JAVA'NIN TARİHÇESİ

Java'yı iyi bir şekilde anlamak için, geçmişini, oluşturulmasının amaçlarını ve motivasyonunu bilmek gerekir. Diğer başarılı programlama dillerinde de olduğu gibi, Java kendinden önce gelen dillerin başarısız özelliklerini azaltmış veya tamamen yok etmiş, iyi özelliklerini ise bünyesinde toplamış ve geliştirmiştir.

Java, doğrudan C++ ile bağlantılıdır. C++ ise C'nin devamıdır. Java, karakteristik özelliklerinin birçoğunu bu iki dilden almıştır. C'nin sözdizimi (*syntax*), C++'ın ise nesne yönelimli programlama (*object oriented programming- OOP*) kavramları Java'ya miras kalmıştır. Bunun yanı sıra, Java, bu dillerdeki karmaşıklığı ortadan kaldırmış, eksik özelliklerini ise tamamlamıştır.

Modern Programlamanın Doğuşu: C

Bugünkü teknolojinin geldiği noktada sanırım en büyük paylardan biri de C programlama diline aittir. C'nin ortaya çıkışı yazılım dünyasını derinden etkilemiştir.

C'nin başarısının sırrı, kendisinden önce iyi tasarlanmış ve verimli bir sistem programlama dilinin olmayışıydı. C bu boşluğu başarılı bir şekilde doldurmuştu. C'den önce, programlama dilleri belirli bir amaç için geliştirilirdi. Genel bir sistem tasarlamak üzere üretilmiş başarılı bir programlama dili yoktu. Örneğin, FORTRAN bilimsel çalışmalarda kullanmak üzere geliştirilmiş çok başarılı bir dildi, fakat bir sistem tasarlayamazdınız. BASIC öğrenmesi kolay bir dildi, fakat tasarımında bazı eksikler vardı ve güçlü bir dil değildi.

1970'lerde C'nin geliştirilmesi, birçokları tarafından modern bilgisayar programcılığının başlangıcı olarak kabul edilir. Kendisinden önceki dillerin sorunlu yönlerini çözmüş ve ortaya güçlü, verimli ve iyi tasarlanmış bir dil çıkarmıştı. C o kadar başarılı olmuştur ki, kendisinden sonraki birçok programlama diline ilham vermiştir. Ayrıca bugün bile hâlâ aktif olarak kullanılmaktadır. Örneğin, hepimizin bilgisayarlarında kullandığı işletim sistemlerinin büyük bir kısmı hâlâ C dili ile yazılır.

Bir Sonraki Adım: C++

1970'lerde ve 1980'lerin başlarında, C en çok kullanılan programlama dili oldu. Buna rağmen, C'nin de kendi içinde, gelişen teknolojilerin ve programcıların ihtiyaçlarını karşılamayan bazı eksikleri vardı. Nesne yönelimli programlama (OOP) metodolojisi, o yıllarda hızla benimsenmiş ve programcılar tarafından aranır olmuştu. C bundan yoksundu. Diğer yandan, sistem programlama için mükemmel bir dil olmasına rağmen, bunu sağlayabilmek için sunduğu özelliklerin fazlalığı, dilin karmaşıklığının artmasına neden oluyordu.

Bu gibi sorunları çözmek için, 1979 yılında yeni bir dil geliştirildi. 1983 yılında bu dilin adına C++ dendi. C'nin ne kadar geliştirildiğini anlatabilmek için bir yerine iki + konulmuştu. C++, C'nin bütün özelliklerine sahipti; aynı zamanda OOP metodolojisine göre oluşturulmuştu. Başarısının sırrı da burada yatıyordu. 1980'lerde ve 1990'ların başında, C++ artık C'nin yerini almış ve en çok kullanılan dil olmuştu. Diğer yandan, sahne Java'nın çıkışına hazırlanıyordu.

Java'ya neden ihtiyaç duyuldu?

Java, C++'tan hayli etkilenmiş, diğer yandan C ve C++'ın bazı özelliklerini barındırmayan bir dildi. Ortaya çıkışından bugüne kadar Java en başarılı programlama dillerinden biri olmuştur. Günümüzde dünyada en çok kullanılan 3 programlama dilinden biridir. Peki, zaten başarılı olmuş iki tane dil varken, Java'nın geliştirilmesine neden ihtiyaç duyulmuştur? Bunu cevaplayabilmek için önce bilgisayarların nasıl çalıştığına bakmamız gerekiyor.

Bilgisayarlar nasıl çalışır?

Basitçe tarif etmek gerekirse bilgisayar, birçok karmaşık elektronik devreden oluşan bir sistemdir. Amerikalı matematikçi John von Neumann, 1945 yılında bilgisayarları kavramsal olarak tarif etmiştir. Bu tarife göre, bir bilgisayar sistemi 3 temel parçadan oluşur:

- Merkezi işlemci birimi (CPU)
- Bellek (RAM)
- Girdi/Çıktı birimleri (Sabit disk, ekran, klavye, fare vs.)

Kodların ve verilerin tutulması için bir bellek gereklidir. Bellekte tutulan bu kodlar işlemci tarafından çalıştırılır. Girdi/Çıktı birimleri ise zorunlu olmamakla birlikte, bilgisayarın ek iş yapmasına olanak sağlar.

Bu tarif bugün bile geçerlidir. Bu yüzdendir ki, bu tarife uyan cihazlara (bilgisayarlar, telefonlar, tabletler vs.) "*Von Neumann makinesi"* denir.

Bilgisayar elektronik bir devredir. Her işlem elektrik kullanarak gerçekleştirilir. Bilgisayarda yaptığımız işlemleri fiziksel olarak gerçekleştiren donanım ise CPU'dur. CPU'ların çalışma mantığı ise basitçe şöyledir: CPU üreticileri (Intel, AMD vs.), CPU'ları belli bir komut kümesine (*instruction set*) yanıt verecek şekilde üretirler. Bu komut kümesi önceden belirlenmiş, sınırlı sayıda komutu içerir. Aslında bu küme CPU'nun dilini oluşturur. CPU'nun anladığı bu dile makine dili (*assembly language*) denir. Bilgisayar üzerinde çalıştırılan her program, makine diline dönüştürülmek zorundadır. Fakat makine dili, öğrenmesi ve kullanması oldukça zor bir dildir. Bunun yerine insanların kolayca öğrenip kullanabileceği, metin tabanlı diller geliştirilmiştir. Yine de bu dillerde yazılan kodların da nihayet makine diline dönüştürülmesi gerekir. Bunun için derleyiciler (*compiler*) geliştirilmiştir. Yeni bir programlama dili geliştirilirken, aynı zamanda derleyicisi de yazılır. Derleyici, kodladığınız programı makine diline dönüştürür ve CPU tarafından çalıştırılmaya hazır hale getirir.

C veya C++ ile bir program yazdığınızda, o programın farklı platformlarda çalışması için farklı derleyicilerle derlemeniz gerekiyordu. Örneğin, Windows işletim sisteminde çalışması için başka bir derleyici, Linux için başka bir derleyici kullanmak zorundaydınız. Bu durum programcılar için bir yük oluşturuyordu. Java'nın geliştirilmesindeki en büyük amaç bu yükü ortadan kaldırmaktı. Java, ilk olarak **platform bağımsız** bir dil olmak hedefiyle yola çıkmıştı.

Java'nın ortaya çıkışı

1991'de Sun Microsystems şirketi tarafından geliştirilmeye başlayan Java'nın çalışan ilk sürümünü üretmek 18 ay aldı. İlk olarak "Oak" denilse de 1995 yılında ismi Java olarak değiştirildi.

Java, yukarıda da belirttiğimiz gibi platform bağımsız bir dil olmak amacıyla üretilmişti. Bu amaç, **WORA** (*Write Once, Run Anywhere- Bir kere yaz, her yerde çalıştır*) sloganıyla belirtilmiştir. Bu dilin tek bir derleyicisi vardır ve yazdığınız kod bütün platformlarda aynı şekilde çalışır.

C# etkisi

Java'nın ortaya çıkışı birçok programcıyı etkilemiştir. Microsoft, 1990'ların sonunda C# dilini geliştirmiştir. C# doğrudan Java'dan etkilenen bir dildir. İki dil arasındaki benzerlikler o kadar fazladır ki, bu dillerden birini öğrenen diğerini de öğrenmiş gibi olur.

Anahtar kelimeler

Java'yı geliştiren ekip, Java'nın sahip olduğu özellikleri bazı anahtar kelimelerle belirtmiştir. Bu kelimelere kısaca göz atalım:

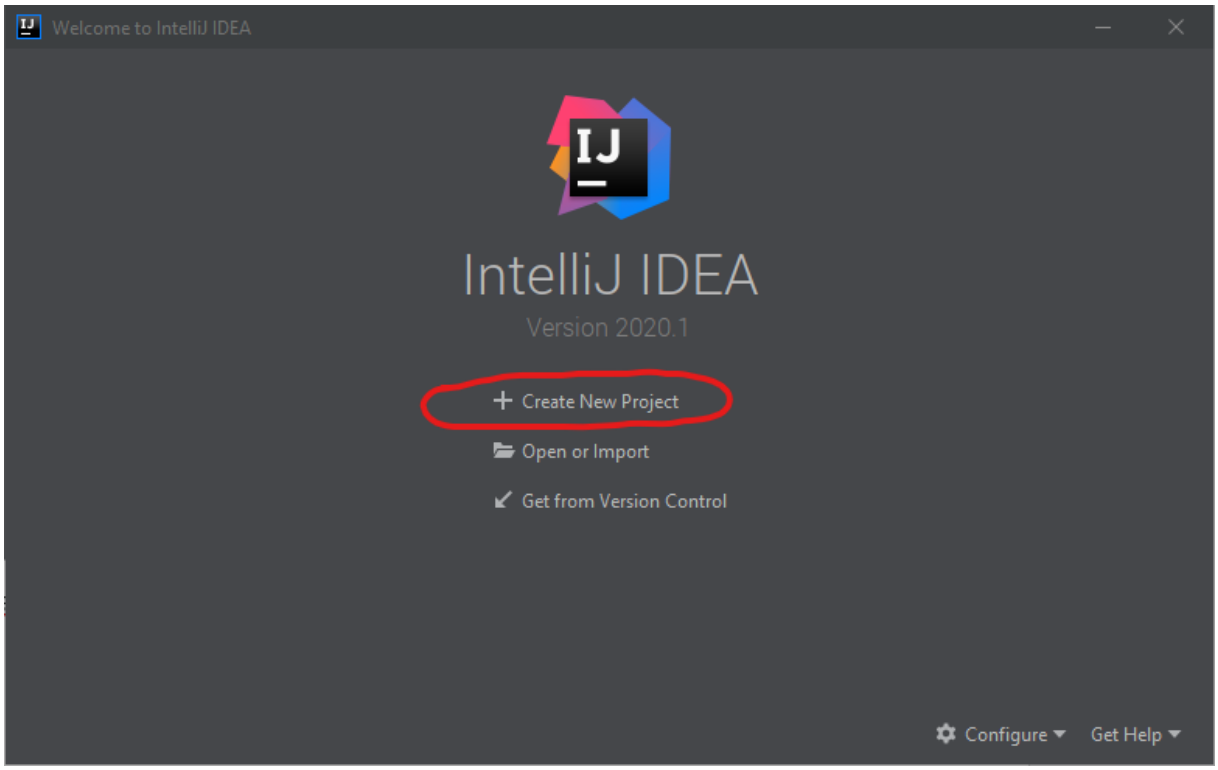
- **Simple (basit):** Java, programcıların kolayca öğrenmesi ve verimli bir şekilde kodlayabilmesi amacıyla tasarlanmıştır. C ve C++ dillerinden etkilense de bu dillerdeki bazı zorluklar Java için söz konusu değildir. Bu dilleri bilen kişiler için Java'yı öğrenmek ve kullanmak zor olmayacaktır.
- **Object-oriented (nesne yönelimli):** Java, nesne yönelimli bir programlama dilidir. Nesne yönelimli programlama daha sonra ayrıntıyla açıklanacaktır.
- **Robust (güçlü):** Java, güçlü ve stabil bir dildir. Güçlü hata yönetim mekanizması sayesinde hataları tespit etmek, tekrar etmek ve çözmek kolaydır. Java ile yazılan kodlar hem derleme aşamasında (*compile time*) hem de çalışma aşamasında (*runtime*) sıkı bir şekilde kontrol edildiği için istisnai durumlara nadiren rastlanır. Üstelik daha önce de belirttiğimiz gibi platform bağımsız olması, yazdığınız kodun farklı platformlarda stabil bir şekilde çalışmasını sağlar.
- **Multithreaded (çok kanallı):** Java, gelişen teknolojileri destekleyen bir dildir. İşlemci mimarisinin gelişmesi ve çok kanallı uygulamaların yaygınlaşması nedeniyle, Java dili doğrudan çok kanallı programlamayı destekleyecek şekilde geliştirilmiştir. Java'nın kullanması kolay

senkronizasyon yöntemleri sayesinde çok kanallı uygulamalar geliştirmek oldukça kolaydır.

