NESNE YÖNELİMLİ TASARIM VE PROGRAMLAMA - GİRİŞ

Ön Bilgi

Nesne yönelimli tasarım ve programlama dersinin başlıca hedefleri arasında öğrencilere; daha doğal, günlük hayatta iç içe bulunduğumuz kavramlarla daha çok örtüşen ve benzeşen, bünyesinde hiyerarşik bir yapıyı bulunduran, modüler, daha akılcı ve daha ergonomik bir program tasarım biçimi olan nesne yönelimli tasarım hakkında temel bilgilerin verilmesi, nesne yönelimli tasarımın gerekliliğinin kavratılması ve nesne yönelimli tasarım mantığından hareketle nesne yönelimli programlama aracılığı ile bahsedilen tasarım yaklaşımının hayata geçirilmesi yer almaktadır. Bu ders boyunca aktarılacak olan kavramların pratiğe dökülebilmesi için nesne yönelimli bir programlama dili olan C# kullanılarak gerçekleştirilen konsol uygulamaları ele alınacaktır.

Dersin, programlama dili öğretimi amacından çok nesne yönelimli tasarım bilgilerinin verilmesi amacını taşıması nedeniyle C# programlama dili hakkında giriş niteliğinde temel bilgiler verildikten sonra nesne yönelimli tasarım ve programlamaya ait kavramlar uygun bir seviyede ve sırada aktarılacaktır. Öğrencilerin nesne yönelimli tasarım ve programlamanın önemini kavramaları ve bu alanda gerekli beceriyi kazanmaları, dönem içerisinde adım adım gelişerek dönem sonunda tamamlanacak bir olgudur.

C# Programlama Dili

C#, Microsoft tarafından geliştirilmiş, nesne yönelimli programlamayı destekleyen bir programlama dilidir. C# programlama dilinin sözdizimsel yapısı, kısıtlamaları, makine diline dönüştürülmesine, tasarım ve kullanım amacına (örneğin nesne yönelimliliği desteklemesi) dair kurallar, Avrupa Bilgisayar Üreticileri Birliği (ECMA: European Computer Manufacturers Association) tarafından belirlenen standartlara uygun biçimde düzenlenmiştir.

Programlama dilleri kullanılarak hazırlanan programların bilgisayar ortamında çalıştırılabilmeleri için, makinenin (yani bilgisayarın) anlayabileceği tek dil olan **makine dili**ne dönüştürülmeleri gerekmektedir. Makine dili, sadece '**1**' ve '**0**' lardan oluşmaktadır. "Çalıştırılabilir dosya" olarak nitelendirdiğimiz dosyaların içerisinde doğrudan makine dili ile yazılmış ve donanım üzerinde çalıştırılmaya hazır veriler vardır. Windows işletim

sistemlerinde çalıştırılabilir dosyaların uzantıları genellikle "exe" iken, Unix işletim sistemlerinde çalıştırılabilir dosyalar için herhangi bir uzantı sınırlaması yoktur.

```
using System;

class Sinif1
{
    static void Main()
    {
        Console.WriteLine("Merhaba dunya!");
    }
}
```

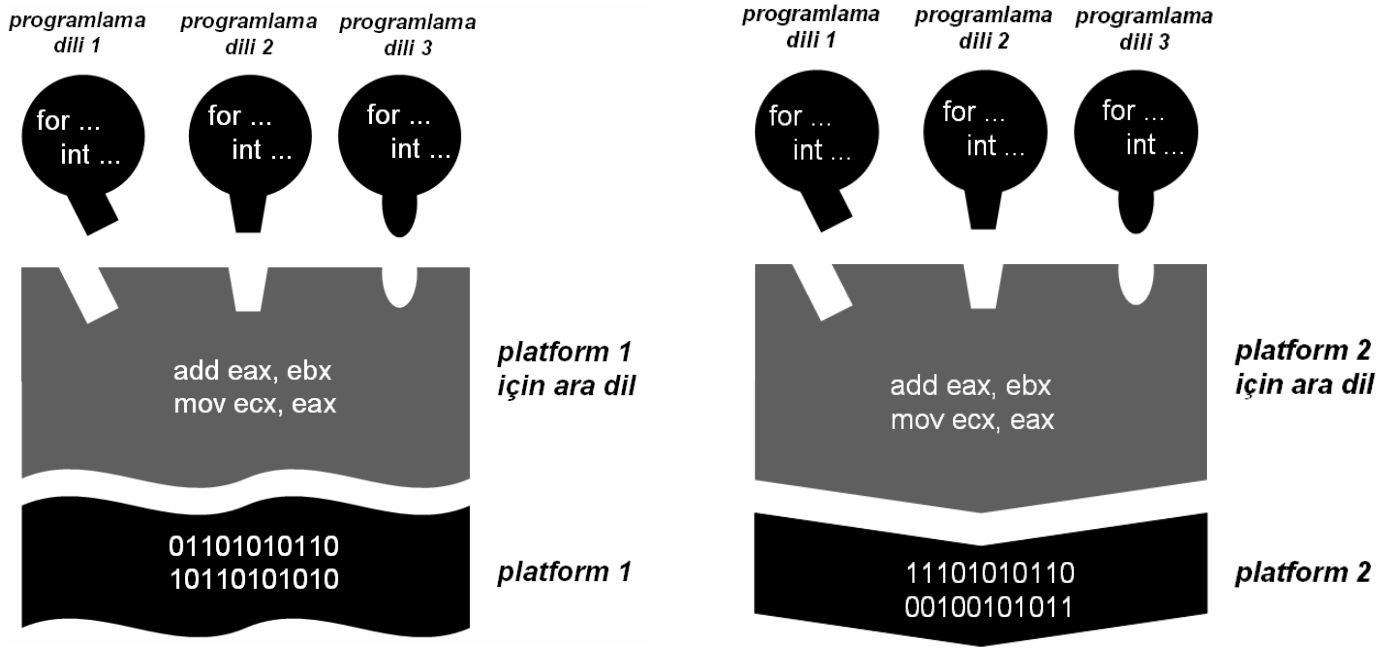
Sözgelimi yukarıdaki kutuda, C# programlama dili ile yazılmış olan ve ekrana "**Merhaba dunya!**" yazdırmaya yarayan küçük bir program verilmiştir. Bu program, programcının anlayabileceği düzeyde olup, bilgisayar için işlenebilirlik konusunda bir anlam ifade etmemektedir. Bilgisayar, sadece **1** ve **0**' larla muhatap olabildiği için, bu programın, bilgisayarın anlayabileceği formata dönüştürülmesi gerekmektedir. Bu dönüşüm işleminden sonra elde edilen ***1 ve 0' lar kümesi***, gelişigüzel oluşturulmuş olmayacaktır. Bilgisayarın, bu ***1 ve 0' lar kümesinden*** hareketle ekrana "**Merhaba dunya!**" yazısını yazdırabilmesi için bu kümede yer alan **1** ve **0**' ların uygun sayıda olması ve belirli bir sıraya göre düzenlenmiş olması şarttır. Bu sayı ve sıranın insan tarafından anlaşılması çok zor, hatta imkânsızdır.

