JAVA'DA PROGRAM DENETİMİ

VE OPERATÖRLER

Java programlama dilinde temel tipleri ve nesneleri yönlendirmek ve değiştirmek için operatörler kullanılır. Bu operatörler yapısı ve işlevleri bakımından C ve C++ programlama dillerinden miras alınmıştır; ancak bu miras üzerine kendisi de birçok şey eklemiştir. (yorum ekle)

2.1. Atamalar

Değer atamalar sağ taraftaki değerin (-ki bu bir sabit değer veya başka bir değişken olabilir) sol taraftaki değişkene atanması ile gerçekleşir. (yorum ekle)

Gösterim-2.1:

```
int a ;a=4 ;      // doğru bir atama4=a ;      // yanlış bir atama!
```

2.1.1. Temel Tiplerde Atama

Atama işlemi, temel (*primitive*) tipler için basittir. Temel tipler, değerleri doğrudan kendileri üzerlerinde tuttukları için, bir temel tipi diğerine atadığımız zaman değişen sadece içerikler olur. (yorum ekle)

Gösterim-2.2:

```
int a, b ;a=4 ;b=5 ;a=b ;
```

Sonuç olarak `a` ve `b` değişkenleri içerikleri aynı olur...

```
a=5, b=5
```

2.1.2. Nesneler ve Atamalar

Nesneler için atama işlemleri, temel tiplere göre biraz daha karmaşıktır. Nesneleri yönetmek için referanslar kullanılır; eğer, nesneler için bir atama işlemi söz konusu ise, akla gelmesi gereken ilk şey, bu nesnelere bağlı olan referansın gösterdiği hedeflerde bir değişiklik olacağıdır. (yorum ekle)

Örnek: *NesnelerdeAtama.java* (yorum ekle)

```
class Sayi {
    int i;
 }

public class NesnelerdeAtama {
   public static void main(String[] args) {
     Sayi s1 = new Sayi();
     Sayi s2 = new Sayi();
     s1.i = 9;
     s2.i = 47;
     System.out.println("1: s1.i: " + s1.i +", s2.i: " + s2.i);
     s1 = s2; // referanslar kopyalaniyor.. nesneler degil
     System.out.println("2: s1.i: " + s1.i +", s2.i: " + s2.i);
     s1.i = 27;
     System.out.println("3: s1.i: " + s1.i +", s2.i: " + s2.i);  }
}
```

Yukarıda verilen uygulama adım adım açıklanırsa: Önce 2 adet *Sayi* nesnesi oluşturulmaktadır; bunlar *Sayi* tipindeki referanslara bağlı durumdadırlar; **`s1`** ve **`s2`**... Bu referanslar artık 2 ayrı *Sayi* nesnesini göstermektedirler. Daha sonra `s1` referansının işaret ettiği *Sayi* nesnesinin **`i`** alanına 9 sayısını atandı; benzer şekilde `s2` referansının işaret ettiği *Sayi* nesnesinin **i** alanına da 47 sayısını atandı. Yapılan işlemlerin düzgün olup olmadıklarının görülmesi için ekrana yazdırıldığında aşağıdaki sonuç ile karşılaşır: (yorum ekle)

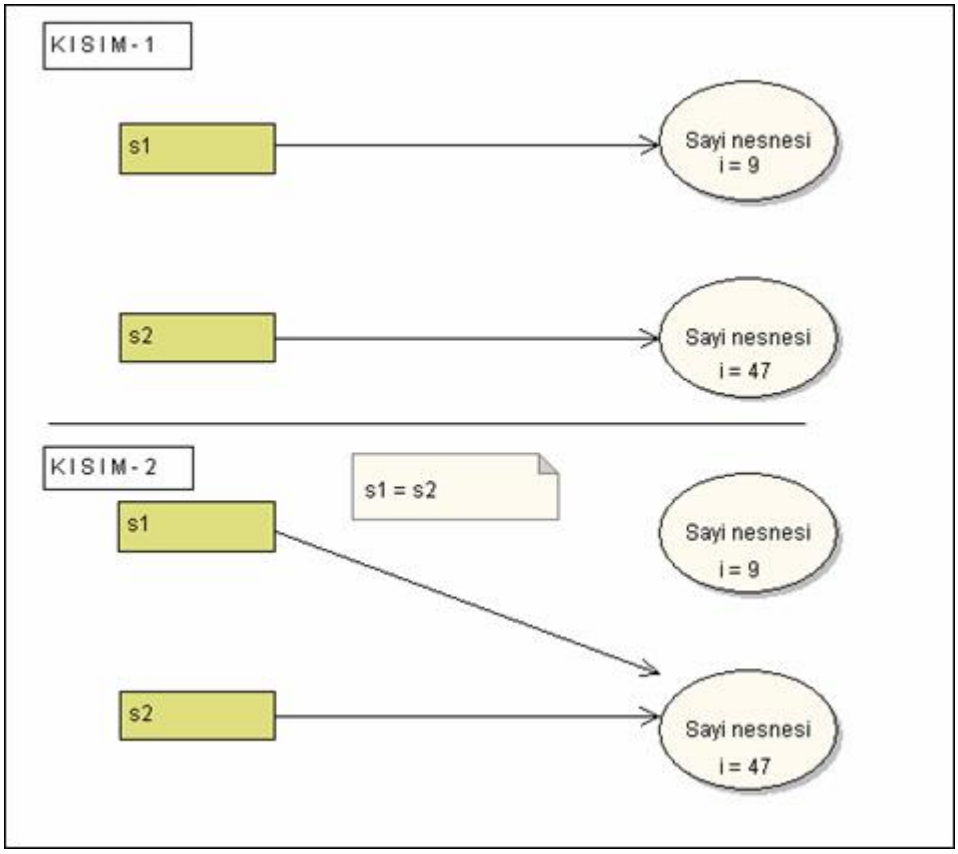
```
1: s1.i: 9, s2.i: 47
```

Şu ana kadar bir sorun olmadığını anlaşılıp rahatladıktan sonra önemli hamleyi yapıyoruz. (yorum ekle)

Gösterim-2.3:

```
s1 = s2 ;          // referanslar kopyalanıyor... nesneler değil
```

Burada gerçekleşen olay şudur; `s1` artık `s2`'nin işaret ettiği nesneyi göstermektedir. Şekil-2.1, verilen bu örneğin daha iyi anlaşılmasına yardımcı olabilir. Kısım-1 durumun, `s2`'nin `s1`'e atanmadan önceki halini göstermektedir. Kısım-2 ise `s2`'nin `s1`'eatandıktan sonraki halini göstermektedir. (yorum ekle)



Şekil-2.1. Nesnelerde atama ve referans değişikliği

Kalınan yerden devam edilirse, şimdi `s1` ve `s2`'nin değerlerini ekrana yazdırılırsa, `s1.i` ve `s2.i` alanları aynı içeriği taşıdığı görülür. (yorum ekle)

```
2: s1.i: 47, s2.i: 47
```

Bunun nedeni ise bu iki referansın (`s1` ve `s2`) aynı nesneyi göstermeleridir. Son olarak `s1` referansının işaret ettiği nesnenin `i` alanı değiştirilip ekrana yazdırıldığında... (yorum ekle)

```
3: s1.i: 27, s2.i: 27
```

Görüldüğü gibi `s2.i` alanının da değeri değişmiş oldu; nedeni ise yine `s1` ve `s2` _referanslarının aynı nesneyi göstermeleridir. (yorum ekle)

Peki, **s1** referansının daha önceden işaret etmiş olduğu *Sayi* nesnesine ne olacaktır? Cevap vermek için henüz erken ama yinede söylenirse, bu nesne kullanılmayacağından dolayı çöp haline gelecektir ve çöp toplayıcısı (*Garbage Collector*) tarafından temizlenecektir. Görüldüğü gibi tasarımcının nesne temizliği konusunda endişeye kapılmasına gerek yoktur. Çöp toplayıcısını ilerleyen bölümlerde daha ayrıntılı ele alınacaktır. (yorum ekle)

Bu örneğimizde `s1` referansının `s2`'nin işaret etmiş olduğu nesneyi göstermesini istemeyip yalnızca `s2.i` alanı değerinin `s1.i` alanı değerine atanmasını istenmiş olsaydı, aşağıdaki gibi yazılması yeterli olacaktı... (yorum ekle)

Gösterim-2.4:

```
s1.i = s2.i;
```

Bu ifade referansların gösterdikleri nesnelerde herhangi bir değişiklik yapmayacaktır; değişen sadece `s1.i` alanının değeri olacaktır. (yorum ekle)

2.2. **Yordamların (*Method*) Çağrılması**

Yordamların parametre kabul ettiklerini ve bu parametreleri alarak işlemler gerçekleştirdiğini biliyoruz. Peki yordamlara parametre olarak neler gitmektedir? Nesnelerin kendisi mi? Yoksa nesnelere ait referanslar mı? (yorum ekle)

