacağımızı netleştirmede bize oldukça yardımcı olan bir detaydır, hemen aşağıda açıklamasını bulabilirsiniz)

Abstract ve interface farkları

ABSTRACT CLASS

- Constructor içerebilir
- Static üyeler içerebilir
- Farklı tiplerde Access Modifier (public, private vb) içerebilir
- Sınıfın ait olduğu içeriği belirtmek için kullanılır
- Bir sınıf sadece bir tane abstract sınıf inherit edebilir.
- Eğer birçok sınıf aynı türden ve ortak davranışlar sergiliyorsa abstract sınıfı base class olarak kullanmak doğru olacaktır.
- Abstract sınıf method, fields vb içerebilir
- Türetilen sınıflar abstract sınıfı tamamen yada kısmen implemente edebilir
- Method imzaları yada implementasyonları içerebilir.

INTERFACE

- Constructor içeremez
- Static üyeler içeremez
- Farklı tiplerde Access Modifier içeremez. Interface te tanımlanan her method default olarak public kabul edilir.
- Sınıfın yapabileceği kabiliyetleri belirtmek için kullanılır
- Bir sınıf birden fazla interface'i inherit edebilir.
- Eğer birçok sınıf yalnızca ortak methodları kullanıyorsa interface'ten türetilmeleri doğru olacaktır.
- Interface yalnızca method imzalarını içerebilir
- Türetilen sınıflar interface'i tamamen implemente etmek zorundadır
- Yalnızca method imzalarını içerebilir.

Boxing, Unboxing

- .Net Platformunda kullanmış olduğumuz veri tipleri, Değer tipleri ve Referans tipleri olarak ikiye ayrılmaktadır. Değer Tipleri stack bölgesinde tutulurken, Referans Tipleri heap bölgesinde tutulmaktadır.
- **Değer Tipleri:** “int”, “long”, “float”, “double”, “decimal”, “char”, “bool”, “byte”, “short”, “struct”, “enum”
- **Referans Tipleri:** “string”, “object”, “class”, “interface”, “array”, “delegate”, “pointer”
- Bir Değer Tipinin, Referans Tipine dönüştürülmesi işlemine Boxing, tersi bir işlemede Unboxing denmektedir. Koleksiyonlar verileri object olarak tutmaktadır. Bu yüzden koleksiyonlara her değer tipli eleman eklediğimizde Boxing işlemi gerçekleşecektir. Yani verimiz object’e dönüştürülecektir. Koleksiyona eklenen verileri değer tipli bir değişene aktarmak istediğimizde de Unboxing işlemi gerçekleşecektir. Koleksiyonun eleman sayısındaki artışa bağlı olarak boxing ve unboxing işlemleri artacaktır ve buna bağlı olarak da uygulamamızın performansı düşecektir.

Enum

Numaralandırmalar (enumerations) kod içerisinde sayısal karşılaştırma veya işlem gerektiren yerlerde yazılımcı için daha okunabilirlik sunan, kod karmaşasını azaltan yardımcı bir yapıdır.

Enum yapısı

Enum'ın en temel yapısı aşağıdaki gibidir.

```
enum enum_ismi { deger1,deger2,deger3};
```

Farklı Türde enum yapısı

```
enum Sonuç : byte { Kaldi,Gecti };
```

Enum

- Enum içerisinde değer vermezsek, değerler 0'dan başlar ve birer birer artar.
- Enum'ların varsayılan değer "int"dir.
- Enumları; byte,sbyte, short,ushort, int, uint,long, ulong türlerin oluşturabiliriz.
- Enum içerisine verdiğimiz değerlerde, değişken isimlendirmede dikkat edilen kurallar geçerlidir. Örneğin sayı ile başlayan veya içerisinde boşluk bulunan isimlendirmeler veremeyiz.

```
using System;
enum Mevsim
{
    İlkBahar = 1, Yaz, SonBahar, Kış
}
class Uygulama
{
    public static void Main()
    {
        int a = (int)Mevsim.İlkBahar;
        int b = (int)Mevsim.Yaz;
        int c = (int)Mevsim.SonBahar;
        int d = (int)Mevsim.Kış;

        Console.WriteLine("İlkbahar = {0} , Yaz = {1}, Sonbahar = {2} ,
Kış = {3}", a, b, c, d);
    }
}
```