*Fikir vermek için en basit ifadelerle örneklemek gerekirse, yazmış olduğumuz program derlenerek sayısı ve sırası belli bir **1' ler ve 0' lar** kümesine dönüştürülecek, bu sıralanmış **1**' ler ve **0**' lar kümesi bellekte tutulacak, bir yandan da işlemci aracılığı ile işlenecektir. Bu "işleme" süreci, özetle birer statik elektrik yükü olan bu **1** ve **0**' ların donanım seviyesinde "and", "or", "xor" gibi bit bazlı işlemlere tabi tutulması olarak düşünülebilir. Bu işlemler sonucunda elde edilen ve bellekte tutulan belli sayıda ve sırada olan yeni yeni **1' ler ve 0' lar kümeleri**nin de işletim sistemine ait **1 ve 0 kümeleri** ile belirli kurallar çerçevesinde etkileşime girdikten sonra işlemci ve veri yolları vasıtası ile diğer çevresel birimlere aktarılabileceği düşünülürse, "**Merhaba dunya!**" yazısının yazılması durumunda ekrandaki hangi piksellerde hangi değişikliklerin yapılacağı da gene **1** ve **0**' larla belirlenmektedir.*

İnsanın (programcının) anlayabildiği programların, bilgisayarın anlayabileceği **1' ler ve 0' lar kümesi**ne dönüştürülmesi işlemi, kimi zaman **derleyici**ler, kimi zaman da **yorumlayıcı**lar tarafından yapılmaktadır. **Derleyiciler** ve **yorumlayıcılar** da birer yazılımdır ancak burada aralarında ne fark olduğu ele alınmayacaktır.

C# programlama dilinde yazılmış olan programlar, genellikle derleyiciler vasıtası ile makinenin anlayabileceği forma dönüştürülür. C# programlarını derlemek için Windows işletim sistemleri altında kullanılabilen Microsoft Visual Studio .NET programı ile Unix işletim sistemleri altında kullanılabilen Mono programı birer derleyicidir. Programcı tarafından yazılan bir program ile bu programın derlenmesiyle makinenin kullanabileceği hale dönüştürülerek elde edilen yeni program, anlam bakımından birbirinin aynıdır (İkisi eşdeğerdirler.). Programcının geliştirdiği program makine için bir anlam taşımazken, makine diline çevrilmiş program da programcı için bir anlam ifade etmeyecektir.

Microsoft Visual Studio .NET içerisinde yer alan C# ve diğer programlama dilleri (VB: Visual Basic, C++) makine koduna direkt olarak dönüştürülmek yerine ilk önce ortak bir ara dile dönüştürülürler. Bu ara dile "CIL (Common Intermediate Language)" adı verilir (Microsoft Visual Studio .NET tarafından kullanılan CIL, "MSIL" adıyla da bilinir.). Bu ara dil, ECMA/ISO standartları çerçevesinde tasarlanmıştır. Visual Studio .NET ortamında VB ya da C++ dilleri kullanılarak yazılan programlar da tıpkı C# programları gibi ara dile dönüştürülürler. Benzer şekilde, Unix altında Mono derleyicisi kullanıldığında da bir ara dile dönüşüm söz konusudur. Yani, derlenen bir C# programının ilk önce CIL ara diline, sonrasında ise makine diline çevrilmesi söz konusu olmaktadır. CIL ara dilinin makine diline dönüştürülmesi ise platforma özgü olarak tasarlanmış olan "CLR (Common Language Runtime)" yazılımı tarafından yapılır. Böylesi bir yöntemin tercih edilmesinin nedeni, programlama dilinin platformdan bağımsız olarak kullanılabilmesine olanak sağlamaktır.



Özetle ".NET Framework" kavramı: Sözgelimi, bir Japon balığının yaşaması için gerekli olan şeyler içerisinde belli oranda çözünmüş oksijen bulunan su, bir miktar yem ve ortalama 20 santigrat derece sıcaklık olsun. Eğer evimizde akvaryum yoksa, bir Japon balığını beslememiz mümkün değildir. Bizim yaşam standartlarımıza uygun olan ancak bir Japon balığı için elverişsiz olan evimizde bu balığı beslemek için, içerisinde ısıtıcı ve oksijen kaynağı olan bir akvaryum alıp yemle doldurmamız gereklidir. Aynı mantıktan hareketle, bazı uygulamalar (kimi zaman web uygulamaları, kimi zamansa diğer konsol / form uygulamaları) da direkt olarak kullandığımız işletim sistemi üzerinde çalışmaz ve ayrı bir ortama, ayrı kütüphanelere gereksinim duyar. .NET Framework de bu tür uygulamaların çalıştırılabilmesi için ortam hazırlayan, bulundurduğu çeşitli kütüphanelerle bu uygulamalara işletim sisteminin sağlayamadığı birtakım özellikleri sağlayan bir yazılımdır. Bir önceki cümlede geçen ".NET Framework tarafından sağlanan ortam" kavramını "akvaryum"; "özellik" kavramını ise "oksijen", "yem" olarak düşünebiliriz. Buna göre bahsedilen "işletim sistemi" ise, bizim yaşayabildiğimiz ancak Japon balığının yaşayamadığı "ev" olarak düşünülebilir. İşletim sistemi üzerinde doğrudan çalıştırılamayan ancak .NET Framework üzerinde çalışabilen uygulama ise "Japon balığı" na karşılık gelmektedir.

"Merhaba dunya!" Programını İnceleyelim

Aşağıda, ekranda "**Merhaba dunya!**" çıktısını üretmek amacı ile hazırlanmış basit bir C# programı yer almaktadır.

```
using System;                                   #1
namespace Uzay1                                 #2
{
    class Sinif1                                #3
    {
        static void Main()                      #4
        {
            Console.WriteLine("Merhaba dunya!"); #5
        }
    }
}
```

Yukarıdaki programda elips içerisine alınarak numaralandırılmış kod parçalarının ne anlama geldiklerini inceleyelim:

#1 : "System" adlı isim uzayının kullanılabileceğini belirtir. Bu isim uzayının kapsamındaki tüm sınıflardan, arayüzlerden, *struct* veri tiplerinden yararlanılabilecektir.

