NESNELERİN BAŞLANGIÇ DURUMU VE TEMİZLİK

Bir nesnenin başlangıç durumuna getirilme işlemini bir sanatçının sahneye çıkmadan evvelki yaptığı son hazırlıklar gibi düşünebilir... Oluşturulacak olan her nesne kullanıma sunulmadan önce bazı bilgilere ihtiyaç duyabilir veya bazı işlemlerin yapılmasına gereksinim duyabilir. Uygulama programlarının çalışması sırasında oluşan hataların önemli nedenlerden birisi de, nesnelerin yanlış biçimde başlangıç durumlarına getirilmesidir; diğeri ise, temizlik işleminin doğru dürüst yapılmamasıdır. Tasarımcı, bilmediği kütüphanelere ait nesneleri yanlış başlangıç durumuna getirmesinden ötürü çok sıkıntılar yaşayabilir. Diğer bir husus ise temizliktir. Temizlik işleminin doğru yapılmaması durumunda, daha önceden oluşturulmuş ve artık kullanılmayan nesneler sistem kaynaklarında gereksiz yer kaplarlar; ve bunun sonucunda ciddi problemler oluşabilir. Bu bölümde nesnelere başlangıç değerleri verilmesi ve temizleme sürecinin Java programlama dilinde nasıl yapıldığı ele alınmıştır. (yorum ekle)

3.1. Başlangıç Durumuna Getirme İşlemi ve Yapılandırıcılar (*Initialization and Constructor*)

Başlangıç durumuna getirme işlemlerin gerçekleştiği yer bir çeşit yordam (*method*) diyebileceğiz yapılandırıcılardır (*constructor*); ancak, yapılandırıcılar normal yordamlardan çok farklıdırlar. Şimdi biraz düşünelim... Elimizde öyle bir yapılandırıcı olacak ki, biz bunun içerisinde nesnenin kullanılmasından önce gereken işlemleri yapacağız; yani, nesneyi başlangıç durumuna getireceğiz. Ayrıca, Java bu yapılandırıcıyı, ilgili nesneyi oluşturmadan hemen önce otomatik olarak çağırabilecek... (yorum ekle)

Diğer bir problem ise yapılandırıcının ismidir; bu isim öyle olmalıdır ki, diğer yordam (*method*) isimleriyle çakışmamalıdır. Ayrıca Java'nın bu yapılandırıcıyı otomatik olarak çağıracağı düşünülürse, isminin Java tarafından daha önce biliniyor olması gerekir. (yorum ekle)
Bu problemlere ilk çözüm C++ dilinde bulunmuştur. Çözüm, yapılandırıcıyla sınıf isimlerinin birebir aynı olmasıdır (büyük küçük harf dahil). Böylece Java sınıfın yapılandırıcısını bularak, buna ait bir nesne oluşturmak için ilk adımı atabilecektir. Küçük bir uygulama üzerinde açıklanmaya çalışırsa, bir sonraki sayfadaki örnek verilebilir: (yorum ekle)

Örnek-3.1: *YapilandirciBasitOrnek.java* (yorum ekle)

```
class KahveFincani {

 public KahveFincani() {
    System.out.println("KahveFincani...");
 }
```

```
}

public class YapilandirciBasitOrnek {
  public static void main(String[] args) {
      for(int i = 0; i < 5; i++)
          new KahveFincani();
  }
}
```

Yukarıda verilen örnekte art arda 5 adet *KahveFincani* nesnesi oluşturuluyor; dikkat edilirse, nesneler oluşturulmadan önce (ellerimizi kahve fincanlarının üzerine götürmeden) ekrana kendilerini tanıtan ifadeler yazdılar; yani, nesnelere ilk değerleri verilmiş oldu... (yorum ekle)
Dikkat edilmesi gereken ikinci unsursa, yapılandırıcıya (*constructor*) verilen isimdir; bu içinde bulunduğu sınıf ismi ile birebir aynıdır. Anımsarsanız, normalde yordam isimleri bir fiil cümlesi içermeliydi (`dosyaAc()`, `openFile()`, `dosyaOku()`,`readFile()`, `dosyaYaz()`, `arabaSur()` vb.); ancak, yapılandırıcılar bu ku
ralın da dışındadır. Yukarıda verilen uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
KahveFincani...KahveFincani...KahveFincani...KahveFincani...KahveFincani...
```

Yapılandırıcılar, yordamlar (*methods*) gibi parametre alabilirler:

Örnek: *YapilandirciBasitOrnekVersiyon2.java* (yorum ekle)

```
class YeniKahveFincani {
  public YeniKahveFincani(int adet) {
      System.out.println(adet + " adet YeniKahveFincani");
  }
}

public class YapilandirciBasitOrnekVersiyon2 {
  public static void main(String[] args) {
      for(int i = 0; i < 5; i++)
          new YeniKahveFincani( i );
  }
}
```

Gönderilen parametre sayesinde nesnenin nasıl oluşacağı belirtebilmektedir. Bu örnekte olduğu gibi *YeniKahveFincani* nesnesi oluşturulurken kaç adet olacağı söylenebiliyor. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
0 adet YeniKahveFincani
1 adet YeniKahveFincani
2 adet YeniKahveFincani
3 adet YeniKahveFincani
4 adet YeniKahveFincani
```

Yapılandırıcılar, yordamlardaki gibi değer döndürme mekanizmasına sahip değildirler; herhangi bir şekilde değer döndüremezler. Bu değer döndürülemez ibaresi yordamlardaki **`void`** ifadesine

karşılık gelmemektedir. Yapılandırıcılardan çıkılmak isteniyorsa **`return`** kullanabilir. (yorum ekle)

3.1.1. Bir İsmin Birden Çok Yordam İçin Kullanılması

- Adaş Yordamlar (*Overloaded Methods*)

İyi bir uygulama yazılması her zaman için iyi bir takım çalışması gerektirir; takım çalışmasının önemli kurallarından birisi de birinin yazdığı kodu diğer kişilerin de kolaylıkla anlayabilmesinden geçer. Uygulamalardaki yordam isimlerinin, yordam içerisinde yapılan işlerle uyum göstermesi önemlidir. Bu sayede bir başka kişi sadece yordamın ismine bakarak, bu yordam içerisinde oluşan olayları anlayabilme şansına sahip olabilir. Örneğin elimizde bulunan müzik, resim ve metin (*text*) formatındaki dosyaları açmak için yordamlar yazılmak istenirse, bunların isimlerinin ne olması gerekir? Müzik dosyasını açan yordamın ismi `muzikDosyasiAc()`, resim dosyası için `resimDosya-sıAc()`, metin dosyasını açmak için ise `textDosyasınıAc()` gibi üç ayrı yordam ismi kullanılması ne kadar akıllıca ve pratik olur? Sonuçta işlem sadece dosya açmaktır; dosyanın türü sadece bir ayrıntıdır. Bir ismin birçok yordam için kullanılması (*method overloading*) bize bu imkanı verebilmektedir. Aynı tür işleve sahip olan yordamların aynı isimlere sahip olabilme özelliği, bizi isim bulma sıkıntısından da kurtarmaktadır. (yorum ekle)

Örnek-3.2: *YordamOverloadingDemo1.java* (yorum ekle)

```
class MuzikDosyasi {
  String m_tur = "Muzik Dosyasi" ;
}

class ResimDosyasi {
  String r_tur = "Resim Dosyasi" ;
}

class TextDosyasi {
  String t_tur = "Text Dosyasi" ;
}

public class YordamOverloadingDemo1 {
  public void dosyaAc(MuzikDosyasi md) {
    System.out.println( "Tur =" + md.m_tur );
  }


  public void dosyaAc(ResimDosyasi rd) {
    System.out.println( "Tur =" + rd.r_tur );
  }

  public void dosyaAc(TextDosyasi td) {
```

```
  System.out.println( "Tur =" + td.t_tur );
 }

 public static void main(String[] args) {
    YordamOverloadingDemo1 mod1 = new YordamOverloadingDemo1();
    MuzikDosyasi md = new MuzikDosyasi();
    ResimDosyasi rd = new ResimDosyasi();
    TextDosyasi td = new TextDosyasi();
    mod1.dosyaAc(md);
    mod1.dosyaAc(rd);
    mod1.dosyaAc(td);
 }
}
```

Uygulamamızın sonucu aşağıdaki gibi olur:

```
Tur =Muzik DosyasiTur =Resim DosyasiTur =Text Dosyasi
```

Yukarıdaki örnekte görüldüğü gibi aynı tür işlemleri yapan yordamların isimleri aynıdır. Peki, Java aynı isimde olan bu üç yordamı birbirinden nasıl ayırt etmektedir? (yorum ekle)

Œ Adaş Yordamlar Nasıl Ayırt Edilirler?

Java aynı isme sahip olan yordamları nasıl ayırt edebilmektedir? Cevap olarak parametrelerine göre denilebilir. Konu biraz daha açılırsa, her yordamın kendisine ait özel ve tek parametresi veya parametre listesi olmak zorundadır. (yorum ekle)

Örnek-3.4: *YordamOverloadingDemo2.java* (yorum ekle)

```
public class YordamOverloadingDemo2 {
 public int toplamaYap(int a , int b){
     int sonuc = a + b ;
     System.out.println("sonuc - 1 = " + sonuc);
     return sonuc ;
 }

 public void toplamaYap(int a , double b){
     double sonuc = a + b ;
     System.out.println("sonuc - 2 = " + sonuc);
 }

 public double toplamaYap(double a , int b){
     double sonuc = a + b ;
     System.out.println("sonuc - 3= " + sonuc);
     return sonuc ;
 }


 public static void main(String[] args) {
     YordamOverloadingDemo2 mod2 = new YordamOverloadingDemo2();
     mod2.toplamaYap(3,4);
     mod2.toplamaYap(3,5.5);
     mod2.toplamaYap(6.8,4);
 }
```

```
}
```

Bu örnekte üç adet `toplamaYap()` yordamının parametreleri birbirinden bir şekilde farklıdır: `toplamaYap()` yordamının ilki, 2 adet temel **`int`** tipinde parametre alarak diğer adaş yordamlarından ayrılmaktadır; geriye kalan 2 adet `toplamaYap()` yordamı ise aynı tip parametreler almaktadır. Bunlar temel `double` tipi ve `int` tipi, bu iki yordamı birbirinden farklı kılan, parametrelerin sırasıdır. Uygulamanın çıktısı aşağıdaki gibidir: (yorum ekle)

```
sonuc - 1 = 7sonuc - 2 = 8.5sonuc - 3= 10.8
```

Dönüş Değerlerine Göre Adaş Yordamlar Ayırt Edilebilir mi ?

Akıllara şöyle bir soru gelebilir: "Adaş yordamlar dönüş tiplerine göre ayırt edilebilir mi?" İnceleyelim:

Gösterim-3.1:

```
void toplamaYap();double toplamaYap();
```

Elimizde 2 adet aynı isimde ve aynı işlemi yapan, fakat biri değer döndürmeyen (**`void`**) diğeri ise **`double`** tipinde değer döndüren yordamların olduğunu varsayalım. (yorum ekle)

Gösterim-3.2:

```
double y = toplamayap();
```

Gösterim-3.2 için, Java bu yordamlardan hangisini seçeceğini tahmin edebilir; `double toplamaYap()`... Peki aşağıdaki gibi bir durum için nasıl bir yol izlenmesi gerekir? (yorum ekle)

Gösterim-3.3:

```
toplamayap() ;
```

Değer döndüren bir yordamı döndürdüğü tipe karşılık gelen değişkene atama zorunluluğu olmadığı hatırlatalım. Kısacası bu koşulda Java hangi yordamı çağıracağını bilemeyecektir. Bu nedenle, Java dilinde dönüş tiplerine göre yordamların ayırt edilmesi kabul görmez; ayırt edilmesini sağlayan tek şey parametrelerindeki farklılıktır. (yorum ekle)

3.1.2. Varsayılan Yapılandırıcılar (Default Constructors)

Eğer uygulamaya herhangi bir yapılandırıcı koyulmazsa, Java bu işlemi kendiliğinden yapmaktadır. Varsayılan yapılandırıcılar aynı zamanda parametresiz yapılandırıcılar (*default constructor* veya "*no-args*" *constructor*) olarak ta anılmaktadır; bunları içi boş yordamlar olarak düşünebilirsiniz. (yorum ekle)

Örnek-3.5: *VarsayilanYapilandirici.java* (yorum ekle)

```
class Kedi {
  int i;
}

public class VarsayilanYapilandirici {
   public static void main(String[] args) {
       Kedi kd = new Kedi();         //Varsayılan yapılandırıcı çağrıldı
   }
}
```

Java'nın yerleştirmiş olduğu varsayılan yapılandırıcı açık bir şekilde gözükmemektedir. Açık şekilde görmek istenirse; (yorum ekle)

Örnek-3.6: *VarsayilanYapilandirici.java (değişik bir versiyon)* (yorum ekle)

```
class Kedi {
  int i;
/*  varsayılan yapılandırıcı bu yapılandırıcıyı eğer biz koymasaydık
Java bizim yerimize zaten koyardı */
  public Kedi() {}
}

public class VarsayilanYapilandirici {
  public static void main(String[] args) {
     Kedi kd = new Kedi();         //Varsayılan yapılandırıcı çağrıldı
 }
}
```

Œ Büyünün Bozulması

Eğer, yapılandırıcı kodu yazan kişi tarafından konulursa, Java varsayılan yapılandırıcı desteğini çekecektir. Bunun nedeni şöyledir: Eğer bir sınıfa ait herhangi bir yapılandırıcı belirtilmezse, Java devreye girip kendiliğinden varsayılan bir yapılandırıcı koyar; eğer, biz kendimize ait özel yapılandırıcılar tanımlarsak, şöyle demiş oluruz: "Ben ne yaptığımı biliyorum, lütfen karışma". Bu durumda olası tüm yapılandırıcılar bizim tarafımızdan yazılması gerekir. Şöyle ki: (yorum ekle)

Örnek-3.7: *VarsayilanYapilandiriciVersiyon2.java* (yorum ekle)

```
class Araba {
  int kapi_sayisi;
  int vites_sayisi ;

  public Araba(int adet) {
     kapi_sayisi = adet ;
  }

  public Araba(int adet, int sayi) {
     kapi_sayisi = adet ;
     vites_sayisi = sayi ;
  }
}

public class VarsayilanYapilandiriciVersiyon2 {
   public static void main(String[] args) {
       Araba ar = new Araba();   //  ! Hata var!  Bu satır anlamlı değil; yapılandırıcısı yok
       Araba ar1 = new Araba(2);
       Araba ar2 = new Araba(4,5);
  }
}
```

Artık *Araba* sınıfına ait bir nesne oluşturmak için parametresiz yapılandırıcıyı (*default constructor*) çağıramayız; böylesi bir nesne oluşturmak isteniyorsa, artık, bu işlem için bir *Araba* sınıfına ait iki yapılandırıcıdan birisini seçip çağırmamız gerekir. (yorum ekle)

3.1.3. this Anahtar Sözcüğü

`this` anahtar sözcüğü, içinde bulunulan nesneye ait bir referans döndürür; bunun sayesinde nesnelere ait global alanlara erişme fırsatı bulunur. Şöyle ki: (yorum ekle)

Örnek-3.8: *TarihHesaplama.java* (yorum ekle)

```
public class TarihHesaplama {
  int gun, ay, yil;
  public void gunEkle(int gun) {
     this.gun += gun ;
  }
  public void gunuEkranaBas() {
      System.out.println("Gun = " + gun);
  }
  public static void main(String[] args) {
      TarihHesaplama th = new TarihHesaplama();
      th.gunEkle(2);  th.gunEkle(3);  th.gunuEkranaBas();
  }
}
```

`gunEkle()` yordamı sayesinde parametre olarak gönderdiğimiz değer, global olan temel `int` tipindeki `gun` alanının değerini arttırmaktadır.