Örnek: *Pas.java* (yorum ekle)

```
class Harf {
   char c;
}

public class Pas {
  static void f(Harf h) {
    // Harf nesnesine yeni bir referans bağlandı (h), yoksa oluşturulan Harf nesnesinin

     //veya  yeni bir Harf nesnesinin bu yordama gönderilmesi gibi birşey söz konusu
    // değildir.
    h.c = 'z';
  }

   public static void main(String[] args) {
     Harf x = new Harf(); // Harf nesnesini oluşturuluyor.
     x.c = 'a';           // Harf nesnesinin c alanına değer atandı

     System.out.println("1: x.c: " + x.c);     f(x); //
dikkat
     System.out.println("2: x.c: " + x.c);
   }
}
```

Yukarıda verilen örnekte *Harf* ve *Pas* olarak adlandırılan 2 adet sınıf bulunmaktadır. *Pas* sınıfı `public` olduğu için fiziksel dosyanın ismi *Pas.java*'dır. Bu kadar ön bilgiden sonra program açıklamasına geçilebilir: İlk olarak *Harf* nesnesi oluşturuluyor ve*Harf* nesnesin `char` tipinde olan `c` alanına `'a'` karakteri atanıyor. Yapılan işlem ekrana yazdırıldıktan sonra *Harf* nesnesine bağlanmış olan *Harf* sınıfı tipindeki **x** referansı `f()` yordamına gönderiliyor; sonra, `f()` yordamı içerisinde daha önceden oluşturulan *Harf* nesnesinin `char` tipinde olan `c` alanına `'z'` karakteri atanıyor. Bilgiler yeniden ekrana yazdırıldığında *Harf* nesnesinin `char` tipinde olan **c** alanındaki değerin değişmiş olduğu görülür. Burada yapılan işlem, kesinlikle, *Harf* nesnesinin yer değiştirmesi değildir; nesnenin bellekteki yeri her zaman sabittir... Yalnızca `f()` yordamı içerisinde *Harf* nesnesine kısa süreli olarak başka bir *Harf* sınıfı tipindeki bir referansın işaret etmiş olmasıdır (böylece *Harf* nesnesine toplam iki referans bağlı olmaktadır biri **x** diğeri ise **h**). Yordamın sonuna gelindiğinde ise **h** referansı geçerlilik alanı bitmektedir; ancak, bu kısa süre içerisinde *Harf* nesnesinin `char` tipinde olan `c` alanını değiştirilmiştir. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
1: x.c: a2: x.c: z
```

Yordam çağrımları temel tipler için biraz daha açıktır. (yorum ekle)

Örnek: *IlkelPas.java* (yorum ekle)

```
public class IlkelPas {

 static void f(double a) {
    System.out.println(a + " gonderildi");    a = 10 ;
    System.out.println("gonderilen parametrenin degeri
10'a"
                                                        +
"esitlendi");
 }
  public static void main(String[] args) {
      double a = 5 ;
      f(a);
      System.out.println("a --> " + a );
  }
}
```

Yukarıdaki uygulamada `double` tipindeki **`a`** değişkenine 5 değeri atanmaktadır; daha sonra bu değişkenimiz `double` tipinde parametre kabul eden **`f()`** yordamına gönderilmektedir. **`f()`** yordamı içerisinde `a=10` ifadesi, gerçekte, `main()` yordamı içerisindeki `double` tipindeki **`a`** değişkeni ile hiç bir ilgisi yoktur. Bu iki değişken birbirlerinden tamamen farklıdırlar. Sonuç olarak, temel tipler değerleri kendi üzerlerinde taşırlar. Yordamlara gönderilen parametreler yerel değişkenler gibidir. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
 5.0 gonderildigonderilen parametrenin degeri 10'a
esitlendia --> 5.0
```

2.3. Java **Operatörleri**

Operatörler programlama dillerinin en temel işlem yapma yeteneğine sahip simgesel isimlerdir. Tüm programlama dillerinde önemli bir yere sahip olup bir işlem operatör ile gerçekleştirilebiliyorsa "en hızlı ve verimli ancak bu şekilde yapılır" denilebilir. Yalnızca bir operatör ile gerçekleştirilemeyen işlemler, ya bir grup operatörün biraraya getirilmesiyle ya da o işlemi gerçekleştirecek bir yordam (*method*) yazılmasıyla sağlanır. Java dili oldukça zengin ve esnek operatör kümesine sahiptir; örneğin matematiksel, mantıksal, koşulsal, bit düzeyinde vs. gibi birçok operatör kümesi vardır; ve, bunlar içerisinde çeşitli operatörler bulunmaktadır: (yorum ekle)

§ Aritmetik Operatör
§ İlişkisel Operatör
§ Mantıksal Operatörler
§ Bit düzeyinde (*bitwise*) Operatörler

Operatörler, genel olarak, üzerinde işlem yaptığı değişken/sabit sayısına göre tekli operatör (*unary operator*) veya ikili operatör (*binary operator*) olarak sınıflanmaktadır; 3 adet değişken/sabite ihtiyaç duyan operatörlere de üçlü operatör denilir. Yani, tek değişken/sabit üzerinde işlem yapan operatör, iki değişken/sabit üzerinde işlem yapan operatör gibi... Tekli operatörler hem ön-ek (*prefix*) hem de son-ek (*postfix*) işlemlerini desteklerler. Ön-ek'ten kastedilen anlam operatörün değişkenden önce gelmesi, son-ke'ta de operatörden sonra gelmesidir... (yorum ekle)

```
à operatör değişken   //ön-ek ifadesi
```

Son-ek işlemlerine örnek olarak,

```
à değişken operatör   // son-ek ifadesi
```

İkili operatörlerde operatör simgesi ara-ek (*infix*) olarak iki değişkenin ortasında bulunur: (yorum ekle)

```
à değişken1 operatör değişken2   //ara-ek
```

Üçlü operatörlerde ara-ek (*infix*) işlemlerde kullanılır. Java'da üçlü operatör bir tanedir. (yorum ekle)

```
à değişken1 ? değişken2 : değişken3   //ara ek
```

2.3.1. **Aritmetik Operatörler**

Java programlama dili kayan-noktalı (*floating-point*) ve tamsayılar için birçok aritmetik işlemi destekleyen çeşitli operatörlere sahiptir. Bu işlemler toplama operatörü (`+`), çıkartma operatörü (`-`), çarpma operatörü (`*`), bölme operatörü (`/`) ve son olarak da artık bölme (`%`) operatörüdür. (yorum ekle)

Tablo-2.1. Java'da aritmetik operatörler (yorum ekle)

Operatör	Kullanılış	Açıklama
`+`	`değişken1 + değişken2`	`değişken1` ile `değişken2`'yi toplar
`-`	`değişken1 - değişken2`	`değişken1` ile `değişken2`'yi çıkarır
`*`	`değişken1 * değişken2`	`değişken1` ile `değişken2`'yi çarpar
`/`	`değişken1 / değişken2`	`değişken1`, `değişken2` tarafından bölünür
`%`	`değişken1 % değişken2`	`değişken1`'in `değişken2` tarafından bölümünden kalan hesaplanır.

Verilenler bir Java uygulamasında aşağıdaki gibi gösterilebilir:

Örnek: *AritmetikOrnek.java* (yorum ekle)

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

```
   //  Değişkenler atanan değerler
   int a = 57, b = 42;
   double c = 27.475,  d = 7.22;
   System.out.println("Degisken Degerleri...");
   System.out.println(" a = " + a);
   System.out.println(" b = " + b);
   System.out.println(" c = " + c);
   System.out.println(" d = " + d);
   // Sayıları  topluyoruz
   System.out.println("Toplama...");
   System.out.println(" a + b = " + (a + b));
   System.out.println(" c + d = " + (c + d));
   //  Sayıları çıkartıyoruz
   System.out.println("Cikartma...");
   System.out.println(" a - b = " + (a - b));
   System.out.println(" c - d = " + (c - d));
   //  Sayıları Çarpıyoruz.
   System.out.println("Carpma...");
   System.out.println(" a * b = " + (a * b));
   System.out.println(" c * d = " + (c * d));
   //  Sayıları bölüyoruz
   System.out.println("Bolme...");
   System.out.println(" a / b = " + (a / b));
   System.out.println(" c / d = " + (c / d));


   //  Bölme işlemlerinden kalan sayıyı hesaplıyoruz
   System.out.println("Kalan sayiyi hesaplama...");
   System.out.println(" a % b = " + (a % b));
   System.out.println(" c % d = " + (c % d));
   //  double ve int tiplerini karışık şekilde
   //kullanıyoruz.
   System.out.println("Karisik tipler...");
   System.out.println(" b + d = " + (b + d));
   System.out.println(" a * c = " + (a * c));
 }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
Degisken Degerleri...
a = 57
b = 42
c = 27.475
d = 7.22
Toplama...
a + b = 99
c + d = 34.695
Cikartma...
a - b = 15
c - d = 20.255000000000003
Carpma...
a * b = 2394
c * d = 198.36950000000002
Bolme...
a / b = 1
```

```
c / d = 3.805401662049862
Kalan sayiyi hesaplama...
a % b = 15
c % d = 5.815000000000002
Karisik tipler...
b + d = 49.22
a * c = 1566.075
```

Verilen örnek dikkatlice incelenecek olursa, tamsayı ile kayan noktalı sayılar bir operatörün değişkenleri olursa sonuç kayan noktalı sayı olmaktadır. Bu işlemde tamsayı, kendiliğinden kayan noktalı sayıya çevrilir. Aşağıdaki tabloda böylesi dönüştürme işleminde izlenen yol gösterilmiştir: (yorum ekle)

Tablo-2.2. Operatörlerin veri tipini etkilemesi/dönüştürmesi (yorum ekle)

Sonuç Veri Tipi	Değişkenlerin Veri Tipleri
`long`	Değişkelerin `float` veya `double` tipinden farklı olması ve en az bir değişkenin `long` tipinde olması
`int`	Değişkelerin `float` veya `double` tipinden farklı olması ve değişkenlerin `long` tipinden farklı olması
`double`	En az bir değişkenin `double` tipinde olması
`float`	Değişkenlerin hiçbirinin `double` tipinde olmaması ve değişkenlerden en az birinin `float` tipinde olması

+ ve – operatörleri, aynı zamanda, karakter tipindeki verileri sayısal tipe dönüştürme misyonları da vardır. (yorum ekle)

Tablo-2.3. Toplama ve Çıkartma operatörlerinin tip etkilemesi (yorum ekle)

Operatör	Kullanılış Şekli	Açıklama
+	`+ değişken`	Eğer değişken `char`, `byte` veya `short` tipinde ise `int` tipine dönüştürür
–	`- değişken`	Değişkenin değerini eksi yapar (-1 ile çarpar).