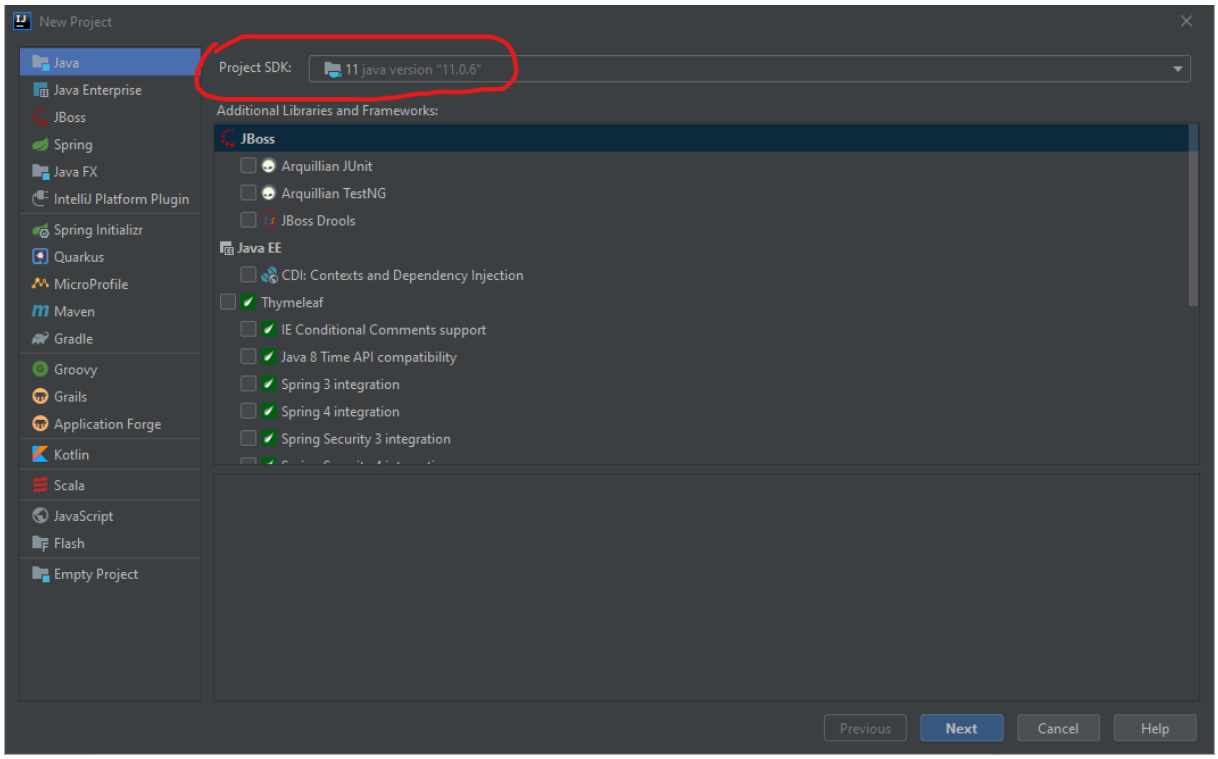
- **Distributed (dağıtık):** Java, dağıtık sistemler geliştirmeyi destekleyen bir dildir.

JAVA İLE İLK PROGRAM

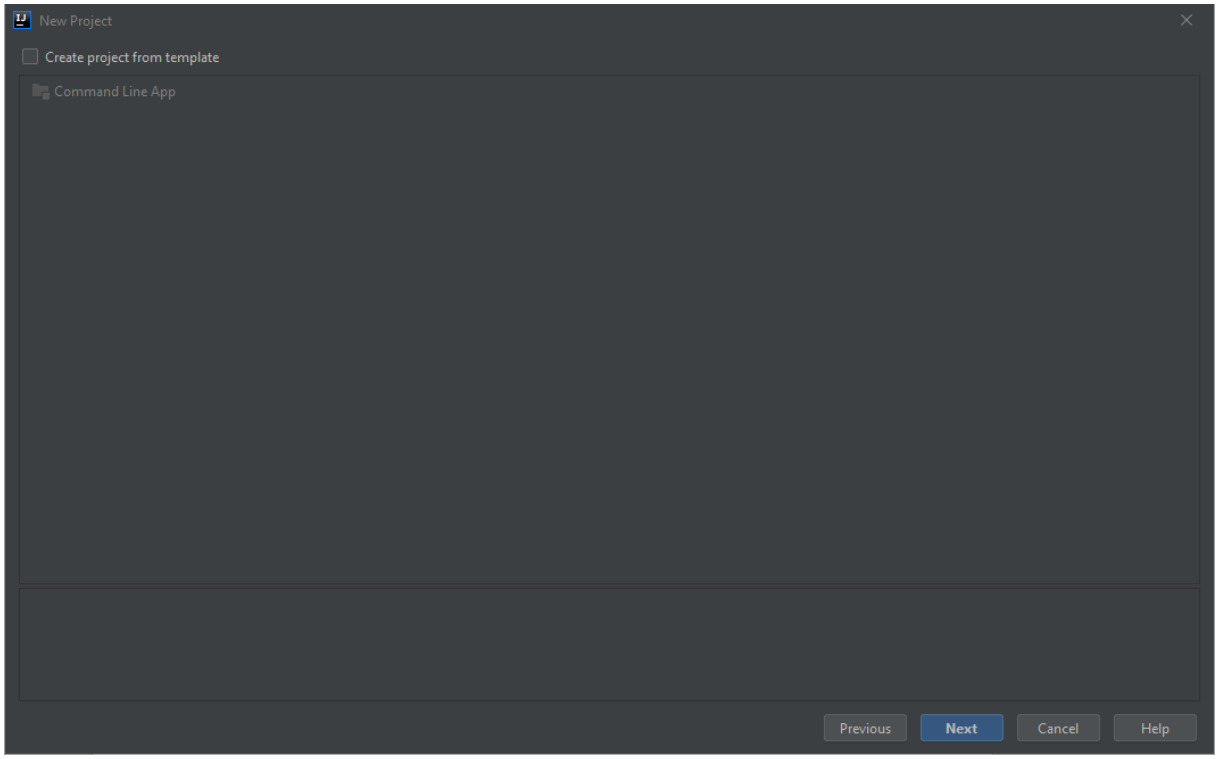
Intellij IDEA programını açalım. Karşımıza aşağıdaki ekran çıkacaktır:



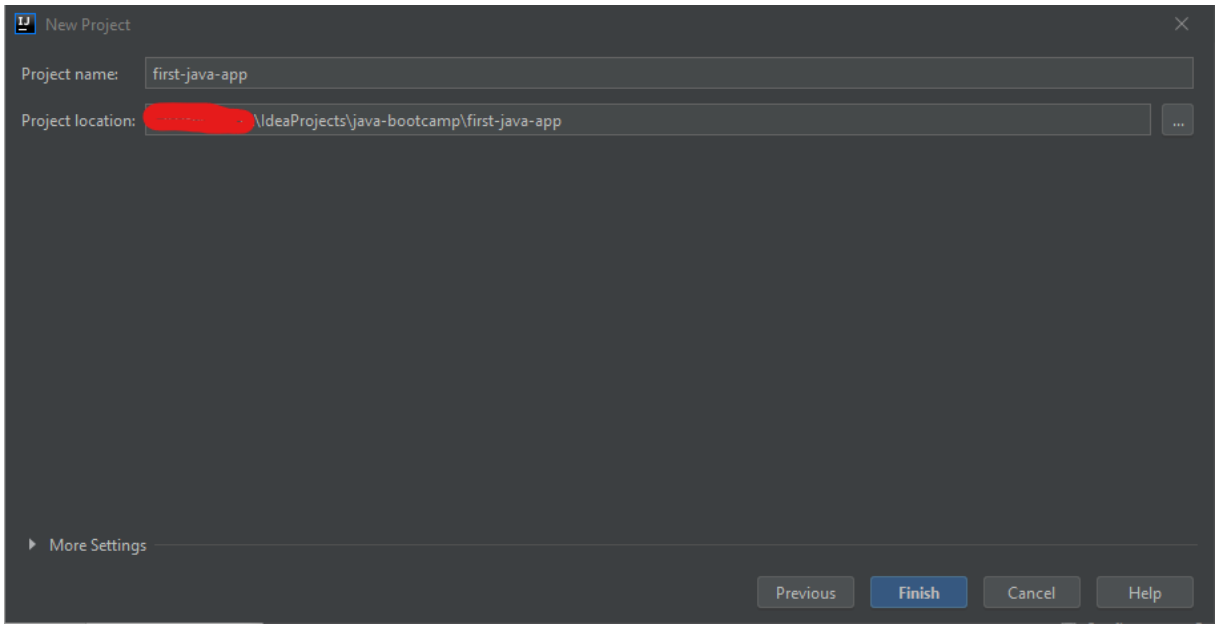
"Create New Project" butonuna tıklayalım.



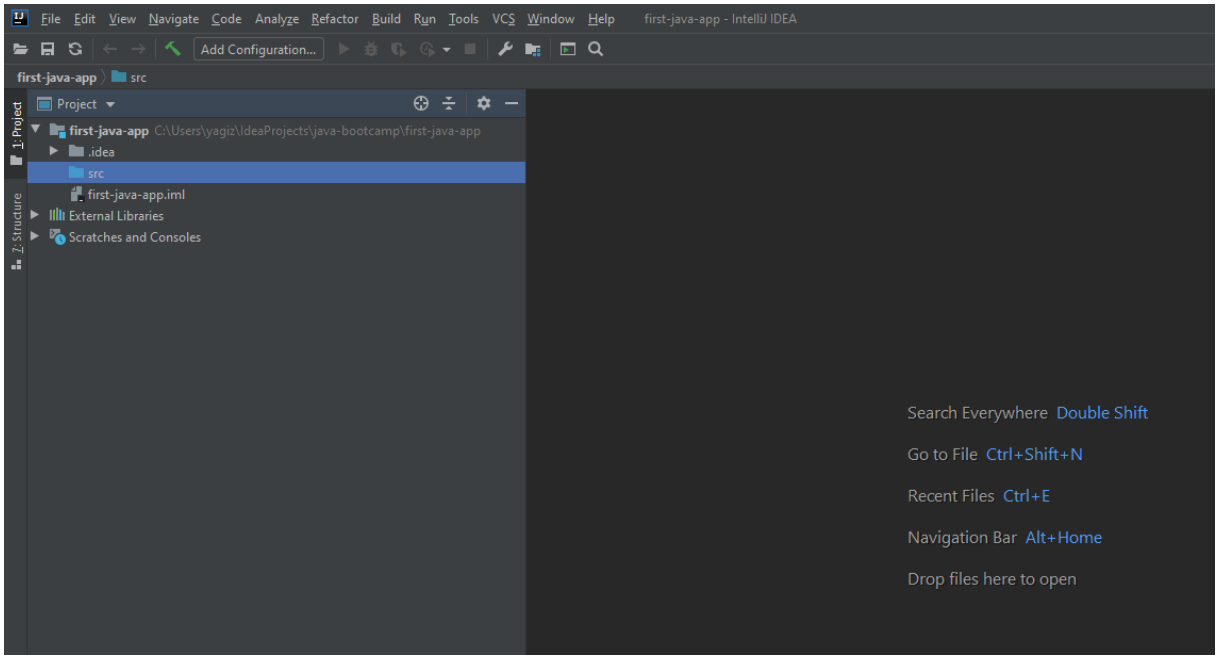
Soldaki bölümden "Java"yı ve Project SDK olarak 11'i seçelim. "Next" butonuna tıklayalım.



Bu ekranda hiçbir şey yapmadan "Next" butonuna basarak devam edelim.