Nesnelere ait global alanlar, içinde bulundukları nesnelere ait alanlardır ve nesne içerisindeki her statik olmayan yordam tarafından doğrudan erişilebilirler. Yerel değişkenler ise yordamların içerisinde tanımlanırlar; ve, ancak tanımlandığı yordam içerisinde geçerlidir. (yorum ekle)
`gunEkle()` yordamına dikkat edilirse, `gun` ismini hem *TarihHesaplama* nesnesine ait global bir alanının adı olarak hem de yerel değişken adı olarak kullanıldığı görülür. Burada herhangi bir yanlışlık yoktur. Çünkü bunlardan biri nesneye ait bir alan, diğeri ise `gunEkle()` yordamına ait yerel değişkendir. Bizim gönderdiğimiz değer, `gunEkle()` yordamının yerel değişkeni sayesinde *TarihHesaplama* nesnesinin global olan alanına eklenmektedir. En sonunda `gunuEkranaBas()` yordamı ile global olan`gun` alanının değerini görebilmekteyiz. (yorum ekle)
Özet olarak, `gunEkle()` yordamının içerisinde kullandığımız `this.gun` ifadesiyle *TarihHesaplama* nesnesinin global olan `gun` alanına erişebilmekteyiz. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
Gun = 5
```

Peki, `gunEkle()` yordamının içerisinde `this.gun` ifadesi yerine sadece `gun` ifadesi kullanılsaydı sonuç nasıl değişirdi? (yorum ekle)

Örnek-3.9: *TarihHesaplama2.java* (yorum ekle)

```
public class TarihHesaplama2 {
  int gun, ay, yil;

  public void gunEkle(int gun) {
    gun += gun ;
  }

  public void gunuEkranaBas() {
     System.out.println("Gun = " + gun);
  }

   public static void main(String[] args) {
      TarihHesaplama2 th = new TarihHesaplama2();
      th.gunEkle(2);
      th.gunEkle(3);
      th.gunuEkranaBas();
   }
}
```

Uygulamanın çıktısı aşağıdaki gibi olur:.

```
Gun = 0
```

TarihHesaplama nesnesine ait olan global **gun** alanına herhangi bir değer ulaşmadığı için sonuç sıfır olacaktır. (yorum ekle)
Œ Yordam Çağrımlarımda this Kullanımı

Gösterim-3.4:

```
class Uzum {     void sec() { /* ... */ }     ...     void cekirdeginiCikar() { sec(); /* ... */ } }
```

Bir yordamın içerisinde diğer yordamı çağırmak gayet basit ve açıktır ama sahne arkasında derleyici, çağrılan bu yordamın önüne **`this`** anahtar kelimesini gizlice yerleştirir: yani, fazladan **`this.sec()`** denilmesinin fazla bir anlamı yoktur. (yorumekle)
Aşağıdaki örnek **`this`** anahtar kelimesinin, içinde bulunduğu nesneye ait nasıl bir referansın alındığını çok net bir biçimde göstermektedir. (yorum ekle)

Örnek-3.10: *Yumurta.java* (yorum ekle)

```
public class Yumurta {

  int toplam_yumurta_sayisi = 0;

  Yumurta sepeteKoy() {
    toplam_yumurta_sayisi++;
    return this;
  }

  void goster() {
    System.out.println("toplam_yumurta_sayisi = "
                       + toplam_yumurta_sayisi);
  }

  public static void main(String[] args) {
    Yumurta y = new Yumurta();
    y.sepeteKoy().sepeteKoy().sepeteKoy().goster();
  }
}
```

`sepeteKoy()` yordamı *Yumurta* sınıfı tipinde değer geri döndürmektedir. `return this` diyerek, oluşturulmuş olan *Yumurta* nesnenin kendisine ait bir referans geri döndürülmektedir. `sepeteKoy()` yordamı her çağrıldığında *Yumurta* nesnesine ait,`toplam_yumurta_sayisi` global alanın değeri bir artmaktadır. Burada dikkat edilmesi gereken husus, `this` anahtar kelimesi ancak nesnelere ait olan yordamlar içinde kullanılabilir. Nesnelere ait olan yordamlar ve statik yordamlar biraz sonra detaylı bir şekilde incelenecektir. Uygulama sonucu aşağıdaki gibi olur: (yorum ekle)

```
toplam_yumurta_sayisi = 3
```

Bir Yapılandırıcıdan Diğer Bir Yapılandırıcıyı Çağırmak

Bir yapılandırıcıdan diğerini çağırmak `this` anahtar kelimesi ile mümkündür. (yorum ekle)

Örnek-3.11: *Tost.java* (yorum ekle)

```
public class Tost {
 int sayi ;
 String malzeme ;
```

```
Public Tost() {
   this(5);
    //   this(5,"sucuklu");              !Hata!-iki this kullanılamaz
    System.out.println("parametresiz yapilandirici");
}

public Tost(int sayi) {
   this(sayi,"Sucuklu");
    this.sayi = sayi ;
    System.out.println("Tost(int sayi) " );
}

public Tost(int sayi ,String malzeme) {
   this.sayi = sayi ;
   this.malzeme = malzeme ;
   System.out.println("Tost(int sayi ,String malzeme) " );
}

public void siparisGoster() {
     //   this(5,"Kasarli");         !Hata!-sadece yapılandırıcılarda kullanılır
     System.out.println("Tost sayisi="+sayi+ "malzeme =" + malzeme );
}

public static void main(String[] args) {
     Tost t = new Tost();
     t.siparisGoster();
}
}
```

- Bir yapılandırıcıdan `this` ifadesi ile diğer bir yapılandırıcıyı çağırırken dikkat edilmesi gereken kurallar aşağıdaki gibidir: (yorum ekle)

- Yapılandırıcılar içerisinden `this` ifadesi ile her zaman başka bir yapılandırıcı çağrılabilir. (yorum ekle)
- Yapılandırıcı içerisinden, diğer bir yapılandırıcıyı çağırırken `this` ifadesi her zaman ilk satırda yazılmalıdır. (yorum ekle)
- Yapılandırıcılar içerisinde birden fazla `this` ifadesi ile başka yapılandırıcı çağrılamaz. (yorum ekle)

Uygulama sonucu aşağıdaki gibi olur:

```
Tost(int sayi,String malzeme)Tost(int sayi)parametresiz yapilandiriciTost
sayisi =5 malzeme =Sucuklu
```

3.1.4. Statik Alanlar (Sınıflara Ait Alanlar)

Sadece global olan alanlara statik özelliği verilebilir. Yerel değişkenlerin statik olma özellikleri yoktur. Global alanları tür olarak iki çeşide ayırabiliriz: statik olan global alanlar ve nesnelere ait global alanlar. Statik alanlar, bir sınıfa ait olan alanlardır ve bu sınıfa ait tüm nesneler için ortak

bir bellek alanında bulunurlar, ayrıca statik alanlara sadece bir kez ilk değerleri atanır. (yorum ekle)

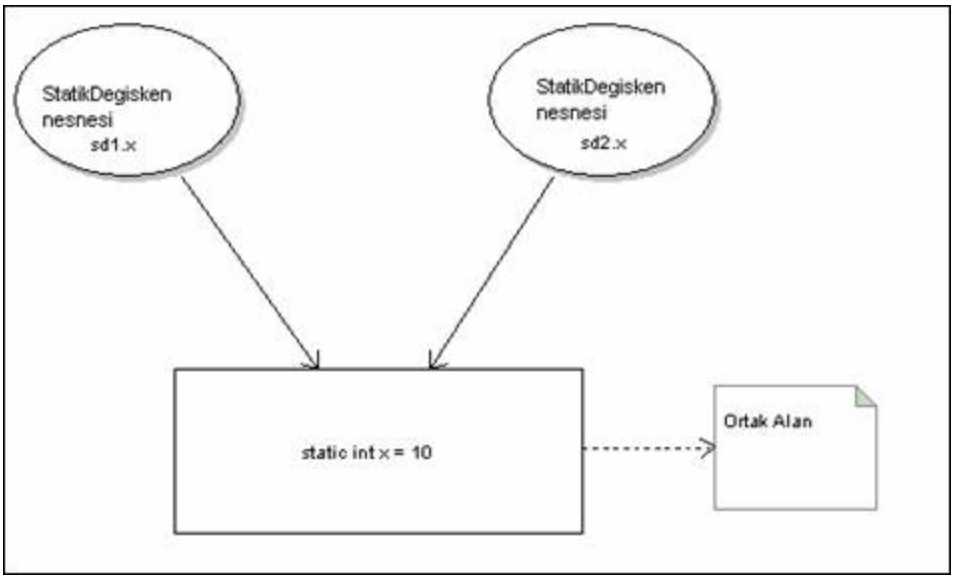
Örnek-3.12: *StatikDegisken.java* (yorum ekle)

```
public class StatikDegisken {
 public static int x ;
 public int y ;

 public static void ekranaBas(StatikDegisken sd ) {
    System.out.println("StatikDegisken.x = " + sd.x +
                       " StatikDegisken.y = " + sd.y );
 }

 public static void main(String args[]) {
    StatikDegisken sd1 = new StatikDegisken();
    StatikDegisken sd2 = new StatikDegisken();
    x = 10 ;
    // sd1.x = 10 ; //  x = 10 ile ayni etkiyi yapar
          // sd2.x = 10 ; //  x = 10 ile ayni etkiyi yapar
    sd1.y = 2 ;
    sd2.y = 8;
    ekranaBas(sd1);
    ekranaBas(sd2);
 }
}
```

Bu uygulamada *StatikDegisken* sınıfına ait iki adet nesne oluşturulmaktadır; daha sonradan *StatikDegisken* sınıfının statik olan **x** alanına 10 değeri atanmaktadır. Artık oluşturulacak olan tüm *StatikDegisken* tipindeki nesneler için bu **x** değeri ortaktır. Yalnız dikkat edilecek olursa *StatikDegisken* nesnelerine ait olan global **y** alanı her bir *StatikDegisken* nesnesi için farklıdır. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
StatikDegisken.x = 10 StatikDegisken.y = 2StatikDegisken.x = 10
StatikDegisken.y = 8
```

Statik alanların etkisi Şekil-3.1'de verilen çizimden incelenebilir:



Şekil-3.1. Statik alanlar

3.1.5. Statik Yordamlar (Static Methods)

Statik yordamlar (sınıf yordamları) nesnelerden bağımsız yordamlardır. Statik bir yordamı çağırmak için herhangi bir nesne oluşturulmak zorunda değildir. Statik olmayan yordamlardan (nesneye ait yordamlar) statik yordamları rahatlıkla çağırabilmesine karşın, statik yordamlardan nesne yordamlarını doğrudan çağrılamaz. (yorum ekle)

Örnek-3.13: *StatikTest.java* (yorum ekle)

```
public class StatikTest {

 public static void hesapla(int a , int b) {
    /*static yordam doğrudan  nesneye ait bir yordamı çağıramaz */
          // islemYap(a,b);             // !Hata!
 }

 public void islemYap(int a , int b) {
    /*doğru , nesneye ait bir yordam, static bir yordamı çağırabilir*/

    hesapla(a,b);
 }
}
```

3.1.6. Bir Yordamın Statik mi Yoksa Nesne Yordamı mı Olacağı Neye Göre Karar Verilecek?

Bir yordamı statik olarak tanımlamak için, "Bu yordamı çağırmak için, bu yordamın içerisinde bulunduğu sınıfa ait bir nesne oluşturmaya gerek var mı?" sorusuna yanıt vermemiz gerekir. (yorum ekle)

Acaba bir yordamı statik olarak mı tanımlasak? Yoksa nesne yordamı olarak mı tanımlasak? Bu sorunun cevabı, sınıf yordamları ile nesne yordamları arasındaki farkı iyi bilmekte gizlidir. Bu farkı anlamak için aşağıdaki uygulamamızı inceleyelim. (yorum ekle)

Örnek-3.14: *MutluAdam.java* (yorum ekle)

```
public class MutluAdam {

 private String ruh_hali = "Mutluyum" ;

 public void ruhHaliniYansit() {
 System.out.println( "Ben " + ruh_hali );
 }
 public void tokatAt() {
   if( ruh_hali.equals("Mutluyum" ) ) {
   ruh_hali = "Sinirlendim";
    }
 }
 public void kucakla() {
   if( ruh_hali.equals( "Sinirlendim" ) ) {
   ruh_hali = "Mutluyum";
     }
}

 public static void main(String[] args) {
     MutluAdam obj1 = new MutluAdam();
     MutluAdam obj2 = new MutluAdam();

obj1.ruhHaliniYansit();
obj2.ruhHaliniYansit();

obj1.kucakla();
obj2.tokatAt();

obj1.ruhHaliniYansit();
obj2.ruhHaliniYansit();
  }
}
```

`ruhHaliniYansit()`, `tokatAt()`, `kucakla()`'nın hepsi birer nesne yordamlarıdır; yani, bu yordamları çağırmak için *MutluAdam* sınıfına ait bir nesne oluşturmalıdır. (yorum ekle)
MutluAdam tipindeki `obj1` ve `obj2` referanslarına *MutluAdam* nesnelerini bağladığı anda bu iki nesnenin ruh halleri aynıdır (ruh_hali = "Mutluyum"). Fakat zaman geçtikçe her nesnenin kendisine ait ruh hali değişmektedir. Bu değişimin sebebi çevre koşulları olabilir.
Örneğin `obj1` referansına bağlı olan *MutluAdam* nesnesini kucaklandık, böylece mutlu olmaya devam etti ama `obj2` referansına bağlı *MutluAdam* nesnesine tokat attığımızda, mutlu olan ruh hali değişti ve sinirli bir ruh haline sahip oldu. (yorum ekle)
Nesne yordamları, nesnenin durumuna ilişkin işlemleri yapmak için kullanılırlar ama statik yordamlar ilgili nesnenin durumuna ilişkin genel bir işlem yapmazlar; Şöyle ki: (yorum ekle)