Anlaşılması açısından kısa bir uygulama örneği yazılırsa,

Örnek: *OperatorTest.java* (yorum ekle)

```
public class OperatorTest {
  public static void main(String args[] ) {
    char kr = 'a' ;
    int b = +kr ;       // otomatik olarak int temel tipine çevrildi
    int c = -b ;        // değeri eksi yaptı
    System.out.println("kr = " + kr );
    System.out.println("b = " + b );
    System.out.println("c = " + c );
  }
}
```

`char` temel (*primitive*) bir tiptir; ve, bu tiplere değer atanırken veri tek tırnak içerisinde verilmelidir. Bu örnekte girilen değer '**a**' harfidir. Daha sonra + operatörü kullanılarak `char` değerini `int` tipine dönüştürülüyor ve son olarak ta bu `int` değeri -operatörüyle eksi hale getiriliyor. Uygulamanın sonucu aşağıdaki gibi olur: (yorum ekle)

```
kr = ab = 97c = -97
```

Dönüştürme *(Casting)* İşlemi

Temel bir veri tipi diğer bir temel tipe dönüştürebilir; fakat, oluşacak değer kayıplarından tasarımcı sorumludur. (yorum ekle)

Örnek: *IlkelDonusum.java* (yorum ekle)

```
public class IlkelDonusum {
 public static void main(String args[]) {
   int a = 5;
   double b = (double) a;
   double x = 4.15 ;


   int y = (int) x ;
   long z = (long) y ;
   System.out.println("b = " + b + " y = " + y + " z = " + z);
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
b = 5.0 y = 4 z = 4
```

Bir Arttırma ve Azaltma

Java dilinde, aynı C dilinde olduğu gibi, birçok kısaltmalar vardır; bunlar yaşamı bazen daha güzel, bazen de çekilmez kılabilmektedir... İşe yarayan kısaltmalardan iki tanesi arttırma ve azaltma operatörleridir; bu operatörler değişkenin içeriğini bir arttırmak veya azaltmak için kullanılır. (yorum ekle)
Arttırma ve azaltma operatörleri iki farklı konumda kullanılabilirler: Birincisi ön-ek (*prefix*) - ki bu (--) veya (++) operatörünün, kullanılan değişkenin önüne gelmesi anlamını taşır, diğeri ise son-ek'dir (*postfix*), bu da (--) veya (++) operatörünün değişkenin sonuna gelmesi anlamına gelir. Peki bu operatörlerin değişkenin başına gelmesi ile sonuna gelmesi arasında ne gibi farklar vardır? (yorum ekle)

Tablo-2.4. Artırma ve azaltma operatörü (yorum ekle)

Operatör	Kullanılış Şekli	Açıklama
`++`	`değişken++`	Önce değişkenin değerini hesaplar sonra değişkenin değerini bir arttırır.
`++`	`++değişken`	Önce değişkenin değerini arttırır sonra değişkenin değerini hesaplar.
`--`	`değişken--`	Önce değişkenin değerini hesaplar sonra değişkenin değerini bir azaltır.
`--`	`--değişken`	Önce değişkenin değerini azaltır sonra değişkenin değerini hesaplar.

Örneğin (++a veya --a) şeklinde verilmesinde, önce matematiksel toplama/çıkartma işlemi gerçekleşir; daha sonra değer üretilir. Ancak, (a++ veya a--) şeklinde verilmesi durumunda ise, önce değer üretilir; daha sonra matematiksel toplama/çıkartma işlemi gerçekleşir. Aşağıda verilen kısa program bunu güzel bir şekilde ifade etmektedir: (yorum ekle)

Örnek: *OtomatikArtveAz.java* (yorum ekle)

```
public class OtomatikArtveAz {

 static void ekranaYaz(String s) {
   System.out.println(s);
 }

 public static void main(String[] args) {

   int i = 1;
   ekranaYaz("i : " + i);
   ekranaYaz("++i : " + ++i);     // önek artırım
   ekranaYaz("i++ : " + i++);     // sonek artırım
   ekranaYaz("i : " + i);
   ekranaYaz("--i : " + --i);     // önek azaltma
   ekranaYaz("i-- : " + i--);     // sonek azaltma
   ekranaYaz("i : " + i);
 }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
i : 1++i : 2i++ : 2i : 3--i : 2i-- : 2i : 1
```

2.3.2. **İlişkisel Operatörler**

İlişkisel operatörler iki değeri karşılaştırarak bunlar arasındaki mantıksal ilişkiyi belirlemeye yararlar. Örneğin iki değer birbirine eşit değilse, == operatörüyle bu ilişki sonucu yanlış **(false)** olur. Tablo-2.5'de ilişkisel operatörler ve anlamları verilmiştir: (yorum ekle)

Tablo-2.5. İlişkisel operatörler

Operatör	Kullanılış Şekli	True değeri döner eğer ki.....
>	değişken1 > değişken2	değişken1, değişken2'den büyükse
>=	değişken1 >= değişken2	değişken1, değişken2'den büyükse veya eşitse
<	değişken1 < değişken2	değişken1, değişken2'den küçükse
<=	değişken1 <= değişken2	değişken1, değişken2'den küçükse veya eşitse
==	değişken1 == değişken2	değişken1, değişken2'ye eşitse
!=	değişken1 != değişken2	değişken1, değişken2'ye eşit değilse

İlişkisel operatörlerin kullanılması bir Java uygulaması üzerinde gösterilirse,

Örnek: *IliskiselDeneme.java* (yorum ekle)

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

   // değişken bildirimleri
   int i = 37, j = 42, k = 42;

   System.out.println("Degisken degerleri...");
   System.out.println(" i = " + i);
   System.out.println(" j = " + j);
   System.out.println(" k = " + k);

   //Büyüktür
```

```
    System.out.println("Buyuktur...");
    System.out.println(" i > j = " + (i > j));  //false - i, j  den
küçüktür
    System.out.println(" j > i = " + (j > i));  //true - j, i den Büyüktür
    System.out.println(" k > j = " + (k > j));  //false - k, j ye eşit

    //Büyüktür veya eşittir
    System.out.println("Buyuktur veya esittir...");
    System.out.println(" i >= j = " + (i >= j)); //false - i, j  den
küçüktür
    System.out.println(" j >= i = " + (j >= i)); //true - j, i den büyüktür
    System.out.println(" k >= j = " + (k >= j)); //true - k, j'ye eşit

    //Küçüktür
    System.out.println("Kucuktur...");
    System.out.println(" i < j = " + (i < j));    //true - i, j'den
küçüktür
    System.out.println(" j < i = " + (j < i));    //false - j, i' den
büyüktür
    System.out.println(" k < j = " + (k < j));    //false - k, j'ye eşit

    //Küçüktür veya eşittir
    System.out.println("Kucuktur veya esittir...");
    System.out.println(" i <= j = " + (i <= j)); //true - i, j'den küçüktür
    System.out.println(" j <= i = " + (j <= i)); //false - j, i  den
büyüktür
    System.out.println(" k <= j = " + (k <= j)); //true - k, j  ye eşit


        //Eşittir
    System.out.println("Esittir...");
    System.out.println("i == j =" + (i == j)); //false - i, j'den küçüktür
    System.out.println(" k == j = " + (k == j)); //true - k, j'ye eşit