Kolleksiyonlar

- **Kolleksiyonların Sağladığı Avantajlar**
- Bir dizi nasıl birden fazla elemanı temsil ediyorsa, bir koleksiyon nesnesi de aynı şekilde birden fazla nesneyi temsil etmektedir. Diziler ile koleksiyonlar arasındaki farklılıklardan bahsedecek olursak;
- Diziler sabit boyutludur ve eleman sayısının önceden belirtilmesi gerekir. Koleksiyonlar ise dinamik yapıdadır yani sabit boyutlu değildir. Eleman eklendikçe boyutu dinamik olarak artmaktadır.
- Diziler aynı veri tipindeki elemanları içermektedir. Koleksiyonlarda ise böyle bir kısıtlama bulunmamaktadır. Farklı veri tipindeki elemanları genel amaçlı koleksiyonlar üzerinde tutabiliriz.
- .Net Platformunda kullanmış olduğumuz veri tipleri, Değer tipleri ve Referans tipleri olarak ikiye ayrılmaktadır. Değer Tipleri stack bölgesinde tutulurken, Referans Tipleri heap bölgesinde tutulmaktadır.
- **Değer Tipleri:** “int”, “long”, “float”, “double”, “decimal”, “char”, “bool”, “byte”, “short”, “struct”, “enum”
- **Referans Tipleri:** “string”, “object”, “class”, “interface”, “array”, “delegate”, “pointer”
- Bir Değer Tipinin, Referans Tipine dönüştürülmesi işlemine Boxing, tersi bir işlemde Unboxing denmektedir. Koleksiyonlar verileri object olarak tutmaktadır. Bu yüzden koleksiyonlara her değer tipli eleman eklediğimizde Boxing işlemi gerçekleşecektir. Yani verimiz object’e dönüştürülecektir. Koleksiyona eklenen verileri değer tipli bir değişene aktarmak istediğimizde de Unboxing işlemi gerçekleşecektir. Koleksiyonun eleman sayısındaki artışa bağlı olarak boxing ve unboxing işlemleri artacaktır ve buna bağlı olarak da uygulamamızın performansı düşecektir.

Kolleksiyonlar

- **Genel Amaçlı Koleksiyonlar:** Her tipten veriyi saklamak için kullanılabilirler.