Örnek-3.15: *Toplama.java* (yorum ekle)

```
public class Toplama {

 public static double topla(double a , double b ) {
    double sonuc = a + b ;
    return sonuc ;
 }
}
```

Bu örneğimizde görüldüğü üzere `topla()` yordamının amacı sadece kendisine gelen iki `double` değerini toplamak ve sonucu geri döndürmektir. `topla()` yordamı *Toplama* sınıfına ait bu nesnenin durumu ile ilgili herhangi bir görev üstlenmediği için statik yordam (sınıf yordamı) olarak tanımlanması gayet mantıklıdır. Ayrıca `topla()` yordamını çağırmak için *Toplama* sınıfına ait bir nesne oluşturmak da çok gereksiz durmaktadır.
Şimdi *Toplama* sınıfı içerisindeki statik olan `topla()` yordamını kullanan bir uygulama yazalım. (yorum ekle)

Örnek-3.16: *ToplamaIslemi.java* (yorum ekle)

```
public class ToplamaIslemi {

 public static void main(String args[]) {

    if (args.length < 2) {
      System.out.println("Ltf iki adet sayi giriniz");
      System.exit(-1);       // uygulama sonlanacaktır
    }

    double a = Double.parseDouble(args[0]);
    double b = Double.parseDouble(args[1]);

    double sonuc = Toplama.topla(a,b);      // dikkat
    System.out.println("Sonuc : " + sonuc );
 }
}
```

ToplamaIslemi sınıfı, kullanıcıdan aldığı parametreleri öncelikle `double` tipine dönüştürmektedir. Daha sonra bu değerleri *Toplama* sınıfının `topla()` yordamına göndererek, toplatmaktadır. Dikkat edileceği üzere `topla()` yordamını çağırmak için*Toplama* sınıfına ait bir nesne oluşturma zahmetinde bulunmadık. Yukarıdaki uygulamamızı aşağıdaki gibi çalıştıracak olursak, (yorum ekle)

```
> java ToplamaIslemi 5.5 9.2
14.7
```

Yukarıda uygulamaya gönderilen iki parametre toplanıp sonucu ekrana yazdırılmıştır.

Sonuç olarak eğer bir yordam, nesnenin durumuna ilişkin bir misyon yükleniyorsa, o yordamı nesne yordamı olarak tanımlamamız en uygun olanıdır. Yani ilgili sınafa ait bir nesne oluşturulmadan, nesneye ait bir yordam (statik olmayan) çağrılamaz. Ama eğer bir yordam sadece atomik işlemler için kullanılacaksa (örneğin kendisine gelen *String* bir ifadenin hepsini büyük harfe çeviren yordam gibi) ve nesnenin durumuna ilişkin bir misyon yüklenmemiş ise o yordamı rahatlıkla statik yordam (sınıf yordamı) olarak tanımlayabiliriz. Dikkat edilmesi gereken bir başka husus ise uygulamalarınızda çok fazla statik yordam kullanılmasının tasarımsal açıdan yanlış olduğudur. Böyle bir durumda, stratejinizi baştan bir kez daha gözden geçirmenizi öneririm. (yorum ekle)

3.2. Temizlik İşlemleri: finalize() ve Çöp Toplayıcı (Garbage Collector)

Yapılandırıcılar sayesinde nesnelerimizi oluşturmadan önce, başlangıç durumlarının nasıl verildiğine değinildi; Peki, oluşturulan bu nesneler daha sonradan nasıl bellekten silinmektedir? Java programlama dilinde, C++ programla dilinde olduğu gibi oluşturulan nesneleri, işleri bitince yok etme özgürlüğü kodu yazan kişinin elinde değildir. Java programlama dilinde, bir nesnenin gerçekten çöp olup olmadığına karar veren mekanizma çöp toplayıcısıdır. (yorum ekle)
Çalışmakta olan uygulamanın içerisinde bulunan bir nesne artık kullanılmıyorsa, bu nesne çöp toplayıcısı tarafından bellekten silinir. Çöp toplama sistemi, kodu yazan kişi için büyük bir rahatlık oluşturmaktadır çünkü uygulamalardaki en büyük hataların ana kaynağı temizliğin (oluşturulan nesnelerin, işleri bitince bellekten silinmemeleri) doğru dürüst yapılamamasıdır. (yorum ekle)

3.2.1. finalize() Yordamı

Çöp toplayıcı (*garbage collector*) bir nesneyi bellekten silmeden hemen önce o nesnenin `finalize()` yordamını çağırır. Böylece bellekten silinecek olan nesnenin yapması gereken son işlemler var ise bu işlemler `finalize()` yordamı içerisinde yapılır. Bir örnek ile açıklanırsa, örneğin bir çizim programı yaptık ve elimizde ekrana çizgi çizen bir nesnemiz olduğunu düşünelim. Bu nesnemiz belli bir süre sonra gereksiz hale geldiğinde, çöp toplayıcısı tarafından bellekten silinecektir. Yalnız bu nesnemizin çizdiği çizgilerin hala ekranda olduğunu varsayarsak, bu nesne çöp toplayıcısı tarafından bellekten silinmeden evvel, ekrana çizdiği çizgileri temizlemesini `finalize()` yordamında sağlayabiliriz. (yorum ekle)
Akıllarda tutulması gereken diğer bir konu ise eğer uygulamanız çok fazla sayıda çöp nesnesi (kullanılmayan nesne) üretmiyorsa, çöp toplayıcısı *(garbage collector)* devreye girmeyebilir. Bir başka nokta ise eğer `System.gc()` ile çöp toplayıcısını tetiklemezsek, çöp toplayıcısının ne zaman devreye girip çöp haline dönüşmüş olan nesneleri bellekten temizleyeceği bilinemez. (yorum ekle)

Örnek-3.17: *Temizle.java* (yorum ekle)

```
class Elma {

   int i = 0 ;
   Elma(int y) {
     this.i = y ;
     System.out.println("Elma Nesnesi Olusturuluyor = " + i );
   }
```

```
    public void finalize() {
      System.out.println("Elma Nesnesi Yok Ediliyor = "+ i );
    }
}

public class Temizle {

    public static void main(String args[]) {
      for (int y=0 ; y<5 ;y++) {
        Elma e = new Elma(y);
      }

      for (int y=5 ; y<11 ;y++) {
        Elma e = new Elma(y);
      }
    }
}
```

Temizle.java örneğinde iki döngü içerisinde toplam 11 adet *Elma* nesnesi oluşturulmaktadır. Kendi bilgisayarımda 256 MB RAM bulunmaktadır; böyle bir koşulda çöp toplayıcısı devreye girmeyecektir. Nedeni ise, *Elma* nesnelerinin Java için ayrılan bellekte yeterince yer kaplamamasındandır. Değişik bilgisayar konfigürasyonlarına, sahip sistemlerde, çöp toplayıcısı belki devreye girebilir! Uygulamanın çıktısı aşağıdaki gibi olur: (yorum ekle)

```
 Elma Nesnesi Olusturuluyor = 0Elma Nesnesi Olusturuluyor = 1Elma Nesnesi
Olusturuluyor = 2Elma Nesnesi Olusturuluyor = 3Elma Nesnesi Olusturuluyor =
4Elma Nesnesi Olusturuluyor = 5Elma Nesnesi Olusturuluyor = 6Elma Nesnesi
Olusturuluyor = 7Elma Nesnesi Olusturuluyor = 8Elma Nesnesi Olusturuluyor =
9Elma Nesnesi Olusturuluyor = 10
```

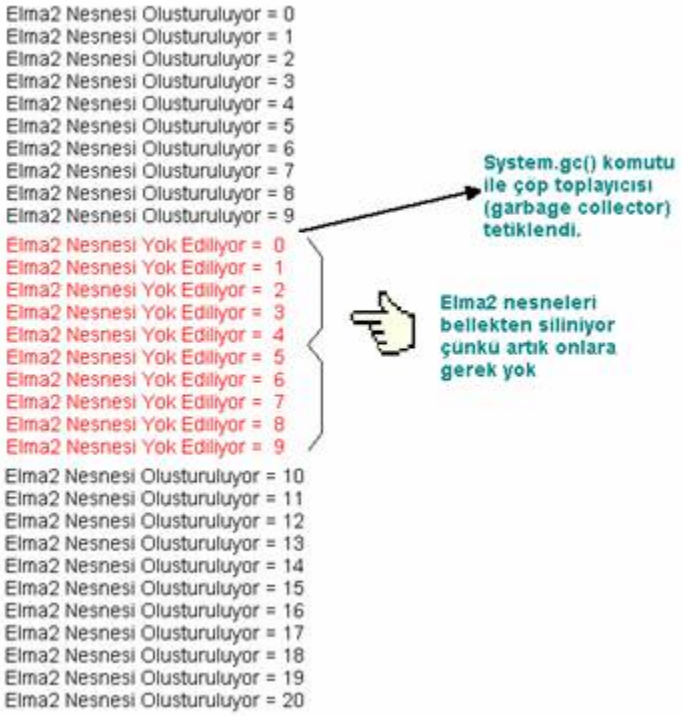
Uygulama sonucundan anlaşılacağı üzere, *Elma* nesnelerinin `finalize()` yordamı hiç çağrılmadı. Nedeni ise, çöp toplayıcısının hiç tetiklenmemiş olmasıdır. Aynı örneği biraz değiştirelim (bkz.. Örnek-3.18): (yorum ekle)

Örnek-3.18: *Temizle2.java* (yorum ekle)

```
class Elma2 {

 int i = 0 ;
 Elma2(int y) {
    this.i = y ;
    System.out.println("Elma2 Nesnesi Olusturuluyor = " + i );
 }

 public void finalize() {
    System.out.println("Elma2 Nesnesi Yok Ediliyor = "+ i );
 }
}

public class Temizle2 {
 public static void main(String args[]) {
```

```
    for (int y=0 ; y<10 ;y++) {
      Elma2 e = new Elma2(y);
    }
    System.gc() ;      // çöp toplayıcısını çağırdık
    for (int y=10 ; y<21 ;y++) {
      Elma2 e = new Elma2(y);
    }
  }
}
```

System sınıfının statik bir yordamı olan **`gc()`**, çöp toplayıcısının kodu yazan kişi tarafından tetiklenmesini sağlar. Böylece çöp toplayıcısı, çöp haline gelmiş olan nesneleri (kullanılmayan nesneleri) bularak (eğer varsa) bellekten siler. Yukarıdaki uygulamamızı açıklamaya başlarsak; İlk `for` döngüsünde oluşturulan *Elma2* nesneleri, döngü bittiğinde çöp halini alacaklardır. Bunun sebebi, ilk `for` döngüsünün bitimiyle, oluşturulan 10 adet *Elma2* nesnesinin erişilemez bir duruma geleceğidir. Doğal olarak, erişilemeyen bu nesneler, çöp toplayıcısının iştahını kabartacaktır. Uygulamanın sonucu aşağıdaki gibi olur; (yorum ekle)



Not: Sonuç üzerindeki oklar ve diğer gösterimler, dikkat çekmek amacıyla, özel bir resim uygulaması tarafından yapılmıştır. (yorum ekle)

`System.gc()` komutu ile çöp toplayıcısını tetiklediğimizde, gereksiz olan bu on adet *Elma2* nesnesi bellekten silinecektir. Bu nesneler bellekten silinirken, bu nesnelere ait `finalize()` yordamlarının nasıl çağrıldıklarına dikkat çekmek isterim. (yorumekle)

3.2.2. Bellekten Hangi Nesneler Silinir?

Çöp toplayıcısı bellekten, bir referansa bağlı olmayan nesneleri siler. Eğer bir nesne, bir veya daha fazla sayıdaki referansa bağlıysa, bu nesnemiz uygulama tarafında kullanılıyordur demektir ve çöp toplayıcısı tarafından bellekten silinmez. (yorumekle)

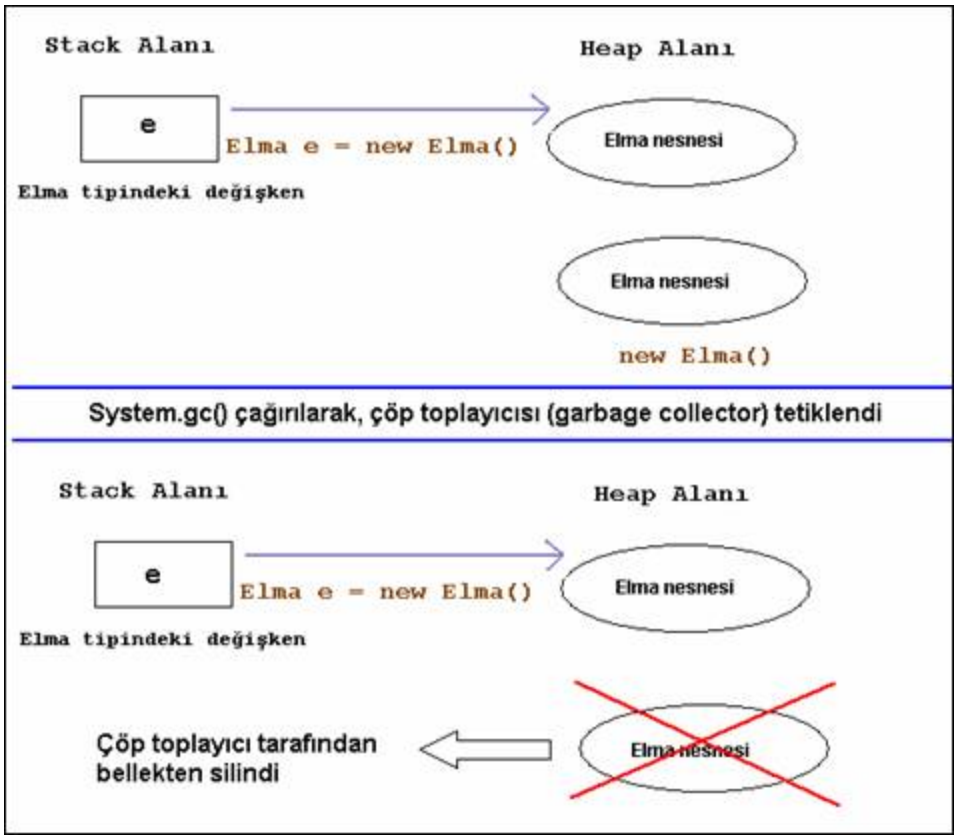
Örnek-3.19: *CopNesne.java* (yorum ekle)

```
public class CopNesne {

  public static void main(String args[]) {
     Elma e = new Elma(1);
     new Elma(2);
     System.gc() ; //  çöp toplayıcısını çağırdık
  }
}
```

Verilen örnekte 2 adet *Elma* nesnesinden biri çöp toplayıcısı tarafından bellekten silinirken, diğer nesne bellekte yaşamanı sürdürmeye devam eder. `System.gc()` yordamının çağrılması ile çöp toplayıcımız tetiklenir. *Elma* sınıfı tipindeki **e** referansına bağlı olan nesnemize çöp toplayıcısı tarafından dokunulmamıştır. Bunun sebebi bu *Elma* nesnesinin, *Elma* sınıfı tipindeki **e** referansına bağlı olmasıdır. Yapılandırıcısına 2 sayısını göndererek oluşturduğumuz *Elma* nesnesi ise çöp toplayıcısı tarafından bellekten silinmiştir çünkü bu nesnemize herhangi bir referans bağlı değildir. Uygulama sonucu aşağıdaki gibi olur: (yorum ekle)

```
Elma Nesnesi Olusturuluyor = 1Elma Nesnesi Olusturuluyor = 2Elma Nesnesi Yok
Ediliyor = 2
```

Uygulamamızı şekil üzerinde gösterirsek:



Şekil-3.2. Hangi nesne bellekten silinir?