    //Eşit değil
    System.out.println("Esit degil...");
    System.out.println(" i != j = " + (i != j)); //true - i, den küçüktür
    System.out.println(" k != j = " + (k != j)); //false - k, ye eşit
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
Degisken degerleri...
 i = 37
 j = 42
 k = 42
Buyuktur...
i > j = false
j > i = true
k > j = false
Buyuktur veya esittir...
i >= j = false
j >= i = true
k >= j = true
Kucuktur...
i < j = true
j < i = false
k < j = false
Kucuktur veya esittir...
i <= j = true
j <= i = false
k <= j = true
Equal to...
 i == j = false
```

k == j = true
Not equal to...
i != j = true
k != j = false

2.3.3. **Mantıksal Operatörler**

Mantıksal operatörler birden çok karşılaştırma işlemini birleştirip tek bir koşul ifadesi haline getirilmesi için kullanılır. Örneğin bir koşul sağlanması için hem **a**'nın 10'dan büyük olması hem de **b**'nin 55 küçük olması gerekiyorsa iki karşılaştırma koşulu VE mantıksal operatörüyle birleştirilip `a>10 && b<55` şeklinde yazılabilir. Aşağıda, Tablo-2.6'da mantıksal operatörlerin listesi ve anlamaları verilmiştir: (yorum ekle)

Tablo-2.6. İlişkisel ve koşul operatörlerinin kullanımı (yorum ekle)

Operatör	Kullanılış Şekli	İşlevi/Anlamı
&&	**değişken1 && değişken2**	VE operatörü
||	**değişken1 || değişken2**	VEYA operatörü
^	**değişken1 ^ değişken2**	YA DA operatörü
!	**! değişken**	DEĞİLini alma operatörü

Aşağıda birden çok ilişkisel karşılaştırmanın mantıksal operatörler birleştirilmesi için örnekler verilmiştir (`m`, `n`, `r` ve `z` değişken adları):

```
m>10 && m<55
(m>0 && r<55) && z==10
a>10 && b<55 || r<99
```

Not: Mantıksal operatörlerden && (VE), || (VEYA) operatörleri sırasıyla tek karakterli olarak ta kullanılabilmektedir. Örneğin **a&&b** şeklinde bir ifade **a&b** şeklinde de yazılabilinir. Aralarında fark, eğer tek karakterli, yani `&` veya `|` şeklinde ise, operatörün heriki yanındaki işlemler/karşılaştırmalar kesinkes yapılır. Ancak, çift karakterli kullanılırsa, yani `&&` veya `||` şeklinde ise, işleme soldan başlanır; eğer tüm ifade bitmeden kesin sonuca ulaşılırsa ifadenin geri kalan kısmı gözardı edilir. Örneğin VE işleminde sol taraf yanlış (*false*) ise sonuç kesinkes yanlış olacaktır ve ifadenin sağına bakmaya gerek yoktur. (yorum ekle)
Mantıksal operatörlerin doğruluk tabloları bit düzeyinde operatörler ayrıtında Tablolarda verilmiştir.

-

Örnek: *KosulOp.java* (yorum ekle)

```
public class KosulOp {
  public static void main( String args[] ) {
    int a = 2 ;
    int b = 3 ;
    int c = 6 ;
    int d = 1 ;
    /*  (a < b) = bu ifadenin doğru (true) olduğunu biliyoruz
            (c < d) = bu ifadenin yanlış (false) olduğunu biliyoruz   */

    System.out.println(" (a<b)&&(c<d) --> " + ((a<b)&&(c<d)) );
    System.out.println(" (a<b)||(c<d) --> " + ((a<b)||(c<d)) );
    System.out.println(" ! (a<b) --> " + ( ! (a<b)) );
    System.out.println(" (a<b)&(c<d) --> " + ((a<b)&(c<d)) );
```

```
    System.out.println(" (a<b)|(c<d) --> " + ((a<b)|(c<d)) );
    System.out.println(" (a<b)^(c<d) --> " + ((a<b)^(c<d)) );
  }
}
```

Uygulamamızın çıktısı aşağıdaki gibidir.

```
(a < b) && (c < d) --> false
(a < b) || (c < d) --> true
! (a < b) --> false
(a < b) & (c < d) --> false
(a < b) | (c < d) --> true
(a < b) ^ (c < d) --> true
```

2.3.4. bit Düzeyinde (*bitwise*) Operatörler

bit düzeyinde operatörler, adı üzerinde, değişkenlerin/sabitlerin tuttuğu değerlerin doğrudan ikili kodlarının bitleri üzerinde işlem yaparlar. Örneğin 6 sayısının iki karşılığı 0110'dır. Bu değer sonuç 4 bit üzerinde kalmak koşuluyla bir sola kaydırılırsa 1100, tümleyeni alınırsa 1001 olur. Görüldüğü gibi bit düzeyinde operatörler veriyi bit düzeyde etkilemektedir. bit düzeyinde operatörlerin listesi Tablo-2.8.'da ve doğruluk tabloları da sırasıyla Tablo-2.9, 2-10 ve 2-11'de gösterilmişlerdir. (yorum ekle)

Tablo-2.7. bit düzeyinde operatörler (yorum ekle)

Operatör	Kullanılış Şekli	Açıklama
&	`değişken1 & değişken2`	bit düzeyinde VE
|	`değişken1 | değişken2`	bit düzeyinde VEYA
^	`değişken1 ^ değişken2`	bit düzeyinde YA DA
~	`~değişken`	bit düzeyinde tümleme
>>	`değişken1 >> değişken2`	bit düzeyinde sağa öteleme
<<	`değişken1 << değişken2`	bit düzeyinde sağa öteleme
>>>	`değişken1 >>> değişken2`	bit düzeyinde sağa öteleme (unsigned) ?????

§ VE (AND) İşlemi/Operatörü

VE işleminde heriki taraf ta doğru (*true*) ise sonuç doğru diğer durumlarda sonuç yanlış (*false*) olur. VE işlemi doğruluk tablosu Tablo-2.9'da verildiği gibidir. bit düzeyinde VE işleminde operatörün hemen sağ ve sol yanında bulunan parametrelerin ikili karşılıkları bit bit VE işlemine sokulur; işlem en sağdaki bitten başlantrı. Örneğin 10 ve 9 sayılarının ikili karşılıkları sırasıyla `1010` ve `1011`'dir. Herikisi bit düzeyinde VE işlemine sokulursa, sonuç 1000 çıkar: (yorum ekle)

```
1010     ð  10 & 1001     ð  9---------   1000
```

Tablo-2.8. VE (AND) işlemi doğruluk tablosu (yorum ekle)

değişken1	değişken2	Sonuç

0	0	0
0	1	0
1	0	0
1	1	1

§ VEYA (OR) İşlemi/Operatörü

VEYA işleminde heriki taraftan birinin doğru (*true*) olması sonucun doğru çıkması için yeterlidir. VEYA işlemi doğruluk tablosu Tablo-2.10'da verildiği gibidir. bit düzeyinde VEYA işleminde operatörün hemen sağ ve sol yanında bulunan parametrelerin ikili karşılıkları bit bit VEYA işlemine sokulur; işlem en sağdaki bitten başlatılır. Örneğin 10 ve 9 sayılarının ikili karşılıkları sırasıyla `1010` ve `1011`'dir. Herikisi bit düzeyinde VE işlemine sokulursa, sonuç 1011 çıkar: (yorum ekle)

```
1010     ð  10 | 1001     ð  9---------   1011
```

Tablo-2.9. VEYA (OR) işlemi doğruluk tablosu (yorum ekle)

değiken1	değişken2	Sonuç
0	0	0
0	1	1
1	0	1
1	1	1

§ YA DA *(Exclusive Or)* İşlemi/Operatörü

YA DA işleminde heriki taraftan yalnızca birinin doğru (*true*) olması sonucu doğru yapar; heriki tarafın aynı olması durumunda sonuç yanlış (*false*) çıkar. YA DA işlemi doğruluk tablosu Tablo-2.10'da verildiği gibidir. (yorum ekle)

Tablo-2.10. Dışlayan YA DA işlemi doğruluk tablosu (yorum ekle)

Değişken1	değişken2	Sonuç
0	0	0
0	1	1
1	0	1
1	1	0

VE ve VEYA işlemlerinde kullanılan örnek sayıları YA DA için de gösterilirse, sonuç aşağıdaki gibi olur:

```
1010     ð  10 ^ 1001     ð  9---------   0011
```

TÜMLEME (NOT) İşlemi/Operatörü

`a` bir değişken adı ise **`~a`** ifadesi açılımı : `~a` = (-`a`) - 1, yani, ~10= (-10) -1 = -11 sonucunu verir. (yorum ekle)

Örnek: *BitwiseOrnek2.java* (yorum ekle)

```
public class BitwiseOrnek2 {

 public static void main( String args[] ) {
   int a = 10, b = 9, c = 8 ;
   System.out.println(" (a & b) --> " + (a & b) );
   System.out.println(" (a | b) --> " + (a | b ) );
```

```
    System.out.println(" (a ^ b) --> " + (a ^ b) );

    System.out.println(" ( ~a ) --> " + ( ~a ) );
    System.out.println(" ( ~b ) --> " + ( ~b ) );
    System.out.println(" ( ~c ) --> " + ( ~c ) );
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
(a & b) --> 8
(a | b) --> 11
(a ^ b) --> 3
( ~a ) --> -11
( ~b ) --> -10
( ~c ) --> -9
```

VE, VEYA ve YA DA (*Exclusive Or*) operatörleri birden çok mantıksal sonuç içeren ifadelerde kullanılabilir! (yorum ekle)

Öteleme (*Shift*) Operatörleri

bit düzeyinde işlem yapan bir grup operatörün adı öteleme operatörleri olarak adlandırılırlar; bunlar >>, >> ve >>> simgeleriyle gösterilmektedir. Öteleme operatörleri veri üzerindeki bitlerin sağa veya sola kaydırılması amacıyla kullanılır. (yorum ekle)
Aşağıdaki örneğimiz bu operatörlerin Java uygulamalarında nasıl kullanılacaklarına ait bir fikir verebilir. (yorum ekle)

Örnek: *Bitwise.java* (yorum ekle)