#2 : "Uzay1" adında bir isim uzayı oluşturulmaktadır. İlerleyen adımlarda bu isim uzayının içerisine sınıflar, arayüzler vs. koyulabilir.

#3 : "Uzay1" isim uzayının içerisinde, "Sinif1" isminde bir sınıf oluşturulmaktadır. İlerleyen adımlarda bu sınıfın içerisinde değişken tanımlanabilir, fonksiyon gerçekleştirimi yapılabilir.

#4 : "Sinif1" sınıfı içerisinde, "Main" isminde, argüman almayan ve değer döndürmeyen bir fonksiyon tanımla ("static" deyiminin ne işe yaradığı ilerleyen konularda ele alınacaktır.).

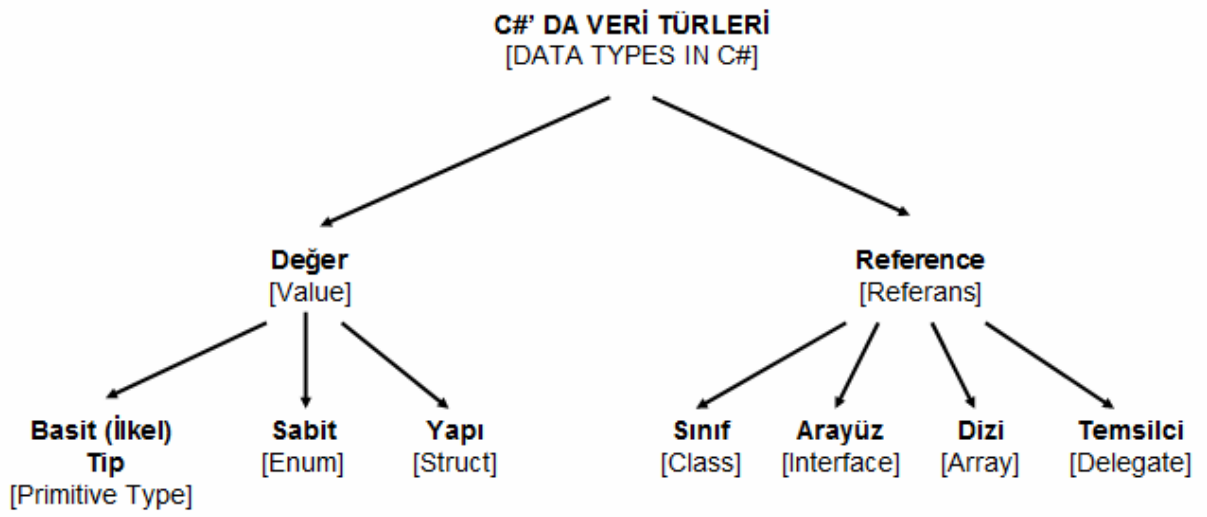
NOT: Hazırladığımız programın çalıştırılabilmesi için, giriş kapısı görevi üstlenen bir ana fonksiyonun tanımlanması şarttır. Bu ana fonksiyonun ismi "**Main**" olmak zorundadır ve statik erişimli olması gerekmektedir.

#5 : "Console" sınıfı içerisindeki "WriteLine" fonksiyonuna argüman olarak ***"Merhaba dunya!"*** karakter dizisini ver.

NOT: "Console" sınıfı, "System" isim uzayı içerisinde bulunmaktadır. Eğer **#1** numaralı deyim yazılmasaydı, bu fonksiyona ulaşabilmek için **#5** numaralı deyimin "**System.Console.WriteLine("Merhaba dunya!");**" biçiminde olması gerekirdi.

C# Programlama Dilinde Veri Türleri

C# programlama dilinde, **değer** (value) ve **referans** (reference) olmak üzere iki farklı veri türü bulunmaktadır.



Değer Türü → Her değer, direkt olarak kendi verisini bulundurur. Bir değer üzerinde yapılan herhangi bir işlemin diğerini etkilemesi mümkün değildir.

Referans Türü → Referans türünden olan değişkenler ise nesneleri refere ederler, yani gösterirler. Dolayısı ile bu tür değişkenlere "tutacak" da denebilir. Referans türünden iki değişkenin aynı nesneyi refere etmesi mümkündür. Refere edilen nesnedeki olası bir değişiklik, refere eden değişkenlerin (tutacakların) tamamı tarafından da görülmüş (izlenmiş) olur.

```
using System;

namespace Uzay1
{
    public class Sinif1
    {
        public int sayi = 0;
    }

    public class Test
    {
        static void Main()
        {
            int deger1 = 0;
            int deger2 = deger1;
            deger2 = 123;

            Sinif1 referans1 = new Sinif1();
            Sinif1 referans2 = referans1;
            referans2.sayi = 456;
            Console.WriteLine("Degerler    : {0}, {1}", deger1,
                                                        deger2);
            Console.WriteLine("Referanslar : {0}, {1}",
                                  referans1.sayi, referans2.sayi);
            System.Console.Read();
        }
    }
}
```

Çıktı:

```
Degerler    : 0, 123
Referanslar : 456, 456
```

Yukarıda verilen programda atama yapılan satırları görsel örneklerle açıklayalım:

```
int deger1 = 0;
```

deger1 adında bir çanta oluşturulur ve içerisine *0* koyulur.

```
int deger2 = deger1;
```

deger2 adında başka bir boş **<u>çanta</u>** (***<u>değişken</u>***) oluşturulur ve içerisine **deger1** çantasında bulunan değerin aynısı koyulur.

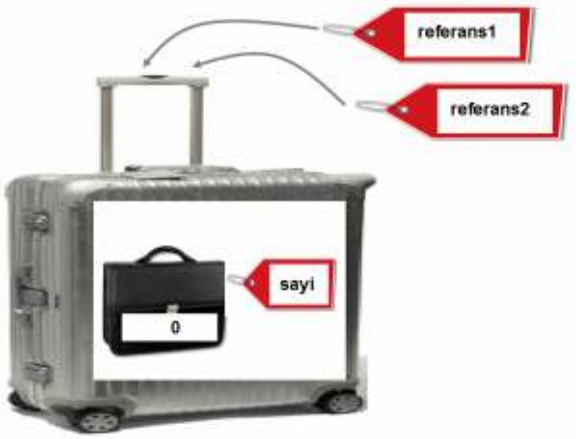
```
deger2 = 123;
```

deger2 çantasının içerisindeki değer *123* yapılır.

```
Sinif1 referans1 = new Sinif1();
```