3.2.3. finalize() Yordamına Güvenilirse Neler Olur?

Örnek-3.20: *BenzinDepo.java* (yorum ekle)

```
class Ucak {
 String ucak_isim ;
 boolean benzin_deposu_dolu = false ;
 boolean benzin_deposu_kapagi_acik_mi = false ;

 Ucak(boolean depoyu_doldur ,String ucak_isim) {
    benzin_deposu_kapagi_acik_mi = true ;   //  kapağı açıyoruz
    benzin_deposu_dolu = depoyu_doldur ;    // depo dolu
    this.ucak_isim =ucak_isim ;
 }

 /* Kapakların kapatılmasını finalize() yordamına bıraktık    */
 public void finalize() {
    if (benzin_deposu_kapagi_acik_mi) {          //  kapak açıksa
      benzin_deposu_kapagi_acik_mi = false ;     //  kapağı kapa
      System.out.println(ucak_isim + "- kapaklari kapatildi ");
    }
 }
}

public class BenzinDepo {
```

```
public static void main(String args[]) {

    Ucak ucak_1 = new Ucak(true,"F-16");        // benzin doldur
    new Ucak(true,"F-14");                      // benzin doldur

    System.gc();          // kapakları kapat
    System.out.println("Ucaklara benzin dolduruldu, kapaklari"
                                        + "kapatildi");
 }
}
```

Bu örnekte 2 adet *Ucak* nesnesi oluşturuluyor. Nesneler oluşturulur oluşturmaz kapaklar açılıp depolarına benzin doldurulmaktadır. Kapakları kapatma işlemi için `finalize()` yordamını kullanılıyor. Sonuçta `System.gc()` çağrılınca çöp toplayıcısının tetiklendiği bilinmektedir. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
F-14 - kapaklari kapatildiUcaklara benzin dolduruldu,kapaklari kapatildi
```

Uygulama çıktısının gösterdiği gibi, sadece F-14 isimli *Ucak* nesnesine ait benzin deposunun kapağı kapatılmış durumda, F-16 isimli *Ucak* nesnesinin kapağı ise hala açık durmaktadır –ki istenen durum F-16 isimli *Ucak* nesnesininde kapağının kapalı olmasıdır. Bu yanlışlığın sebebi, F-16 isimli *Ucak* nesnesine, yine *Ucak* tipinde olan `ucak_1` referansının bağlanmış olmasıdır. Çöp toplayıcısı boşta olan nesneleri bellekten siler, yani herhangi bir nesne, bir referansa bağlanmamış ise o nesne boşta demektir. Bu örneğimizde boşta olan nesne ise F-14 isimli *Ucak* nesnesidir ve çöp toplayıcısı tarafından temizlenmiştir ve bu temizlik işleminden hemen önce F-14 isimli *Ucak* nesnesinin `finalize()` yordamı çağrılmıştır. (yorum ekle)
Özet olarak, `System.gc()` ile çöp toplayıcısını tetikliyebiliriz ama referanslar, ilgili nesnelere bağlı kaldığı sürece**,** bu nesnelerin bellekten silinmesi söz konusu değildir. Dolayısıyla `finalize()` yordamı kullanılırken dikkatli olunmalıdır. (yorum ekle)

3.2.4. Çöp Toplayıcısı (Garbage Collector) Nasıl Çalışır?

Çöp toplayıcısının temel görevi kullanılmayan nesneleri bularak bellekten silmektir. *Sun Microsystems* tarafından tanıtılan Java HotSpot VM (*Virtual Machine*) sayesinde *heap* bölgesindeki nesneler nesillerine göre ayrılmaktadır. Bunlar **eski nesil** ve**yeni nesil** nesneler olmak üzere iki çeşittir. Belli başlı parametreler kullanarak Java HotSpot VM mekanizmasını denetlemek mümkündür; bizlere sağlandığı hazır parametreler ile normal bir uygulama gayet performanslı çalışabilir. Eğer sunucu üzerinde uygulama geliştiriliyorsa, Java HotSpot VM parametrelerini doğru kullanarak uygulamanın performansının arttırmak mümkündür. (yorum ekle)
Daha önceden söz edildiği gibi nesnelerin bellekten silinmesi görevi programcıya ait değildir. Bu işlem tamamen çöp toplayıcısının sorumluluğundadır. Java HotSpot VM ait çöp toplayıcısı iki konuyu kullanıcılara garanti etmektedir. (yorum ekle)

- Kullanılmayan nesnelerin kesinlikle bellekten silinmesini sağlamak. (yorum ekle)

- Nesne bellek alanının parçalanmasını engellemek ve belleğin sıkıştırılmasını sağlamak. (yorum ekle)

Bu bölümde dört adet çöp toplama algoritmasından bahsedilecektir, bunlardan ilki ve en temel olanı referans sayma yöntemidir, bu yöntem modern JVM'ler (*Java Virtual Machine*) tarafından artık kullanılmamaktadır. (yorum ekle)

Œ Eski yöntem

- Referans Sayma Yöntemi

  Bu yöntemde, bir nesne oluşturulur oluşturulmaz kendisine ait bir sayaç çalıştırılmaya başlar ve bu sayacın ilk değeri birdir. Bu nesnemizin ismi **X** olsun. Bu sayacın saydığı şey, oluşturduğumuz nesneye kaç adet referansın bağlı olduğudur. Ne zaman yeni bir referans bu **X** nesnesine bağlanırsa, bu sayacın değeri bir artar. Aynı şekilde ne zaman bu **X** nesnesine bağlı olan bir referans geçerlilik alanı dışına çıksa veya bu referans `null` değerine eşitlenirse, **X** nesnesine ait bu sayacın değeri bir eksilir. Eğer sayaç sıfır değerini gösterirse, **X** nesnemizin artık bu dünyadan ayrılma zamanı gelmiş demektir ve çöp toplayıcısı tarafından bellekten silinir.

  Bu yöntem, kısa zaman aralıkları ile çalıştırıldığında iyi sonuçlar vermektedir ve gerçek zamanlı uygulamalar için uygun olduğu söylenebilir. Fakat bu yöntemin kötü yanı döngüsel ilişkilerde referans sayacının doğru değerler göstermemesidir. Örneğin iki nesnemiz olsun, bunlardan biri **A** nesnesi diğeri ise **B** nesnesi olsun. Eğer **A** nesnesi, **B** nesnesine, **B** nesnesi de, **A** nesnesine döngüsel bir biçimde işaret ediyorsa ise bu nesneler artık kullanılmıyor olsa dahi bu nesnelere ait sayaçların değerleri hiç bir zaman sıfır olmaz ve bu yüzden çöp toplayıcısı tarafından bellekten silinmezler. (yorum ekle)

Yeni Yöntemler

Toplam 3 adet yeni çöp toplama yönetimi vardır. Her üç yöntemin de yaşayan nesneleri bulma stratejisi aynıdır. Bu strateji bellek içerinde yer alan statik ve yığın (*stack*) alanlarındaki referansların bağlı bulunduğu nesneler aranarak bulunur. Eğer geçerlibir referans, bir nesneye bağlıysa, bu nesne uygulama tarafından kullanılıyor demektir. (yorum ekle)

- Kopyalama yöntemi

  Oluşturulan bir nesne, ilk olarak *heap* bölgesindeki yeni nesil alanında yerini alır. Eğer bu nesne zaman içinde çöp toplayıcısı tarafından silinmemiş ise belirli bir olgunluğa ulaşmış demektir ve *heap* bölgesindeki eski nesil alanına geçmeye hak kazanır. Yeni nesilbölgeleri arasında kopyalanma işlemi ve bu alandan eski nesil alanına kopyalanma işlemi, kopyalama yöntemi sayesinde gerçekleşir. (yorum ekle)

- İşaretle ve süpür yöntemi

  Nesneler zaman içinde belli bir olgunluğa erişince *heap* bölgesindeki eski nesil alanına taşındıklarını belirtmiştik. Eski nesil alanındaki nesneleri bellekten silmek ve bu alandaki parçalanmaları engellemek için işaretle ve süpür yöntemi kullanılır. İşaretle ve süpüryöntemi, kopyalama yöntemine göre daha yavaş çalışmaktadır. (yorum ekle)

- Artan (sıra) yöntem

  Kopyalama yöntemi veya işaretle ve süpür yöntemi uygulamanın üretmiş olduğu büyük nesneleri bellekten silerken kullanıcı tarafından fark edilebilir bir duraksama oluşturabilirler. Bu fark edilir duraksamaları ortadan kaldırmak için Java HotSpot VM artan yönetimini geliştirmiştir.

  Artan yöntem büyük nesnelerin bellekten silinmeleri için orta nesil alanı oluşturur. Bu alan içerisinde küçük küçük bir çok bölüm vardır. Bu sayede büyük nesneleri bellekten silerken oluşan fark edilir duraksamalar küçük ve fark edilmez duraksamalara dönüştürülmektedir.

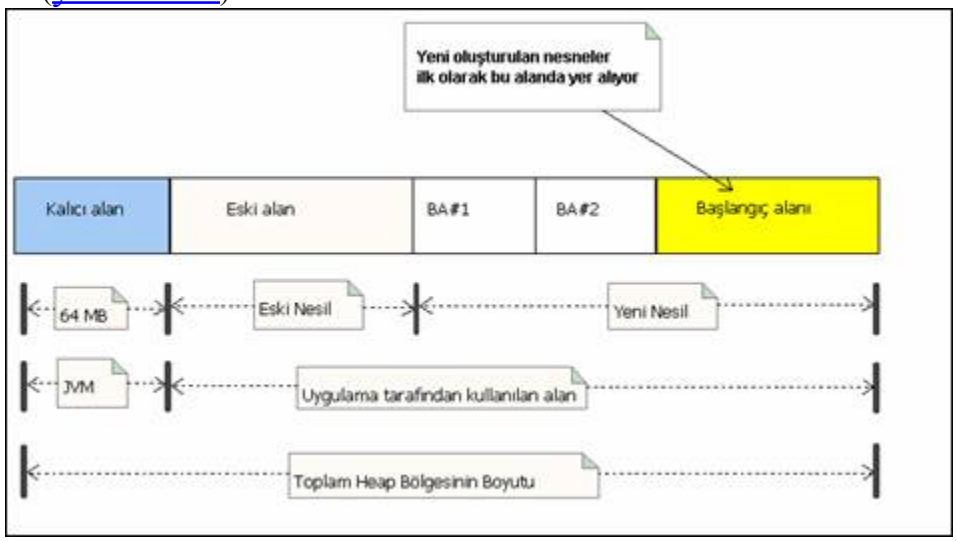
Artan yöntemi devreye sokmak için `-Xincgc` , çıkartmak için ise `-Xnoincgc` parametreleri kullanılır. Java HotSpot VM normal şartlarda bu yöntemi kullanmaz eğer kullanılmasını istiyorsak bu işlemi kendimiz yapmak zorundayız. (yorum ekle)

Gösterim-3.5:

```
java -Xincgc BenzinDepo
```

3.2.5. Heap bölgesi

Java HotSpot VM, *heap* bölgesini nesillere göre yönetir. Bellek alanında değişik nesillere ait nesneler bulunur. Aşağıdaki şeklimizde *heap* bölgesinin nesillere göre nasıl ayrıldığını görebilirsiniz. (yorum ekle)



Şekil-3.3. *Heap* bölgesi

Kalıcı alan özel bir bölgedir (*32 MB, 64 MB veya daha fazla olabilir*). Bu bölgede JVM'e ait bilgiler bulunur. `-XX:MaxPermSize=??M` komutu ile bu alanın boyutları kontrol edilebilir. (*??=ne kadarlık bir alan gerekiği, ör: java -X:MaxPermSize=64M*)(yorum ekle)

3.2.6. **Yeni Nesil**

Yeni Nesil bölümü toplam 3 alandan oluşur: Başlangıç alanı ve iki adet boş alan (*BA#1 ve BA#2*). Bu iki boş alandan bir tanesi bir sonraki kopyalama (*kopyalama yöntemi sayesinde*) için her zaman boş tutulur. Başlangıç alanındaki nesneler belli bir olgunluğa ulaştıkları zaman boş olan alanlara kopyalanırlar. (yorum ekle)

3.2.7. **Eski Nesil**

Eski nesil nesneler, *heap'deki* eski alanında bulunurlar. Uygulama tarafından kullanılan uzun ömürlü nesneler yeni nesil alanından, eski nesil alanına taşınırlar. Eski nesil alan içerisinde de zamanla kullanılmayan nesneler olabilir. Bu nesnelerin silinmesi için işaretle ve süpür yöntemi kullanılır. (yorum ekle)

3.2.8. ***Heap* bölgesi Boyutları Nasıl Denetlenir?**

Heap bölgesine minimum veya maksimum değerleri vermek için `-Xms` veya `-Xmx` parametreleri kullanılır. Performansı arttırmak amacı ile geniş kapsamlı sunucu (*server-*

side) uygulamalarında minimum ve maksimum değerler birbirlerine eşitlenerek sabit boyutlu *heap* bölgesi elde edilir. (yorum ekle)
JVM herhangi bir çöp toplama yöntemini çağırdıktan sonra *heap* bölgesini, boş alanlar ile yaşayan nesneler arasındaki farkı ayarlamak için büyütür veya azaltır. Bu oranı yüzdesel olarak minimum veya maksimum değerler atamak istenirse `-Xminf` veya (yorum ekle) `-Xmaxf` parametreleri kullanılır. Bu değer (*SPARC Platform versiyonu*) için hali hazırda minimum %40, maksimum %70'dır. (yorum ekle)

Gösterim-3.6:

```
java -Xms32mb Temizle
```

Gösterim-3.7:

```
java -Xminf%30 Temizle
```

Heap bölgesi JVM tarafından büyütülüp küçültüldükçe, eski nesil ve yeni nesil alanları da `NewRatio` parametresine göre yeniden hesaplanır. `NewRatio` parametresi eski nesil alan ile yeni nesil alan arasındaki oranı belirlemeye yarar. Örneğin `-X:NewRatio=3` parametresinin anlamı eskinin yeniye oranının 3:1 olması anlamına gelir; yani, eski nesil alanı *heap* bölgesinin 3/4'inde, yeni nesil ise 1/3'inde yer kaplayacaktır. Bu şartlarda kopyalama yönteminin daha sık çalışması beklenir. Eğer ki eski nesil alanını daha küçük yaparsak o zaman işaretle ve süpür yöntemi daha sık çalışacaktır. Daha evvelden belirtildiği gibi işaretle ve süpür yöntemi , kopyalama yöntemine göre daha yavaş çalışmaktadır. (yorum ekle)

Gösterim-3.8:

```
java -XX:NewRatio=8 Temizle
```

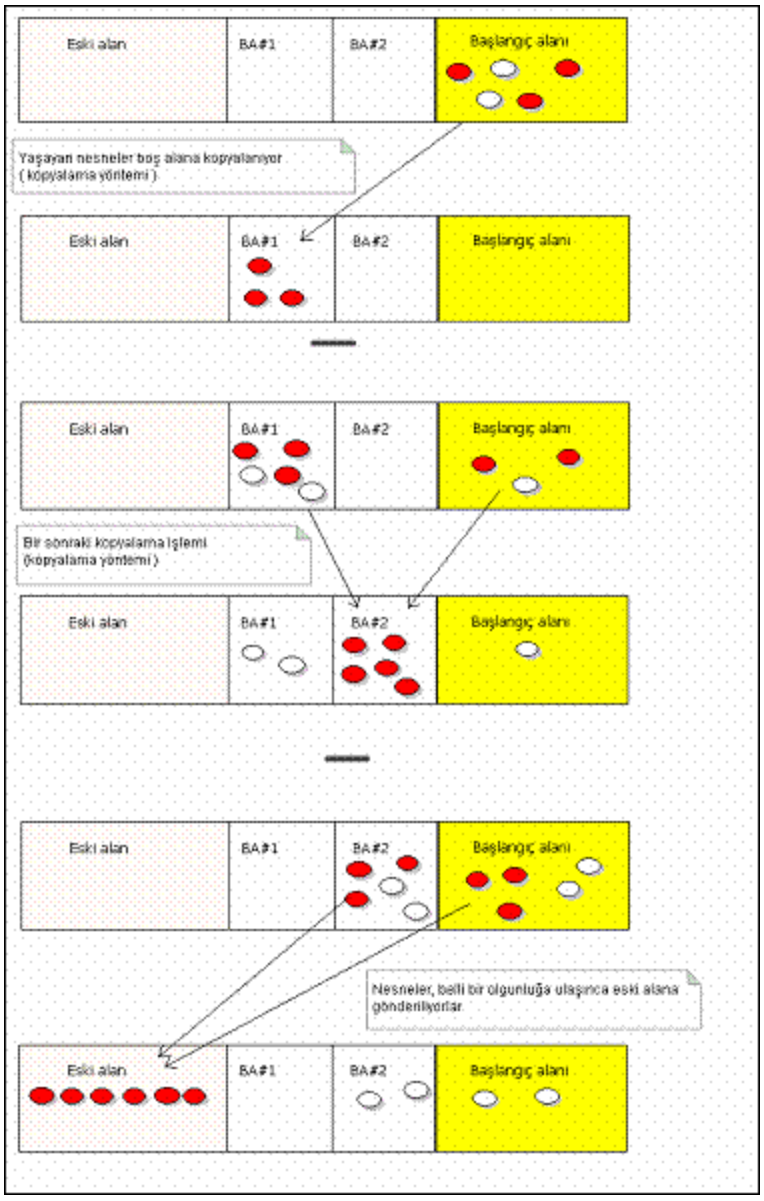
`NewSize` ve `MaxNewSize` parametreleri, yeni nesil alanının minimum ve maksimum değerlerini belirmek için kullanılır. Eğer bu parametreler birbirlerine eşitlenirse, sabit uzunlukta yeni nesil alanı elde edilmiş olur. Aşağıdaki gösterimde yeni nesil alanlarının 32MB olacağı belirtilmiştir. (yorum ekle)

Gösterim-3.9:

```
java -XX:MaxNewSize=32m Temizle
```

3.2.9. Kopyalama yönteminin gösterimi

Aşağıda, Şekil 3-4'de yaşayan nesneler koyu renk ile gösterilmişlerdir:

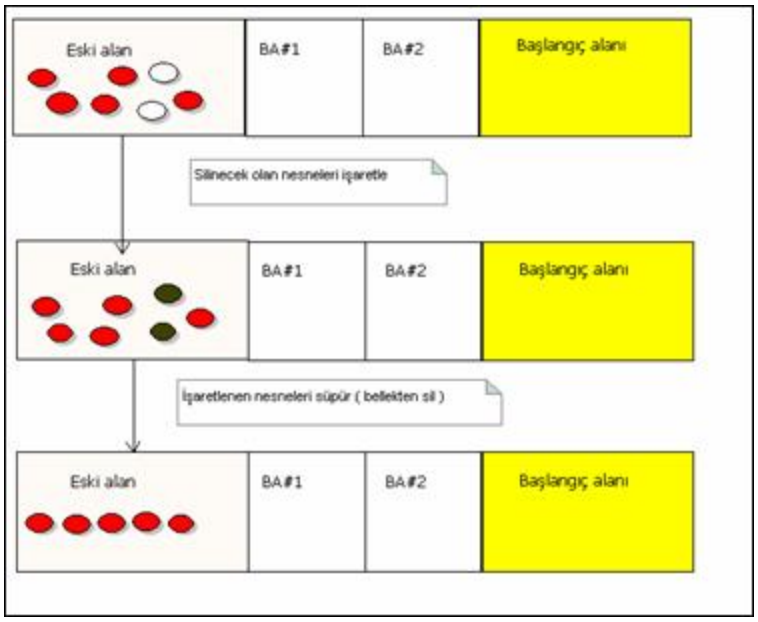


Şekil-3.4. *Heap* Bölgesindeki Gelişmeler

Nesneler belirli bir olgunluğa ulaşıncaya kadar, yeni nesil alanının bölümleri arasında geçirilirler. Bu sırada kopyalama yöntemi kullanılmaktadır. Belli bir olgunluğa ulaşan nesneler son aşamada eski alana gönderilirler. (yorum ekle)

3.2.10. İşaretle ve süpür yönteminin gösterimi

İyi tasarlanmış bir sistemde, çöp toplayıcısı bir kaç defa devreye girmesiyle, kullanılmayan nesneler bellekten silinir; geriye kalan yaşayan nesneler ise, eski nesil alanına geçmeye hak kazanırlar. Eski nesil alanında, işaretle ve süpür yöntemi kullanılır; bu yöntem kopyalama yöntemine göre daha yavaş fakat daha etkilidir. (yorum ekle)



Şekil-3.5. İşaretle ve Süpür Yönteminin Gösterimi

Aşağıdaki örnekte, kopyalama yönteminin ne zaman çalıştığını, işaretle ve süpür yönteminin ne zaman çalıştığını gösterilmektedir: (yorum ekle)

Örnek-3.21: *HeapGosterim.java* (yorum ekle)

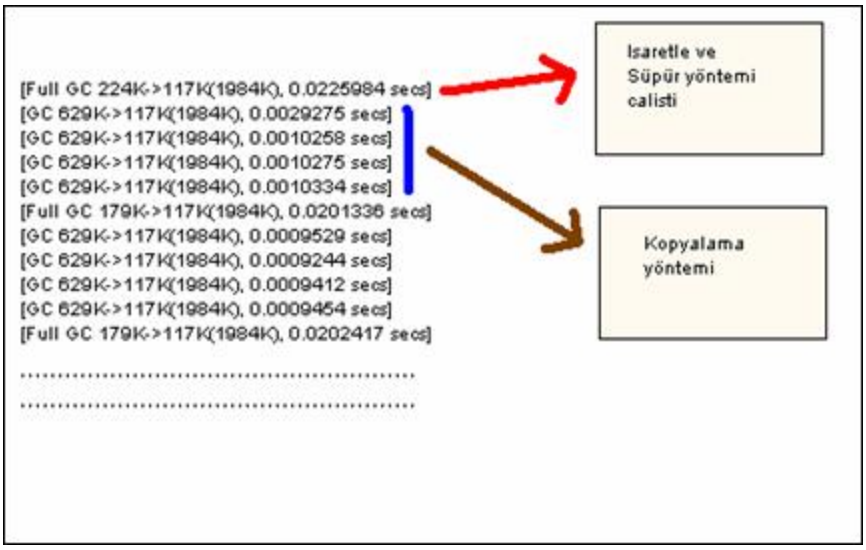
```
public class HeapGosterim {
  public static void main(String args[]) {
     for (int y=0 ; y<100000 ;y++) {
          new String("Yer Kaplamak icin");
          new String("Yer Kaplamak icin");
          new String("Yer Kaplamak icin");
          if ( (y%10000) == 0 ) {
            System.gc();
          }
     }
  }
}
```

Bu örnekte 3 adet *String* nesnesi `for` döngüsünün herbir çevriminde oluşturularak bellekten yer alınmaktadır. Nesneler herhangi bir referansa bağlı olmadıklarından dolayı, çöp toplayıcısının devreye girmesiyle birlikte bellekten silineceklerdir. Eğer uygulama aşağıdaki gibi çalıştırılacak olursa, gerçekleşen olaylar daha iyi görebilir: (yorum ekle)

Gösterim-3.10:

```
java -verbosegc HeapGosterim
```

Uygulamanın sonucu aşağıdaki gibi olur:



Şekil-3.6. Yöntemlerin Gösterimi

Not: Ekran sonuçları üzerindeki oklar ve diğer gösterimler, dikkat çekmek amacıyla özel bir resim uygulaması tarafından yapılmıştır. (yorum ekle)

3.2.11. İlk Değerlerin Atanması

Java uygulamalarında üç tür değişken çeşidi bulunur: yerel (*local*) değişkenler, nesneye ait global alanlar ve son olarak sınıfa ait global alanlar (*statik alanlar*). Bu değişkenlerin tipleri temel (*primitive*) veya herhangi bir sınıf tipi olabilir. (yorum ekle)

Örnek-3.22: *DegiskenGosterim.java* (yorum ekle)

```
public class DegiskenGosterim {

  int x ;                    //nesneye ait global alan
  static int y ;           // sınıfa ait global alan
  public void metod () {
     int i ; //yerel degisken
     //static int y = 5 ;       // yanlış
  }
}
```

3.2.12. Yerel Değişkenler

Yerel değişkenlere ilk değerleri programcı tarafından verilmelidir; ilk değeri verilmeden kullanılan yerel değişkenlere ilk tepki derleme (*compile-time*) anında verilir. (yorum ekle)

Gösterim-3.11:

```
public int hesapla () { // yerel değişkenlere ilk değerleri
                        // her zaman verilmelidir.
  int i ;
  i++;                  // ! Hata ! ilk deger verilmeden üzerinde işlem
yapılamaz
  return i ;
}
```

3.2.13. **Nesnelere Ait Global Alanlar**

Nesnelere ait global alanlara ilk değerleri programcının vermesi zorunlu değildir; Java bu alanlara ilk değerleri kendiliğinden verir. (yorum ekle)

3.2.13.1. **Nesnelere Ait Global Alanların Temel Bir Tip Olması Durumunda**

Java'nın temel tipteki global alanlara hangi başlangıç değerleri atadığını görmek için aşağıdaki örneği inceleyelim, (yorum ekle)

Örnek-3.23: *IlkelTipler.java*

```
public class IlkelTipler {

  boolean mantiksal_deger;
  char krakter_deger;
  byte byter_deger;
  short short_deger;
  int int_deger;
  long long_deger;
  float float_deger;
  double double_deger;

  public void ekranaBas() {
    System.out.println("Veri Tipleri Ilk Degerleri");
    System.out.println("boolean " + mantiksal_deger );
    System.out.println("char [" + krakter_deger + "] "+
                                (int)krakter_deger );
    System.out.println("byte " + byter_deger );
    System.out.println("short " + short_deger );
    System.out.println("int " + int_deger );
    System.out.println("long " + long_deger );
    System.out.println("float " + float_deger );
    System.out.println("double " + double_deger);
  }



  public static void main(String args[]) {
    new IlkelTipler().ekranaBas();

    /*
      // yukaridaki ifade yerine
      // asagidaki ifadeyide kullanabilirsiniz.
      IlkelTipler it = new IlkelTipler();
      İt.ekranaBas();
      */
  }
}
```

Bu uygulama sonucu aşağıdaki gibidir; buradan nesne alanlarına hangi başlangıç değerleri görülebilir: (yorum ekle)

```
Veri Tipleri Ilk Degerleri
boolean         false
char [ ]     0
byte         0
short        0
int          0
long         0
float        0.0
double       0.0
```

3.2.13.2. Nesnelere Ait Global Alanların Sınıf Tipi Olması Durumunda

Aksi belirtilmediği sürece nesnelere ait global alanlara, herhangi bir sınıf tipinde olması durumunda, başlangıç değeri olarak "`null`" atanır; yani boş değer... Eğer bu alanlar bu içeriğiyle kullanılmaya kalkışılırsa, çalışma anında (*run time*) hata ile karşılaşılır. Hata ile karşılaşmamak için ilgili alanları uygun bir nesnelere bağlamak gerekir. (yorum ekle)

Örnek-3.24: *NesneTipleri.java* (yorum ekle)

```
public class NesneTipleri {
  String s ;

  public static void main(String args[]) {
     NesneTipleri nt = new NesneTipleri();
     System.out.println("s = " + nt.s );
     // nt.s = nt.s.trim(); //hata
  }
}
```

Uygulama sonucu aşağıdaki gibi olur:

```
s = null
```

3.2.14. Sınıflara Ait Global Alanlar (Statik Alanlar)

Sınıflara ait global alanlara (statik alanlar) değer atamakla nesnelere ait global alanlara değer atamak arasında bir fark yoktur. Buradaki önemli nokta statik alanların ortak olmasıdır. (yorum ekle)

3.2.14.1. Sınıflara Ait Global Alanların (Statik Alanların) Temel Bir Tip Olması Durumunda

Örnek-3.25: *IlkelTiplerStatik.java* (yorum ekle)

```
public class IlkelTiplerStatik {

  static boolean mantiksal_deger;
  static char krakter_deger;
  static byte byter_deger;
  static short short_deger;
  static int int_deger;
```

```
  static long long_deger;
  static float float_deger;
  static double double_deger;

  public void ekranaBas() {
    System.out.println("Veri Tipleri Ilk Degerleri");
    System.out.println("static boolean " + mantiksal_deger );
    System.out.println("static char [" + krakter_deger + "] "+
                                (int)krakter_deger );
    System.out.println("static byte " + byter_deger );
    System.out.println("static short " + short_deger );
    System.out.println("static int " + int_deger );
    System.out.println("static long " + long_deger );
    System.out.println("static float " + float_deger );
    System.out.println("static double " + double_deger);
  }
  public static void main(String args[]) {
     new IlkelTiplerStatik().ekranaBas();

   /*
            // yukaridaki ifade yerine
      // asagidaki ifadeyi de kullanabilirsiniz.
      IlkelTiplerStatik its = new IlkelTiplerStatik();
      its.ekranaBas();   */
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:.

```
Veri Tipleri   Ilk Degerleri
static boolean         false
static char [ ]        0
static byte    0
static short   0
static int     0
static long    0
static float   0.0
static double 0.0
```

3.2.14.2. Sınıflara Ait Global Alanların (Statik Alanların) Sınıf Tipi Olması Durumunda

Aksi belirtilmediği sürece, sınıflara ait global alanlar herhangi bir sınıf tipinde olması durumunda, bu alana başlangıç değeri olarak "`null`" atanır. (yorum ekle)

Örnek-3.26: *StatikNesneTipleri.java* (yorum ekle)

```
public class StatikNesneTipleri {
  static String s ;

  public static void main(String args[]) {
     StatikNesneTipleri snt = new StatikNesneTipleri();
     System.out.println("s = " + snt.s );
     // snt.s = snt.s.trim(); //hata
  }
```

```
}
```

Uygulamanın çıktısı aşağıdaki gibi olur:

```
S = null
```

3.2.15. İlk Değerler Atanırken Yordam (*Method*) Kullanımı

Global olan nesnelere ait alanlara veya statik alanlara ilk değerleri bir yordam aracılığı ile de atanabilir: (yorum ekle)

Örnek-3.27: *KarisikTipler.java* (yorum ekle)

```
public class KarisikTipler {

  boolean mantiksal_deger = mantiksalDegerAta(); //   doğru (true)
değerini alır

  static int int_deger = intDegerAta();  // 10 değerini alır
  String s ;
  double d = 4.17 ;

  public boolean mantiksalDegerAta() {
     return true ;
  }

  public static int intDegerAta() {
     return 5*2 ;
  }

  public static void main(String args[]) {
       new KarisikTipler();
  }
}
```

Dikkat edilirse, statik olan `int_deger` alanına başlangıç değeri statik bir yordam tarafından verilmektedir. (yorum ekle)

3.2.16. İlk Değer Alma Sırası

Nesnelere ait global alanlara başlangıç değerleri hemen verilir; üstelik, yapılandırıcılardan (*constructor*) bile önce... Belirtilen alanların konumu hangi sırada ise başlangıç değeri alma sırasında aynı olur. (yorum ekle)

Örnek-3.28: *Defter.java* (yorum ekle)

```
class Kagit {
  public Kagit(int i) {
     System.out.println("Kagit ("+i+") ");
  }
}

public class Defter {
  Kagit k1 = new Kagit(1);       // dikkat
```

```
  public Defter() {
     System.out.println("Defter() yapilandirici ");
     k2 = new Kagit(33);          //artık başka bir Kagit nesnesine
bağlı
  }
  Kagit k2 = new Kagit(2); //dikkat

  public void islemTamam() {
     System.out.println("Islem tamam");
  }

  Kagit k3 = new Kagit(3);        //dikkat

  public static void main (String args[]) throws Exception {
    Defter d = new Defter();
    d.islemTamam();
  }
}
```

Uygulama sonucu aşağıdaki gibi olur:

```
Kagit (1)
Kagit (2)
Kagit (3)
Defter() yapilandirici
Kagit (33)
Islem tamam
```

Görüleceği gibi ilk olarak `k1` daha sonra `k2` ve `k3` alanlarına değerler atandı; sonra sınıfa ait yapılandırıcı çağrıldı... Ayrıca `k3` alanı, yapılandırıcının içerisinde oluşturulan başka bir *Kagit* nesnesine bağlanmıştır. Peki?, `k3` değişkeninin daha önceden bağlandığı *Kagit* nesnesine ne olacaktır? Yanıt: çöp toplayıcısı tarafından bellekten silinecektir. Uygulama `islemTamam()` yordamı çağrılarak çalışması sona ermektedir... (yorum ekle)

3.2.17. **Statik ve Statik Olmayan Alanların Değer Alma Sırası**

Statik alanlar, sınıflara ait olan alanlardır ve statik olmayan alanlara (*nesne alanları*) göre başlangıç değerlerini daha önce alırlar. (yorum ekle)

Örnek-3.29: *Kahvalti.java* (yorum ekle)

```
class Peynir {
  public Peynir(int i, String tur) {
  System.out.println("Peynir ("+i+") -->" + tur);
  }
}

class Tabak {
  public Tabak(int i, String tur) {
```

```
      System.out.println("Tabak ("+i+") -->" + tur);
   }
   static Peynir p1 = new Peynir(1,"statik alan");
   Peynir p2 = new Peynir(2,"statik-olmayan alan");
}

class Catal {
   public Catal(int i , String tur) {
      System.out.println("Catal ("+i+") --> " + tur);
   }
}

public class Kahvalti {
   static Catal c1 = new Catal(1,"statik alan");    // dikkat!

   public Kahvalti() {
      System.out.println("Kahvalti() yapilandirici");
   }

   Tabak t1 = new Tabak(1,"statik-olmayan alan");   // dikkat!

   public void islemTamam() {
      System.out.println("Islem tamam");
   }

   static Catal c2 = new Catal(2,"statik alan");   // dikkat!

   public static void main (String args[] ) throws Exception {
      Kahvalti d = new Kahvalti();
      d.islemTamam();
   }

   static Tabak t4 = new Tabak(4,"statik alan");    // dikkat!
   static Tabak t5 = new Tabak(5,"statik alan");    // dikkat!
}
```

Sırayla gidilirse, öncelikle `c1`, `c2`, `t4`, `t5` alanlarına başlangıç değerleri verilecektir. *Catal* nesnesini oluşturulurken, bu sınıfın yapılandırıcısına iki adet parametre gönderilmektedir; birisi `int` tipi, diğeri ise *String* tipindedir. Böylece hangi *Catal* tipindeki alanların, hangi *Catal* nesnelerine bağlandığı anlaşılabilir. *Catal* nesnesini oluştururken olaylar gayet açıktır; yapılandırıcı çağrılır ve ekrana gerekli bilgiler yazdırılır... (yorum ekle)

```
Catal (1) --> statik alan
Catal (2) --> statik alan
```

Tabak sınıfına ait olan yapılandırıcıda iki adet parametre kabul etmektedir. Bu sayede oluşturulan *Tabak* nesnelerinin ne tür alanlara (*statik mi? Değil mi?*) bağlı olduğunu öğrenme şansına sahip olunur. Olaylara bu açıdan bakılırsa; statik olan *Catal* sınıfı tipindeki `c1` ve `c2` alanlarından sonra yine statik olan *Tabak* sınıfı tipindeki `t4` ve `t5` alanları ilk değerlerini alacaklardır. Yalnız *Tabak* sınıfını incelersek, bu sınıfın içerisinde de global olan

statik ve nesnelere ait alanların olduğunu görürüz. Bu yüzden*Tabak* sınıfına ait yapılandırıcının çağrılmasından evvel *Peynir* sınıfı tipinde olan bu alanlar ilk değerlerini alacaklardır. Doğal olarak bu alanlardan ilk önce statik olan `p1` daha sonra ise nesneye ait olan `p2` alanına ilk değerleri verilecektir. Statik olan alanların her zaman daha önce ve yalnız bir kere ilk değerini aldığı kuralını hatırlarsak, ekrana yansıyan sonucun şaşırtıcı olmadığını görülür. (yorum ekle)

```
Peynir (1) àstatik alan
```

Daha sonradan da statik olmayan *Peynir* tipindeki alana ilk değeri verilir. (yorum ekle)

```
Peynir (2) -->statik-olmayan alan
```

Ve nihayet *Tabak* sınıfının yapılandırıcısı çalıştırılarak, *Tabak* nesnesi oluşturulur. (yorum ekle)

```
Tabak (4) -->statik alan
```

Bu işlemlerde sonra sıra statik `t5` alanına, *Tabak* nesnesini bağlanır. Bu işlemde sadece statik olmayan *Peynir* tipindeki `p2` alanına ilk değeri atanır. Statik olan `p1` alanına bu aşamada herhangi bir değer atanmaz (*daha evvelden değer atanmıştı*). Bunun sebebi ise statik alanlara sadece bir kere değer atanabilmesidir (*daha önceden belirtildiği üzere*). Bu işlemden sonra *Tabak* nesnesinin yapılandırıcısı çağrılır. (yorum ekle)

```
Peynir (2) -->statik-olmayan alan
Tabak (5) -->statik alan
```

Daha sonra *Kahvalti* nesnesine ait olan `Tabak` sınıfı tipindeki `t1` alanına ilk değeri verilir: (yorum ekle)

```
Peynir (2) -->statik-olmayan alan
Tabak (1) -->statik-olmayan alan
```

Dikkat edilirse, *Tabak* sınıfının içerisinde bulunan *Peynir* tipindeki global `p2` isimli alan (*nesneye ait alanı*), her yeni *Tabak* nesnesini oluşturulduğunda başlangıç değerini almaktadır. (yorum ekle)
Sondan bir önceki adım ise, *Kahvalti* sınıfının yapılandırıcısının çağırılması ile *Kahvalti* nesnesinin oluşturulmasıdır: (yorum ekle)

```
Kahvalti() yapilandirici
```

Son adım, *Kahvalti* nesnesinin `islemTamam()` yordamını çağırarak ekrana aşağıdaki mesajı yazdırmaktadır: (yorum ekle)

```
Islem tamam
```

Ekran yazılanlar toplu bir halde aşağıdaki gibi görülür:

```
Catal (1) --> statik alan
Catal (2) --> statik alan
Peynir (1) -->statik alan
Peynir (2) -->statik-olmayan alan
Tabak (4) -->statik alan
Peynir (2) -->statik-olmayan alan
Tabak (5) -->statik alan
Peynir (2) -->statik-olmayan alan
Tabak (1) -->statik-olmayan alan
Kahvalti() yapilandirici
Islem tamam
```

3.2.18. **Statik Alanlara Toplu Değer Atama**

Statik alanlara toplu olarak da değer atanabilir; böylesi bir örnek :

Örnek-3.30: *StatikTopluDegerAtama.java* (yorum ekle)

```
class Kopek {

  public Kopek() {
     System.out.println("Hav Hav");
  }
}

public class StatikTopluDegerAtama {

  static int x ;
  static double y ;
  static Kopek kp ;
  {
     x = 5 ;
     y = 6.89 ;
     kp = new Kopek();
  }

  public static void main(String args[]) {
     new StatikTopluDegerAtama();
  }
}
```

Statik alanlara toplu değer vermek için kullanılan bu öbek,
yalnızca, *StatikTopluDegerAtama* sınıfından nesne oluşturulduğu zaman veya bu sınıfa ait herhangi bir statik alana erişilmeye çalışıldığı zaman (*statik alanlara erişmek için ilgili sınafa ait bir nesne oluşturmak zorunda değilsiniz*), bir kez çağrılır. (yorum ekle)

3.2.19. **Statik Olmayan Alanlara Toplu Değer Atama**

Statik olmayan alanlara toplu değer verme, şekilsel olarak statik alanlara toplu değer verilmesine benzer: (yorum ekle)

Örnek-3.31: *NonStatikTopluDegerAtama.java* (yorum ekle)

```
class Dinozor {
  public Dinozor() {
     System.out.println("Ben Denver");
  }
}

public class NonStatikTopluDegerAtama {

  int x ;
  double y ;
  Dinozor dz ;
  {
     x = 5 ;
     y = 6.89 ;
     dz= new Dinozor();
  }

  public static void main(String args[]) {
     new NonStatikTopluDegerAtama();
  }
}
```

3.3. Diziler (*Arrays*)

Diziler nesnedir; içerisinde belirli sayıda eleman bulunur. Eğer bu sayı sıfır ise, dizi boş demektir. Dizinin içerisindeki elemanlara eksi olmayan bir tam sayı ile ifade edilen dizi erişim indisi ile erişilir. Bir dizide `n` tane eleman varsa dizinin uzunluğu da `n`kadardır; ilk elemanın indisi/konumu **0**'dan başlar**,** son elemanı ise `n-1`'dir. (yorum ekle)

Dizi içerisindeki elemanlar aynı türden olmak zorundadır. Eğer dizi içerisindeki elemanların türü `double` ise, bu dizinin türü için `double` denilir. Bu `double` tipinde olan diziye *String* tipinde bir nesne atanması denenirse hata ile karşılaşılır. Diziler temel veya herhangi bir sınıf tipinde olabilir... (yorum ekle)

3.3.1. Dizi Türündeki Referanslar

Dizi türündeki referanslar, dizi nesnelerine bağlanmaktadır. Dizi referansları tanımlamak bu dizinin hemen kullanılacağı anlamına gelmez... (yorum ekle)

Gösterim-3.12:

```
double[] dd ;     // double tipindeki dizi
double dd[] ;     // double tipindeki dizi
float [] fd ;    // float tipindeki dizi
Object[] ao ;     // Object tipindeki dizi
```

Gösterim-3.12'de yalnızca dizi nesnelerine bağlanacak olan referanslar tanımlandı; bu dizi nesneleri bellek alanında henüz yer kaplamamışlardır. Dizi nesnelerini oluşturmak için **new** anahtar sözcüğü kullanılması gereklidir. (yorum ekle)