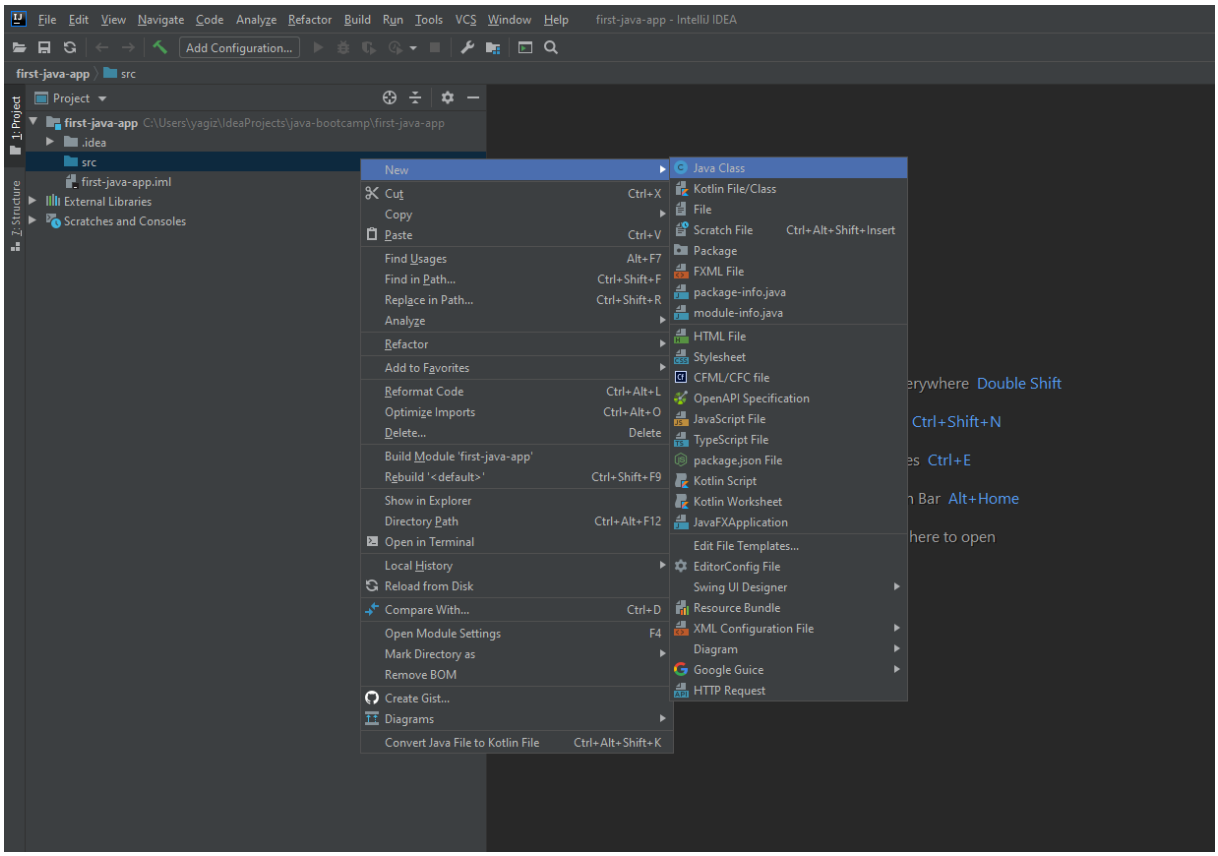


"Project name" ile belirtilen bölüme "first-java-app" yazalım ve "Finish" butonuna tıklayalım. Karşımıza şu şekilde bir ekran çıkacaktır:

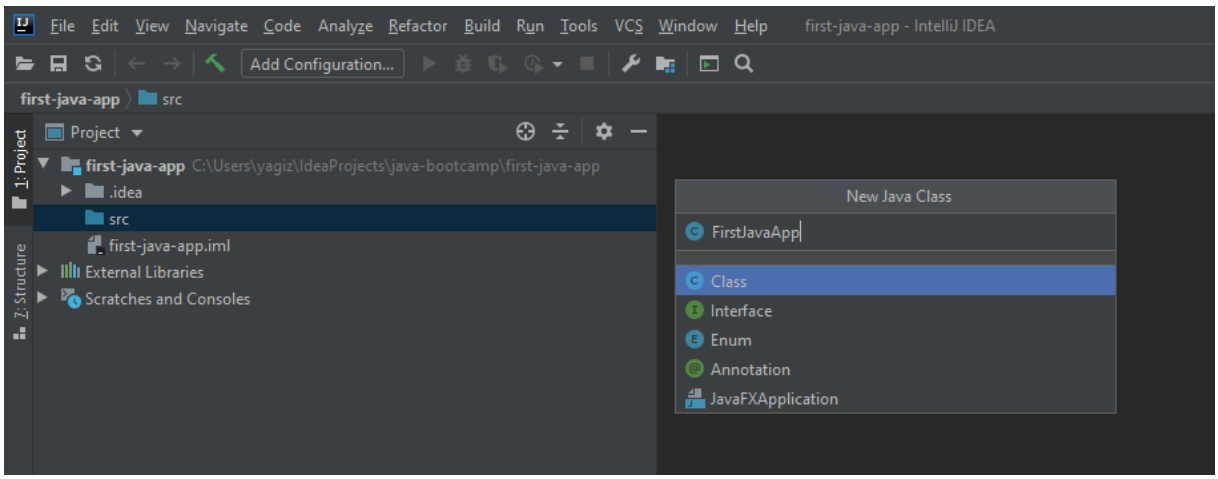


Solda bulunan "Project" başlıklı pencere projemizdeki dosyaları ve klasör yapısını gösterir. Buradan projemizi inceleyebilir, yeni dosyalar ekleyebilir veya var olan dosyaları silebiliriz.

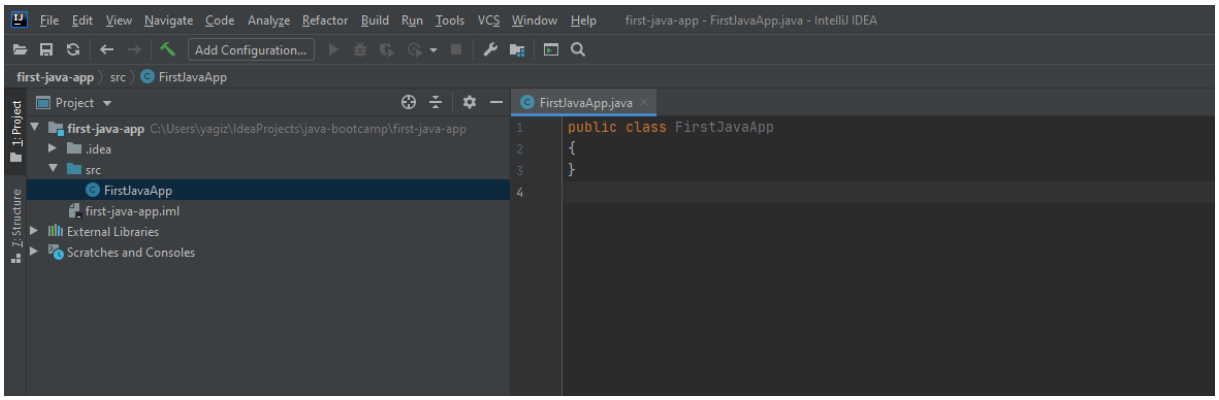
"src" adındaki klasörün üzerine gelip sağ tıklayalım ve açılan menüden sırasıyla "New" ve "Java Class" seçeneklerini seçelim. Bu sayede projemize yeni bir sınıf ekleyeceğiz.



Açılan pencerede bize, sınıfımıza vermek istediğimiz ismi soracaktır.



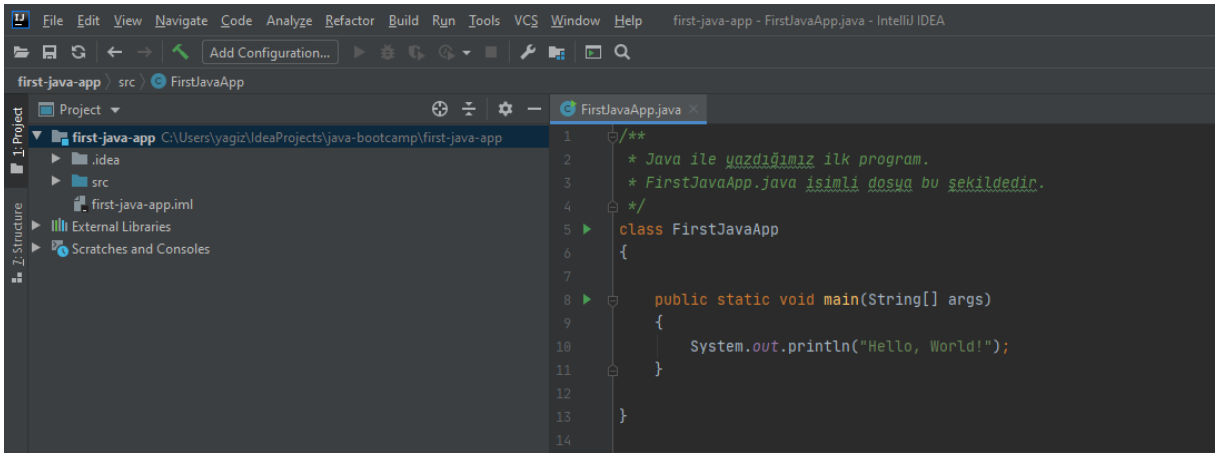
Buraya “FirstJavaApp” yazalım ve enter’a basalım. Karşımıza çıkacak ekran şu şekildedir:



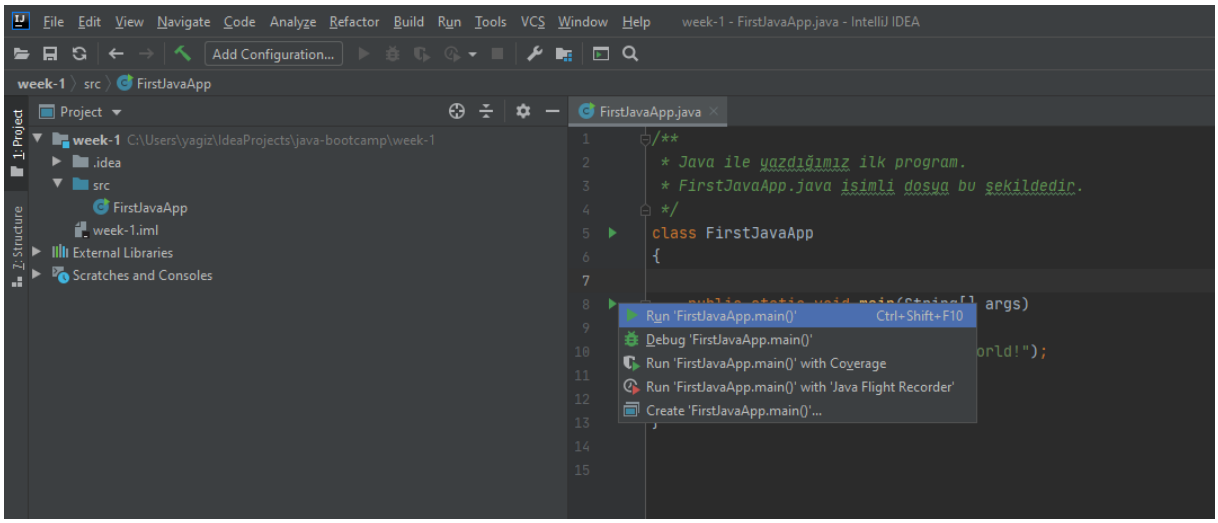
Sağda bulunan pencere bir metin editörüdür. Burada yeni oluşturduğumuz Java sınıfını görüyoruz. Bu pencereyi kullanarak kodlarımızı istediğimiz gibi düzenleyebiliriz. Şimdi sağdaki pencerede bulunan bütün metni silelim ve aşağıdaki metni kopyalayarak yapıştıralım:

```
/**
 * Java ile yazdığımız ilk program.
 * FirstJavaApp.java isimli dosya bu şekildedir.
 */
class FirstJavaApp
{

    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }

}
```