```
public class Bitwise {
 public static void main( String args[] ) {
   int a = 9 ;
   System.out.println(" (a >> 1) -->" + (a >> 1) );
   System.out.println(" (a >> 2) -->" + (a >> 2) );
   System.out.println(" (a << 1) -->" + (a << 1) );
   System.out.println(" (a << 2) -->" + (a << 2) );
   System.out.println(" (a >>> 2) -->" + (a >>> 2) );
 }
}
```

Verilen örnekte a değişkenine 9 sayısı atanmıştır; bu sayının ikili karşılığı aşağıdaki gibi bulunur:

$$9 = (1001)_2 = 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0$$

Yani, 9_{10} sayısının ikili tabandaki karşılığı 1001 olmaktadır. Buna göre a değişkeni üzerinde öteleme operatörünün etkisi aşağıda açıklandığı gibi olur: (yorum ekle)

(a >> 1) şeklinde ifade ile, 9 sayısı ikili karşılığı olan 1001 bitleri sağa doğru 1 basamak kaydırılır; _100, boşlan yere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 0100 elde edilir ve bunun ondalık karşılığı 4 çıkar. (yorum ekle)

(a >> 2) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sağa doğru 2 basamak kaydırılır; __10, boşalan yerlere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 0010 elde edilir ve bunun ondalık karşılığı 2 çıkar. (yorum ekle)

(a << 1) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sola doğru 1 basamak kaydırılır; 1001_, boşalan yere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 10010 elde edilir ve bunun ondalık karşılığı 18 çıkar. (yorum ekle)

(a << 2) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sola doğru 2 basamak kaydırılır; 1001__, boşalan yerlere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 100100 elde edilir ve bunun ondalık karşılığı 36 çıkar. (yorum ekle)

(a >>> 2) şeklinde verilen ifadenin (a >> 2) ile arasında sonuç olarak bir fark yoktur, sonuç olarak yine 2 elde edilir. ">>>" operatörü işaretsiz (unsigned) sağa doğru kaydırıp yapar. (yorum ekle)

Eğer **`char`**, **byte,** veya **short** tiplerinde kaydırım işlemi yapacaksanız bu tipler ilk önce **int** tipine dönüştürülürler. Eğer **long** tipinde kaydırma işlemi gerçekleştiriyorsanız o zaman yine **long** tipinde bir sonuç elde ederseniz. (yorum ekle)

Uygulamanın sonucu aşağıdaki gibi olur:.

```
(a >> 1) -->4
(a >> 2) -->2
(a << 1) -->18
(a << 2) -->36
(a >>> 2) -->2
```

bit düzeyinde operatörleri tamsayı veriler üzerinde uygulamak anlamlıdır. (yorum ekle)

2.3.5. **Atama Operatörleri**

Atama operatörü en temel operatördür denilebilir; atama işlemi, bir değeri veya değişkenini içeriğini bir başka değişkene yerleştirmektir. Hemen hem tüm programlama dillerinde atama operatörü olarak = simgesi kullanılır; yalnızca *Pascal* ve benzeri dillerde := karakter çifti kullanılır. (yorum ekle)

Örnek: *EnBuyukSayilar.java* (yorum ekle)

```
public class EnBuyukSayilar {

  public static void ekranaBas(String deger) {
      System.out.println(deger);
  }
  public static void main( String args[] ) {

      // tamsayılar
      byte enbuyukByte = Byte.MAX_VALUE;
      short enbuyukShort = Short.MAX_VALUE;
      int enbuyukInteger = Integer.MAX_VALUE;
      long enbuyukLong = Long.MAX_VALUE;

      ekranaBas("enbuyukByte-->" + enbuyukByte );
      ekranaBas("enbuyukShort-->" + enbuyukShort );
      ekranaBas("enbuyukInteger-->" + enbuyukInteger );
      ekranaBas("enbuyukLong-->" + enbuyukLong );
      ekranaBas("");

      // gerçek sayılar
      float enbuyukFloat = Float.MAX_VALUE;
      double enbuyukDouble = Double.MAX_VALUE;
      ekranaBas("enbuyukFloat-->" + enbuyukFloat );
      ekranaBas("enbuyukDouble-->" + enbuyukDouble );
```

```
        ekranaBas("");

        // diğer temel (primitive) tipler
        char birChar = 'S';
        boolean birBoolean = true;

        ekranaBas("birChar-->" + birChar );
        ekranaBas("birBoolean-->" + birBoolean );
    }
}
```

Java'da C dilinde olduğu gibi bitişik atama operatörleri de vardır; bunlar atama operatörüyle diğer operatörlerden birinin birleştirilmesinden oluşurlar. Böylece kısa bir yazılımla hem aritmetik, öteleme gibi işlemler yaptırılır hem de atama yapılır. Yani, ifade yazımı kolaylaştırır. Örneğin, `int` tipinde olan `toplam` değişkeninin değeri 1 arttırmak için aşağıda gibi bir ifade kullanılabilir: (yorum ekle)

```
toplam = toplam + 1 ;
```

Bu ifade bitişik atama operatörüyle aşağıdaki gibi yazılabilir. Görüldüğü gibi değişken adı yukarıdaki yazımda 2, aşağıda yazımda ise 1 kez yazılmıştır... (yorum ekle)

```
toplam += 1 ;
```

Tablo-2-12'de bitişik atama operatörlerinin listesi görülmektedir; bu operatör, özellikle, uzun değişken kullanıldığı durumlarda yazım kolaylığı sağlarlar. (yorum ekle)

Tablo-2.11. Java'daki bitişik atama operatörleri (yorum ekle)

Operatör	Kullanılış Şekli	Eşittir
+=	değişken1 += değişken2	değişken1 = değişken1 + değişken2
-=	değişken1-= değişken2	değişken1 = değişken1 - değişken2
*=	değişken1*= değişken2	değişken1 = değişken1 * değişken2
/=	değişken1/= değişken2	değişken1 = değişken1 / değişken2
%=	değişken1%= değişken2	değişken1 = değişken1 % değişken2
&=	değişken1&= değişken2	değişken1 = değişken1 & değişken2
|=	değişken1|= değişken2	değişken1 = değişken1 | değişken2
^=	değişken1^= değişken2	değişken1 = değişken1 ^ değişken2
<<=	değişken1<<= değişken2	değişken1 = değişken1 << değişken2
>>=	değişken1>>= değişken2	değişken1 = değişken1 >> değişken2
>>>=	değişken1>>>= değişken2	değişken1 = değişken1 >>> değişken2

2.3.6. *String* (+) Operatörü

"+" operatörü *String* verilerde birleştirme görevi görür; eğer ifade *String* ile başlarsa, onu izleyen veri tipleri de kendiliğinden *String*'e dönüştürülür. Bu dönüştürme sırrı ve ayrıntısı ilerleyen bölümlerde ele alınmaktadır: (yorum ekle)

Örnek: *OtomatikCevirim.java* (yorum ekle)

```
public class OtamatikCevirim {

  public static void main(String args[]) {
    int x = 0, y = 1, z = 2;
    System.out.println("Sonuc =" + x + y + z);
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
Sonuc =012
```

Görüldüğü gibi *String* bir ifadeden sonra gelen tamsayılar toplanmadı; doğrudan *String* nesnesine çevrilip ekrana çıktı olarak gönderildiler... (yorum ekle)

2.3.7. **Nesnelerin Karşılaştırılması**

Nesnelerin eşit olup olmadığını = = veya ! = operatörleriyle sınanabilir! (yorum ekle)

Örnek: *Denklik.java* (yorum ekle)

```
public class Denklik {

  public static void main(String[] args)
  {
    Integer a1 = new Integer(47);
    Integer a2 = new Integer(47);

    System.out.println(a1 == a2);
    System.out.println(a1 != a2);
  }
}
```

Önce *Integer* sınıfı tipinde olan `n1` ve `n2` referansları, içlerinde 47 sayısını tutan *Integer* nesnelerine bağlı durumdadırlar. Uygulamanın sonucu olarak aşağıdaki gibi değerler bekliyor olabiliriz... (yorum ekle)

```
TrueFalse
```

Ancak ne yazık ki, sonuç yukarıdaki gibi değildir! Nedeni ise, elimizde iki adet farklı *Integer* nesnesi bulunmaktadır. Bu nesnelerin taşıdıkları değerler birbirlerine eşittir; ancak, `a1==a2` ifadesi kullanılarak şu denilmiş oldu "a1 ve a2 referanslarının işaret etmiş oldukları nesneler aynı mı?" Yanıt tahmin edilebileceği gibi hayırdır. Yani, *false*'dur. `a1` ve `a2` ayrı *Integer* nesnelerini işaret etmektedirler; eşit olan tek şey, bu iki ayrı nesnenin tuttukları değerin 47 olmasıdır (-ki bu eşitliği `a1=a2` ifadesi ile yakalayamayız). Programımızın çıktısı aşağıdaki gibidir. (yorum ekle)