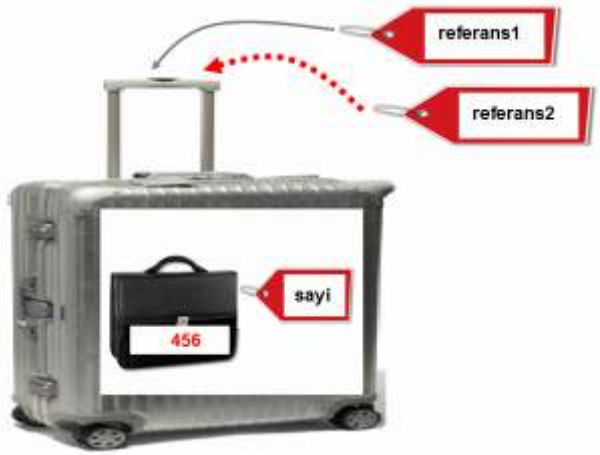
Sinif1 tipinden yeni bir **valiz** (***sınıf nesnesi***) oluştur -ki **Sinif1** nesnelerinin içerisinde "sayi" isimli çantalar olacaktır- ve bu valizin adresini, gene **Sinif1** tipinden olan **referans1** tutacağına kaydet.

```
Sinif1 referans2 = referans1;
```



Sinif1 tipinden **referans2** isminde bir tutacak oluştur ve **referans1** tutacağının göstermekte olduğu nesneyi bu tutacağa da paylaştır.

```
referans2.sayi = 456;
```



referans2 tutacağının refere ettiği valizde yer alan **sayi1** çantasının içindeki değeri *456* yap.

Görsel örnekler de incelendiğinde, değer türlerinin kendilerine ait verileri olduğu, referans türlerinin ise nesneleri paylaşma özelliklerinin olduğu anlaşılmaktadır.

Tip Güvenliği (Type Safety): C# programlama dili, tip güvenliğine sahip bir dildir. Yani, değer türünden bir değişken ya da referans türünden bir nesne tutacağı tanımladığımızda, bunun hangi tipte (*tamsayı, karakter dizisi vb.*) olduğunu belirtmemiz gerekir. Programın tamamında, bu değişken ve tutacaklara tanımlanmış oldukları tipin dışında bir tipe sahip değer ataması yapılamaz. Oysa Python programlama dilinde, bir önceki cümlede belirtilen anlamı ile bir tip güvenliğinden söz edemeyiz.

KOD	ÇIKTI

`a = 4`
`print a`
 	
`a = 'karakter'`
`print a`
	`4`
	`karakter`

Görüldüğü gibi, **a** değişkeninin tipini (*int*, *string* vb.) önceden (yani statik olarak) belirtmedik ve **a** değişkenine her atama yapıldığında, atanan değere göre **a**' nın tipi de dinamik bir biçimde değişti (yani yeniden belirlendi). Elbette bu durum, Python' da hiç tip güvenliği olmadığı, yani üzerinde işlem yapılan nesnelerin tiplerinin hiçbir önem arz etmediği anlamına gelmez:

```
a = 4
b = 'bes'
c = a + b
```

Yukarıdaki Python programı, 3. satırda "+" operatörünün tamsayı ve karakter dizisi tipindeki iki farklı değişken üzerinde tanımlı bir işleve sahip olmaması nedeniyle hata verecektir. Bu, değişken tiplerinin Python' da da önem arz ettiğinin bir göstergesi olarak düşünülebilir.

```
using System;

namespace Uzay1
{
    public class Sinif1
    {
        public int sayi = 0;
    }
    public class Sinif2
    {
        public string sozcuk = "bilgisayar";
    }

    public class Test
    {
        static void Main()
        {
            // Deger Turlerinde Tip Guvenligi
            int deger1;
            deger1 = 3;
            deger1 = 5;
            deger1 = "KarakterDizisi"; // hatali

            // Referans Turlerinde Tip Guvenligi
            Sinif1 referans1a, referans1b;
            Sinif2 referans2;

            referans1a = new Sinif1();
            referans1a.sayi = 7;
            referans1b = referans1a;
            referans2 = referans1a; // hatali

            System.Console.Read();
        }
    }
}
```

Yukarıdaki C# programında, **deger1** isminde ve *int* tipinde bir değişken (Bunun gibi ilkel tipten değişkenlerin "değer" türünden olduğunu hatırlayalım.) tanımlanmıştır. Sonrasında ise bu değişkene 3 farklı atama yapılmıştır. Bu atamaların ilk ikisinde atanan değer tamsayı olduğu için herhangi bir sorunla karşılaşılmamıştır. Ancak 3. atamada atanan değer bir karakter dizisi olduğu için bir derleme anı hatası ile karşılaşılacaktır. Atanan değerin, değişkenin tipi ile aynı tipte olması (ya da dönüşüm ile değişkenin tipine çevrilebilir olması) gerekir.

Programın devamında ise, **referans1a**, **referans1b** ve **referans2** olmak üzere üç farklı nesne tutacağı (nesne adreslerini tutan adres defterleri) tanımlanmış olup bunlardan ilk ikisi **Sinif1**, sonuncusu ise **Sinif2** tipindedir. Burada da bir önceki paragrafta anlatılan durumla benzer bir durum söz konusudur. Bir nesne tutacağı tanımlandığında, hangi tip nesneleri işaret etmekle