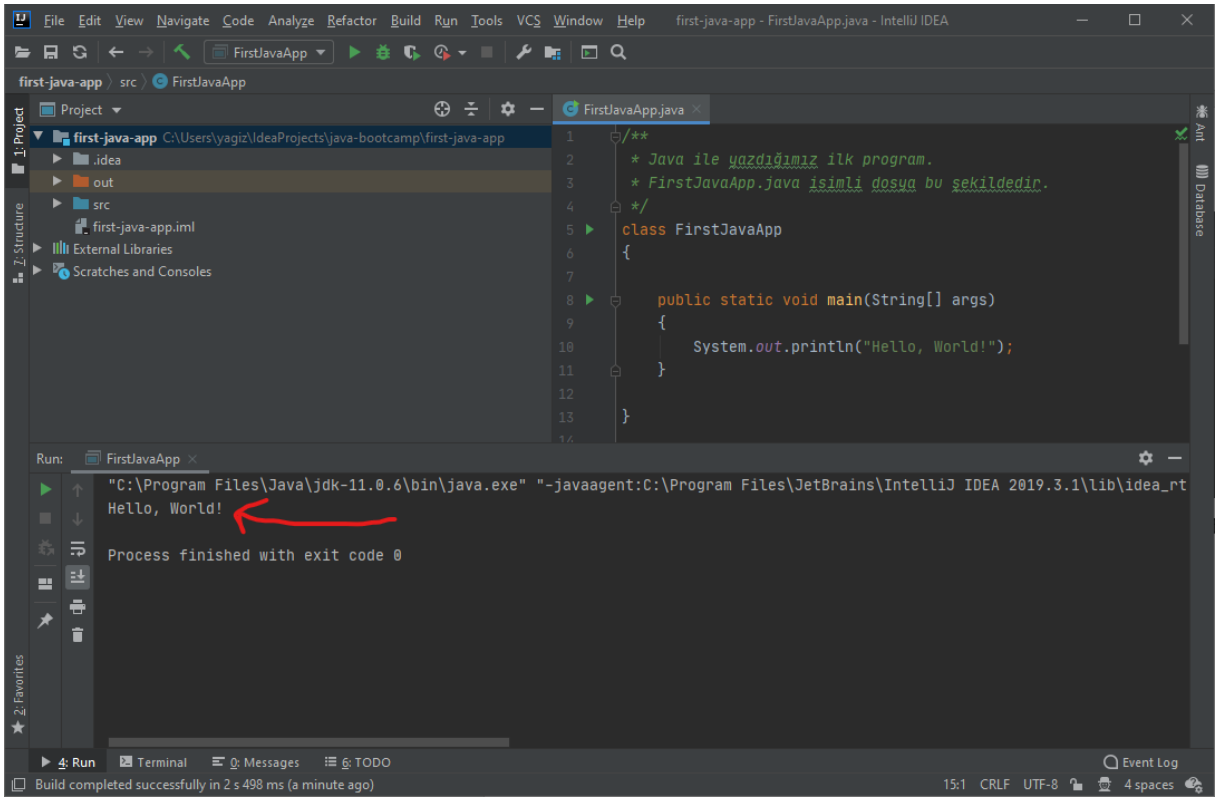
Sonuçta ekranımız şu şekilde gözükmelidir:



Görüldüğü üzere 5. ve 8. satırlarda üçgen şeklinde yeşil butonlar belirdi. Bu butonlardan herhangi birine tıklayalım ve açılan menüden ilk seçeneği seçelim:



Bu seçenek yazdığımız programı çalıştırmamızı sağlayacaktır. Çalıştırdıktan sonra sonuç şu şekilde olur:



Görüldüğü gibi aşağıda yeni bir pencere açıldı ve bu pencerede “Hello, World” yazdı. Şimdi programımızı daha ayrıntılı inceleyelim.

İlk programımızda ne yaptık?

Kodu incelerseniz, ilk satırın /**, 4. satırın ise */ karakterlerinden başka bir şey içermediğini görürsünüz. Bunlar Java’da çok satırlı yorumlar yazmak için kullandığımız işaretlerdir. Yorumlar, kaynak kodun içine yazdığınız,

programın çalışmasına hiçbir şekilde etki etmeyen yazılardır. Kodunuzun üzerine yorum satırları yazarak kodunuzu açıklayabilir veya birden fazla programcının çalıştığı bir projede diğer programcılar için notlar düşebilirsiniz. Java derleyicisi, kodunuzu derlerken yazdığınız yorumları görmezden gelecektir. Kısacası, yorumlar yalnızca kodun içerisinde olan ve yalnızca programcı için anlam taşıyan metinlerdir.

Eğer yazacağınız yorum tek satırdan oluşuyorsa, // ile belirtebilirsiniz. Aşağıdaki örnekte, tek ve çok satırlı yorumlar ayrı ayrı gösterilmiştir:

```
/**
 * Burası
 * çok satırlı
 * bir yorumdur.
 */

 // Burası tek satırlı bir yorumdur.

 // Tek satırlı yorumlarınızı kodunuzla aynı satıra da yazabilirsiniz:

 System.out.println("Hello, World!"); // Bu yorum kodun ardından geliyor
```

Kodumuzu incelemeye devam edelim. 5. satırda bir Java sınıfının tanımlandığını görüyoruz. Sınıfımızın ismi FirstJavaApp olarak belirlenmiş. Sınıflarla ilgili ayrıntıya daha sonra gireceğiz.

8. ve 11. satırlarda bir metot tanımlamasının yapıldığını görüyoruz. Metotlar, bir iş yapan yapısal bölümlerdir. Burada yazdığımız metot özel bir metottur. Bu metoda **ana metot** (*main method*) denir. Her uygulamanın bir giriş noktası olması gerekir; uygulama buradan çalışmaya başlar ve devam eder. Ana metotlar uygulamamızda çalışan ilk metottur. Ana metodun tanımı her zaman aynıdır ve yukarıda yazdığımız gibidir. Diğer yandan, başka metotların daha farklı tanımlamaları olabilir. Metotlarla ilgili ayrıntıya daha sonra gireceğiz.

9. satırda ise metodun içeriğini görüyoruz. Burada konsola “Hello, World” metnini yazdırıyoruz. “System.out” varsayılan konsolu belirtirken; “println” ise konsola metin yazdırmamızı sağlayan metodun ismini belirtir.

Karar mekanizmalarına giriş

Şimdi bir örnek daha görelim ve karar mekanizmalarına giriş yapmış olalım. Kodumuzu tamamen silelim ve aşağıdaki metni alıp kopyalayalım:

```
import java.util.Random;

class FirstJavaApp
{

    public static void main(String[] args)
    {
        Random random = new Random();
        int number = random.nextInt(11);

        System.out.println("Rastgele bir sayı tuttum: " + number);

        if (number < 5)
        {
            System.out.println("Sayı 5'ten küçük.");
        }
    }

}
```

Burada kısaca anlatmak gerekirse, bilgisayardan 0 ile 10 arasında rastgele bir sayı seçmesini istiyoruz, daha sonra bu sayıyı konsola yazdırıyoruz. Buna ek olarak, eğer sayı 5'ten küçükse konsolda bunu belirtiyoruz. Programı arka arkaya birkaç kez çalıştırırsanız şuna benzer sonuçlar alırsınız:

```
Rastgele bir sayı tuttum: 6
```

```
Rastgele bir sayı tuttum: 3
Sayı 5'ten küçük.
```

Görüldüğü gibi ilk çalıştırmamızda çıkan sonuçla ikincisindeki farklıdır. Buna **if bloğu** sebep olmaktadır. Java'da belli bir koşula bağlı olarak bir şey yapmak istediğimizde *if bloğunu* kullanırız. Yukarıdaki örnekte, "Sayı 5'ten küçük." yazısını yalnızca sayı gerçekten 5'ten küçükse yazdırıyoruz. Eğer sayı 5 veya 5'ten büyükse kodumuz if bloğuna girmez.

Belli bir koşula bağlı olarak iş yapmamızı sağlayan bu tarz ifadelere, **seçim ifadeleri** (*selection statements*) denir. Seçim ifadelerini daha sonra ayrıntılı olarak inceleyeceğiz.

Şimdi başka bir karar mekanizmasına göz atalım. Kodumuzu aşağıdaki gibi değiştirelim:

```
class FirstJavaApp
{

    public static void main(String[] args)
    {
        System.out.println("1'den 10'a kadar sayıyorum...");

        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }

        System.out.println("Saydım.");
    }

}
```

Bu programı çalıştırdığınızda aşağıdaki çıktıyı alırsınız:

```
1'den 10'a kadar sayıyorum...
1
2
3
4
5
6
7
8
9
10
Saydım.
```

Bu örneğimizde bilgisayardan bir işlemi defalarca yapmasını istedik. 1'den 10'a kadar olan sayıları konsola yazdırdık. Bu şekilde, belli bir kod bloğunu birden fazla kez çalıştırmamızı sağlayan ifadelere **döngü ifadeleri** (*iteration statements*) denir. Biz buradaki örnekte bir *for döngüsü* kullandık. Döngü ifadelerini ileride ayrıntıyla inceleyeceğiz.

VERİ TÜRLERİ, DEĞİŞKENLER VE SABİTLER

Bu bölümde, bir programlama dilinin en temel taşlarından olan veri türlerini ve değişkenleri öğreneceğiz. Diğer dillerin çoğunda da olduğu gibi, Java birçok veri türünü destekler. Bu veri türlerini değişken ve dizi oluştururken kullanırız.

Her şeyden önce şunu anlamalıyız ki, Java **tür kesinliği olan** (*strongly typed*) bir dildir. Bunun anlamı, her değişkenin ve her ifadenin bir türü vardır. Bu türler kesin bir şekilde tanımlanmıştır. Ayrıca, yapılan atamaların (*assignment*) hepsi, tür uyumluluğu açısından sıkı bir şekilde kontrol edilir. Tür uyumluluğuna aykırı olan herhangi bir şey kodumuzun derlenmesini engeller. Eğer kodumuzda herhangi bir tür uyumsuzluğu varsa, derleme aşamasından önce düzeltilmelidir.

İlkel veri türleri (*primitive data types*)

Java'da 8 farklı ilkel veri türü tanımlanmıştır. Bunlara ilkel denmesinin sebebi, çok sık kullanılan basit türler olmasıdır. Şöyle diyebiliriz: bu veri türlerini kullanmadığınız bir Java programı yazmayacaksınız.

İlkel veri türlerini 4 başlık altında toplayabiliriz:

- **Tamsayı türleri (*integers*):** 4 farklı tamsayı türü vardır: **byte**, **short**, **int** ve **long**. Bunları tamsayıları temsil etmek amacıyla kullanırız.
- **Ondalıklı sayı türleri (*floating point numbers*):** 2 farklı ondalıklı sayı türü vardır: **float** ve **double**.
- **Karakter türü:** 1 adet karakter türü vardır: **char**. Bunu kullanarak karakterleri temsil ederiz: harfler, rakamlar, noktalama işaretleri vs.
- **Mantıksal veri türü:** 1 adet mantıksal veri türü vardır: **boolean**. Bu türün yalnızca iki değeri vardır: **true** veya **false**. Bu türü kullanarak evet/hayır, var/yok, **doğru/yanlış** gibi ifadeleri temsil edebiliriz.

İlkel türler, karmaşık verileri değil, tek bir değeri temsil eder. Java nesne yönelimli bir dil olmasına rağmen, ilkel veri türleri bunun dışındadır. Nesne yönelimli bir dilde ilkel türlerin olmasının sebebi verimliliktir. Eğer ilkel veri türleri olmasaydı, bunun yerine kullanacağımız nesneler verimliliği düşürecekti. Bu türlere çok fazla ihtiyaç duyacağımız için bunlar ilkel olarak dile dahil edilmiştir.

İlkel veri türleri kesin bir şekilde tanımlanmıştır:

- Hafızada kaplayacağı alanın büyüklüğü belirlidir.
- Alabileceği değerler belli bir aralıkla sınırlandırılmıştır.

Şimdi ilkel türlerin hepsini ayrı ayrı inceleyelim.

Tamsayılar (integers)

4 farklı tamsayı türü vardır. Bunların hepsi işaretli (*signed*) türdür, yani hem pozitif (0'dan büyük) hem de negatif (0'dan küçük) değer alabilir. Bazı başka dillerde (örneğin C#) pozitif ve negatif tamsayılar için farklı veri türleri tanımlanmıştır; fakat Java'nın tasarımcıları bunu gereksiz bulmuştur.

byte

En küçük tamsayı türü byte'tır. 8 bit'ten oluşur ve adından da anlaşılacağı üzerede hafızada **1 byte** alan kaplar. **-128'den 127'ye kadar olan** değerleri tutabilir. Bu türü bir dosyanın içeriğini veya ağ üzerinden gelen bir veriyi okurken sıklıkla kullanırız.

short

16 bit'ten oluşur ve hafızada **2 byte** alan kaplar. **–32,768'den 32,767'ye kadar olan** değerleri tutabilir. Muhtemelen Java'da en az kullanılan veri türü budur.

int

32 bit'ten oluşur ve hafızada **4 byte** alan kaplar. **–2,147,483,648'den 2,147,483,647'ye kadar olan** değerleri tutabilir. Muhtemelen Java'da en çok kullanılan veri türü budur. Diğerlerine göre hafızada daha çok yer kapladığından, birçok durumda int yerine byte veya short kullanmanın daha yararlı olacağını düşünebilirsiniz; fakat bu yanlış bir varsayım olur. Birçok durumda, Java derleyicisi byte ve short değerleri int'e dönüştürür. Bu nedenle, eğer bir tamsayıya ihtiyaç duyuyorsanız, int en iyi tercih olacaktır.

long

64 bit'ten oluşur ve hafızada **8 byte** alan kaplar. int veri türünün yetmediği durumlar için üretilmiştir. Bu veri türünün değer aralığı çok geniştir. Alabileceği değerler **-9,223,372,036,854,775,807 ile 9,223,372,036,854,775,808** arasındadır.