3.3.2. **Dizileri Oluşturmak**

Diziler herhangi bir nesne gibi oluşturalabilir:

Gösterim-3.13:

```
double[] d = new double[20];   //   20 elemanlı double tipindeki dizi
double dd[]= new double[20];   //   20 elemanlı double tipindeki
dizi
float []fd = new float [14];   //   14 elemanlı float tipindeki dizi
Object[]ao = new Object[17];   //   17 elemanlı Object tipindeki dizi
String[] s = new String[25];   //   25 elemanlı String tipindeki dizi
```

Örneğin, **new** `double[20]` ifadesiyle 20 elemanlı temel `double` türünde bir dizi elde edilmiş oldu; bu dizi elemanları başlangıç değerleri `0.0`'dır. Java'nın hangi temel türe hangi varsayılan değeri atadığını görmek için Bölüm 1'e bakılabilir. (yorumekle)

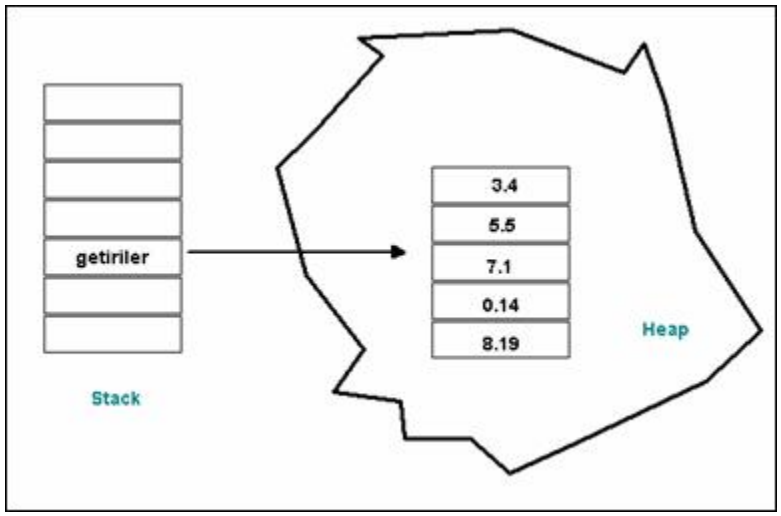
3.3.2.1. **Temel Türlerden Oluşan Bir Dizi**

Bir dizi temel türde ise, elemanları aynı temel türde olmak zorundaydı; bu tekrar anımsatıldıktan sonra aşağıdaki gösterim yakından incelenirse, (yorum ekle)

Gösterim-3.14:

```
double[] getiriler = { 3.4, 5.5, 7,1, 0.14, 8.19 } ;
```

Gösterim-3.14'deki dizi tanımını, aşağıdaki şekil üzerinde açıklanabilir:



Şekil-3.7. Temel (*primitive*) türde bir dizi

Temel `double` türdeki `getiriler` referansı yığın (*stack*) alanında yer almaktadır. Bu referansın bağlı olduğu dizi nesnesi ise *heap* alanında bulunmaktadır. Bu dizimizin elemanları

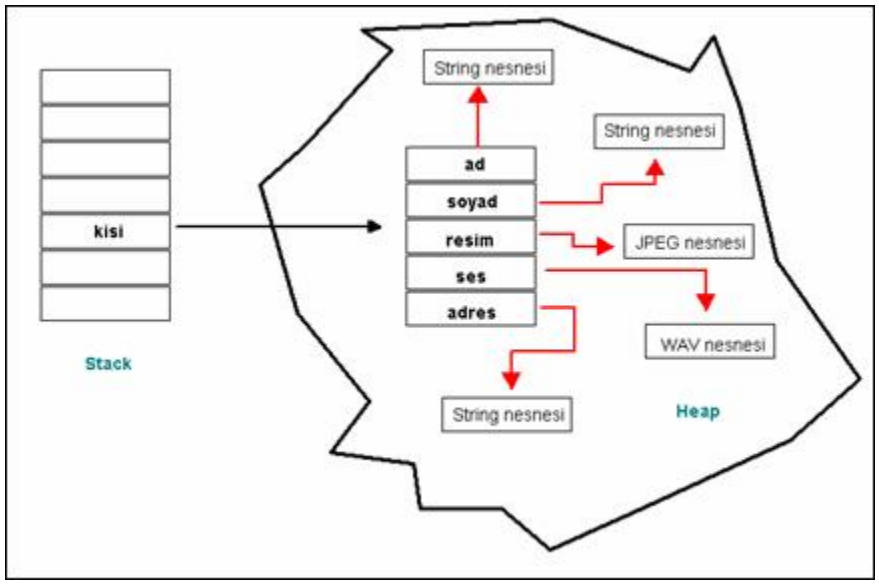
temel `double` tipinde olduğu için dizi elemanlarının değerleri kendi üzerlerinde dururlar. (yorum ekle)

3.3.2.2. **Nesnelerden Oluşan Bir Dizi**

Nesnelerden oluşan bir dizi, temel türlerden oluşan bir diziden çok daha farklıdır. Nesnelerden meydana gelmiş bir dizi oluşturulduğu zaman, bu dizinin elemanlarının içerisinde, ilgili nesnelere ait referanslar tutulur. Örneğin aşağıdaki gibi bir dizi yapısı bulunsun: (yorum ekle)

- `Ad`, *String* sınıfı tipinde
- `Soyad`, *String* sınıfı tipinde
- `Resim`, *JPEG* sınıfı tipinde (*böyle bir sınıf olduğunu varsayalım*)
- `Ses`, *WAV* sınıfı tipinde (*böyle bir sınıf olduğunu varsayalım*)
- `Adres`, *String* sınıfı tipinde.

Bu dizinin içerisinde 5 adet farklı nesne bulunmaktadır. Tüm nesnelerin içerisine koyulacak olan bu dizinin tipi *Object* sınıfı tipinde olmalıdır. *Object* sınıfı, Java programlama dilinde, sınıf ağacının en tepesinde bulunur. *Object* sınıfının ayrıntılarını bölümlerde ele alınacaktır... Verilen dizi aşağıdaki gibi gösterilebilir: (yorum ekle)



Şekil-3.8. *Object* sınıfı türündeki bir dizi

Bu dizinin içerisinde ilgili nesnelerin kendileri değil, bu nesnelere bağlı olan referanslar durur. (yorum ekle)

3.3.3. **Dizi Boyunun Değiştirilmesi**

Dizi boyutu birkez verildi mi, artık değiştirilemezler! (yorum ekle)

Gösterim-3.15:

```
int liste[] = new int[5] ;
liste = new int[15] ; // yeni bir dizi nesnesi bağlandı
```

Gösterim-3.15'de dizi boyutlarının büyütüldüğünü sanmayın; burada, yalnızca yeni bir dizi nesnesi daha oluşturulmaktır. `liste` dizi referansının daha evvelden bağlanmış olduğu dizi nesnesi (*`new int[5]`*), çöp toplama işlemi sırasında çöp toplayıcısı tarafından bellekten silinecektir. (yorum ekle)

3.3.4. **Dizi Elemanlarına Erişim**

Java dilinde dizi kullanımı diğer dillere nazaran daha az risklidir; anlamı, eğer tanımladığımız dizinin sınırları aşılırsak, çalışma-anında (*runtime*) hata ile karşılaşacağımızdır. Örneğin 20 elemanlı bir `double` dizisi tanımlanmış olsun. Eğer bu dizinin 78. elemanına ulaşılmak istenirse (- *ki böyle bir indisli eleman yok*), olanlar olur ve çalışma-anında hata alınır; böylesi hatanın (*ArrayIndexOutOfBounds Exception*) çalışma-anında alınması güvenlik açısından güzel bir olaydır. Böylece dizi için ayrılmış bellek alanından dışarı çıkılıp başka verilere müdahale edilmesi engellenmiş olunur. (yorum ekle)

Örnek-3.32: *DiziElemanlariGosterimBir.java* (yorum ekle)

```
public class DiziElemanlariGosterimBir {

  public static void main(String args[]) {
    double[] d = { 2.1, 3.4, 4.6, 1.1, 0.11 } ;
    String[] s = { "defter", "kalem", "sarman", "tekir", "boncuk" }; ;
    // double tipindeki dizimizi ekrana yazdırıyoruz
    for (int i = 0 ; i < d.length ; i ++) {
      System.out.println("d["+i+"] = " + d[i] );
      // System.out.println("d["+78+"] = " + d[78] );   //   Hata !
    }
    System.out.println("---------------");

    // String tipindeki dizimizi ekrana yazdırıyoruz
    for (int x = 0 ; x < s.length ; x ++) {
       System.out.println("s["+x+"] = " + s[x] );
       // System.out.println("s["+78+"]=" + s[78] );     //   Hata !
    }
  }
}
```

`length` ifadesiyle bir dizinin içerisindeki eleman sayısı öğrenilir. Bu örnekte iki adet dizi tanımlandı: `double` ve `String` türündeki dizilerin içerisine 5'er adet eleman yerleştirildi ve sonradan bunları for döngüsü ile ekrana yazdırıldı. `i < d.length` ifadesine dikkat edilirse, döngü sayacın 4'e kadar artmaktadır; ancak, döngü sayacının 0'dan başladığı unutulmamalıdır. (yorum ekle)

Eğer 5 elemana sahip olan dizinin 78. elemanına erişilmeye kalkışılırsa, derleme anında (*compile-time*) bir hata ile karşılaşılmaz; ancak, uygulama yürütüldüğü zaman; yani, çalışma-anında (*runtime*) hata ile karşılaşılır. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
d[0] = 2.1
```

```
d[1] = 3.4
d[2] = 4.6
d[3] = 1.1
d[4] = 0.11
---------------
s[0] = defter
s[1] = kalem
s[2] = sarman
s[3] = tekir
s[4] = boncuk
```

Bir önceki uygulamanın çalışma anına hata vermesi istenmiyorsa, yorum satırı olan yerler açılması ve uygulamanın baştan derlenip çalıştırması gerekmektedir. Aynı örnek daha değişik bir şekilde ifade edilebilir: (yorum ekle)

Örnek-3.33: *DiziElemanlariGosterimIki.java* (yorum ekle)

```
public class DiziElemanlariGosterimIki {
  double[] d ;
  String[] s ;

  public DiziElemanlariGosterimIki() {  // double tipindeki diziye
eleman atanıyor
        d = new double[5];
        d[0] = 2.1 ;
        d[1] = 3.4 ;
        d[2] = 4.6 ;
        d[3] = 1.1 ;
        d[4] = 0.11 ;
     // d[5] = 0.56 ;    //   Hata !

     // String tipindeki diziye eleman atanıyor
     s = new String[5] ;
     s[0] = new String("defter");
     s[1] = new String("kalem");
     s[2] = new String("sarman");
     s[3] = new String("tekir");
     s[4] = new String("boncuk");
     // s[5] = new String("duman");    //  Hata !
  }

  public void ekranaBas() {   // double tipindeki diziyi ekrana
yazdırıyoruz
     for (int i = 0 ; i < d.length ; i ++) {
       System.out.println("d["+i+"] = " + d[i] );
     }
     System.out.println("-------------------"); // String dizi ekrana
yazdırılıyor
     for (int x = 0 ; x < s.length ; x ++) {
       System.out.println("s["+x+"] = " + s[x] );
     }
```

```
  }

  public static void main(String args[]) {
      DiziElemanlariGosterimIki deg = new
            DiziElemanlariGosterimIki();
            deg.ekranaBas();
  }
}
```

Bu örnekte 5 elemanlı dizilere 6. eleman eklenmeye çalışıldığında, derleme anında (*compile-time*) herhangi bir hata ile karşılaşmayız. Hata ile karışılacağımız yer çalışma anındadır. Çünkü bu tür hatalar çalışma anında kontrol edilir. Yalnız çalışma anında hata oluşturabilecek olan satırlar kapatıldığı için şu an için herhangi bir tehlike yoktur; ancak, çalışma anında bu hatalar ile tanışmak isterseniz, bu satırların başında "//" yorum ekini kaldırmanız yeterli olacaktır. Uygulamanın sonucu aşağıdaki gibi olacaktır: (yorum ekle)

```
d[0] = 2.1
d[1] = 3.4
d[2] = 4.6
d[3] = 1.1
d[4] = 0.11
------------------------
s[0] = defter
s[1] = kalem
s[2] = sarman
s[3] = tekir
s[4] = boncuk
```

3.3.5. **Diziler Elemanlarını Sıralama**

Dizi elemanlarını büyükten küçüğe doğru sıralatmak için *java.util* paketini altındaki *Arrays* sınıfı kullanılabilir. Bu sınıfın statik **`sort()`** yordamı sayesinde dizilerin içerisindeki elemanlar sıralanabilir: (yorum ekle)

Örnek-3.34: *DiziSiralama.java* (yorum ekle)

```
import java.util.*; // java.util kütüphanesini kullanmak için

public class DiziSiralama {
  public static void ekranaBas(double[] d) {
     for (int i = 0 ; i < d.length ; i++) {
       System.out.println("d["+i+"] = " + d[i]);
     }
  }

  public static void main(String args[]) {
     double d[] = new double[9];
     d[0] = 2.45;     d[1] = 3.45 ; d[2] = 4.78;
     d[3] = 1.45;     d[4] = 15.12; d[5] = 1;
    d[6] = 9;  d[7] = 15.32 ; d[8] = 78.17;
```

```
        System.out.println("Karisik sirada");  ekranaBas(d);
        Arrays.sort( d ) ;
        System.out.println("Siralanmis Sirada"); ekranaBas(d);
    }
}
```

Uygulama sonucu aşağıdaki gibi olur:

```
Karisik sirada
d[0] = 2.45
d[1] = 3.45
d[2] = 4.78
d[3] = 1.45
d[4] = 15.12
d[5] = 1.0
d[6] = 9.0
d[7] = 15.32
d[8] = 78.17
Siralanmis Sirada
d[0] = 1.0
d[1] = 1.45
d[2] = 2.45
d[3] = 3.45
d[4] = 4.78
d[5] = 9.0
d[6] = 15.12
d[7] = 15.32
d[8] = 78.17
```

3.3.6. **Dizilerin Dizilere Kopyalanması**

Bir dizi tümden diğer bir diziye kopyalanabilir: (yorum ekle)

Örnek-3.35: *DizilerinKopyalanmasi.java* (yorum ekle)

```
public class DizilerinKopyalanmasi {

  public static void main(String args[]) {
     int[] dizi1 = { 1,2,3,4 };  // ilk dizi
     int[] dizi2 = { 100,90,78,45,40,30,20,10}; // daha geniş olan
2. dizi