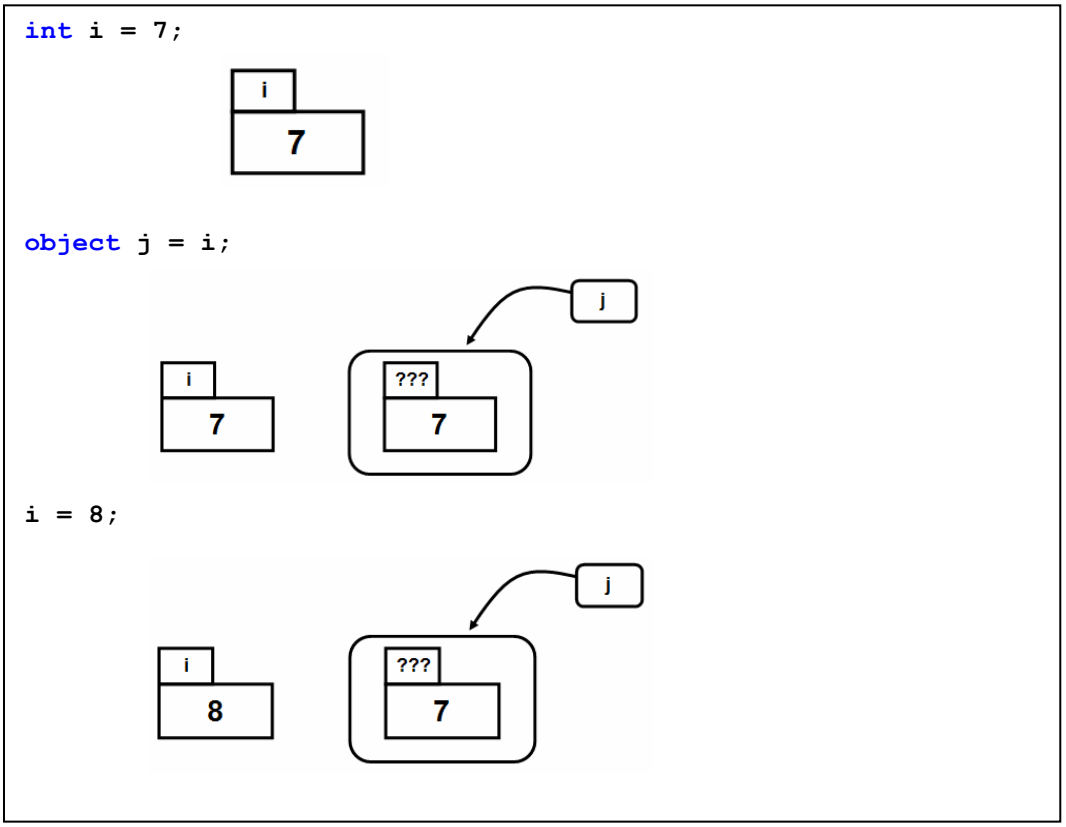
görevlendirileceklerinin belirtilmesi gerekir. Programda ilk önce yeni bir **Sinif1** nesnesi yaratılmış ve **referans1a** tutacağı bu nesneye bağlanmıştır. Sonrasında, **referans1a** tutacağı vasıtası ile yaratılan nesnenin "**sayi**" isimli alt alanına değer ataması yapılmıştır. Daha sonra, **referans1b** tutacağına, **referans1a** tutacağı tarafından gösterilen nesneyi gösterme görevi verilmiş, bu sayede tek bir nesne iki farklı tutacak tarafından işaret edilir olmuştur. Sonraki adımda ise yaratılmış olan nesnenin, **referans2** tutacağı tarafından da gösterilmesi istenmiştir ancak bu durum, derleme anı hatasına yol açmaktadır çünkü **referans2** tutacağı sadece **Sinif2** tipindeki nesnelerin adresini tutabilir, diğer tipten bir nesneyi işaret etmesi söz konusu olamaz.

Önceden Tanımlı Tipler

C# programlama dilinde, **önceden tanımlı referans tipleri** ve **önceden tanımlı değer tipleri** mevcuttur.

Önceden tanımlı referans tipleri, **object** ve **string**' dir.

object → Tüm diğer tipleri kapsayan genel bir tiptir.



Bir **değer** türü bir **referans** türüne dönüştürülürken bir nesne kutusu (*object box*) üretilir ve değer bu kutuya **kopyalanır**.

string → *Unicode* karakter değerlerini tutmak içindir. Sonradan değiştirilemezler.

```
string str = "abcde";
Console.WriteLine(str[2]); // dogru
str[2] = "f"; // hatali
str[3] = 'g'; // hatali
```

Önceden tanımlı değer tipleri ise şu şekilde sıralanabilir:

- Tamsayı Tipleri
  - işaretli
    - **sbyte** → 8 bit
    - **short** → 16 bit
    - **int** → 32 bit
    - **long** → 64 bit
  - işaretsiz
    - **byte** → 8 bit
    - **ushort** → 16 bit
    - **uint** → 32 bit
    - **ulong** → 64 bit
- Kayan Noktalılar
  - **float** → 32 bit *(`float flt1 = 1.23F;`)*
  - **double** → 64 bit *(`double dbl1 = 1.23; double dbl2 = 4.56D;`)*
- **bool** → 1 bit
- **char** → 8 bit
- **decimal** → 128 bit

Operatörlere Yeni Anlam Yükleme *(Operator Overloading)*

Önceden tanımlı tipler, operatörlere yeni anlamlar yükleyebilmektedirler. "==" ve "**!**=" operatörlerinin anlamı değer tipleri (*int, double*) için farklı, referans tipleri (*object, string*) içinse farklıdır:

- **int** → Referans türünden bir veri tipidir. İki tane **int** tipinden değişkenin değerleri aynıysa bunlar eşit, değilse farklıdırlar:

```
int x = 5;
int y = 5;
Console.WriteLine(x == y);
```

Çıktı:

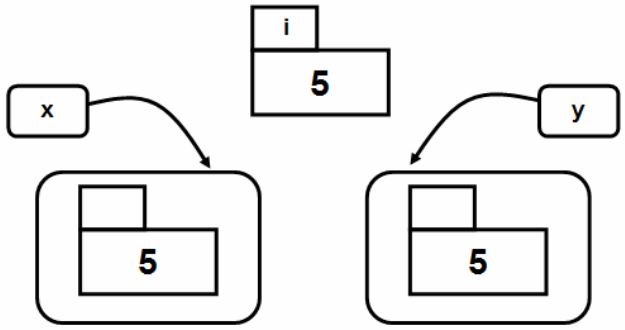
```
True
```

- **object** → **object** tipinden iki farklı değişkenin (tutacağın, referansın) refere ettiği nesneler aynı nesne ise ya da bu değişkenlerden her ikisi de **null** değere sahipse (hiçbir nesneyi refere etmiyorlarsa) bu değişkenler eşit, diğer durumlarda ise farklıdırlar:

```
int i = 5;
object x = i;
object y = i;
Console.WriteLine(x == y);
```

Çıktı:

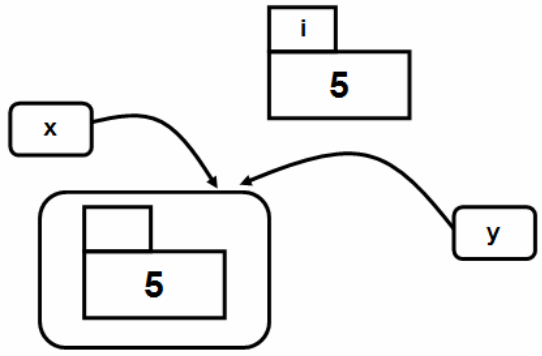
```
False
```




```
int i = 5;
object x = i;
object y = x;
Console.WriteLine(x == y);
```

Çıktı:

```
True
```



- **string** → **string** tipinden iki farklı değişkenin (tutacağın) gösterdikleri karakter dizilerinin uzunlukları eşitse ve indekslerinde yer alan değerler birebir aynıysa; ya da her ikisi de **null** değere sahipse bunlar eşit, diğer durumlarda farklıdırlar:

```
string x = "ABC.def789";
string y = "ABC.def789";
Console.WriteLine(x == y);
```

Çıktı:
```
True
```

```
string x = "ABC.def789";
string y = "ABC.def78";
Console.WriteLine(x == y);
```