ArrayList

Dictionary

HashTable

Queue

SortedList

Stack

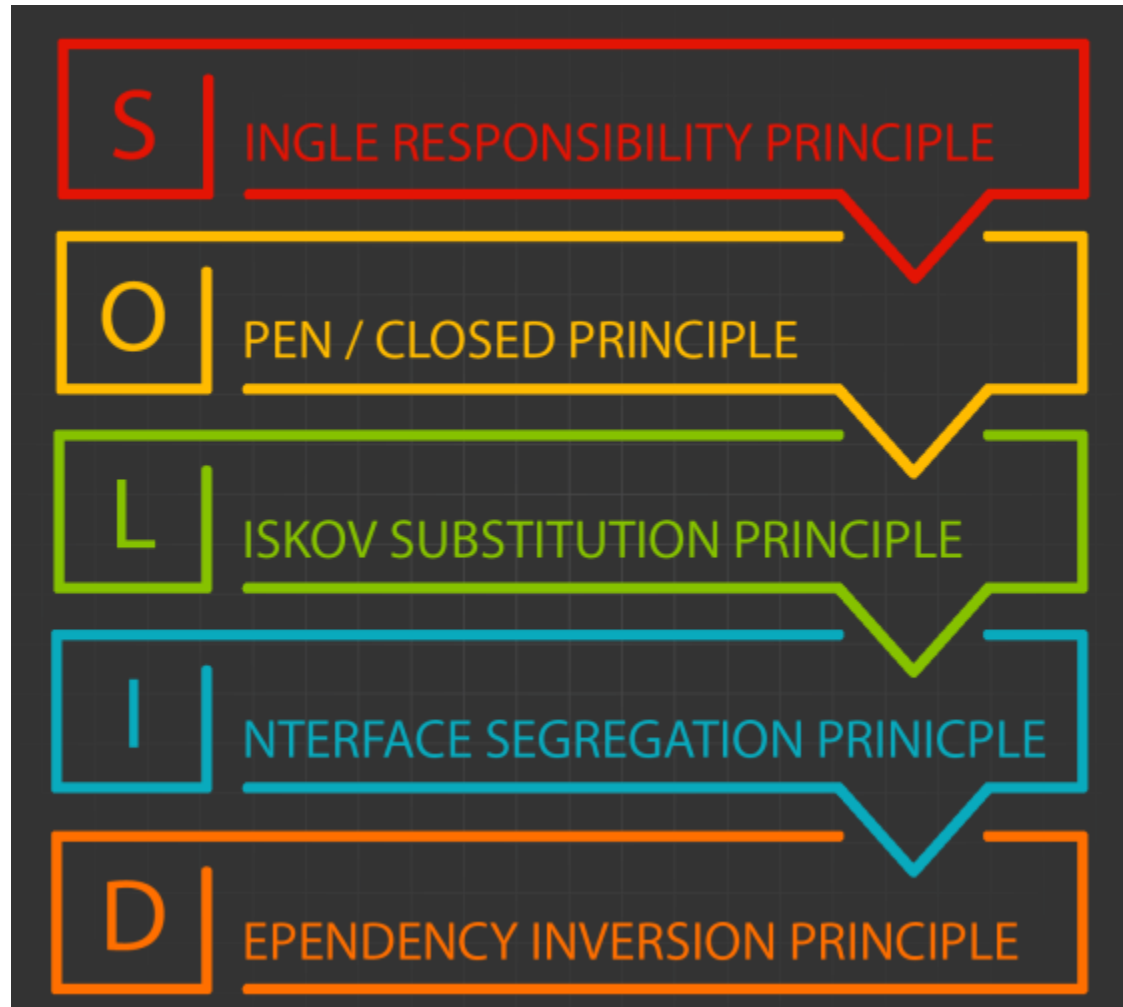
Kolleksiyonlar

- **Özel Amaçlı Koleksiyonlar:** Belirli bir veri tipi veya çalışma şekli için optimize edilmişlerdir.
 - ***CollectionsUtil***
 - ***ListDictionary*:** Anahtar-Değer çiftlerini bir bağlı liste içinde saklar. Küçük veri kümeleri için tercih edilir.
 - ***HybridDictionary*:** Belirli bir büyüklüğü geçene kadar Anahtar-Değer çiftlerini ListDictionary koleksiyonunda tutmaktadır, geçtikten sonra otomatik olarak bir Hashtable koleksiyonuna geçiş yapar.
 - ***NameValueCollection*:** Anahtar-Değer çiftlerinin her ikisinde string tipinde olduğu durumlarda tercih edilebilir.
 - ***StringCollection*:** Karakter katarlarını saklamak için optimize edilmiş bir koleksiyondur.
 - ***StringDictionary*:** Anahtar-Değer çiftlerinin her ikisinde string tipinde olduğu bir hash tablodur. Anahtar-Değer çiftleri küçük harflere çevrilerek saklanır.

Kolleksiyonlar

- **Bit Tabanlı Koleksiyonlar:** İsminden de anlaşılacağı üzere bu koleksiyonlar bir grup biti içerisinde saklamaktadır. Bit tabanlı koleksiyonlara örnek olarak [BitArray](#)'i verebiliriz. [BitArray](#) bitleri saklamak haricinde VE – VEYA gibi mantıksal işlemleri de gerçekleştirebilmektedir.

SOLID



SOLID

- **S** is single responsibility principle (SRP)
- **O** stands for open closed principle (OCP)
- **L** Liskov substitution principle (LSP)
- **I** interface segregation principle (ISP)
- **D** Dependency injection principle (DIP)

SOLID Kuralları

- Bir class bir sorumluluk almalı yani bir class'ın isviçre çakısı gibi bir sürü görevi, özelliği olmamalıdır. Aşağıdaki ilk örnekteki gibi, hem nesnesinin özelliklerini barındırıp hem de bir kaç metodun birleşiminden oluşan bir metodumuz olmamalıdır. Bu nedenle, SRP prensibine göre, bir class bir sorumluluk almalıdır.
- Gelişime açık fakat değişiklik için kapalıdır.
- Bir class ta bulunan özellikler, kendisinden kalıtım alan class'larda kullanılmayacak sabu durum LSP'ye aykırı bir durumdur. Yani kalıtım alınan class'ın içindeki özellikler kalıtımı alan class ta kullanılmalıdır.
- Interface Segregation prensibine göre, “istemcilerin kullanmadıkları arayüzleri uygulamaya zorlanmaması gerektiğini” savunulmaktadır. Herbir interface'in belirli bir amacı olmalıdır. Tüm metodları kapsayan tek bir interface kullanmak yerine, herbiri ayrı metod gruplarına hizmet veren birkaç interface tercih edilmektedir.
- Üst seviye (High-Level) sınıflar alt seviye (Low-Level) sınıflara bağlı olmamalıdır, ilişki abstraction veya interface kullanarak sağlanmalıdır, Abstraction(soyutlama) detaylara bağlı olmamalıdır, tam tersi detaylar abstraction(soyutlama)'lara bağlı olmalıdır.