Ondalıklı sayı türleri (*floating point numbers*)

Ondalıklı sayı türleri, matematikte reel sayılar olarak tanımlanan ondalıklı değerleri temsil eder. 2 farklı ondalıklı sayı türü vardır ve aralarındaki tek fark hafızada kapladıkları alan ve değer aralığıdır.

float

32 bit'ten oluşur ve hafızada **4 byte** alan kaplar. **Tek kesinlikli** (*single precision*) değerleri temsil eder.

double

64 bit'ten oluşur ve hafızada **8 byte** alan kaplar. Çift kesinlikli (*double precision*) değerleri temsil eder. Bunun anlamı şudur: virgülden sonra tutabileceği rakam sayısı float türüne göre daha fazladır. Çoğu durumda float ihtiyacınızı karşılayacak olsa da özellikle kesinlik gerektiren matematiksel işlemlerde double daha çok işe yarayacaktır.

Karakter türü

Karakterleri temsil eden veri türü **char**'dır. Şunu öncelikle belirtmeliyiz ki, Java **Unicode** karakter setini kullanır. Java'nın yazıldığı dönemde Unicode için en az 16 bit gerekiyordu. Bu nedenle, char veri türü hafızada **2 byte** alan kaplar. **0'dan 65,536'ya kadar olan** değerleri tutabilir. Negatif karakter yoktur.

Anlamamız gereken önemli bir nokta vardır. Bu veri türünün iki yönü vardır. Asıl amacı karakterleri temsil etmektir. Karakterleri temsil etmek için evrensel bazı standartlar geliştirilmiştir. Java'nın Unicode standardını kullandığını daha önce söylemiştik. Bu yönteme göre her bir karaktere karşılık bir sayı değeri atanır. Bu nedenle diyebiliriz ki, char aynı zamanda sayısal bir veri türüdür. Bu türü karakterleri temsil etmek için kullanabileceğiniz gibi, char değişkenler üzerinde matematiksel işlemler de yapabilirsiniz. Hatta daha sonra göreceğimiz gibi, char ve int türleri arasında dönüşüm de yapabilirsiniz.

Mantıksal veri türü

Java'da mantıksal değerleri temsil etmek amacıyla **boolean** türü vardır. Bu türdeki değişkenlerin iki değeri olabilir: **true** veya **false**. Daha sonra göreceğimiz ilişkisel operatörlerin döndürdüğü veri türü boolean'dır. Ayrıca if ve for gibi karar mekanizmalarında da boolean türüne ihtiyaç duyarız.

Bazı dillerde boolean türler sayısal olarak temsil edilir. Bu tip durumlarda 0 false değerini temsil ederken, diğer bütün sayılar true belirtir. Java'da ise durum böyle değildir. Java'nın geliştiricileri boolean türünü tam olarak

mantıksal bir veri türü olarak tasarlamışlardır. Dolayısıyla boolean ile sayısal türler arasında dönüşüm yapılamaz.

DEĞİŞKENLER VE SABİTLER (Variables and constants)

Hafızada veri tutmak için değişkenleri kullanırız. Değişkenlerle ilgili 5 önemli kavram vardır:

- **Tür:** Java'nın tür kesinliği olan bir dil olduğunu söylemiştik. Bunun bir sonucu olarak, her değişkenin bir türü vardır. Bu tür değişken tanımlarken belirtilir ve bir daha değiştirilemez. Türünü belirtmeden değişken tanımlayamazsınız.
- **İsim:** Her değişkene bir isim verilir. Bu ismi değişkene değer atamak ve gerekirse bu değeri değiştirmek için kullanırız. Değişkene verilecek isim tek bir kelimeden oluşmalıdır (boşluk içermemelidir). Değişken isimleri harflerden, rakamlardan ve alt çizgi (_) karakterinden oluşabilir. Değişken isimleri rakam ile başlayamaz.
- **Değer:** Değişkenler hafızada değer tutmak için kullanılır. Değişkene tanımlandığı anda bir değer verebileceğiniz gibi, daha sonra da değer atayabilirsiniz. Değişkenin değeri istenilen bir anda değiştirilebilir. Bu değerler değişkenin türüne göre sınırlandırılmıştır. Değişkene vereceğimiz değer türüyle uyumlu olmalıdır. Aksi halde Java derleyicisi kodumuzun derlenmesine izin vermez. Örneğin, boolean bir değişkene tamsayı değer atayamazsınız.
- **Kapsam (Scope):** Her değişkenin bir kapsamı vardır. Bu kapsam, değişkenin program içerisinde geçerli olduğu alanı belirler. Bir değişkene kapsamı dışında erişemezsiniz.
- **Yaşam süresi (Lifetime):** Sürekli değişken oluşturursak bir süre sonra bilgisayarın hafızası tükenebilir. Bunun için her programlama dilinde bir **çöp toplama** (*garbage collection*) mekanizması vardır. Java'da her değişkenin bir ömrü vardır ve gerektiği anda hafızadan silinirler. Yaşam süresi çoğu zaman değişkenin kapsamıyla bağlantılıdır.

Değişken tanımlamak

Java'da bir değişken, sırasıyla önce türü ve sonra ismi belirterek tanımlanır.

```
int number;
// number isminde, int türünde bir değişken tanımlanmış
```

Aynı satırda birden fazla değişken tanımlayabilirsiniz, fakat türleri aynı olmak zorundadır:

```
double a, b, c;
// double türünde 3 ayrı değişken tanımlanmış
```

Değişkeni tanımladıktan sonra, atama operatörü (=) kullanarak değişkene bir değer verebilirsiniz:

```
double pi;    // Önce double türünde bir değişken tanımladık
pi = 3.14;    // Daha sonra bu değişkene bir değer verdik
```

Eğer bir değişkene hemen değer atayacaksanız, bunu iki satırda yapmak yerine tek bir satırda halledebilirsiniz:

```
double pi = 3.14;
```

Aynı satırda birden fazla değişken tanımlıyorsanız bunlara şu şekilde değer verebilirsiniz:

```
int year = 2020, age = 25;
// Aynı satırda 2 farklı değişken tanımlanmış ve ikisine de değer
verilmiş
```

Değişkene verilen değer herhangi bir anda değiştirilebilir:

```
int year = 2020;    // Bir değişken tanımlanmış ve değer verilmiş
year = 2021;        // Değişkenin değeri değiştirilmiş
year = 2022;        // Değişkenin değeri tekrar değiştirilmiş
```

Bir değişkeni tanımladığınız zaman, aynı kapsam içinde aynı isimde başka bir değişken tanımlayamazsınız:

```
boolean a = true;   // a isminde bir değişken tanımlanmış
boolean a = false;  // Bu satır hataya neden olur, a değişkeni zaten var
```

Buraya kadar olan örneklerimizde değişkene hep kesin bir değer atadık; fakat Java'da bir metodun sonucunu da değişkene atayabilirsiniz:

```
double result = Math.sqrt(16.0);
// Karekök metodu çağrılıyor ve sonucu bir değişkene atanıyor
// Bu işlem sonucunda result değişkeninin değeri 4.0 olur
```

Değişkenlerin kapsamı ve yaşam süresi

Değişkenlerin kapsamını ve yaşam süresini anlamak için önce **kod bloğu** (*block*) kavramını incelemeliyiz.

Java'da kodlarımızı satırlar halinde yazarız. Her bir satırın sonuna noktalı virgül işareti konur. Bunu satırın bittiğini belirtmek için yaparız. Birden fazla satırdan oluşan kodlarımızı ise bir blok içine alırız. Bunun için küme parantezleri ({ ve }) kullanılır. Her sınıfın ve her metodun kendine ait kod blokları vardır. Bunun yanında, bazı özel kod blokları da bulunur; hatta kendimiz de kod blokları açabiliriz. Aşağıdaki örneği inceleyelim:

```
class CodeBlocksDemo
{ // sınıfın kod bloğu başlıyor

    public static void main(String[] args)
    { // main metodunun kod bloğu başlıyor

        int year = 2020;

        if (year >= 2000)
        { // if bloğu başlıyor

            System.out.println("Milenyum çağındayız.");

        } // if bloğu bitiyor

        for (int i = 0; i < 10; i++)
        { // for bloğu başlıyor

            System.out.println(i);

        } // for bloğu bitiyor

        { // kod bloğu başlıyor

            System.out.println("Burası isimsiz bir kod bloğudur.");

        } // kod bloğu bitiyor


    } // main metodunun kod bloğu bitiyor

} // sınıfın kod bloğu bitiyor
```

Kısaca belirtmek gerekirse, bir değişkenin kapsamı tanımlandığı kod bloğuyla sınırlıdır. Bu blok içinde değişkene erişebilirsiniz. Kod bloğunun dışına çıktığınızda ise artık değişkeni kullanamazsınız. Yukarıdaki örnekteki kod bloklarını aynen bırakalım ve aşağıdaki örneği inceleyelim:

```
class CodeBlocksDemo
{

    int a = 1;

    public static void main(String[] args)
    {
        // Burada a değişkenine erişimimiz var

        int b = 2;
        // Burada a ve b değişkenlerine erişimimiz var

        if (b >= 2000)
        {
            int c = 3;
            // Burada a, b ve c değişkenlerine erişimimiz var
        }

        // c'nin kapsamı bitti, artık erişemeyiz

        for (int i = 0; i < 10; i++)
        {
            int d = 4;
            // Burada a, b ve d değişkenlerine erişimimiz var
        }

        // d'nin kapsamı bitti, artık erişemeyiz

        {
            int e = 5;
            // Burada a, b ve e değişkenlerine erişimimiz var
        }

        // e'nin kapsamı bitti, artık erişemeyiz
    }

    // b'nin kapsamı bitti, burada yalnızca a değişkenine erişebiliriz

}
```

Yukarıdaki örnekten de anlaşılacağı üzere, bir kod bloğunun içinde tanımlanan değişkene dışarıdaki bir bloktan erişilemez. Diğer yandan, bunun tam tersi geçerli değildir. Bir kod bloğunda tanımlanan değişkene içerideki bir bloktan da erişilebilir.

İlkel veri türüne sahip değişkenler kapsam dışına çıkınca otomatik olarak hafızadan da silinirler. Diğer bir deyişle, ilkel veri türüne sahip değişkenlerin yaşam süresi kapsamlarıyla aynıdır. Fakat bu diğer veri türündeki değişkenler için geçerli değildir. İlkel olmayan veri türündeki değişkenler kapsam dışına çıksa da hafızada kalmaya devam edebilir. Bunu daha sonra ayrıntıyla anlatacağımız için şimdilik kısaca geçiyoruz.

Sabitler (constants)

Değeri değiştirilemeyen değişkenlere **sabit** (*constant*) denir. Bazen, yazdığımız programlarda bazı değişkenlerin bir kere tanımlanmasını ve daha sonra değerlerinin değiştirilmemesini isteriz. Bu gibi durumlarda sabit tanımlarız. Sabitlerin değişkenlerden iki temel farkı vardır:

- Bir değişkeni sabit yapmak istiyorsanız **final** belirteci ile tanımlamalısınız.
- Sabitlerin değeri sonradan değiştirilemediği için tanımladığınız anda değer atamalısınız.

```
boolean someVariable = false;
// Bir değişken tanımlanmış. Değeri daha sonra değiştirilebilir.
final double pi = 3.14;
// Bir sabit tanımlanmış. Değeri daha sonra değiştirilemez.
```

Sabitlerle ilgili hataya neden olabilecek aşağıdaki örnekleri inceleyelim:

```
final byte x;    // Bu satır hataya neden olur; çünkü sabit olarak
belirlenmesine rağmen bir değer atanmamış
```

```
final int year = 2020;
year = 2021;    // Bu satır hataya neden olur; çünkü sabitin değeri
değiştirmeye çalışılıyor
```

Tür dönüşümleri (type casting)

Türler arasında kurallara aykırı olmadığı sürece dönüşüm yapılabilir. Tür dönüşümüne, türleri birbirinden farklı değişkenler arasında atama yaparken ihtiyaç duyulur. Örneğin, int türündeki bir değişkenin değerini long türündeki bir değişkene aktarmak istiyorsanız. Tür dönüşümleri ikiye ayrılır.

