

VERİ TIPLERİ

1. Temel Veri Tipleri
2. Referans Veri Tipleri

Temel veri tiplerini öğrenmiştik; dolayısıyla referans veri tipleri ile devam edelim.

REFERANS VERİ TIPLERİ

1. Nesneler
 - Kütüphanede tanımlı nesneler
 - Vector (vektör)
 - String (karakter serisi)
 - vb.
 - Kullanıcı tarafından tanımlanan nesneler
2. Diziler

NESNELER

Java nesne yönelimli bir dil olduğu için, Java’da nesneler modellenir.

Nesnelerden bazıları, Java Development Kit (JDK)’e eşlik eden kütüphanelerde tanımlı durumdadır.

örnek:

String nesneleri

Vector nesneleri

Hashtable nesneleri

Java programcıları kendi nesnelerini de yaratabilirler:

örnek:

Ogrenci nesnesi

Ders nesnesi

BankaHesabi nesnesi

Referans veri tipi isimleri, Java sınıf isimleri olmalıdır.

Java sınıf isimlerinin iki bileşeni vardır :

1. Paket (package) isimleri: Kural gereği tüm karakterleri küçük olmalıdır.
2. Sınıf (class) isimleri: İlk karakterleri büyük olmalıdır.

örnek:

java.lang.Object

java.awt.Button



paket
ismi

sınıf
ismi



PAKETLER

Paketler, ilişkili sınıfların oluşturduğu gruplardır. Çoğu Java paketi “java.” ile başlar. Çekirdek Java sınıfları “java.lang” paketinde yer alır. Bu sınıflar genellikle sadece isimleri ile kullanılır.

örnek:

```
java.lang.Object myObject;  
    yerine,  
Object myObject;
```

NESNE YARATMAK

Değişken Tanımlama:

Nesnelere, değişkenler tarafından referans verilir. Değişkenler belirli bir tipte tanımlanmalıdır. Tipi tanımlarken, sınıfın ismi kullanılır (gerekli durumlarda paket ismi ile birlikte). Değişkenler, kendilerine bir değer atanmadan önce null'dır.

örnek:

```
Object birObject;           // bir Object nesnesi referansı  
String ogrenciAdi;          // bir String nesnesi referansı  
Java.util.Date bugun;       // bir Date nesnesi referansı
```

Yapılandırıcılar:

Bir değişkeni nesne tipinde tanımlamakla, o değişken kullanılır hale getirilmiş olmaz. Bir başka deyişle, bir nesneye referans vermeyen bir değişken ile bir mesaj göndermeye çalışırsak sorunla karşılaşırız. Bu yüzden nesneler, bir “yapılandırıcı (constructor)” çağrılarak yaratılır. Yapılandırıcılar, nesneleri yaratan metotlardır. Ait oldukları sınıf ile aynı ismi taşırlar. Tanımlanırken parametre de kullanabilirler. “new” anahtar kelimesi ile birlikte kullanılırlar.

örnek:

```
new java.lang.Object()           // bir Object nesnesi
new Object()                     // başka bir Object nesnesi
new Java.util.Date()             // bir Date nesnesi
```

Bir yapılandırıcının geri döndürdüğü değer, yeni bir nesnedir; yani ait olduğu sınıfa ait bir örnektir. Bu değere bir değişken tarafından referans verilmelidir. Değişkenler daha önceden tanımlanmış olabilir ya da yapılandırıcının çağrıldığı ifadede aynı zamanda değişken de tanımlanabilir.

örnek:

ayrı ifadelerde tanımlama

```
Java.util.Date bugununTarihi;
bugununTarihi = new java.util.Date();    // bir Date nesnesi
```

tek bir ifadede tanımlama

```
Java.util.Date bugununTarihi = new java.util.Date();
// bir Date nesnesi
```

Sabit değerler (literals), değeri değişmeyen değerlerdir. Nesneler tanımlanırken, referansları “null” olarak atanabilir:

örnek:

```
String selam = null;
```

Karakter dizesi nesneleri, sabit değerli olarak yaratılabilir:

örnek:

```
String selam = "Merhaba";
```

Ya da, tanımlandıktan sonra kendilerine sabit değer atanabilir:

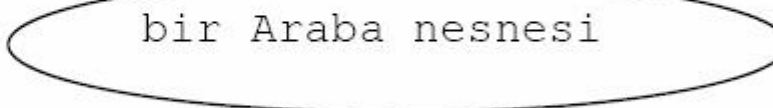
örnek:

```
String selam = null;
selam = "Merhaba";
```

Bir değişken, bir nesneye referans olabilir; yani, o nesneye işaret eder. Nesnenin fiziksel adresine erişilemez ve nesnenin fiziksel adresi yönetilemez.

örnek:

```
Araba benimki = new Araba ();           // bir Araba nesnesi
```

benimki → 

Buradaki durumda, “benimki” değişkeni bir işaretçi görevini görür. Görüldüğü gibi, Java’da da işaretçiler (pointers) bulunur; ancak C ve C++’ın tersine Java’da, programcılar işaretçilere erişemez.

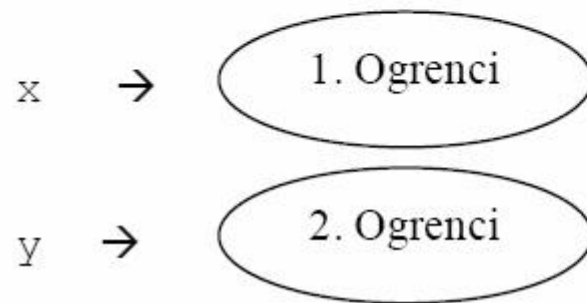
NESNE ATAMA

Nesne atama, nesneler için bir bağlantı işlemidir. Bir nesnenin bir örneğini aynı nesnenin başka bir örneğine bağlayarak, nesne atama gerçekleştirilir.

örnek:

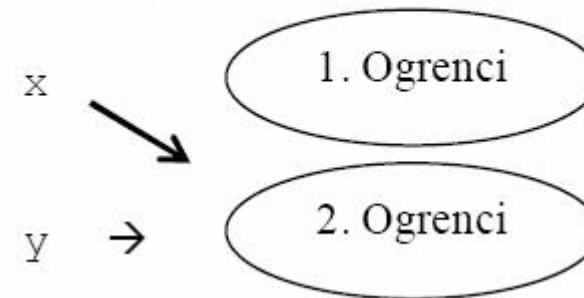
ilk durum

```
Ogrenci x = new Ogrenci(); // bir Ogrenci nesnesi
Ogrenci y = new Ogrenci(); // bir baska Ogrenci nesnesi
```



son durum

```
x = y;
```



Bu örnekteki son durumda, 2. Ogrenci nesnesinin BİR KOPYASI OLUŞTURULMAZ; a değişkeni, b değişkeninin referans verdiği nesne ile aynı nesneye referans verir. 1. Ogrenci nesnesi ise boşta kalır; kendisine referans veren bir değişken bulunmamaktadır.

== ve != karşılaştırma operatörleri, nesneler için kullanılabilir. Karşılaştırılan, nesneler değil nesnelerin referanslarıdır.

Üstteki örnek ele alındığında;

```
x == y; // sonuç: false (ilk durum için)
x == y; // sonuç: true (son durum için)
```

ÇÖP TOPLAMA

Kendisine artık referans verilemeyen nesneler, çöp toplayıcısı için uygun olan nesnelerdir. Yine yukarıdaki örnek ele alındığında, 1. Öğrenci çöp durumuna gelmiştir ve çöp toplayıcısı tarafından yok edilir.