Çıktı:
```
False
```

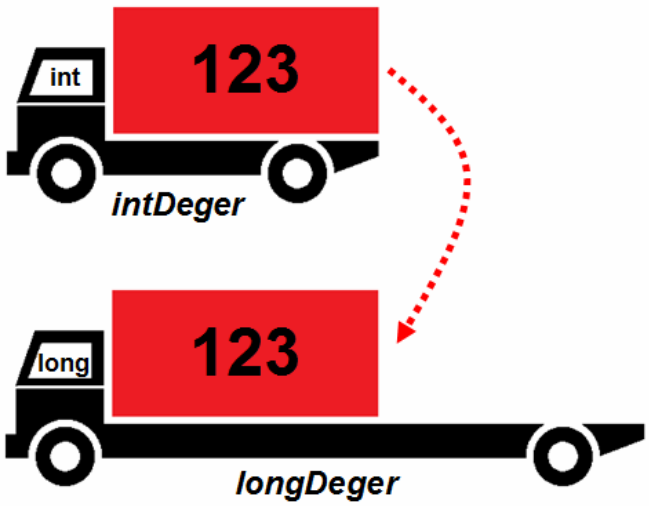
Dönüşümler *(Conversions)*

Önceden tanımlı tipler arasında dönüşüm yapmak mümkündür ve bu dönüşümün kuralları da önceden belirlenmiştir. Örneğin, **int** ve **long** arasında dönüşüm söz konusudur. C# programlama dilinde iki tür dönüşüm vardır: **kapalı** (***implicit***) ve **açık** (***explicit***).

- **Kapalı Dönüşümler** → Bilgi kaybı olmaz. Her zaman başarılı bir biçimde gerçekleşir.

```
int intDeger = 123;
long longDeger = intDeger;
Console.WriteLine("{0}\n{1}", intDeger, longDeger);
```

Çıktı:
```
123
123
```



Uzun gösterim (yukarıdaki kutunun 2. satırı):

```
long longDeger = (long)intDeger; // tip donusumu (cast)
```

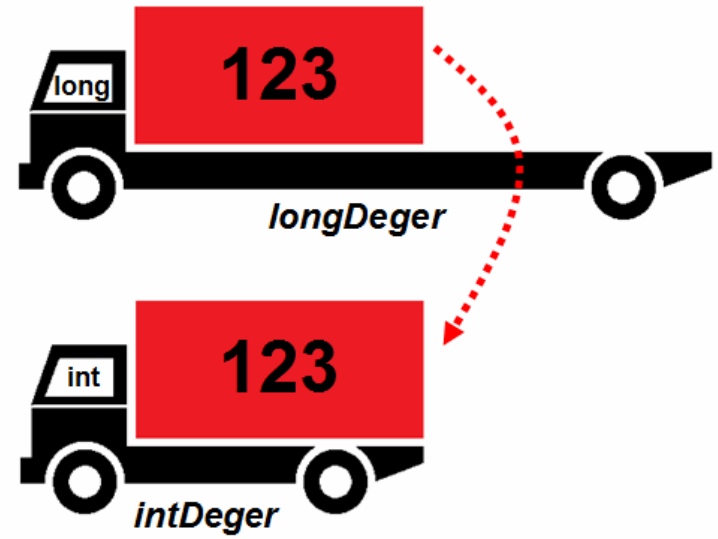
- **Açık Dönüşümler** → Bilgi kaybı olabilir de, olmayabilir de. Tip dönüşüm (*cast*) işlemi programcı tarafından yapılmalıdır, otomatik olarak gerçekleşmez. Her zaman başarılı bir biçimde gerçekleşmeyebilir:

Örnek 1 (Bilgi kaybı yok.):

```
long longDeger = 123;
int intDeger = (int) longDeger;
Console.WriteLine("{0}\n{1}", longDeger, intDeger);
```

Çıktı:

```
123
123
```

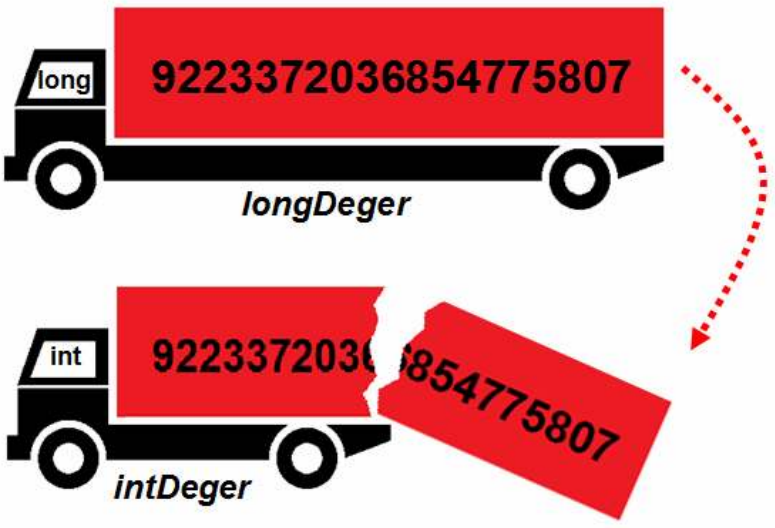


Örnek 2 (Bilgi kaybı var.):

```
long longDeger = 9223372036854775807;
int intDeger = (int) longDeger;
Console.WriteLine("{0}\n{1}", longDeger, intDeger);
```

Çıktı:

```
9223372036854775807
-1
```



Yukarıda verilen 2. örnekte bilgi kaybı olmuştur. **int** veri tipi 32 bit' lik, **long** veri tipi ise 64 bit' lik kapasiteye sahip olduğundan ve **longDeger** değişkeninde tutulan değer 32 bit ile temsil edilemeyecek kadar büyük olduğundan, bir **taşma** (***overflow***) söz konusu olmuştur. Tip dönüşüm işlemi ile **int** veri tipi **long**' a dönüştürülmüştür ancak veri kaybı yaşanmıştır.

Diziler Hakkında Temel Bilgiler

C# programlama dilinde tek boyutlu, çok boyutlu ve iç içe dizi tipleri mevcuttur. Dizilerin temel türü, **System** ad uzayında bulunan "**Array**" isimli bir sınıftır. Pratikte ise çok boyutlu birer değişken olarak kullanılırlar.