          // kopyalama işlemi başlıyor
     // 0. indisinden dizi1 uzunluğu kadar kopyalama yap
     System.arraycopy(dizi1,0,dizi2,0,dizi1.length);
     for (int i = 0 ; i < dizi2.length ; i++) {
       System.out.println("dizi2["+i+"] = "+ dizi2[i] );
     }
  }
}
```

System sınıfının statik yordamı olan `arraycopy()` sayesinde `dizi1` `dizi2`'ye kopyalandı. Sonuç aşağıdaki gibi olur: (yorum ekle)

```
dizi2[0] = 1
dizi2[1] = 2
dizi2[2] = 3
dizi2[3] = 4
dizi2[4] = 40
dizi2[5] = 30
dizi2[6] = 20
dizi2[7] = 10
```

3.3.7. Çok Boyutlu Diziler

Çok boyutlu diziler, Java'da diğer programlama dillerinden farklıdır. Sonuçta dizinin tek türde olması gerekir; yani, dizi içerisinde *diziler (dizilerin içerisinde dizilerin içerisindeki diziler şeklinde de gidebilir...)* tanımlayabilirsiniz. (yorum ekle)

Gösterim-3.16:

```
int[][] t1 = {
   { 1, 2, 3, },
   { 4, 5, 6, },
};
```

Gösterim-3.16 'de ifade edildiği gibi iki boyutlu temel türden oluşmuş çok boyutlu dizi oluşturulabilir. Çok boyutlu dizileri oluşturmanın diğer bir yolu ise, (yorum ekle)

Gösterim-3.17:

```
int [] [] t1 = new int [3][4] ;
int [] [] t1 = new int [][4] ; // ! Hata !
```

Çok boyutlu dizileri bir uygulama üzerinde incelersek; (yorum ekle)

Örnek: *CokBoyutluDizilerOrnekBir.java* (yorum ekle)

```
public class CokBoyutluDizilerOrnekBir {
  public static void main(String args[]) {
     int ikiboyutlu[][] = new int[3][4] ;
     ikiboyutlu[0][0] = 45 ;
     ikiboyutlu[0][1] = 83 ;
     ikiboyutlu[0][2] = 11 ;
     ikiboyutlu[0][3] = 18 ;

     ikiboyutlu[1][0] = 17 ;
     ikiboyutlu[1][1] = 56 ;
```

```
      ikiboyutlu[1][2] = 26 ;
      ikiboyutlu[1][3] = 79 ;

      ikiboyutlu[2][0] = 3 ;
      ikiboyutlu[2][1] = 93 ;
      ikiboyutlu[2][2] = 43 ;
      ikiboyutlu[2][3] = 12 ;

      // ekrana yazdırıyoruz
      for (int i = 0 ; i<ikiboyutlu.length ; i++ ) {
        for (int j = 0 ; j < ikiboyutlu[i].length ; j++ ) {
          System.out.println(" ikiboyutlu["+i+"]["+j+"] = "
                                    + ikiboyutlu[i][j] );
        }
      }
    }
  }
```

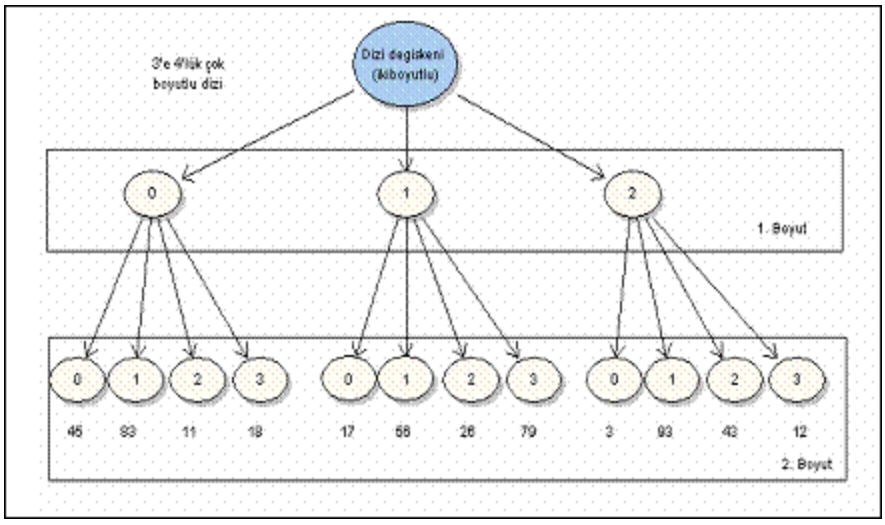
Verilen örnekte `int` türünde 3'e 4'lük (3x4) çok boyutlu dizi oluşturuldu; bu diziyi 3'e 4'lük bir matris gibi de düşünülebilir. Uygulama sonucu aşağıdaki gibi olur: (yorum ekle)

```
ikiboyutlu[0][0] =45
ikiboyutlu[0][1] =83
ikiboyutlu[0][2] =11
ikiboyutlu[0][3] =18
ikiboyutlu[1][0] =17
ikiboyutlu[1][1] =56
ikiboyutlu[1][2] =26
ikiboyutlu[1][3] =79
ikiboyutlu[2][0] =3
ikiboyutlu[2][1] =93
ikiboyutlu[2][2] =43
ikiboyutlu[2][3] =12
```

Uygulama sonucu matris gibi düşünülürse aşağıdaki gibi olur:

```
45 83 11 18
17 56 26 79
 3 93 43 12
```

Uygulama şekilsel olarak gösterilirse:



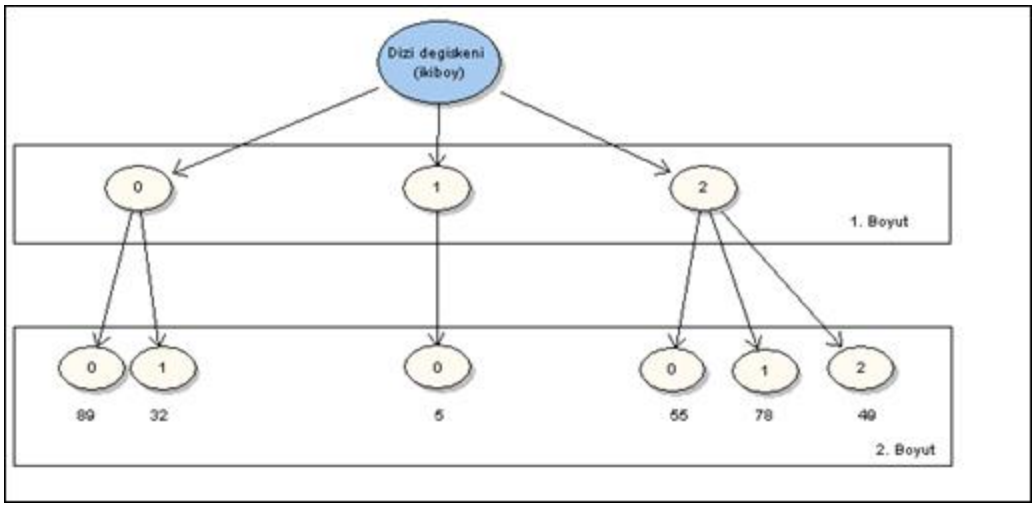
Şekil-3.9. Aynı elemana sahip çok boyutlu dizi

Dizilere bağlı diğer dizilerin aynı boyutta olma zorunluluğu yoktur. (yorum ekle)

Örnek-3.36: *CokBoyutluDiziler.java* (yorum ekle)

```
public class CokBoyutluDiziler {

  public static void main(String args[]) {
     int ikiboy[][] = new int[3][];

     ikiboy[0] = new int[2] ;
     ikiboy[1] = new int[1] ;
     ikiboy[2] = new int[3] ;

     ikiboy[0][0] = 89 ;
     ikiboy[0][1] = 32 ;

     ikiboy[1][0] = 5 ;

     // ikiboy[1][1] = 3 ; // ! Hata !

     ikiboy[2][0] = 55 ;
     ikiboy[2][1] = 78 ;
     ikiboy[2][2] = 49 ;
  }
}
```

Yukarıda verilen örnekte görüldüğü gibi, dizilere bağlı her farklı dizinin boyutları birbirinden farklı olmuştur. Şekil üzerinde ifade etmeye çalışırsak. (yorum ekle)



Şekil-3.10. Farklı elemana sahip çok boyutlu dizi

Çok boyutlu dizilerin içerisine sınıf tiplerinin yerleştirilmesi de olasıdır. Örneğin *String* sınıfı tipinde olan çok boyutlu bir dizi oluşturulabilir: (yorum ekle)

Örnek-3.37: *HayvanlarAlemi.java* (yorum ekle)

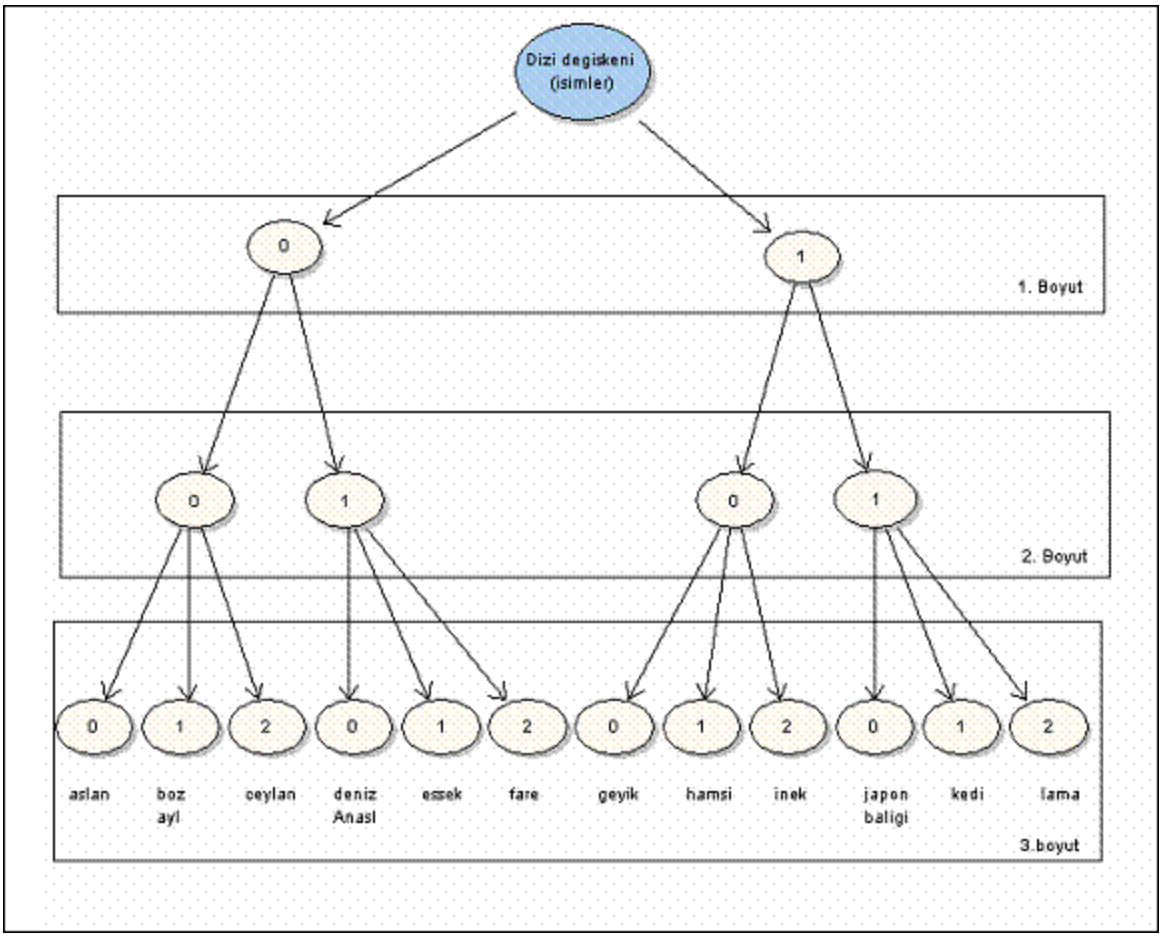
```
public class HayvanlarAlemi {

  String isimler[][][] ;

  public HayvanlarAlemi() {
     isimler = new String[2][2][3] ;
     veriAta();
  }

  public void veriAta() {
     isimler[0][0][0] = "aslan" ;
     isimler[0][0][1] = "boz AyI" ;
     isimler[0][0][2] = "ceylan";

     isimler[0][1][0] = "deniz AnasI" ;
     isimler[0][1][1] = "essek" ;
     isimler[0][1][2] = "fare" ;

     isimler[1][0][0] = "geyik" ;
     isimler[1][0][1] = "hamsi" ;
     isimler[1][0][2] = "inek" ;

     isimler[1][1][0] = "japon baligi" ;
     isimler[1][1][1] = "kedi" ;
     isimler[1][1][2] = "lama" ;

  ekranaBas() ;
  }
```

```
  public void ekranaBas() {
     for (int x = 0 ; x < isimler.length ; x++) {
       for (int y = 0 ; y < isimler[x].length ; y++) {
         for (int z = 0 ; z < isimler[x][y].length ; z ++) {
             System.out.println("isimler["+x+"]["+y+"]["+z+"] =" +
                                      isimler[x][y][z]);
       }
      }
     }
  }

  public static void main(String args[]) {
     HayvanlarAlemi ha = new HayvanlarAlemi();
  }
}
```

Bu örnekte *HayvanlarAlemi* nesnesinin oluşturulmasıyla olaylar tetiklinmiş olur. İster tek boyutlu olsun ister çok boyutlu olsun, diziler üzerinde işlem yapmak isteniyorsa, onların oluşturulması (`new` *anahtar kelimesi ile*) gerektiği daha önceden belirtilmişti... Bu örnekte olduğu gibi, dizi ilk oluşturulduğunda, dizi içerisindeki *String* tipindeki referansların ilk değeri `null`'dır. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
isimler[0][0][0] =aslan
isimler[0][0][1] =boz AyI
isimler[0][0][2] =ceylan
isimler[0][1][0] =deniz AnasI
isimler[0][1][1] =essek
isimler[0][1][2] =fare
isimler[1][0][0] =geyik
isimler[1][0][1] =hamsi
isimler[1][0][2] =inek
isimler[1][1][0] =japon baligi
isimler[1][1][1] =kedi
isimler[1][1][2] =lama
```

Oluşan olaylar şekilsel olarak gösterilmeye çalışılırsa:



Şekil-3.11. *String* sınıfı tipindeki çok boyutlu dizi

x