Çöp toplama işlemi, JVM'in hafızaya ihtiyacı olduğu zamanlarda otomatik olarak gerçekleşir. Ancak, aşağıdaki ifade kullanılarak istenildiğinde çöp toplayıcısı programcı tarafından çalıştırılabilir :

```
System.gc();
```

NESNE KULLANIMI

Nesneler,

- veriler (fields)
- metotlar (programlar/işlemler)

içerir.

Nesnelerin içerdığı verilere ve metotlara “ . ” (nokta) operatörü kullanılarak erişilir.

yapısı:

```
nesne.veriAdi
```

```
nesne.metotAdi(parametreler)
```

Bazı Java sınıfları, verilerine doğrudan erişime izin verir.

Bir nesneye mesaj göndermek, o nesnenin ait olduğu sınıfın içerdiği bir metodun çalıştırılması ile gerçekleşir. Bazı metotların parametreleri vardır.

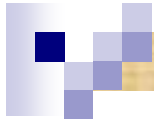
System.out, “System” sınıfının bir public verisidir; “java.io.PrintStream” paketinin bir örneğini tutar. Ekranı metin göndermek için kullanılır. En çok kullandığı metotlar:

- `print()` → devam eden satırda
- `println()` → yeni satırda

java.lang.String, bir karakter serisini temsil eder. Karakter serileri, yaratıldıktan sonra değiştirilemezler. Erişmek, arama yapmak ve karşılaştırma yapmak için metotları vardır. Dizinleri 0-tabanlıdır (sıfır ile başlar).

örnek:

```
new String()           // boş String
"Merhaba Dünya"       // sabit String
```

M	e	r	h	a	b	a		D	ü	n	y	a
0	1	2	3	4	5	6	7	8	9	10	11	12

```
String selam = "Merhaba Dünya"  
selam.length();           // 13  
selam.charAt(5);          // b  
selam.indexOf('a');       // 4  
selam.lastIndexOf('a');   // 12
```

STRING EKLEME

+ ve += operatörleri karakter dizileri için de geçerlidir. Karakter dizileri, hem temel veri tipleri ile hem de nesneler ile eklenebilir.

örnek:

```
System.out.println("Merhaba " + "Dünya");  
                        //"Merhaba Dünya"  
  
int x = 4;  
System.out.println("x " + x + "'e eşittir.");  
                        //"x 4'e eşittir."
```

String ekleme işlemi sonucunda YENİ bir String nesnesi yaratılır:



örnek:

```
String selam = "Sevgili";  
String isim = " Ayşe,";  
selam = selam + isim;  
System.out.println(selam);           // "Sevgili Ayşe,"
```

ilk durum:

selam → "Sevgili"

isim → " Ayşe,"

son durum:

selam → "Sevgili Ayşe,"

DIZILER

Diziler; homojen, dizinli koleksiyonlardır. Hem temel veri tipleri hem de nesnelerden oluşabilir. Dizilerin kendileri de birer nesnedir. Diziler, daha ileri düzeydeki koleksiyonların temel taşlarıdır.

Özellikleri arasında aşağıdakiler sayılabilir:

- Sabit boyutludur.
- Sınırları kontrollüdür.
- Dizini 0-tabanlıdır.

DİZİ TANIMLAMAK

Diziler, belirli bir tipte tanımlanır. [] bloğunu takip eden tip ismi, bir dizi belirtir.

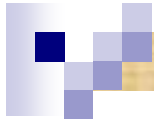
örnek:

```
int[] aylar;  
// integer tipinde bir dizinin elemanlarını tutmak için bir  
değişken tanımlar.
```

Java ya da C tarzında tanımlama yapılabilir:

örnek:

```
int[] aylar;           // Java tarzı  
int aylar[];          // C tarzı
```



Dizi tanımlamakla, dizi yaratılmış olmaz. Dizi oluşturma adımları:

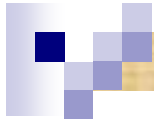
1. Diziye referans verecek olan değişkeni tanımlamak
2. Dizi yaratmak
3. Dizinin elemanlarını yerleştirmek

DİZİ YARATMAK

Köşeli parantez [] içine dizinin boyutu yazılır. Dizi hücreleri, dizi oluşturulurken başlangıç değeri atma işlemi yapılmazsa varsayılan değerler (null) içerir.

örnek:

```
String[] gunler;  
gunler = new String[7];  
// dizi elemanları = {null, null, null, null, null, null, null}  
  
int[] aylar;  
aylar = new String[12];  
  
// dizi elemanları = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Tek bir ifadede dizi tanımlanabilir ve yaratılabilir:

örnek:

```
String[] gunler = new String[7];  
int[] aylar = new String[12];
```

BAŞLANGIÇ DEĞERİ ATAMAK

Bir dizideki her hücre için teker teker başlangıç değeri atanabilir:

örnek:

```
String[] gunler = new String[7];  
    // dizi elemanlar = {null, null, ...}  
gunler[0] = "Pazar";  
    // dizi elemanları = {Pazar, null, ...}  
gunler [1] = "Pazartesi";  
    // dizi elemanları = {Pazar, Pazartesi, null, ...}
```

Tek bir adımda bir dizi yaratılabilir ve her hücresi için başlangıç değerleri atanabilir.

örnek:

```
String[] gunler = {Pazar, Pazartesi, Salı, ... };
```

Başlangıç değerleri, sadece değişken tanımlıyken geçerlidir.

örnek:

```
String[] gunler;  
gunler = {Pazar, Pazartesi, Salı, ... };           // hata verir
```

DİZİLERE ERİŞİM

Dizi hücreleri, 0-tabanlı dizinle belirtilir. Dizi hücrelerindeki değerler, değer atama kullanılarak güncellenir.

örnek:

```
int[] aylar = new int[12];  
                                // dizi elemanları = {0, 0, 0, ...}  
aylar[0] = 1;                  // dizi elemanları = {1, 0, 0, ...}  
aylar[1] = 2;                  // dizi elemanları = {1, 2, 0, ...}  
System.out.println(aylar[0]);
```

VERİ UZUNLUĞU (LENGTH)

Diziler, uzunluk verisine sahiptir.

örnek:

```
String[] gunler = new String[7];  
System.out.println(gunler.length);           // sonuç = 7
```

ÇOK BOYUTLU DİZİLER

Çok boyutlu bir dizi, dizilerin dizisi olarak düşünülebilir. Çok sayıda alt dizini vardır.

örnek:

```
String[] gunler = new String[7][2];
gunler[0][0] = "P";
gunler[0][1] = "Pazar";
...
gunler[6][0] = "C";
gunler[6][1] = "Cumartesi";
```

Tek boyutlu diziler gibi, çok boyutlu diziler de başlangıç değeri atayıcıları kullanılarak yaratılabilirler:

örnek:

```
String[][] gunler = {{ "P", "Pazar"}, { "Pt", "Pazartesi"},
..., { "Ct", "Cumartesi" }};
```

İFADELER VE DEYİMLER

Ifade; Java dilinin sözdiziminde yer alan değişkenler, sabit değerler, operatörler gibi birimlerin her hangi bir kombinasyonla bir araya gelmesiyle oluşan yapılardır. İfadelerin derlenebilir ve dolayısıyla çalıştırılabilir duruma gelmeleri için sonlarına ";" işaretinin konması gerekir. Bu şekilde, deyimler oluşturulur.

Deyimler, bir JAVA programını oluşturan bloklardır. ";" karakteri ile sonlandırılırlar.

Birçok şey deyim olabilir:

örnek:

```

; // boş deyim
selaml = "Merhaba"; // atama
x++; // arttırma
--x; // azaltma
System.out.println(selaml); // metot çağırma
if(x < 3) ... ; // if deyimi
switch(x) ... ; // switch deyimi
for(int i = 0; i < 10; i++) ... ; // for deyimi
while(x < 10) ... ; // while deyimi

```

İfadeler ve deyimler sadece Java dilinde değil, diğer programlama dillerinde de aynı şekilde yer alır.

BİLEŞİK DEYİMLER

Birden fazla deyim, kıvrımlı parantezler {} içine alınarak bir blok halinde gruplanabilir. Böyle bir blok içinde değişken tanımlı yapılabilir. Blok sonlarında, sonlandırıcı bir “;” işareti kullanılmaz.

örnek:

```
{  
    String selam1 = "Merhaba ";  
    String selam2 = "Dünya";  
    System.out.print(selam1);  
    System.out.println(selam2);  
}
```

FAALİYET ALANI VE ÖMÜR

Faaliyet alanı; değişkenlere, kendilerinin tanımlandığı isimlerle ulaşılabilen kodun yer aldığı bölgedir. Ömür, değişkenlerin bellekte yaratılmalarından itibaren yok edilmelerine kadar geçen süredir.

Bir blok içinde tanımlanan değişkenler sadece o blok içinde geçerlidir; blok dışında varlıklarını yitirirler. Değişkenlerin faaliyet alanı, geçerli oldukları bölgedir; dolayısıyla, blok içinde tanımlanan değişkenlerin faaliyet alanı, tanımlandıkları bloğun içidir.

Varlığını yitiren yani artık kullanılmayan, silinen bir değişkene erişilemez. Referans verdikleri nesneler de çöp durumuna gelir. Bu durumda, o değişkenin ömrünün sona erdiği söylenir.

örnek:

```

...
{
    String selam1 = "Merhaba ";
    String selam2 = "Dünya";
    System.out.print(selam1);    // derlenir
    System.out.println(selam2);  // derlenir
}
System.out.println(selam1);    // derleme hatası verir
...

```

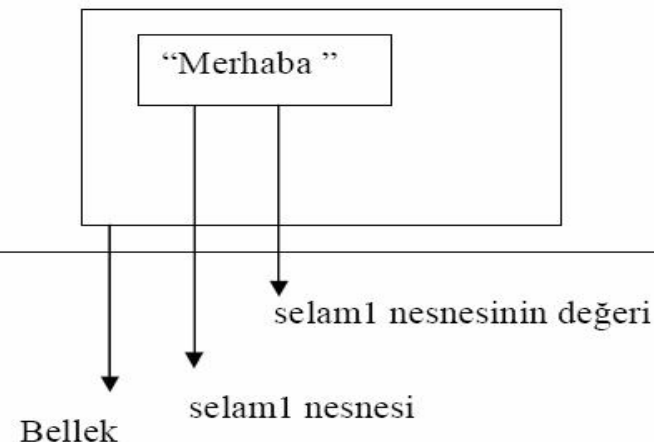
Bu örnekte, blok içinde tanımlanan “selam1” nesnesinin faaliyet alanı bloğun içidir. Blok dışına çıkıldığında bu nesnenin geçerliliği kaybolur ve bu nesne için yazılan komutlar da geçersizdir.

Blok içinde:

```

String selam1;
selam1 = "Merhaba ";

```



Blok içindeyken, “selaml” nesnesi tanımlıdır. Bir nesne tanımlandığında, bellkete o nesne için bir yer ayrılır ve nesnenin değeri o yerde saklanır. Bu değere, sadece o nesnenin faaliyet alanı içindeki kodlar tarafından erişilebilir. Dolayısıyla blok dışına çıkıldığında “selaml” nesnesi faaliyet alanı içinden de çıkıldığı için blok dışındaki kodlar tarafından bu nesnenin değerine erişilemez.

Bu açıklama doğrultusunda nesnenin ömrü hakkında da şu söylenebilir: Blok dışında “selaml” nesnesi çöp durumuna geldiği için çöp toplayıcısı tarafından silinir. Bu durumda, ömrü de sona erer.

if DEYİMİ

if deyiminin iki formatı vardır:

1. if (eğer)
2. if-else (eğer-değilse)

if yapısı:

```
if (booleanİfade) {
    deyim
}
```

if-else yapısı:

```
if (booleanİfade) {
    deyim
}
else{
    deyim
}
```

if anahtar kelimesinden sonra parantezler yazılır; parantezlerin arasında, sonucu boolean olan bir ifade bulunur. Parantezlerden sonra yazılan deyim, bir tane olabilir ya da bir deyim bloğu olabilir. Yalnızca bir deyimin kullanıldığı durumlarda kıvrımlı parantezlerin {} kullanılmasına gerek yoktur.

örnek

```
int x = 3;
if(x < 4)
    System.out.println("x = " + x);    // x = 3
```

```
int x = 3;
if(x > 4)
    System.out.println("x = " + x);
    // çıktı olarak hiç bir şey gösterilmez
```

```
int x = 3;
if(x > 4)
    System.out.println("x 4'ten büyüktür");
else
    System.out.println("x 4'ten küçüktür");
    // x 4'ten küçüktür
```


switch DEYİMİ

yapısı:

```

switch (İfade) {
    case sabitDeğer:
        deyim/deyimler
    case sabitDeğer:
        deyim/deyimler
    default:
        deyim/deyimler
}

```

switch deyiminde, kodlar belirtilen şarta bağlı olarak etiketli (labelled) deyimlere dallanır. Belirtilen şart bir ifadedir ve bu ifade sadece byte, char, short ya da int türünde olabilir. Dallanma; her durum (case) için belirtilen sabit değerlere göre gerçekleşir. Sabit değerler de sadece byte, char, short ya da int türünde olabilir. Belirtilen sabit değerler dışındaki durumlar için, varsayılan (default) bir deyime dallanma gerçekleşir.

örnek:

```

// "deste" isimli bir nesne tanımlandığı ve bu nesnenin ait
olduğu sınıfta "value" adında bir metot tanımlandığı
varsayılıyor. "value" metodu, "deste" nesnesinin elemanları
arasından birini rasgele seçerek değerini alıyor.

```

```

// "valueOf" metodu, String sınıfına ait bir metotdur;
sayıları karakter dizesine (string) çevirir.

```

```
int[] deste = {1, 2, 3, ..., 10, 11, 12, 13};
String kart;
switch(deste.value()){
    case 11:
        kart = "Joker";
    case 12:
        kart = "Queen";
    case 13:
        kart = "King";
    case 1:
        kart = "A";
    default:
        kart = String.valueOf(deste.value());
}
```

break DEYİMİ

Bir blok içinde kullanılan break deyimi, bloktan çıkmak için kullanılır. switch deyimi içinde kullanılan break deyimi, yazıldığı yerden önceki durum geçerli olduğunda daha sonraki durumları kontrol etme gereği duyulmadan switch deyiminden hemen çıkmak için kullanılır.

örnek:

```
int[] deste = {1, 2, 3, ..., 10, 11, 12, 13};
String kart;
switch(deste.value()) {
    case 11:
        kart = "Joker";
        break;
    case 12:
        kart = "Queen";
        break;
    case 13:
        kart = "King";
        break;
    case 1:
        kart = "A";
        break;
    default:
        kart = String.valueOf(deste.value());
}
```

Bu örnekte; eğer desteden seçilen kartın değeri 11 ise, programın diğer durumları kontrol etme gereği duymadan kart değişkeninin değerini "Joker" olarak atayarak bloktan çıkma işlemini gerçekleştirir.

Birden fazla durum için aynı işlemin yapılması istenirse, durumlar teker teker yazıldıktan sonra yapılması istenen işlem bir kere yazılır.

örnek:

```
int[] deste = {1, 2, 3, ..., 10, 11, 12, 13};
String kartTipi;
switch(deste.value()){
    case 11:
    case 12:
    case 13:
        kartTipi = "Resimli Kart";
        break;
    default:
        kartTipi = "Diğer Kart";
}
```

Bu örnekte, desteden seçilen kartın değerinin 11, 12 ya da 13 olduğu durumların her biri için yapılması istenen işlem aynıdır.

for DEYİMİ

for deyimi üç bölümden oluşur:

1. init (başlangıç): Döngü başlamadan önce bir kere yürütülür.
2. test (kontrol): Her döngüden bir kere yürütülür. Eğer değeri true ise, döngünün gövdesi (body) yürütülür.
3. update (güncelleme): Her döngünün sonunda yürütülür.

yapısı:

```
for(init; test; update){
    deyim
}
```


örnek:

```

int toplam = 0;
for(int sayi = 1; sayi <= 10; sayi++){
    toplam += sayi;
}
System.out.println(toplam);                // sonuç = 55

```

init ve update bölümleri birden fazla değer içerebilir.

örnek:

```

for(int toplam = 0, sayi = 1; sayi <= 10; sayi++){
    toplam += sayi;
}

```

Bölümlerden herhangi biri boş bırakılabilir. Ancak, eğer test bölümü boş bırakılırsa sonsuz döngü yaratılmış olur.

örnek:

```

for(int sayi = 1; ; sayi++){
    System.out.println(toplam);
}                // sonuç = sonsuz döngü

```

```

int i = 0;
for(; ;){
    System.out.println(i++);
}                // sonuç = sonsuz döngü

```

for deyimi, bir dizinin elemanları ile sık sık kullanılır. Test bölümünde dizinin uzunluğuna göre kontrol yapılır.

örnek:

```
int[] aylar = {1,2,3,4,5,6,7,8,9,10,11,12};
for(int i = 0; i < aylar.length; i++){
    System.out.println(aylar[i]);
}
```

// sonuç =
1
2
3
...

Bu örnekte, “aylar” dizisinin uzunluğunu bulmak için, Array (dizi) sınıfına ait olan “length (uzunluk)” metodu kullanılmıştır.

while DEYİMİ

Sonlandırma testi, döngüden çıkılmasını gerektiren şart ifadesi döngünün başında yer alır. Sonlandırma testine göre döngü hiç yürütülmez ya da bir veya birden fazla kere yürütülür. while deyimi için, for deyiminin init ve update bölümleri olmadan sadece test bölümü ile kullanımı denebilir.

yapısı:

```
while (booleanİfade) {
    deyim
}
```

örnek:

```

int toplam = 0, i =1;
while(i < 11){
    toplam += i;
    i++;
}
// i = 11

```

do DEYİMİ

Sonlandırma testi, döngünün sonunda yer alır. Deyim en az bir kere yürütülür.

yapısı:

```

do{
    deyim/deyimler
}while (booleanİfade);

```

örnek:

```

int toplam = 0, i =1;
do{
    toplam += i;
    i++;
}while(i < 10);
// i = 10

```



ETİKETLİ (LABELLED) DEYİMLER

Java'da herhangi bir deyimin bir etiketi (label) olabilir. Etiketler, geçerli Java belirleyicisi olmalıdır.

örnek:

dis:

```
for (i = 0; i < tablo.rowSize(); i++) {  
    ic:  
    for (j = 0; j < tablo.columnSize(); j++) {  
        deyimler  
    }  
}
```

Bu örnekte kullanılan “dis:” ve “ic:” birer etiketli deyimdir.

Etiketler, “break” ve “continue” deyimleri ile birlikte kullanılırlar; “goto” kullanılmaz!

break DEYİMİ

break deyimi, akış kontrolünü içinde bulunduğu etiketli deyimden, switch deyiminden ya da en içteki yinleme (iteration) deyiminden sonraki deyime geçirir.

örnek:

dis:

```
for (i = 0; i < tablo.rowSize(); i++){
    ic:
    for (j = 0; j < tablo.columnSize(); j++){
        if(tablo.at(i,j) == 0) break;
        if(tablo.at(i,j) < 0) break dis;
        ...
    }
    System.out.println("Döndürülen satır: " + i);
}
System.out.println("Tablo döndürüldü");
```

Bu örnekte, “break” deyimi, içinde bulunduğu “ic” etiketli deyiminden çıkılması için; “break dis” deyimi, “dis” etiketli deyiminden çıkılması için kullanılmıştır.

continue DEYİMİ

continue deyimi, sadece bir yineleme deyiminin içinde yer alabilir. Akış kontrolünü, içinde bulunduğu döngünün ya da etiketli deyimin en başına geçirir.

örnek:

dis:

```
for (i = 0; i < tablo.rowSize(); i++){
    ic:
    for (j = 0; j < tablo.columnSize(); j++){
        if(tablo.at(i,j) == 0) continue;
        if(tablo.at(i,j) < 0) continue dis;
        ...
    }
    System.out.println("Döndürülen satır: " + i);
}
System.out.println("Tablo döndürüldü");
```

Bu örnekte, “continue” deyimi, içinde bulunduğu “ic” etiketli deyimin başına dönülmesi için; “continue dis” deyimi, “dis” etiketli başına dönülmesi için kullanılmıştır.

13/11/2009 devam

Dokümantasyon & Açıklamalar

■ Üç türlü yöntem vardır:

// Bu işaret satır sonuna kadar olan tüm ifadeyi açıklama olarak belirler.

/* ve */ işaretleri arasında kalan tüm ifadeler satır sonu gözetmeksizin açıklama olarak kabul edilir.

/**

* Bu sentaks biçimi Javadoc açıklamaları için kullanılır,

* HTML biçimlendirme özelliklerini sunar.

*/



Bilginizi sınavın

- İşte problem:

```
int iVar = 10;  
float fVar = 23.26f;
```

// derleme zamanı hatası verir

```
iVar = iVar * fVar
```

- Hangi çözüm en iyi çalışır?

1

```
iVar = (int) (iVar * fVar)
```

232

2

```
iVar = (int) iVar * fVar
```

Aynı derleme hatası

3

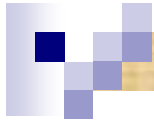
```
iVar = iVar * (int) fVar
```

230

4