```
FalseTrue
```

Peki, verilen örnekteki *Integer* nesneleri yerine temel tip olan `int` tipi kullanılsaydı sonuç ne olurdu? (yorum ekle)

Örnek: *IntIcinDenklik.java* (yorum ekle)

```
public class IntIcinDenklik {
```

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

    int s1 = 47;
    int s2 = 47;
    System.out.println(s1 == s2);
    System.out.println(s1 != s2);
  }
}
```

Bu uygulamanın sonucu aşağıdaki gibi olur:

```
True
False
```

Temel (*primitive*) tipler değerleri doğrudan kendi üzerlerinde taşıdıkları için `==` operatörüyle `s1` ve `s2` değişkeleri değerleri karşılaştırıldı ve doğru (*true*) yanıtı döndürüldü. Benzer şekilde `!=` operatörü de `s1` ve `s2` değişkenleri değerlerini karşılaştırdı ve yanlış (*false*) döndürüldü. Sonuçlar beklendiği gibi... (yorum ekle)

2.4. **Kontrol Deyimleri/İfadeler**

Kontrol deyimleri bir uygulamanın yürütülmesi sırasında program akışını yönlendiren yapılar/ kalıplardır. Kontrol deyimi olmaksızın bir uygulama yazılması neredeyse olanaksıdır denilebilir. Java programlama dilinde toplam 4 adet kontrol ifadesi bulunur:(yorum ekle)

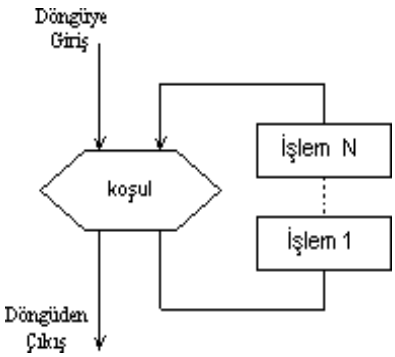
§ Döngü : **`while`**, **`do-while`**, **`for`**
§ Karşılaştırma : **`if-else`**, **`switch-case`**
§ Dallanma : **`break`**, **`continue`**, **`label:`**, **`return`**
§ İstisna : **`try-catch-finally`**, **`throw`** (yorum ekle)

2.4.1. **Döngü Deyimleri**

Döngü deyimleri aynı işlemin farklı parametre değerleri üzerinde yapılması için kullanılan yineleme/tekrarlama işleri için kullanılır. Java'da C dilinde olduğu gibi `while`, `do-while` ve `for` olarak adlandırılan üç farklı döngü deyimi vardır.(yorum ekle)

§ **`while` Döngü Deyimi**

`while` deyimi belirli bir grup kod öbeğini döngü koşulu doğru (*true*) olduğu sürece devamlı yineler. Genel yazım şekli aşağıdaki gibidir:




```
while (koşul) { çalışması istenen kod bloğu }
```

Program akışı `while` deyimine geldiğinde döngü koşuluna bakılır; olumlu/doğru ise çevrime girerek çalışması istenen kod öbeği yürütülür; yineleme döngü koşulu olumsuz/yanlış olana kadar sürer. (yorum ekle)

Örnek: *WhileOrnek.java* (yorum ekle)

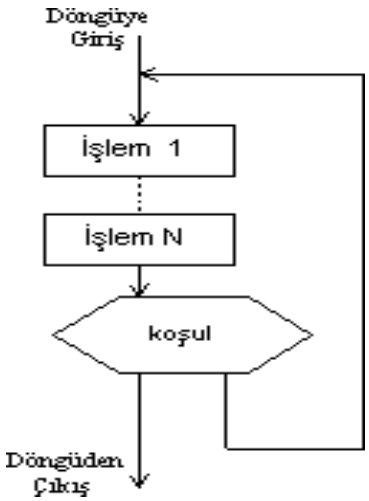
```
public class WhileOrnek {
  int i = 0 ; //döngü kontrol değişkeni

   while (i < 5 ) {
       System.out.println("i = " + i);
       i++ ;
   }
   System.out.println("Sayma islemi tamamlandi.");
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
i = 0i = 1i = 2i = 3i = 4 Sayma islemi tamamlandi.
```

§ `do-while` Döngü Deyimi

Bu döngü deyimde koşul sınaması döngü sonunda yapılır; dolayısıyla çevrim kod öbeği en az birkez yürütülmüş olur. Genel yazım şekli ve çizimsel gösterimi aşağıdaki gibidir: (yorum ekle) *




```
do {
  çalışması istenen kod bloğu
} while(koşul);
```

-

-

Örnek: *WhileDoOrnek.java* (yorum ekle)

```
public class WhileDoOrnek {

  public static void main(String args[])
  {
      int i = 0 ;       //döngü koşul değişkeni
      do {
        System.out.println("i = " + i);
        i++ ;
      } while ( i < 0 );
      System.out.println("Sayma islemi tamamlandi.");
  }
}
```

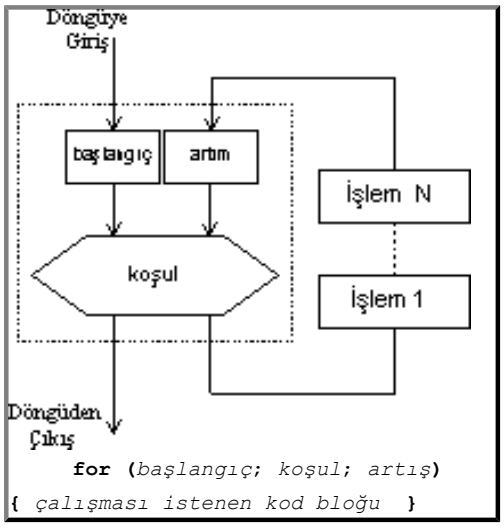
Verilen örnekte `while` kullanılmış olsaydı ekranda sadece "Sayma islemi tamamladi" cümlesi görülecekti; ancak, `do-while` kullanılmasıyla yürütülmesi istenen kod öbeği koşul değişkeni çevri m koşulunu sağlamasa da çevrime girilir... (yorum ekle)

`while` ve `do-while` döngü deyimleri kullanırken dikkat edilmesi gereken unsurlar aşağıdaki gibi belirtilebilir;

1. Döngü koşul değişkenine uygun bir şekilde değer atandığına dikkat ediniz.
2. Döngü durumunun doğru (*true*) ile başlamasına dikkat ediniz.
3. Döngü koşul değişkeninin çevrim içerisinde güncellediğinden emin olunuz; aksi durumda sonsuz çevrime girilebilir! (yorum ekle)

§ `for` Döngü Deyimi

for deyimde çevrim işlemleri daha bir derli toplu yapılabilir; bu döngü deyiminde koşulda kullanılan çevrim değişkeni, koşul ifadesi ve çevrim sayacı artımı for ifadesi içerisinde verilir. Genel yazım şekli ve çizimle gösterilmesi aşağıdaki gibi verilebilir:(yorum ekle)



Görüldüğü gibi `for` deyimi içerisinde "`;`" ile ayrılmış 3 parametre vardır; birincisi çevrim sayacı, ikincisi koşul ifadesi ve üçüncüsü de sayacın artım miktarı ifadesidir. Eğer, kodun daha önceki kısımlarda sayaç değişkeni halihazırda varsa `başlangıç`, artışı kod öbeği kısmında yapılıyorsa `artış` bilgisi verilmeyebilir. Bu durumda bu alanlar bol bırakılır. (yorum ekle)

Örnek: *ForOrnek.java* (yorum ekle)

```
public class ForOrnek {
  public static void main(String args[]) {
    for (int i=0 ; i < 5 ; i ++) {
      System.out.println("i = " + i);
    }
  }
}
```

Uygulamanın çıktısı aşağıdaki gibi olur:

```
i = 0i = 1i = 2i = 3i = 4
```

`for` deyimi kullanılarak sonsuz çevrim oluşturulmak istenirse aşağıdaki gibi yazılması yeterlidir. (yorum ekle)

```
for ( ;1; ) {        // sonsuz döngü ...}
```

`for` ifadesinde birden fazla değişken kullanabilirsiniz; yani, birden çok sayaç değişkeni olabilir veya koşul mantıksal operatörler kullanılarak birden çok karşılaştırma içerebilir. (yorum ekle)

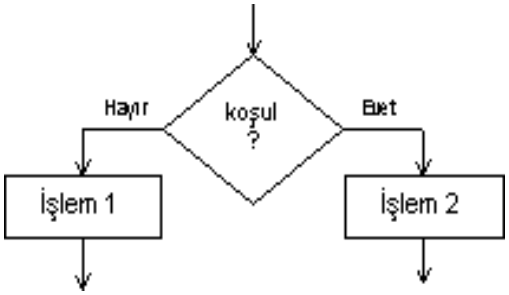
Örnek: *ForOrnekVersiyon2.java* (yorum ekle)

```
public class ForOrnekVersiyon2 {
  public static void main(String args[]) {
    for ( int i = 0, j = 0 ; i < 20 ; i++, j++ ) {
      i *= j ;
      System.out.println("i = " + i + " j = " + j);
    }
  }
}
```