Dolaylı tür dönüşümü (Implicit type casting)

İlkel veri türlerini anlatırken, her bir türün kendine ait bir değer aralığı olduğundan söz etmiştik. Eğer değer aralığı düşük bir türden yüksek bir türe dönüşüm yapılıyorsa burada dolaylı tür dönüşümü söz konusudur.

Örneğin, int türünde bir değişkeniniz var. Bunun değerini long türündeki bir değişkene aktarmak istiyorsunuz. Bildiğiniz gibi, int türünün alabileceği bütün değerler long türünün değer aralığında zaten tanımlıdır. Dolayısıyla bu dönüşüm sorunsuz bir şekilde gerçekleşecektir. Aşağıdaki örneği inceleyelim:

```
int a = 5;
long b = a;
```

Yukarıdaki örnekte ilk önce int türünde bir değişken tanımlanıyor ve bu değişken üzerinden long türündeki bir değişkene atama yapılıyor. Burada gördüğünüz gibi, atama operatörü (=) kullanarak değişkenin ismini yazmanız yeterlidir. İlk bakışta burada bir tür uyumsuzluğu varmış gibi gözükebilir, int türünde bir değeri long türüne aktarmaya çalışıyorsunuz. Fakat burada arka planda bir tür dönüşümü yapılmaktadır. Bizim bu dönüşüm için ekstra kod yazmamız gerekmediğinden, bu tarz dönüşümlere **dolaylı tür dönüşümü** denir.

Dolaylı tür dönüşümü yalnızca daha az kapsayıcı bir türden daha çok kapsayıcı bir türe doğru yapılabilir. Bu nedenle bu tür dönüşümler **genişleyen dönüşüm** (*widening conversion*) olarak da adlandırılır.

Doğrudan tür dönüşümü (Explicit type casting)

Dolaylı tür dönüşümünün aksine, daha kapsayıcı bir türden daha az kapsayıcı bir türe doğru yapılan dönüşümlere **doğrudan tür dönüşümü** denir. Doğrudan denmesinin sebebi, yapılacak dönüşümün yönünü belirtmemiz gerektiğindendir.

Bunu gösterebilmek için yukarıdaki örneğin tam tersini inceleyelim:

```
long a = 5;
int b = (int) a;
```

Görüldüğü gibi, doğrudan tür dönüşümü yaparken, dönüştürülecek türün adı değişkenin adından önce parantez içinde yazılır. Bunu yaparak Java'ya, türü dönüştüreceği yönü belirtmiş oluruz.

Doğrudan tür dönüşümleri, **daralan dönüşüm** (*narrowing conversion*) olarak da adlandırılır.

Operatörler

Kaynak kod içerisinde bazı özel işlevleri olan karakterlere **operatör** denir. Kod yazarken, bazı basit işlemler yapma ihtiyacı duyarız. Örneğin, tamsayı türündeki iki değişkenin değerlerini toplamak isteriz. Bunu toplama operatörüyle (+) yaparız. Java'da bunun gibi birçok operatör vardır. Şimdi bunları gruplar halinde inceleyelim.

Aritmetik operatörler

Sayısal türler üzerinde aritmetik işlemler yapmamızı sağlayan operatörlere aritmetik operatörler denir. Aritmetik operatörleri önce tablo halinde görelim, sonra ayrıntılı inceleyelim:

+	Toplama
-	Çıkarma
*	Çarpma
/	Bölme
%	Mod alma (Bölümden kalanı bulma)
++	1 artırma (*increment*)
+=	Artırma ve atama
--	1 azaltma (*decrement*)
-=	Azaltma ve atama
*=	Çarpma ve atama
/=	Bölme ve atama
%=	Mod alma ve atama

Yukarıda listelenen operatörleri sayısal türler ve char üzerinde kullanabilirsiniz; fakat boolean üzerinde kullanamazsınız.

Dört işlem operatörleri

Toplama, çıkarma, çarpma ve bölme operatörleri, sayısal türler üzerinde dört işlem yapmamızı sağlar.

```
int a = 6 + 3;    // a'nın değeri 9 olur
int b = 6 – 3;    // b'nin değeri 3 olur
int c = 6 * 3;    // c'nin değeri 18 olur
int d = 6 / 3;    // d'nin değeri 2 olur
```

Burada dikkat etmemiz gereken nokta bölme işlemidir. Matematikte hiçbir sayı sıfıra bölünemeyeceğinden, bu tarz işlemler *ArithmeticException* hatası fırlatır.

```
int a = 0;
int b = 7 / a; // Bu satır hataya neden olur: sıfıra bölme
```

Mod Alma (%) Operatörü

Bir bölme işlemindeki kalanı bulmak için % operatörünü kullanırız:

```
int a = 10 % 2;    // a'nın değeri 0 olur
int b = 10 % 3;    // b'nin değeri 1 olur
int c = 10 % 4;    // c'nin değeri 2 olur
int d = 10 % 6;    // d'nin değeri 4 olur
```

Bileşik aritmetik operatörler (Compound arithmetic operators)

Bazen bir değişkenin üzerinde bir aritmetik işlem yapmak ve bu işlemin sonucunu yine o değişkene atamak isteriz. Bunu şu şekilde yapmak mümkündür:

```
int a = 5;     // a'nın değeri 5'tir
a = a + 10;    // a'nın değeri 15 olur
```

Java bu gibi işlemleri daha kısa kodla yazmak için bileşik aritmetik operatörleri destekler. Bu sayede, yukarıdaki işlemi aşağıdaki şekilde yapabiliriz:

```
int a = 5;    // a'nın değeri 5'tir
a += 10;      // a'nın değeri 15 olur
```

Yukarıdaki örneği incelersek, += operatörü a değişkeninin değerini 10 ile toplar ve daha sonra sonucu yine a değişkenine atar.

Bileşik operatörleri 1 artırma (++) ve 1 azaltma (--) operatörleri haricindeki bütün aritmetik operatörlerle kullanabilirsiniz. Aşağıdaki örnekleri inceleyelim:

```
int a = 6;   // a'nın değeri 6'dır
a += 3;      // a'nın değeri 9 olur

int b = 6;   // b'nin değeri 6'dır
b -= 3;      // b'nin değeri 3 olur

int c = 6;   // c'nin değeri 6'dır
c *= 3;      // c'nin değeri 18 olur

int d = 6;   // d'nin değeri 6'dır
d /= 3;      // d'nin değeri 2 olur

int e = 7;   // e'nin değeri 7'dir
e %= 3;      // e'nin değeri 1 olur (7'nin 3'le bölümünden kalan 1'dir)
```

1 artırma ve 1 azaltma operatörleri (Increment and decrement operators)

Çoğu zaman yazdığımız programlarda bir tamsayının değerini 1 artırmak veya 1 azaltmak isteriz. Bunu şu şekilde yapabiliriz:

```
int a = 5;    // a'nın değeri 5'tir
a += 1;       // a'nın değeri 6 olur

int b = 5;    // b'nin değeri 5'tir
b -= 1;       // b'nin değeri 4 olur
```

1 artırma ve 1 azaltma işlemleri çok sık ihtiyaç duyulan işlemler olduğundan, Java'nın tasarımcıları bunun için özel operatörler geliştirmiştir. Bu operatörleri kullanarak yukarıdaki işlemi daha kısa bir şekilde aşağıdaki gibi yapabiliriz:

```
int a = 5;    // a'nın değeri 5'tir
a++;          // a'nın değeri 6 olur

int b = 5;    // b'nin değeri 5'tir
b--;          // b'nin değeri 4 olur
```

Bu iki operatörle ilgili özel bir durum vardır. Bu operatörlerinin yönü işlem önceliği bakımından önem arz eder. Bunu açıklayabilmek için aşağıdaki örneği inceleyelim:

```
int a = 10;
System.out.println(a++);
// a'nın değeri önce elde edildi ve konsola yazdırıldı, sonra 1 artırıldı,
// yani konsolda 10 yazar; fakat a'nın değeri 11 olmuştur
System.out.println(a);
// Şimdi a'nın değeri konsola yazdırıldığında konsolda 11 görürüz
```

1 artırma ve 1 azaltma operatörleri aynı zamanda değer döndüren operatörlerdir. Değişkenden sonra yazıldıklarında, önce değişkenin o anki değerini döndürürler, sonra işlemi gerçekleştirip sonucu değişkene atarlar. Eğer değer döndürme işleminin en son yapılmasını istiyorsanız, bu operatörleri değişkenin önüne yazmalısınız:

```
int a = 10;
System.out.println(++a);
// a'nın değeri önce 1 artırıldı, sonra konsola yazdırıldı
// yani konsolda 11 yazar
```

Bit seviyesi operatörleri (Bitwise operators)

Bilgisayarda her veri bit'lerle ifade edilir. Bir bitin değeri 0 veya 1'den oluşur. Hafızanın en küçük birimi ise byte'tır ve 1 byte 8 bit'ten oluşur. Hafızada tuttuğumuz veriler üzerinde bit seviyesinde işlem yapmamızı sağlayan operatörlere **bit seviyesi operatörleri** denir. Bu operatörleri kısaca aşağıdaki tablodan inceleyelim:

~	Tersini alma
&	Ve işlemi
|	Veya işlemi
^	XOR işlemi
>>	1 bit sağa kaydırma
>>>	0 ile doldurarak 1 bit sağa kaydırma
<<	1 bit sola kaydırma
&=	Ve işlemi ve atama
|=	Veya işlemi ve atama
^=	XOR işlemi ve atama
>>=	1 bit sağa kaydırma ve atama
>>>=	0 ile doldurarak 1 bit sağa kaydırma ve atama
<<=	1 bit sola kaydırma ve atama

Boolean mantıksal operatörleri

Mantıksal operatörler, boolean değerler üzerinde ve, veya gibi mantıksal işlemler yapmamızı sağlar. Bu operatörleri kısaca aşağıdaki tablodan inceleyelim:

&	Mantıksal ve işlemi
|	Mantıksal veya işlemi
^	Mantıksal XOR işlemi
&&	Kısa devre ve işlemi
||	Kısa devre veya işlemi
!	Mantıksal tersini alma
&=	Mantıksal ve işlemi ve atama
|=	Mantıksal veya işlemi ve atama
^=	Mantıksal XOR işlemi ve atama
==	Eşitlik kontrolü
!=	Eşitsizlik kontrolü
?:	Üçlü if-then-else

Bu tabloda listelenen operatörlerin 2 tanesi hariç hepsi **ikili** (*binary*) operatördür, yani iki değişken üzerinde işlem yapar. Tersini alma (**!**) **tekli** (*unary*), üçlü if-then-else (**?:**) ise **üçlü** (*ternary*) operatördür.

Şimdi ikili operatörlerin işlevlerini kısaca inceleyelim:

- **Ve operatörü:** İki değişkenin değeri de **true** ise sonuç **true**, aksi halde **false** olur.
- **Veya operatörü:** Değişkenlerden birinin değeri **true** ise sonuç **true**, aksi halde **false** olur.
- **XOR operatörü:** Değişkenlerin değeri birbirinden farklı ise sonuç **true**, aksi halde **false** olur.
- **Kısa devre ve operatörü:** Ve operatörü ile aynı işi yapar; fakat ilk değişken **false** ise ikinci ifadeye hiç bakılmaz.
- **Kısa devre veya operatörü:** Veya operatörü ile aynı işi yapar; fakat ilk değişken **true** ise ikinci ifadeye hiç bakılmaz.
- **Eşitlik operatörü:** İki değişken (nesne) birbirine eşitse **true**, aksi halde **false** döndürür.

- **Eşitsizlik operatörü:** İki değişken (nesne) birbirine eşitse **false**, aksi halde **true** döndürür.