```
iVar = (int)  
((float) iVar * fVar)
```

232



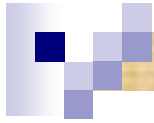
Bir Java sınıfının anatomisi

- Package
- Import(s)
- Comments(açıklamalar)
- Declaration(bildirim)
- Fields(alanlar)
- Constructors(yapıcılar)
- Methods(metotlar)



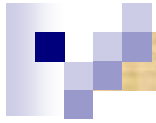
Metotlar

- ◆ Bir metot isimlendirilmiş bir dizi komutlardan oluşur ve bir nesnenin istediğiniz bir işlemi yapmak için gerçekleştirmesi gereken faaliyettir.
- ◆ Metotların bir imzası/*signature* vardır:Bu imza bir isim ve sıfır veya daha fazla sayıda parametreden oluşur
- ◆ Metotlar geri döndürdükleri bir veri yapısı tanımlarlar(primitive veya object veya void)



Metotlar

- ◆ Genelde mümkün olduğunca kısa tutulur
- ◆ Yapıcılar/Constructors – nesnenin ilk oluşum durumunda alacağı değerleri belirleyen metottur.
- ◆ Yapıcılar ve diğer metotlar aşırı yüklenebilir. *over-loaded*
- ◆ Yapıcılar ve diğer metotlar erişim özelliği belirtebilirler(accessibility modifiers)



Java Metodları

hem prosedürler hem de fonksiyonlar için tek bir yapı:

- fonksiyon tanımlaması yapıldığı zaman geri dönüş değerinin türü belirtilmesi gerekir.

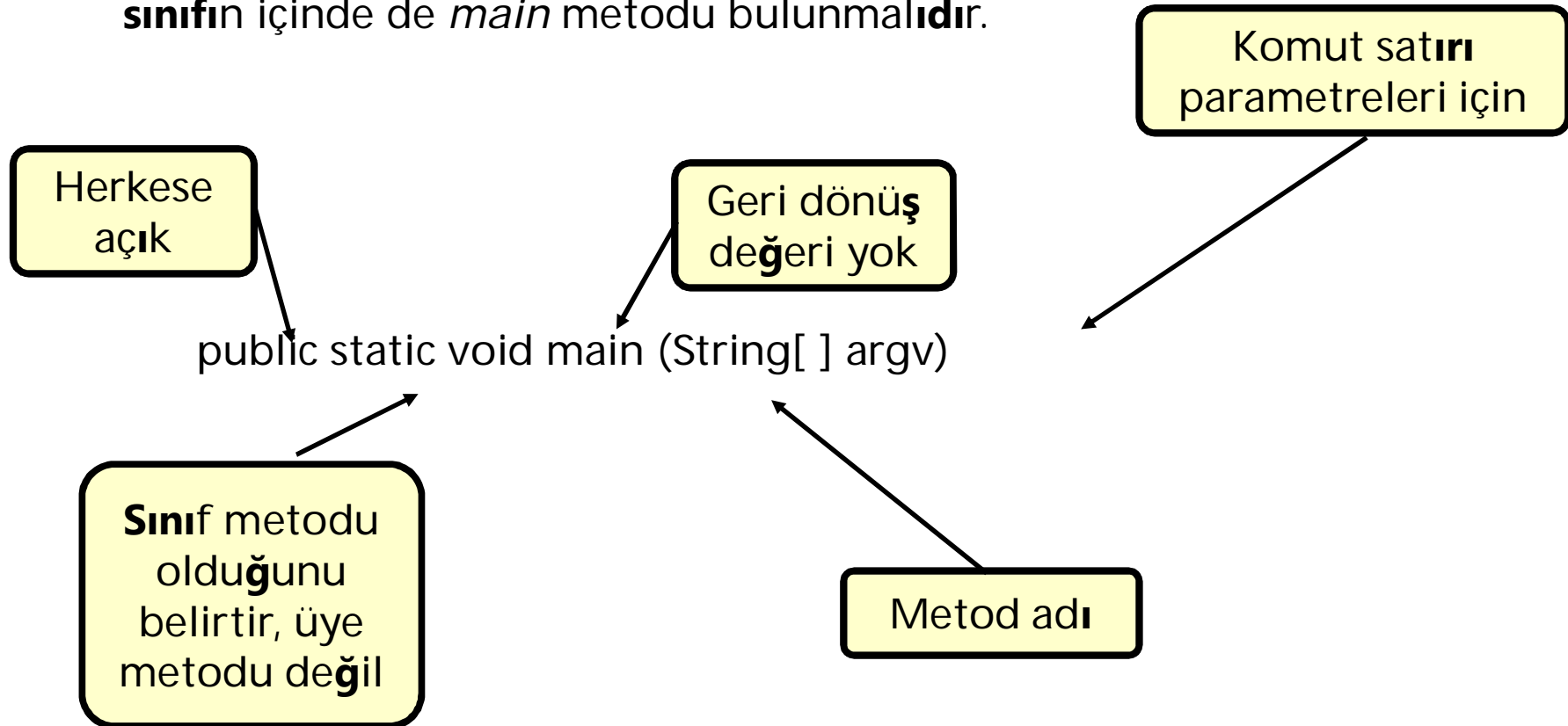
```
public float average (float num1, float num2, float num3)
{
    float answer;
    answer =
        (num1 + num2 + num3)/ 3;
    return (answer);
} // of average
```



Metod Yazımı

Java kuralı:

- Her metod bir nesneye aittir ya da bir sınıfa.
- Bir metod çağrıldığı zaman hangi sınıfa ya da nesneye ait olduğu kesin belli olmalıdır.
- Bir uygulamayı çalıştırmak için programın aynı adında bir sınıf ve bu sınıfın içinde de *main* metodu bulunmalıdır.





```
class A
{
    public static void main(...
}
```

```
class B
{
    public static void main(...
}
```

```
class C
{
    public static void main(...
}
```

Öyleyse, her sınıf kendi *main* metoduna sahip olabilir. Bunlardan hangisinin çalışacağına siz karar vereceksiniz.

Metod İmzaları

“Metod imzası, metod adı, parametre sayısı ve türleriden oluşur. Herhangi bir sınıf aynı imzaya sahip iki metod tanımlayamaz, yoksa derleme zamanı hatası oluşur.

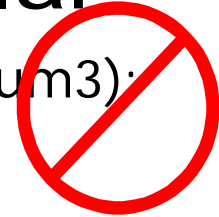
Aynı isme sahip metodlar farklı varyasyonlarda parametrelere sahip olurlarsa *Method overloading* meydana gelir.

```
public int getCube(int num){  
    return num*num*num;  
}  
  
public int getCube(float num){  
    return (int)(num*num*num);  
}  
  
public int getCube(double num){  
    return (int) (num*num*num);  
}
```




Metodlarda yapılan genel hatalar

```
public float average (float num1, float num2, float num3):  
{  
    float answer;  
    answer =  
        (num1 + num2 + num3)/ 3;  
    return (answer);  
} // average
```



‘;’ Noktalı virgül hakkında

- Eğer yukarıdaki gibi bir kullanım söz konusu olursa metod *abstract* metod gibi görünebilir.
- Çözümlemesi zor bir hata mesajı ile karşılaşılabilir.
- Çok kolay yapılan hatalardan biridir.



Veri ve metodlar birlikte bir sınıfa aittir.

Şu anda sadece, değişkenlerin ve metodların sınıflara ait olduğunu bilmeniz yeterli.

Daha sonra, bu özelliği kullanarak durumları (değişken), davranışlarla (behaviour) nasıl sarmalanır göreceğiz...



Yapıcılar

- Yapıcılar sınıfa ait yeni nesneler sınıf tanımındaki kalıp kullanılarak üretilirken başvurulan mekanizmalardır.
 - Yapıcının amacı yeni nesneye ilk değerlerini atamaktır.
 - Yapıcılar metotlara benzer ancak
 - Her zaman isimleri sınıf ismiyle aynı olmak zorundadır
 - Hiçbir zaman dışarıya bir değer döndürmezler



Yapıcılar

- Bir yapıcı metot çağırmak için new işlemi kullanılır ve gerekli parametreler verilir.
- Her sınıf varsayılan bir yapıcıya sahiptir:

```
public ClassName() {}
```

bu yapıcının parametresi yoktur ve herhangi bir ilk değer atamaz.



Yapıcılar

- Eğer siz bir tane tanımlamazsanız, Java sizin için parametresi olmayan varsayılan bir yapıcıyı otomatik olarak oluşturur.
- Eğer siz parametrelili veya parametresiz bir yapıcı metot yazarsanız, Java sizin ne yaptığının farkında olduğunuzu varsayarak herhangi bir yapıcıyı tanımlamaz.



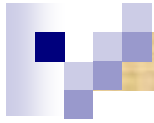
Yapıcılar: Başucu kuralları

- ◆ Hatırlatma: Bir yapıcının amacı yeni üretilmiş bir nesneyi bilinen bir ilk duruma getirmektir.
- ◆ Yapıcılar çok fazla işlem yapmamalıdır
- ◆ Nesneye ilk değer atama işlemi ile nesnenin davranışlarını kesinlikle birbirinden ayırmak gerekir.



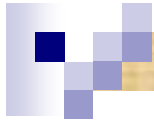
Nesneler

- “Bir nesne değişkenler ve ilişkili metotlar yazılım paketidir. ” - Java Tutorial
- Her şey ya temel bir veri yapısı ya da bir nesnedir.
- Nesneler:
 - Gerçek dünyadaki fiziksel elemanların modeli (Öğrenci gibi) veya
 - Soyut elemanlar olabilir (Dersler, Seçimler, Finansal İşlemler gibi)



Nesneler

- Java'da temel bir veri bildirimi yapılmasıyla bellekte o veri için uygun yer ayrılır
- Ancak bir nesne bildirimi yapılırken bellekte o nesneye (veya o nesneden türetilen bir tipe) erişimi sağlayacak *referansa* (*reference*) değişkeni için yer ayrılır.



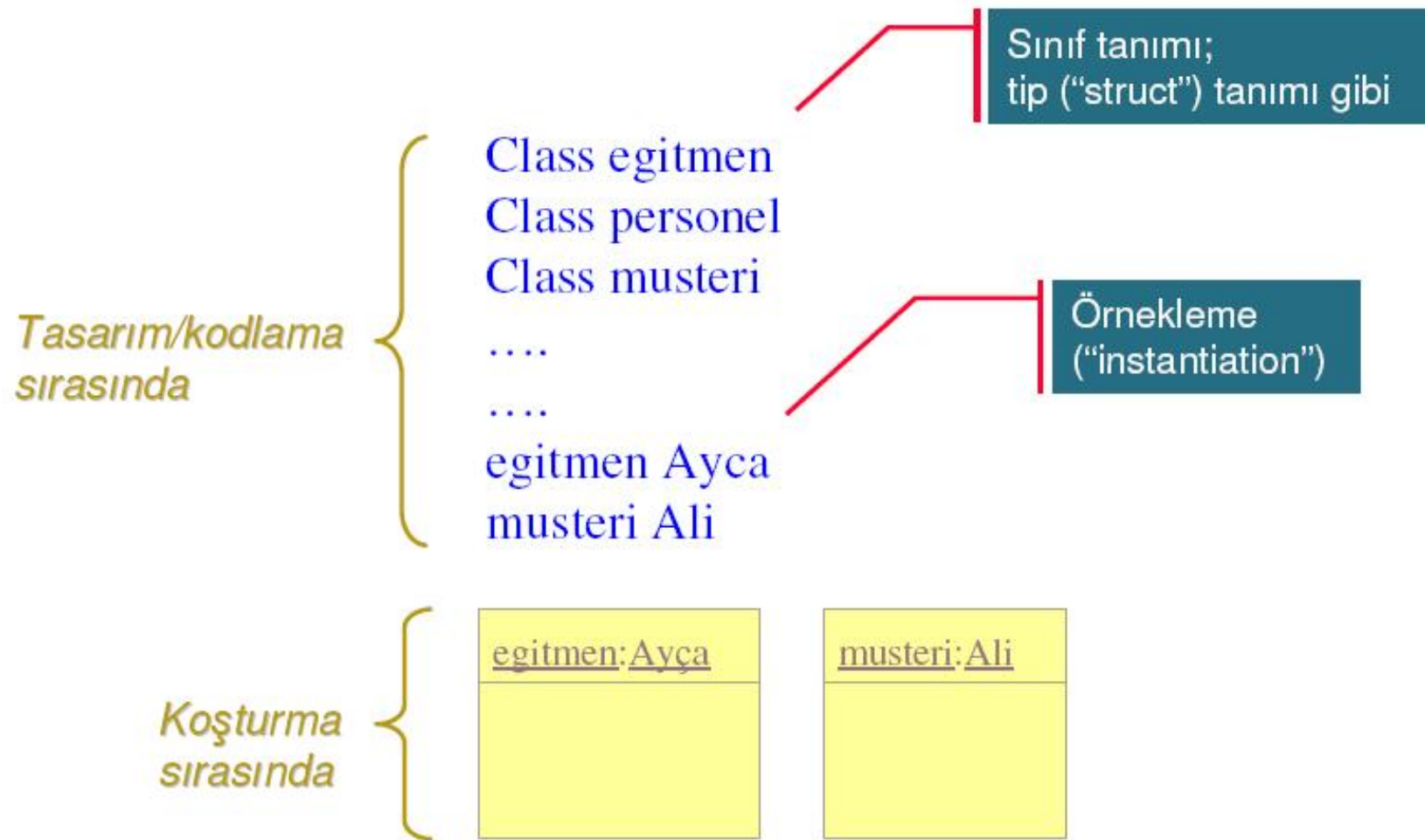
Nesneler

- Nesneyi tutacak değişken bildirimi yapmakla nesne üretilmiş olmaz.
- Nesne örneği için bellekte yer ayrılması dinamik olarak *new* operatorü kullanılarak gerçekleşir.
- Nesne tipleri istenilen şekilde oluşturulur.

“Sınıf” ve “Nesne”

- Her sınıfın sıfır veya daha fazla örneği vardır.
- Sınıf statik, nesne dinamiktir.
 - ▶ Sınıfın varlığı, semantiği ve ilişkileri program çalıştırılmadan önce sabit olarak belirlenmiştir.
 - ▶ Nesneler program çalıştırıldığında sınıf tanımından dinamik olarak yaratılırlar (“construction”).
 - ▶ Nesneler sorumluluklarını tamamladıklarında ortadan kaldırılırlar (“destruction”).
- Nesnenin sınıfı sabittir ve nesne bir kez yaratıldıktan sonra değiştirilemez.

“Sınıf” ve “Nesne”: Örnek



Temel Sınıf Tanımı

```
class sınıf_adi {  
    degisken_tipi d1;  
    degisken_tipi d2;  
    ...  
    donen_tip yordam_adi ( // parametre listesi  
                           parametre_tipi p1, parametre_tipi p2, ...) {  
        ...  
    }  
    donen_tip yordam_adi ( // parametre listesi  
                           parametre_tipi p1, parametre_tipi p2, ...) {  
        ...  
    }  
}
```

Temel Sınıf Tanımı: Örnek - 1

Kutu.java

```
1 class Kutu {
2     double genislik;
3     double yukseklik;
4     double derinlik;
5     public Kutu (double a, double b, double c) {
6         this.genislik = a;
7         this.yukseklik = b;
8         this.derinlik = c;
9     }
10    public double hacimHesapla () {
11        double h;
12        h = genislik * yukseklik * derinlik;
13        return h;
14    }
15 }
```

KutuDemo.java