Uygulamamızın çıktısı aşağıdaki gibidir:

```
i = 0 j = 0i = 1 j = 1i = 4 j = 2i = 15 j = 3i = 64 j = 4
```

2.4.2. **Karşılaştırma Deyimleri**



Karşılaştırma deyimleri belirli bir koşula göre farklı işlemler yaptırılacağı zaman kullanılır. Örneğin `adet` adlı değişken değeri 5'ten küçük olduğundan farklı, 5'ten büyük veya eşit olduğunda farklı işler kodlar yürütülecekse bu işi yapabilmek için karşılaştırma deyimlerine gereksinim duyulur. Java'da `if-else` ve `switch-case` olmak üzere iki farklı karşılaştırma deyimi vardır. (yorum ekle)

§ `if-else` Deyimi

Koşula göre program akışı değiştirilmek isteniyorsa **`if`** kullanılabilir. Genel yazım ifadesi aşağıdaki gibidir:

```
if (koşul) {
   durum true olduğunda çalışması istenen kod bloğu
} else {
   durum false olduğunda çalışması istenen kod bloğu
}
```

Örnek: *IfElseTest.java* (yorum ekle)

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

    int puan = 76;
    char sonuc;
```

```
    if (puan >= 90) {
        sonuc = 'A';
    } else if (puan >= 80) {
        sonuc = 'B';
    } else if (puan >= 70) {
        sonuc = 'C';
    } else if (puan >= 60) {
        sonuc = 'D';
    } else {
        sonuc = 'F';
    }
    System.out.println("Sonuc = " + sonuc);
  }
}
```

`int` tipindeki *puan* değişkenin değeri 70'den büyük olduğu için sonuç aşağıdaki gibi olacaktır:

```
Sonuc = C
```

- **3'lü `if-else`:** 3'lü `if-else` deyimi önceki `if-else` deyimine alternatif olarak kullanılabilir. Genel yazılış biçimi;

  `mantıksal-ifade? deger0: deger1`

  Eğer mantıksal ifade doğrı (*true*) ise `değer0` hesaplanır; eğer yanlış (*false*) ise `deger1` hesaplanır. (yorum ekle)

- **Kestirme sonuç**: VE işleminde (bkz. Mantıksal Operatörler) iki değerin doğru (true) olması durumunda sonuç doğru oluyordu... Eğer if deyiminde VE işlemi kullanılmış ise ve ilk değerden yanlış dönmüş ise, ikinci değer kesinlikle hesaplanmaz. Bunun nedeni, iki değer sonucunun VE işlemine göre doğru dönmesi imkansızlığıdır. Kestirme sonuç özelliği sayesinde uygulamalar gereksiz hesaplamalardan kurtulmuş olur; bununda getirisi performansıdır. (yorum ekle)

Örnek: *Kestirme.java* (yorum ekle)

```
public class Kestirme {
  public static boolean hesaplaBir(int a) {
      System.out.println("hesaplaBir yordamına girildi");
      return a > 1 ;
  }

  public static boolean hesaplaIki(int a) {
      System.out.println("hesaplaIki yordamına girildi");
      return a > 2 ;
  }

  public static void main(String[] args) {
      System.out.println("Baslangic");
      //hesaplaBir(0) --> false deger doner
      //hesaplaIki(3) --> true deger doner
```

```
      System.out.println("hesaplaBir(0) && hesaplaIki(3)");
      if ( hesaplaBir(0) && hesaplaIki(3) ) {
        System.out.println(" 1 -true ");
      } else {
        System.out.println(" 1 -false ");
      }
      System.out.println("--------");
      System.out.println("hesaplaBir(0) || hesaplaIki(3)");

      if (hesaplaBir(0) || hesaplaIki(3)) {
        System.out.println(" 2 -true ");
      } else {
        System.out.println(" 2 -false ");
      }
      System.out.println("--------");
      System.out.println("hesaplaBir(0) & hesaplaIki(3)");

      if (hesaplaBir(0) & hesaplaIki(3)) {
        System.out.println(" 3 -true ");
      } else {
        System.out.println(" 3 -false ");
      }
      System.out.println("--------");
      System.out.println("hesaplaBir(0) | hesaplaIki(3)");

      if (hesaplaBir(0) | hesaplaIki(3)) {
        System.out.println(" 4 -true ");
      } else {
        System.out.println(" 4 -false ");
      }
      System.out.println("--------");
      System.out.println("hesaplaBir(0) ^ hesaplaIki(3)");

      if (hesaplaBir(0) ^ hesaplaIki(3)) {
        System.out.println(" 5 -true ");
      } else {
        System.out.println(" 5 -true ");
      }
      System.out.println("Son..");
  }
}
```

Programı açıklanmaya çalışılırsa, bu uygulamada **`hesaplaBir()`** ve **`hesaplaIki()`** adında iki adet yordam bulunmaktadır. Bunlar `int` tipinde parametre kabul edip mantıksal sonuç döndürmektedirler. Bu yordamlara girildiği zaman ekrana kendilerini tanıtan bir yazı çıkartıyoruz -ki gerçekten kestirme olup olmadığını anlayalım: (yorum ekle)
`hesaplaBir()` yordamı kendisine gelen `int` tipindeki parametreyi alıp 1'den büyük mü diye bir sınamaktadır. Burada **`hesaplaBir()`** yordamına parametre olarak sıfır sayısı gönderildiğinden dönecek değerin olumsuz olacağı biliniyor. **`hesaplaIki()`** yordamına da aynı şekilde üç sayısı gönderilerek bu yordamın bize olumlu değer döndüreceğinden emin olduktan sonra işlemlere başlıyoruz. (yorum ekle)
İlk önce, yordamlardan bize geri dönen değerler VE işlemine tabii tutuluyor. Görüldüğü gibi yalnızca **`hesaplaBir()`** yordamına giriyor. Çünkü **`hesaplaBir()`** yordamından olumsuz değer dönmektedir; VE işleminde olumlu dönebilmesi için iki değerinde olumlu olması

gerektiğinden, **`hesaplaIki()`** yordamı çağrılmayarak kestirme özelliği kullanılmıştır. (yorum ekle)
İkinci olarak, gelen değerler VEYA işlemine tabii tutuluyor; görüldüğü gibi hem **`hesaplaBir()`** hem de **`hesaplaIki()`** yordamları çağrılmaktadır. VEYA tablosu hatırlanırsa, sonucun olumsuz olması için iki değerinde olumsuz olması gerekmekteydi. Burada ilk değerden olumsuz değeri döndü; ancak, ikinci değerin de hesaplanması gerek, aksi durumda sonucun öğrenilmesi imkansız olur. Bu nedenden dolayı, burada kestirme işlemi gerçekleşmedi. Ancak, ilk değer olumlu dönseydi, o zaman, ikinci yordam olan **`hesaplaIki()`** hiç çağrılmayacaktı. Çünkü VEYA işlemleri sonucunun olumlu olabilmesi için parametrelerden birisinin olumlu olması gereklidir. (yorum ekle)
Üçüncü olarak, değerler yine VE işlemine tabii tutuluyor; ancak, burada (&) operatörü kullanıldığı için kestirme işlemi ortadan kalkmaktadır; iki yordam da ayrı ayrı çağrılır. (yorum ekle)
Dördüncü olarak, değerler VEYA işlemine tabii tutuluyor; fakat (|) operatörü zaten kestirme işlemini ortadan kalkmaktadır ve iki yordamda ayrı ayrı çağrılır. (yorum ekle)
Son olarak, değerler YA DA (*Exclusive Or*) işlemine tabii tutuluyor; bu işlemde kesinlikle iki değere de bakılma zorunluluğu olduğundan kestirme işlemi söz konusu olmaz. (yorum ekle)
Uygulamanın sonucu aşağıdaki gibi olur:

```
Baslangic
hesaplaBir(0) && hesaplaIki(3)
hesaplaBir yordamına girildi
1 -false
--------
hesaplaBir(0) || hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
2 -true
--------
hesaplaBir(0) & hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
3 -false
--------
hesaplaBir(0) | hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
4 -true
--------
hesaplaBir(0) ^ hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
5 -true
Son..
```

§ switch Deyimi

`switch` deyimi tamsayıların karşılaştırılması ile doğru koşulların elde edilmesini sağlayan mekanizmadır. `switch` deyimini genel yazım biçimi aşağıdaki gibidir: (yorum ekle)

```
switch(tamsayı) {
```

```
 case uygun-tamsayı-deger1 : çalışması istenen kod bloğu; break;
 case uygun-tamsayı-deger2 : çalışması istenen kod bloğu; break;
 case uygun-tamsayı-deger3 : çalışması istenen kod bloğu; break;
 case uygun-tamsayı-deger4 : çalışması istenen kod bloğu; break;
 case uygun-tamsayı-deger5 : çalışması istenen kod bloğu; break;
 //...
 default: çalışması istenen kod bloğu ;
}
```

`switch` deyimi içersindeki tamsayı ile, bu tamsayıya karşılık gelen koşula girilir ve istenen kod bloğu çalıştırılır. Kod bloklarından sonra `break` koymak gerekir aksi takdirde uygun koşul bulduktan sonraki her koşula girilecektir. Eğer tamsayımız koşullardan hiçbirine uymuyorsa `default` koşulundaki kod bloğu çalıştırılarak son bulur. (yorum ekle)

Örnek: *AylarSwitchTest.java* (yorum ekle)