Bu operatörlerle ilgili aşağıdaki tabloları inceleyelim:

A	B	A & B
true	true	true
true	false	false
false	true	false
false	false	false

A	B	A | B
true	true	true
true	false	true
false	true	true
false	false	false

A	B	A ^ B
true	true	false
true	false	true
false	true	true
false	false	false

A	B	A == B
true	true	true
true	false	false
false	true	false
false	false	true

A	B	A != B
true	true	false
true	false	true
false	true	true
false	false	false

Tersini alma (!) operatörü

Değişkenin değeri true ise false, false ise true döndürür.

```
boolean b = true;
boolean c = !b;
System.out.println(c);    // Konsola false yazar
System.out.println(!c);   // Konsola true yazar
```

Üçlü if-then-else operatörü

Üçlü if-then-else operatörü 2 karakterden oluşur ve 3 değer üzerinde işlem yapar. Soru işaretinden önce boolean bir ifade yazılır. Eğer bu ifade **true** ise soru işareti ile iki noktanın arasındaki değeri, aksi halde iki noktadan sonraki değeri döndürür. Bu operatörün yapısı şu şekildedir:

[boolean ifade] **?** [1. değişken] **:** [2. değişken]

Daha iyi anlamak için aşağıdaki örneği inceleyelim:

```
boolean a = true;
boolean b = false;
System.out.println(a ? 3 : 5);    // Konsola 3 yazar
          System.out.println(b ? 3 : 5);    // Konsola 5 yazar
```

İlişkisel (relational) operatörler

İki değişkenin birbirine göre ilişkisini kontrol eden operatörlere ilişkisel operatörler denir. Bu operatörler aşağıda listelenmiştir:

==	Eşitlik kontrolü
!=	Eşitsizlik kontrolü
<	Küçüktür kontrolü
>	Büyüktür kontrolü
<=	Küçük eşittir kontrolü
>=	Büyük eşittir kontrolü

Bu operatörlerle ilgili aşağıdaki örnekleri inceleyelim:

```
System.out.println(5 < 5);     // Konsola false yazar
System.out.println(5 <= 5);    // Konsola true yazar
System.out.println(6 > 7);     // Konsola false yazar
System.out.println(4 >= 1);    // Konsola true yazar
```

Operatör önceliği

Her operatörün bir öncelik sıralaması vardır. Matematikte çarpma ve bölmenin toplama ve çıkarmaya göre daha öncelikli olması gibi, bazı operatörler diğerlerinden daha öncelikli olarak işletilir. Java ifadeleri yazarken bu öncelik sırasına dikkat etmek gerekir. Aşağıdaki tabloda Java operatörlerinin en yüksekten en düşüğe doğru öncelik sıralamasını inceleyebilirsiniz (aynı satırda yazan operatörlerin öncelik sıralaması birbirine eşittir):

En yüksek	• ++ (Değişkenden sonra yazılan) • -- (Değişkenden sonra yazılan)
	• ++ (Değişkenden önce yazılan) • -- (Değişkenden önce yazılan) • ~ (Bit seviyesinde tersini alma) • ! (Mantıksal tersini alma) • + (Pozitif sayı belirteci) • - (Negatif sayı belirteci) • Tür dönüşümü
	• * (Çarpma işlemi) • / (Bölme işlemi) • % (Mod alma)
	• + (Toplama işlemi) • - (Çıkarma işlemi)
	• >> (1 bit sağa kaydırma) • >>> (0 ile doldurarak 1 bit sağa kaydırma) • << (1 bit sola kaydırma)
	• > (Büyüktür) • >= (Büyük eşittir) • < (Küçüktür) • <= (Küçük eşittir) • instanceof
	• == (Eşitlik kontrolü) • != (Eşitsizlik kontrolü)
	& (Ve işlemi)
	^ (XOR işlemi)
	| (Veya işlemi)

	&& (Kısa devre ve işlemi)
	|| (Kısa devre veya işlemi)
	?: (Üçlü if-then-else)
	-> (Lambda operatörü)
En düşük	• = (Atama operatörü) • Bileşik atama operatörleri (+=, %=, &= vs.)

Parantez kullanmak

Parantez kullanarak operatör önceliğini değiştirebilirsiniz. Örneğin, şu ifadeyi göz önüne alalım:

```
a >> b + 3
```

Toplama operatörünün önceliği sağa kaydırmaya operatörüne göre yüksel olduğu için b'nin değerine 3 eklenir, sonra a'nın bitleri sağa kaydırılır. Eğer sağa kaydırma işleminin toplamadan önce yapılmasını istiyorsanız ifadeyi şu şekilde yeniden yazmalısınız:

```
(a >> b) + 3
```

İşlem önceliğini değiştirmek için parantez kullanmanın programın performansına olumsuz bir etkisi yoktur. Bu nedenle, gereksiz olsa bile, yazılan ifadelerin okunurluğunu artırmak amacıyla parantez kullanabilirsiniz.

DİZİLER (ARRAYS)

Değişkenler hafızada tek bir değer tutmamızı sağlar. Fakat bazı durumlarda, birden çok veriyi bir arada bulundurmak isteriz. Örneğin, bir sınıfta okuyan 20 öğrenci olsun. Bu 20 öğrenci için hafızada ayrı ayrı 20 tane değişken oluşturmak yerine, tek bir değişken kullanarak 20 öğrencinin koleksiyonunu tutmak isteyebiliriz.

Bu gibi durumlarda dizileri kullanırız. Dizi, aynı türden birden fazla değişkeni tutmamızı sağlayan hafıza birimidir. Kısaca, dizileri aynı türden elemanları gruplamak için kullanırız. Dizi oluşturduktan sonra dizinin içerisindeki elemanlara indeks numarasıyla ulaşır ve değiştiririz. Ayrıca, çok boyutlu

diziler oluşturmak da mümkündür. Tek boyutlu dizi kullanabileceğimiz gibi 2 veya 3 boyutlu diziler de oluşturabiliriz. 2 boyutlu dizilere matris denir.

Tek boyutlu diziler

Tek boyutlu diziler basitçe, aynı türden elemanların listesini tutan bir yapıdır.

Dizi oluşturmak için, önce dizide yer alacak elemanların türü belirtilir, sonra diziye bir isim verilir ve isimden sonra köşeli parantezler ([ve]) konulur.

```
int numbers[];    // Burada numbers isminde bir dizi oluşturuluyor
```

Köşeli parantezleri değişken isminden sonra koymak yerine, tür isminden sonra da yazabilirsiniz. Örneğin aşağıdaki kodun yukarıdakiyle bir farkı yoktur:

```
int[] numbers;    // Burada numbers isminde bir dizi oluşturuluyor
```

Diziler **new** deyimiyle oluşturulur. Dizi oluştururken kapasite değeri vermek zorunludur. Kapasite değeri, dizinin kaç eleman barındıracağını belirtir. Aşağıdaki örnekte, 5 adet int değişkeni tutabilecek bir dizi oluşturuluyor:

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

Bu ifade çalıştırıldığında, hafızada 5 adet int değişken için yetecek kadar alan ayrılır. Bu alanı düzenleyebilmek için indeks numaraları kullanırız. **Dizi indeksleri 0'dan başlar** ve kapasitenin 1 eksiğine kadar gider. Örneğin, yukarıdaki dizinin indeksleri 0'dan 4'e kadardır. Şimdi bu dizinin ilk elemanını verelim:

```
numbers[0] = 10;    // Dizinin ilk elemanı 5 olarak ayarlandı.
```

Dizinin diğer elemanlarını şu şekilde verelim:

```
numbers[1] = 15;
numbers[2] = 20;
numbers[3] = 25;
numbers[4] = 30;
```

Bu kodlar çalıştırıldığında dizinin elemanları sırasıyla aşağıdaki gibi olur:

{ 10, 15, 20, 25, 30 }

Aşağıdaki kodu çalıştırdığınızda konsola 25 yazar:

```
System.out.println(numbers[3]);
```

Dizilerle uğraşırken indeks numaralarına çok dikkat etmelisiniz. Eğer dizinin aralığı dışında bir indekse erişmeye çalışırsanız, **IndexOutOfBoundsException** hatası meydana gelir.

```
System.out.println(numbers[5]);    // Hata!
```

Yukarıdaki satır hataya neden olur; çünkü *numbers* dizisinin kapasitesi 5 olmasına rağmen dizinin 6. elemanına erişmeye çalışıyoruz.

Eğer dizinin içindeki elemanlar dizi oluşturulurken belliyse, diziyi oluştururken elemanları küme parantezi içinde ve virgülle birbirinden ayırarak verebiliriz:

```
String[] weekDays = new String[] { "Pazartesi", "Salı", "Çarşamba",
"Perşembe", "Cuma", "Cumartesi", "Pazar" };
```

Bu şekilde oluşturulan dizilere kapasite vermemize gerek yoktur; çünkü kapasite değeri zaten eleman sayısından bellidir. Yukarıdaki örnekde *weekDays* dizisinin kapasitesi otomatik olarak 7 olur.

Yukarıdaki gibi dizi oluştururken *new* deyimini kullanmaya gerek yoktur. Yani, yukarıdaki kodu aşağıdaki gibi yazabiliriz:

```
String[] weekDays = { "Pazartesi", "Salı", "Çarşamba", "Perşembe", "Cuma",
"Cumartesi", "Pazar" };
```

Dizinin kapasitesini öğrenmek

Her dizinin **length** adında bir özelliği bulunur. Bu özelliği kullanarak dizinin kapasitesini öğrenebilirsiniz.

Örneğin, aşağıdaki kodu inceleyelim:

```
int[] numbers = new int[100];
System.out.println(numbers.length);    // Konsolda 100 yazar
```

Çok boyutlu diziler

Bir değişkenin dizi olduğunu köşeli parantezler ile belirtmiştik. Bir tane köşeli parantez tek boyutlu dizi belirtir. Eğer çok boyutlu dizi oluşturmak istiyorsak, boyut sayısı kadar köşeli parantez belirtmeliyiz. Örneğin aşağıdaki satır 2 boyutlu bir dizi (yani matris) belirtir:

```
int matrix[][];
```

İlk köşeli parantez birinci boyutu (satırları), diğeri ise ikinci boyutu (sütunları) belirtir. Aşağıdaki kodu çalıştırırsak, 3 satırlı ve 4 sütunlu bir matris oluşturur:

```
int matrix[][] = new int[3][4];
```

Bu matrisin bütün elemanlarına ulaşmak için kullanmamız gereken indeks numaralarını aşağıdaki tabloda görebilirsiniz:

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

```
matrix[1][2];    // Matrisin 2. satır ve 3. sütunundaki elemana erişiliyor
matrix[0][3];    // Matrisin 1. satır ve 4. sütunundaki elemana erişiliyor
matrix[2][0];    // Matrisin 3. satır ve 1. sütunundaki elemana erişiliyor
```

Şimdi güzel bir örnek yapalım. 3 satırdan ve 4 sütundan oluşan bir matris yaratalım ve bu matrisin elemanlarını sırayla 1'den başlayacak şekilde dolduralım. Aşağıdaki kodu inceleyelim:

```
int[][] matrix = new int[3][4];
int number = 1;

for (int x = 0; x < matrix.length; x++)
{
    int[] row = matrix[x];

    for (int y = 0; y < row.length; y++)
    {
        row[y] = number;
        number++;
    }
}
```

Şimdi yukarıdaki kodu inceleyelim. İki boyutlu diziyi oluşturduktan sonra önce for döngüsüyle dizinin satırlarını geziyoruz. Daha sonra içerideki for döngüsüyle dizinin sütunlarını dolaşıyoruz. Bu örneği vermemizdeki amaç, matrisin elemanlarına ulaşmak için iç içe 2 for döngüsü kullanmak gerektiğini göstermektir. Ayrıca dizinin *length* metodunun faydasını da burada görmüş oluyoruz.

Yukarıdaki kod çalıştığında matrisin elemanları şu şekilde olur:

1	2	3	4
5	6	7	8
9	10	11	12

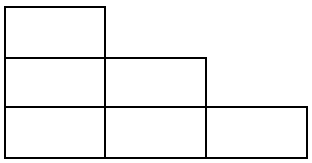
Sütun kapasiteleri farklı matris oluşturmak

2 boyutlu dizilere matris denir. Başka bir açıdan bakıldığında ise, 2 boyutlu dizileri, dizinin dizisi olarak düşünmek doğru olur. Diziler aynı türden elemanlardan oluşur. int türünde bir dizi olabileceği gibi, dizinin dizisi de olabilir. Matrisleri **dizilerin dizisi** (*array of arrays*) olarak düşünebiliriz.