```
1 class KutuDemo {
2     public static void main (String args[]) {
3         Kutu kutuX = new Kutu (10, 20, 15);
4         double hacim = kutuX.hacimHesapla ();
5         System.out.println ("Kutu Hacmi : " + hacim);
6     }
7 }
```

“Sınıf” Tipinde “Nesne” Oluşturmak - 1

■ *Sınıf-adi nesne-değişkeni = new Sınıf-adi ();*

↓
Tip

↓
Değişken

↓
Bellekte yer açar

↘
Oluşturucu (“constructor”)

■ Java’da nesneler sınıf tanımından oluşturulur.

► Nesne (“Object”): Sınıf Örneği (“Class Instance”)

“Sınıf” Tipinde “Nesne” Oluşturmak - 2

■ Örnek:

- ▶ `Kutu kutuX = new Kutu();`
 - ◆ “Kutu” bir tip olarak belirtildiğinden sınıf tanıımıdır.
 - ◆ “kutuX”, “Kutu” sınıfının bir örneğidir.
 - ◆ “kutuX” nesnesi, “Kutu” sınıfına aittir.

■ Sınıf tanımından nesne oluşturulurken 3 temel işlev gerçekleştirilir.

- ▶ Tanımlama (“declaration”)
- ▶ Örnekleme (“instantiation”)
- ▶ İklendirme (“initialization”)

“Sınıf” Tipinde “Nesne” Oluşturmak - 3

- Tanımlama: Nesne için değişken tanımlama
 - ▶ `Kutu kutuX;`
- Örnekleme: Sınıf tipinde bir nesne oluşturma
 - ▶ `kutuX = new Kutu ();`
- İklendirme: Nesnenin özelliklerine değer atama
 - ▶ `kutuX = new Kutu (10, 20, 15);`

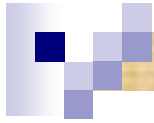
Hepsi birarada:

```
Kutu kutuX = new Kutu (10, 20, 15);
```

Kullanılmayan Nesnelerin Temizlenmesi

- JRE “çöp toplama” (“garbage collection”) özelliğiyle, kullanılmayan nesneler için periyodik olarak belleği boşaltır.
 - ▶ Çöp toplama, programdan gelen bellek atama taleplerine göre, bellekte yer olmadığı değerlendirilirse taleple senkronize olarak gerçekleştirilir.
 - ▶ ***System.gc()*** yordamı çağrılır.
- Sonlandırma: Bazen nesne yok edilmeden önce bazı işlemler yapması istenebilir. Bu işlemler ***finalize()*** yordamı içine yazılır ve nesne bellekten atılmadan hemen önce gerçekleştirilir.

```
protected void finalize () {  
    // sonlandırma kodu  
}
```

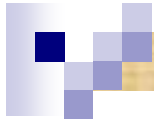
Değişken(Alanlar) / Variable (Fields)

■ Yerel Değişkenler/Local Variables

- Bir metot veya küçük bir blok kodu içerisinde bildirimi yapılan ve sadece o alanda kullanılan değişkenlerdir.

■ Örnek/nesne değişkeni/Instance variables

- İyi bir Java sınıfında *hemen hemen bütün* değişkenlerin örnek veya yerel değişken olması gerekir.
- Her nesne(instance) alanların kendine ait bir kopyasına sahiptir.



Değişken(Alanlar)

■ Sınıf değişkenleri/Class variables

- Bir alan adı veya değişkeni static anahtar kelimesiyle nitelenirse bu değişken bir sınıf değişkeni olur:
- Bu sınıftan kaç tane nesne üretilirse üretilsin bu tür değişkenlerden sadece bir tanesi için yer ayrılır.
- Bütün nesneler bu sınıf değişkenini paylaşırlar.



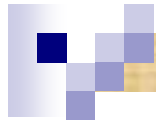
Sınıf Değişkenlerini Kullanma

- ◆ Java'da sınıf değişkeni bildirimi yapmak için iki iyi sebep vardır:
 - ***static final*** bir sabit olarak kullanmak
 - ***private static*** değişken olarak nesneler arasında bilgi paylaşımı sağlayan özel bilgi elemanı olarak kullanmak

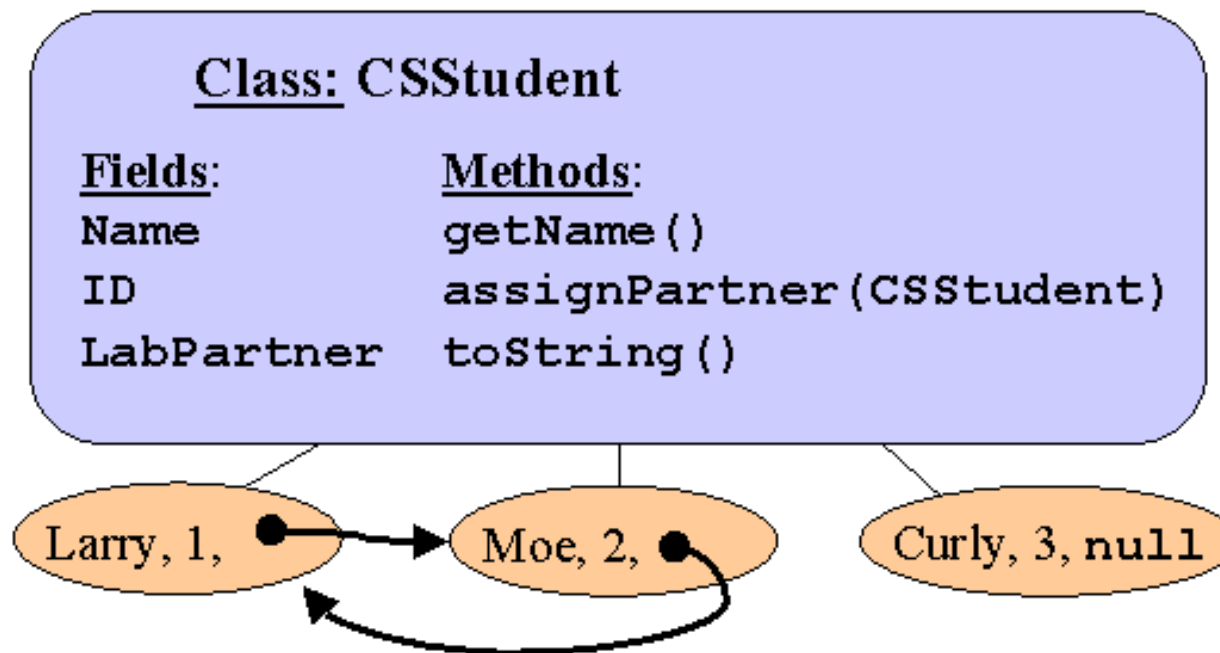
Problem Tanımı – CS Student

■ Kayıt modeli:

- Öğrenci isimlerinin komut satırından girilmesine izin verilecek
- Öğrenci çiftleri lab için eşlenecek. Tek sayıda öğrenci olması durumunda bir kişi eşsiz kalacak.
- Öğrenciler ve varsa labdaki arkadaşı kayıt sırasının tersi olacak şekilde listelenecek



Sınıflar, Alanlar, Metotlar ve Nesneler





Erişim

- Bir Java sınıfı yazılırken, programcı metotlara ve değişkenlere *dışarıdan* nesnelerin ne şekilde müdahale edebileceğini kontrol etmek için bazı *erişim niteleyicileri* belirtebilir.
- C++'dan farklı olarak, her bir metot veya alan için ayrı erişim niteleyicisi kullanılabilir.



Erişim

■ **public**

- Bütün dış nesneler public metotları çağırabilir.
- Bütün dış nesneler public alanları değiştirebilir.

■ **private**

- Metotlar sadece o sınıf içindeki metotlar tarafından çağırılabilir- alt sınıflar da bunları kullanamaz.
- Alanlar sadece o sınıf içindeki metotlar tarafından kullanılabilir- alt sınıflar da bunları kullanamaz.



Erişim

■ **protected**

- Metotlar sadece o sınıf içindeki metotlar ve alt sınıf metotları tarafından çağırılabilir.
- Alanlar sadece o sınıf içindeki metotlar ve alt sınıf metotları tarafından kullanılabilir.

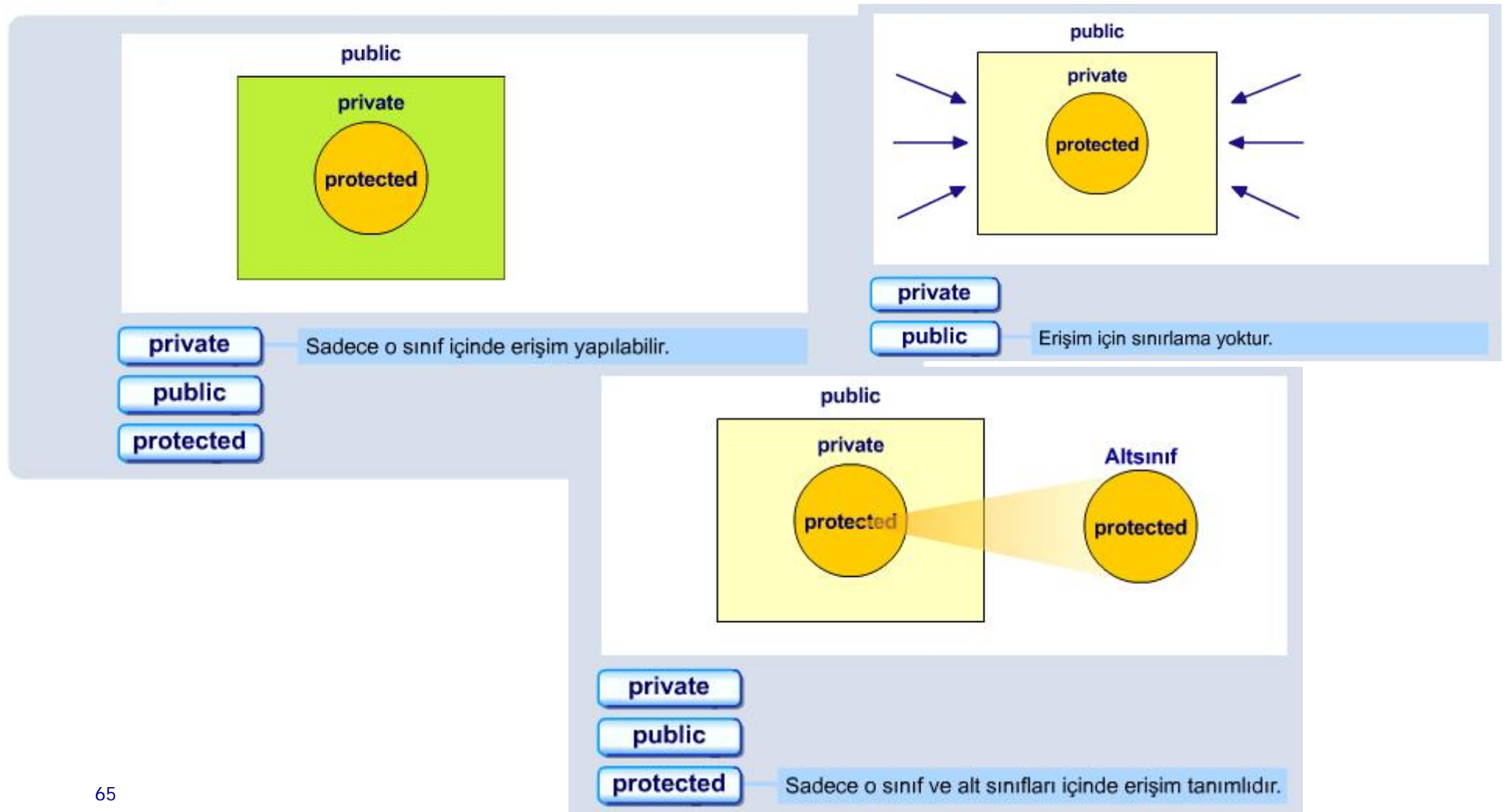
■ "Package erişim"

□ Başka bir erişim niteleyicisi kullanılmazsa varsayılan niteleyicidir:

- Aynı paket içerisinde bulunan sınıflara ait nesneler bu metotları çağırabilir.
- Aynı paket içerisinde bulunan sınıflara ait nesneler bu alanları kullanabilir.

Sınıf Veri ve Metodlarının Saklanması

Çeşitli programlama dilleri, bir sınıfa ilişkin veri ve metodların diğer sınıflardan saklanması için olanaklar sunarlar. C++ ve Java'da üç tür tanımlayıcı ile değişkenlerin erişilebilirlikleri sınırlanabilir.



Temel veri türlerinden Diziler

- Anafikir:

Daha önce öğrendiklerinize benzer
Yazılımda bazı farklılıklar vardır

- Java 'da dizi bildirimi:

```
<VeriTürü>[ ] <DiziAdı> = new <VeriTürü>[<boyut>];
```

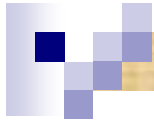
örnek: 10 adet tamsayı türündeki notu tutmak için tamsayı türünden 10 elemanlı bir dizi oluşturalım:

```
int[ ] notDizisi = new int[10];
```

- Dizi bildirim hatası:

köşeli parantez yerine normal parantez kullanmak:

```
int[ ] iGradeArray = new int(10);
```



Detaylar

- Sentaks biraz garip gelebilir çünkü diziler de birer nesnedir.
- Nesneleri anlatmaya başlayınca detaylı olarak inceleyeceğiz...

```
int[ ] iGradeArray = new int[10];  
int iGradeArray[ ] = new int[10];
```

Diziler

Örnek:

- 10 adet tamsayı türündeki notu tutmak için tamsayı türünden iNotDizisi adında 10 elemanlı bir dizi oluşturalım
- dizinin bütün değerlerine 0 değerini atayalım

```
int[ ] iNotDizisi = new int[10];
int i;
/* dizi işlemlerinde indis yani kontrol değişkeni olarak
i,j,k gibi geleneksel olarak herkes tarafından kullanılan ve
tanınan değişken isimleri kullanın. */
for (i=0; i < iNotDizisi.length; i++) {
    iNotDizisi[i] = 0;
} // for döngüsü
```

Güzel düşünce!
Dizinin boyutunu
değiştirdiğinizde,
sadece for imza
yapısının içini
değiştirmeniz yeterli.

Not:

- Diziler kendi uzunluklarını bilirler
- *length* bir özelliktir, metod değil
- Dizi uzunlukları sabittir, bildirim yapıldıktan sonra bir daha değiştirilemez.
- Bütün diziler nesnedir, bu nedenle bir referans değişkeni bişildirimi yapmalı, nesneyi oluşturmalı ve ilk değerini atamalısınız([declare a reference](#), [instantiate](#), [initialize](#))

Diziler

Notlar:

- Dizi indisi her zaman 0 'dan başlar 1 'den değil
- Öyleyse, *length* yani dizi uzunluk değeri indisin maksimum alacağı değerden sayısal olarak 1 fazladır
- Bu nedenle, eğer aşağıdaki gibi yaparsanız hata yaparsınız:

```
int[ ] iGradeArray = new int[10];
int i;
for (i=1; i <= iGradeArray.length; i++)
{
    iGradeArray[i] = 0;
} // for loop
```

- Yukarıdaki program kodunda dizinin 1 'den 10 'a kadar olan elemanlarına erişilmeye çalışılıyor
- Ancak dizinin indis numaraları *0..9 aralığındadır*
- Öyleyse: dizini ilk elemanı olan 0 indis olanı kaçırıyor ve dizinin 10 indis numaralı elemanına erişmeye çalışıyor ki ne böyle bir indis mevcut ne de diziye ait böyle bir alan.