Tek Boyutlu Diziler

```
int[] dizi1; // #1
dizi1 = new int[7]; // #2
Console.WriteLine(dizi1.Length); // #3
Console.WriteLine(dizi1[3]); // #4
```

Çıktı:

```
7
0
```

- Yukarıdaki kodun 1. satırında, **dizi1** isminde bir tek boyutlu dizi tanımlanmıştır. Burada **dizi1** değişkeni aslında, ileriki adımlarda bellekte oluşturulacak ve uzunluğu (eleman sayısı) henüz belli olmayan bir tamsayı dizisinin başlangıç adresini temsil eden bir takma addır.
- 2. satırda, önce bellekte 7 birim uzunluğunda bir tamsayı dizisi oluşturulmuştur. Başka bir deyişle, 7 tane tamsayının bellekte tutulacağı büyüklükte (7 x 32 = 224 bit genişliğinde) ardışık bir bölge tahsis edilmiştir. Bu bellek bölgesinin başlangıç adresi de bellidir ve bu adresi sayısal olarak ifade etmek zor ve anlamsız olacağı için, bu adrese karşılık gelen bir takma ad olarak **dizi1** verilmiştir. Anlaşılırlığı artırmak adına, **dizi1**' den "bir tamsayı dizisi" olarak bahsedilecektir.
- 3. satırda ekrana **dizi1**' e ait uzunluk değeri yazdırılmaktadır. C# programlama dilinde dizilerin uzunluklarını almak için böyle bir yol izlenebilir. Aslında burada yapılan şey, bu dizinin temel sınıfı olan **System.Array** sınıfında yer alan "**Length**" isimli bir özellik (*property*) aracılığı ile dizinin uzunluğunu almaktır.
- 4. satırda ise **dizi1** dizisinin 3. indeksindeki elemanına (C#' ta indeks numaraları 0' dan başladığı için baştan 4. elemanına) ulaşılarak bu elemanın değeri ekrana yazdırılmıştır

(**WriteLine** fonksiyonu tamsayı argüman da alabilmektedir.). C#' ta oluşturulan çok boyutlu dizilerin tüm elemanları ilk başta **0** olacak şekilde otomatik ilkleme yapılır.

```
int[] dizi1 = new int[7];
```

Bir önceki kutunun 1. ve 2. satırlarında verilen işlem, yukarıdaki gibi tek satırda da yapılabilir.

```
int[] dizi1 = new int[];
```

Yukarıdaki kod parçası ise derleme anı hatasına yol açacaktır çünkü bellekte kaç tane tamsayının sığacağı kadar ardışık bölge tahsis edileceği derleyiciye bildirilmemiştir.

```
int[] d1 = new int[3];
d1[0] = 5;
d1[1] = 6;
d1[2] = 7;
```

Yukarıdaki kod parçasında ise **int** tipindeki 3 elemanlı **d1** dizisinin ilk 3 indeksine sırasıyla *5*, *6* ve *7* sayıları yerleştirilmiştir. Bu işlem, aşağıdaki iki kutuda tek satırda yapılmış olup, ikisi de aynı anlam ifade etmektedir:

```
int[] d1 = new int[] {1, 2, 3};
```

```
int[] d1 = {1, 2, 3};
```

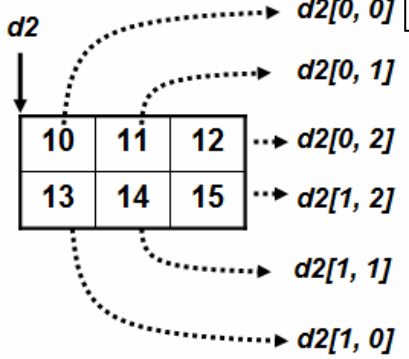
Çok Boyutlu Diziler

C# programlama dilinde çok boyulu diziler (maksimum 32 boyutlu) de tanımlanabilir. 2 ve 3 boyutlu diziler için örnek tanımlamalar aşağıda verilmiştir.

```
int[,] d2 = new int[2, 3]; // boyut: 2x3
d2[0, 0] = 10;
d2[0, 1] = 11;
d2[0, 2] = 12;
d2[1, 0] = 13;
d2[1, 1] = 14;
d2[1, 2] = 15;
Console.WriteLine(d2[0, 2]);
Console.WriteLine(d2[1, 1]);
```

Çıktı:

```
12
14
```

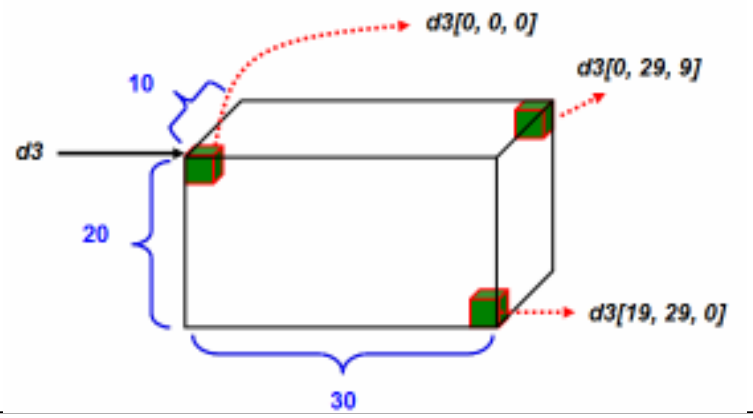


Yukarıda, 2 boyutlu bir **d2** dizisi tanımlanmıştır (2x3 boyutlarında) ve indeks değerleri atama yolu ile belirlenmiştir. **d2** dizisinin tanımlama ve atama işlemleri tek bir satırda da yapılabilir:

```
int[,] d2 = new int[,] {{10, 11, 12}, {13, 14, 15}};
```

3 boyutlu bir dizinin tanımlanması da benzer mantıktan hareketle yapılabilir:

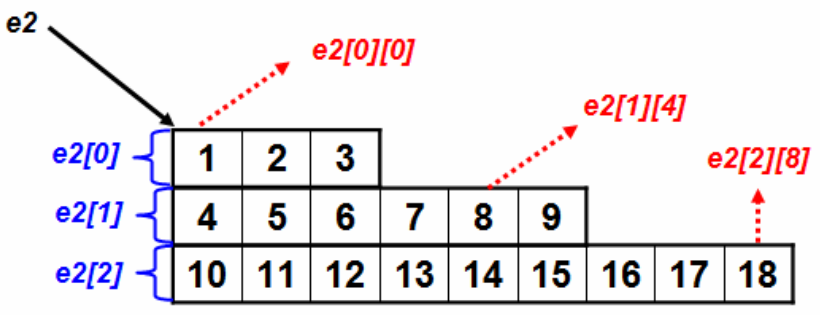
```
int[,,] d3 = new int[20, 30, 10]; // boyut: 20x30x10
```