```
public class AylarSwitchTest {

  public static void main(String[] args) {

    int ay = 8;
    switch (ay) {
case 1: System.out.println("Ocak"); break;
case 2: System.out.println("Subat"); break;
case 3: System.out.println("Mart"); break;
case 4: System.out.println("Nisan"); break;
case 5: System.out.println("Mayis"); break;
case 6: System.out.println("Haziran"); break;
case 7: System.out.println("Temmuz"); break;
case 8: System.out.println("Agustos"); break;
case 9: System.out.println("Eylul"); break;
case 10: System.out.println("Ekim"); break;
case 11: System.out.println("Kasim"); break;
case 12: System.out.println("Aralik"); break;
     }
  }
}
```

Uygulamanın sonucu,

```
Agustos
```

Yukarıdaki uygulamadan `break` anahtar kelimelerini kaldırıp yeniden yazılırsa, (yorum ekle)

Örnek: *AylarSwitchTestNoBreak.java* (yorum ekle)

```
public class AylarSwitchTestNoBreak {

  public static void main(String[] args) {

    int ay = 8;
    switch (ay) {
```

```
case 1: System.out.println("Ocak");
case 2: System.out.println("Subat");
case 3: System.out.println("Mart");
case 4: System.out.println("Nisan");
case 5: System.out.println("Mayis");
case 6: System.out.println("Haziran");
case 7: System.out.println("Temmuz");
case 8: System.out.println("Agustos");
case 9: System.out.println("Eylul");
case 10: System.out.println("Ekim");
case 11: System.out.println("Kasim");
case 12: System.out.println("Aralik");
     }
   }
}
```

Uygulamanın çıktısı aşağıdaki gibi olur. Yalnız hemen uyaralım ki, bu istenmeyen bir durumdur:

```
AgustosEylulEkimKasimAralik
```

Aşağıda verilen uygulama ise `switch` deyiminde `default` kullanımı göstermek amacıyla yazılmıştır: (yorum ekle)

Örnek: *AylarSwitchDefaultTest.java* (yorum ekle)

```
public class AylarSwitchDefaultTest {

  public static void main(String[] args) {

    int ay = 25;
    switch (ay) {
      case 1: System.out.println("Ocak"); break;
      case 2: System.out.println("Subat"); break;
      case 3: System.out.println("Mart"); break;
      case 4: System.out.println("Nisan"); break;
      case 5: System.out.println("Mayis"); break;
      case 6: System.out.println("Haziran"); break;
      case 7: System.out.println("Temmuz"); break;
      case 8: System.out.println("Agustos"); break;
      case 9: System.out.println("Eylul"); break;
      case 10: System.out.println("Ekim"); break;
      case 11: System.out.println("Kasim"); break;
      case 12: System.out.println("Aralik"); break;
      default: System.out.println("Aranilan Kosul"+
"Bulunamadi!!");
    }
  }
}
```

Bu örneğimizde istenen durum hiçbir koşula uymadığı için `default` koşulundaki kod öbeği çalışmaktadır.
Uygulamanın sonucu aşağıdaki gibi olur:

```
Aranilan Kosul Bulunamadi !!
```

2.4.3. **Dallandırma Deyimleri**

Java programlama dilinde dallandırma ifadeleri toplam 3 adettir. (yorum ekle)

- `break`
- `continue`
- `return`

§ `break` Deyimi

`break` deyiminin 2 farklı uyarlaması bulunur; birisi etiketli (*labeled*), diğeri ise etiketsiz (*unlabeled*)'dir. Etiketsiz `break`, `switch` deyiminde nasıl kullanıldığı görülmüştü... Etiketsiz `break` sayesinde koşul sağlandığında `switch` deyimini sonlanması sağlanıyordu. `break` deyimi aynı şekilde `while`, `do-while` veya `for` deyimlerinden çıkılması için de kullanabilir. (yorum ekle)

Örnek: *BreakTest.java* (yorum ekle)

```
public class BreakTest {
 public static void main(String[] args) {
   for ( int i = 0; i < 100; i++ ) {
     if ( i ==9 ) {     // for döngüsünü kırıyor
         break;
     }
     System.out.println("i =" +i);
   }
   System.out.println("Donguden cikti");
 }
}
```

Normalde 0 dan 99'a kadar dönmesi gereken kod bloğu, **i** değişkenin 9 değerine gelmesiyle for dönüsünün dışına çıktı. Uygulamanın çıktısı aşağıdaki gibidir.

```
i =0
i =1
i =2
i =3
i =4
i =5
i =6
i =7
i =8
Donguden cikti
```

Etiketiz break ifadeleri en içteki while, do-while veya for döngü ifadelerini sona erdirirken, etiketli break ifadeleri etiket (label) hangi döngünün başına konulmuş ise o döngü sistemini sona erdirir. (yorum ekle)

Örnek: *BreakTestEtiketli.java* (yorum ekle)

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

```
    kiril :
    for ( int j = 0 ; j < 10 ; j ++ ) {

      for ( int i = 0; i < 100; i++ ) {
        if ( i ==9 ) {// for dongusunu kiriyor break kiril;
        }
        System.out.println("i =" +i);
      }
      System.out.println("Donguden cikti");
      System.out.println("j =" +j);

    }
  }
}
```

Yukarıdaki örneğimizde etiket kullanarak, daha geniş çaplı bir döngü sisteminden çıkmış olduk. Uygulamamızın çıktısı aşağıdaki gibidir.

```
i =0
i =1
i =2
i =3
i =4
i =5
i =6
i =7
i =8
```

§ continue Deyimi

`continue` ifadesi, döngü içersinde o anki devir işleminin pas geçilmesini ve bir sonraki devir işleminin başlamasını sağlayan bir mekanizmadır. `continue` ifadeleri de `break` ifadeleri gibi iki çeşide ayrılır. Etiketsiz `continue` ve etiketli `continue`. Etiketsiz`continue` en içteki döngü içersinde etkili olurken, etiketli `continue` ise başına konulduğu döngü sisteminin etkiler. (yorum ekle)

Örnek: *ContinueTest.java* (yorum ekle)

```
public class ContinueTest {

  public static void main(String[] args) {
    for ( int i = 0; i < 10; i++ ) {
      if ( i == 5 ) {       // for döngüsünü kırıyor
       continue;
    }
      System.out.println("i =" +i);
    }
    System.out.println("Donguden cikti");
  }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
i =0
i =1
```

```
i =2
i =3
i =4
i =6
i =7
i =8
i =9
Donguden cikti
```

Ekrana yazılan sonuca dikkatli bakılırsa 5 değerinin olmadığı görülür; `continue` deyimi `break` gibi döngüleri kırmaz, yalnızca belli durumlardaki döngü işleminin atlanmasını sağlar sağlar. (yorum ekle)

Örnek: *ContinueTestEtiketli.java* (yorum ekle)

```
public class ContinueTestEtiketli {

 public static void main(String[] args) {

   pas :
   for ( int j = 0 ; j < 6 ; j ++ ) {

     for ( int i = 0; i < 5; i++ ) {
       if ( i ==3 ) {   // for döngüsünü kırıyor
            continue pas;
       }
       System.out.println("i =" +i);
     }
     System.out.println("Donguden cikti");
     System.out.println("j =" +j);

   }
 }
}
```

Bu uygulamada, *pas* etiketini kullanılarak `continue` işleminin en dışdaki döngüsel sistemden tekrardan başlamasını (ama kaldığı yerden) sağlandı... Uygulamanın sonucu aşağıdaki gibi olur:

```
i =0
i =1
i =2
i =0
i =1
i =2
i =0
i =1
i =2
i =0
i =1
i =2
i =0
i =1
i =2
i =0
i =1
i =2
```

i değişkeninin her seferinde yeniden 0'dan başladığını ve 2 de kesildiğini görmekteyiz. Bu işlem toplam 6 kez olmuştur. Yani en dışdaki döngünün sınırları kadar. (yorum ekle)

§ return Deyimi

`return` deyimi daha önceden bahsedilmişti; ancak yeni bir anımsatma yararlı olacaktır. `return` deyiminin 2 tür kullanım şekli vardır: Birincisi değer döndürmek için -ki yordamlardan üretilen değerleri böyle geri döndürürüz, ikincisi ise eğer yordamın dönüş tipi buna izin vermiyorsa (*void* ise) herhangi bir taviz vermeden `return` yazıp ilgili yordamı terk edebiliriz: (yorum ekle)

Örnek: *ReturnTest.java* (yorum ekle)

```
public class ReturnTest {

  public double toplamaYap(double a, double b) {

    double sonuc = a + b ;
    return sonuc ;   // normal return kullanımı
  }

  public void biseyYapma(double a) {

    if (a == 0) {
      return ; // yordamı acilen terk et
    } else {
      System.out.println("-->" + a);
    }
  }
}
```

* **Not**: Döngü deyimleri için verilen akış şemaları yine bir Papatya Yayıncılık kitabı olan "Veri Yapıları ve Algoritmaları" (Dr.Rifat ÇÖLKESEN) adlı eserinden yayınevinin ve yazarının izniyle alınmıştır.