İç İçe Diziler

Çok boyutlu diziler, dikdörtgenseldir. İç içe diziler ise, her bir indisinde herhangi bir boyutta dizi bulunabilen diziler olarak tanımlanabilirler. İç içe diziler için dikdörtgensel bir yapı zorunlu değildir.

```
int[][] e2 = new int[3][];
e2[0] = new int[] { 1, 2, 3 }; // ilkleme
e2[1] = new int[] { 4, 5, 6, 7, 8, 9 }; // ilkleme
e2[2] = new int[] { 10, 11, 12, 13, 14, 15, 16, 17, 18 }; // ilkleme
Console.WriteLine(e2[1][4]); // deger : 8
Console.WriteLine(e2[0][5]); // hatali: olmayan indeks
```



Örnek Sınıf Yapısı

```
using System;

namespace cs_giris
{
    public class Snf
    {
        public int x;
        public int y;

        public Snf()
        { // yapici metod
            x = 0;
            y = 0;
            Console.WriteLine("Varsayilan yapici metod");
        }
        public Snf(int a)
        { // yapici metod
            x = a;
            y = a;
            Console.WriteLine("Tek argumanli yapici metod");
        }
        public Snf(int a, int b)
        { // yapici metod
            x = a;
            y = b;
            Console.WriteLine("Iki argumanli yapici metod");
        }
        ~Snf()
        { // yikici metod : cop toplayici tarafindan cagirilir
            Console.WriteLine("Yikici metod");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Snf sn1 = new Snf(4); // sn1 adinda Snf nesnesi, tek arg. yapici metod
            Console.WriteLine("sn1.x : {0}, sn1.y : {1}", sn1.x, sn1.y);
            Snf sn2 = new Snf(7, 9); // sn2 adinda Snf nesnesi, iki arg. yapici metod
            Console.WriteLine("sn2.x : {0}, sn2.y : {1}", sn2.x, sn2.y);
            sn1 = null; // sn1 referansinin bagli oldugu nesne kullanilmiyor, cop
                                                                    olarak toplanacak.
            System.Console.ReadKey();
        }
    }
}
```

"**Nesne**" kavramı, "**sınıf**" yapısı ile hayata geçirilir. Sınıflar, içlerinde değişken ve fonksiyon tipinde öğeler, yapıcı ve yıkıcı metodlar başta olmak üzere pek çok üye barındırabilirler. Yukarıda verilen programda, **cs_giris** isim uzayı içerisinde **Snf** ve **Program** isimlerinde iki farklı sınıf yer almaktadır. **Snf** sınıfında üye olarak tamsayı tipinde **x** ve **y** değişkenleri, varsayılan yapıcı metod (fonksiyon), tek argüman alan yapıcı

metod, iki argüman alan yapıcı metod ve yıkıcı metod yer alırken, **Program** sınıfında sadece **Main** fonksiyonu üye olarak bulunmaktadır. Sınıf ve üye isimlerinin önlerinde bulunan "**public**", "**static**" gibi deyimler birer erişim belirleyicisi olup, bunlar ileriki konularda incelenecektir.

Sınıflar, genellikle (her zaman değil) nesneleri yaratılmak üzere tanımlanırlar. Örneğin yukarıda verilen programda **Program** sınıfındaki **Main** fonksiyonu -ki programın giriş kapısını oluşturur- içerisinde "**new**" operatörü yardımı ile **Snf** sınıfının **sn1** ve **sn2** isminde iki farklı nesnesi yaratılmıştır. Bu nesnelerden her biri, içlerinde **Snf** sınıfı içerisinde tanımlanan değişken üyeleri (**x** ile **y**) birer alt alan (*subfield*) olarak bulundurabildikleri gibi, **Snf** sınıfında yer alan yapıcı metodlardan da yararlanabilirler (**Snf** içerisinde fonksiyonlar da tanımlanmış olsa idi, **sn1** ve **sn2** nesneleri bu fonksiyonlardan da yararlanabileceklerdi.). Bu durumu "patates baskısı" na benzetebiliriz. **Snf** sınıfı oluşturulduğunda baskıda kullanılacak olan patatesin hazırlandığını ve boyandığını, **sn1** ve **sn2** nesnelerinin oluşturulmasıyla da boş kâğıt üzerine patates ile baskı yapıldığını düşünebiliriz.

Yapıcı metodlar, bir sınıfın nesnesinin oluşturulduğu anda otomatik olarak çağrılan ve nesnenin alt alanlarını ilkleyen fonksiyonlardır. **Snf** sınıfında 3 farklı yapıcı metod tanımlanmıştır. Bunlardan birincisi argüman almazken ikincisi tek bir tamsayı argüman almakta, üçüncüsü ise iki tamsayı argüman almaktadır.

```
Snf nesne1 = new Snf();
```

Yukarıdaki kod parçası (örnek programdaki **Main** fonksiyonunda yer aldığı düşünülürse), **Snf** sınıfının **nesne1** adında bir nesnesini yaratacak ve yaratma esnasında da **Snf** sınıfının argüman almayan yapıcı metodunu (varsayılan yapıcı metodunu) çağıracaktır. Dolayısı ile bu nesnenin bünyesinde bulunduracağı **x** ve **y** alt alanlarının değerleri *0* olacaktır.

```
Snf nesne2 = new Snf(25);
```

Yukarıdaki kod parçası ise **Snf** sınıfının **nesne2** adında bir nesnesini yaratacak ve yaratma esnasında da **Snf** sınıfının tek argüman alan yapıcı metodunu çağıracaktır. Dolayısı ile bu nesnenin bünyesinde bulunduracağı **x** ve **y** alt alanlarının değerleri *25* olacaktır.

```
Snf nesne3 = new Snf(60, 35);
```

Yukarıdaki kod parçası ise **Snf** sınıfının **nesne3** adında bir nesnesini yaratacak ve yaratma esnasında da **Snf** sınıfının iki argüman alan yapıcı metodunu çağıracaktır. Dolayısı ile bu nesnenin bünyesinde bulunduracağı **x** ve **y** alt alanlarının değerleri sırasıyla *60* ve *55* olacaktır.

Sınıflarda bulunan yıkıcı metodlar ise, o sınıfa ait herhangi bir nesnenin artık kullanılamaz hale gelmesi (örneğin, nesne tutacağına **null** değerinin atanması) durumunda, o nesne yıkılırken çağrılan fonksiyonlardır. Ancak, kullanılamaz hale gelen bir nesnenin yıkılma (çöp olarak toplanma, geri döndürülme) işlemi, "**çöp toplayıcı (*garbage collector*)**" denen bir program sayesinde gerçekleştiği için, yıkıcı metodun hangi anda çağrılacağını programcının kestirmesi mümkün olmayabilir. Örnek gösterim:

```
Snf nesne3 = new Snf(60, 35);
```


```
nesne3 = null;
```

"**Sınıf**" kavramı, ilerleyen konularda ayrıntılı olarak ele alınacaktır.