Yukarıdaki örneklerde matrisin sütun sayısını 4 olarak belirledik. Bu şekilde oluşturulursa matrisin bütün satırları 4 elemanlı olur. Fakat bu zorunlu değildir. Matris oluştururken sütun sayısı belirlemezsek, her bir satırdaki dizilerin kapasitesi farklı olabilir. Örneğin aşağıdaki kodu inceleyelim:

```
int[][] matrix = new int[3][];
matrix[0] = new int[1];
matrix[1] = new int[2];
matrix[2] = new int[3];
```

Burada önce 3 satırdan oluşan bir matris belirttik, fakat sabit bir sütun sayısı vermedik. Sonra her bir satır için ayrı ayrı sütun sayısı belirledik. Bu kodu çalıştırdığımız zaman aşağıdaki gibi bir matris oluşur:



Çok boyutlu dizi oluştururken, yalnızca ilk boyutun (en soldaki) kapasitesini belirlemeniz yeterlidir. Diğer boyutların kapasitesini dinamik olarak belirleyebilirsiniz.

KONTROL MEKANİZMALARI

Programlama dilleri, kodun akış yönünü belirlemek için kontrol mekanizmaları kullanır. Java'da kodlar sırayla satır satır ilerler; fakat bu her zaman böyle olmak zorunda olduğu anlamına gelmez. Kontrol mekanizmalarını kullanarak kodda atlamalar, seçimler, hata ayıklamalar yapabilir veya döngüye girebiliriz.

Kontrol mekanizmalarını genel olarak 3 gruba ayırabiliriz:

- **Seçim ifadeleri:** Belli bir koşula bağlı olarak koda yön vermemizi sağlar.
- **Döngü ifadeleri:** Kodun belirli kısımlarını defalarca çalıştırmamızı sağlar.
- **Atlama ifadeleri:** Kodun belirli kısımlarını atlamamızı sağlar.

SEÇİM İFADELERİ

İki adet seçim ifadesi vardır: **if** ve **switch**. Bu ifadeler, sonucunu yalnızca çalışma zamanında bilebileceğimiz bazı koşullara göre program akışını değiştirmemizi sağlar.

if

if bloklarına daha önce kısaca giriş yapmıştık. Şimdi ayrıntıyla inceleyelim. Yapısı şu şekildedir:

```
if ( [koşul] )
{
      [ifade1]
}
else
{
      [ifade2]
}
```

Burada belirtilen *koşul*, sonucu boolean olan bir ifadedir. Bu ifade mutlaka parantez içinde yazılmalıdır. Eğer bu boolean koşulumuz **true** ise, if bloğu içindeki kodlar çalışır. Eğer if bloğumuzun hemen ardından bir **else** bloğu geliyorsa, bu bloğun içindeki kodlar yalnızca koşul **false** ise çalışır. else bloğu isteğe bağlıdır, her if ifadesinde else bloğu olmak zorunda değildir.

Eğer bir if ifadesinde else bloğu varsa, koşula bağlı olarak ya if bloğu ya da else bloğu çalışır. Gördüğünüz gibi, if kontrol mekanizmasını kullanarak kodumuzun belli kısımlarını atlamış oluyoruz.

if veya else bloğu içindeki kodlar tek satırdan oluşuyorsa küme parantezi açmaya gerek yoktur; fakat birden fazla kodlar için mutlaka blok açılmalıdır.

```
int year = 2020;

if (year > 0)
{
    System.out.println("Milattan sonra");
}
```

Yukarıdaki örnekte if bloğu çalışır ve konsola “Milattan sonra” yazdırılır.

```
int year = -5;

if (year > 0)
{
    System.out.println("Milattan sonra");
}
```

Bu örnekte ise if bloğunun çalışmadığını görürüz. Kodumuz if bloğunu atlar.

```
int year = -5;

if (year > 0)
{
    System.out.println("Milattan sonra");
}
else
{
    System.out.println("Milattan önce");
}
```

Yukarıdaki örnekte ise if bloğunun atlandığını ve else bloğunun çalıştırıldığını görürüz. Unutmayın: hem if hem else blokları mevcutsa bu bloklardan biri mutlaka çalıştırılır.

if-else-if merdiveni

Bazı durumlarda birden fazla if kontrolü yapmamız gerekir. Bu gibi durumlarda **else-if** blokları yardımımıza koşar.

Birden fazla kontrol yapılacaksa sırayla önce if, sonra else-if blokları ve son olarak (eğer varsa) else bloğu yazılır.

Aşağıdaki örneği inceleyelim:

```
int grade = getStudentGrade();

if (grade < 50)
{
    System.out.println("Notunuz 1");
}
else if (grade < 60)
{
    System.out.println("Notunuz 2");
}
else if (grade < 70)
{
    System.out.println("Notunuz 3");
}
else if (grade < 85)
{
    System.out.println("Notunuz 4");
}
else
{
    System.out.println("Notunuz 5");
}
```

Yukarıdaki kod örneğinde, öğrencinin notu 100 üzerinden elde ediliyor ve 5'lik not sistemine dönüştürülüyor. Bu örnekteki else-if bloklarının kullanımına dikkatinizi çekerim. Birden fazla koşulu if-else-if merdiveni ile kontrol ediyoruz. Burada if bloğunun en önde, else bloğunun en sonra olduğunu da fark etmişsinizdir. Ayrıca, else bloğunun zorunlu değil isteğe bağlı olduğunu da hatırlatalım.

if-else-if merdiveni oluştururken koşulların sıralamasına çok dikkat etmelisiniz. Örneğin yukarıdaki kodda bulunan, notun 60'tan ve 70'den küçük olduğunu test eden else-if bloklarının yerini değiştirelim ve aşağıdaki gibi yazalım:

```
int grade = getStudentGrade();

if (grade < 50)
{
    System.out.println("Notunuz 1");
}
else if (grade < 70)
{
    System.out.println("Notunuz 3");
}
else if (grade < 60)
{
    System.out.println("Notunuz 2");    // Bu if hiçbir zaman çalışmaz
}
else if (grade < 85)
{
    System.out.println("Notunuz 4");
}
else
{
    System.out.println("Notunuz 5");
}
```

Kodu incelerseniz, notun 60'tan küçük olduğunu test eden else-if bloğunun hiçbir zaman çalışmayacağını görürsünüz. Çünkü kendisinden daha kapsayıcı bir if koşulu daha önce yazılmıştır. 60'tan küçük olan her not 70'ten de küçük olacağı için, 60'tan küçük notlar 3. if bloğuna değil, 2. if bloğuna girer. Bunun sebebi, if bloklarının sırasıyla kontrol ediliyor olmasıdır. Bu nedenle, else-if merdiveni oluştururken if bloklarını, koşul kapsayıcılığı az olandan yüksek olana doğru sıralamalısınız. En kapsayıcı olan else bloğu ise en sona yazılır; çünkü else bloğu listelenenler haricinde diğer bütün koşulları kapsar.

Bu başlığı kapatırken, else ve else-if bloklarının zorunlu olmadığını, isteğe bağlı olduğunu tekrar hatırlatalım.

switch

switch bloklarının mantığını anlamak için bir örnek kod yazalım. 100'ün 5 ile bölümünden kalanı elde edelim ve bu kalanın ne olduğunu konsola yazdıralım:

```
int remainder = 100 % 5;

if (remainder == 0)
{
    System.out.println("Kalan: 0");
}
else if (remainder == 1)
{
    System.out.println("Kalan: 1");
}
else if (remainder == 2)
{
    System.out.println("Kalan: 2");
}
else if (remainder == 3)
{
    System.out.println("Kalan: 3");
}
else if (remainder == 4)
{
    System.out.println("Kalan: 4");
}
```

Gördüğünüz gibi, bir if-else-if merdiveni oluşturduk ve *remainder* değişkeninin değerine göre konsola metin yazdırdık. Bu tarz durumlarda, bir değişkenin değerine göre eşitlik koşulu arandığında, if yerine **switch** bloklarını kullanabiliriz. Yukarıdaki kodu aşağıdaki gibi yazabiliriz:

```
int remainder = 100 % 5;

switch (remainder)
{
    case 0:
        System.out.println("Kalan: 0");
        break;
    case 1:
        System.out.println("Kalan: 1");
        break;
    case 2:
        System.out.println("Kalan: 2");
        break;
    case 3:
        System.out.println("Kalan: 3");
        break;
    case 4:
        System.out.println("Kalan: 4");
        break;
}
```

Bu örnekten switch bloklarının yapısını anlamış oluyoruz. Önce **switch** deyimi yazılır, daha sonra değeri test edilecek değişken parantez içinde yazılır ve blok açılır. Test edilecek her bir durum **case** deyimiyle belirtilir. Test edilecek değer yazılır ve ardından iki nokta konulur. Her bir duruma özel kodlar yazıldıktan sonra **break** deyimiyle bu test senaryosunun tamamlandığı belirtilir. switch bloklarıyla sadece **byte, short, int, char** ve **String** türündeki değişkenler test edilebilir. Aynı test birden fazla yazılamaz.

if ifadelerinde, test edilenler haricindeki bütün durumları kapsamak için else bloğu kullanmıştık. switch bloğunda da bunun bir karşılığı vardır. Bütün case'ler test edildikten sonra, diğer testler için bir kod yazmak istiyorsak, bunu **default** bloğu içinde yazarız.

```
int a = 5;

switch (a)
{
    case 1:
        System.out.println("a'nın değeri: 1");
        break;
    case 2:
        System.out.println("a'nın değeri: 2");
        break;
    case 3:
        System.out.println("a'nın değeri: 3");
        break;
    default:
        System.out.println("a'nın değeri 3'ten büyüktür");
        break;
}
```

Yukarıdaki kodda görüldüğü üzere, a'nin değeri için 3 farklı durum test edilmiştir. Eğer a'nın değeri 1, 2 veya 3'ten farklıysa default kod bloğu çalışır.

default bloğu isteğe bağlıdır ve sadece bir kez yazılabilir. Ayrıca default bloğunun bütün case'lerden sonra, en sona yazılması gerektiğini de belirtelim.

Neden break deyimini kullanıyoruz?

Switch bloğunun kendine has bir yapısı vardır. Test edilen case'lerden biri testten geçiyorsa kod oradan itibaren çalışmaya devam eder; fakat break

deyimini görmezse, başka bir case'e girse bile çalışmasını sürdürür. Örneğin aşağıdaki kodu inceleyelim:

```
int a = 1;

switch (a)
{
    case 1:
        System.out.println("a'nın değeri: 1");
    case 2:
        System.out.println("a'nın değeri: 2");
    case 3:
        System.out.println("a'nın değeri: 3");
        break;
    default:
        System.out.println("a'nın değeri 3'ten büyüktür");
        break;
}
```

Bu kodu çalıştırırsanız konsol çıktısı şöyle olur:

```
a'nın değeri: 1
a'nın değeri: 2
a'nın değeri: 3
```

Gördüğünüz gibi, kod case 1'e girmiş; fakat break deyimini görene kadar çalışmaya devam etmiştir. Bu nedenle, switch blokları yazarken break deyimine çok dikkat etmelisiniz. switch blokları içinde unutulan break deyiminin programın çalışmasına çok büyük etkileri olabilir.

Bazı durumlarda, switch bloğunun esnek yapısından faydalanarak, birden fazla test senaryosu için aynı kod bloğunu kullanabilirsiniz. Aşağıdaki örneği inceleyelim:

```
int month;

switch (month)
{
    case 12:
    case 1:
    case 2:
        System.out.println("Mevsim: KIŞ");
        break;
    case 3:
    case 4:
    case 5:
        System.out.println("Mevsim: İLKBAHAR");
        break;
    case 6:
    case 7:
    case 8:
        System.out.println("Mevsim: YAZ");
        break;
    case 9:
    case 10:
    case 11:
        System.out.println("Mevsim: SONBAHAR");
        break;
}
```

DÖNGÜ İFADELERİ

3 farklı döngü ifadesi vardır: **for, while** ve **do-while**. Bu ifadeler kodumuzun belirli kısımlarının döngüye girmesini ve birden fazla kez çalışmasını sağlar. Döngüler bir koşula bağlanır ve bu koşul var olduğu sürece aynı kod bloğu çalışmaya devam eder. Koşul artık sağlanmıyorsa döngü sona erer.

while döngüsü

Java'nın en temel döngüsü **while** döngüsüdür. Hatta diyebiliriz ki, diğer bütün döngü türleri kodumuz derlendiğinde while bloğuna dönüştürülür. while döngüsünün çalışma mantığı basittir: while bloğu içindeki kod, bağlı olduğu koşul sağlandığı sürece çalışır. while bloğunun yapısı şu şekildedir:

```
while ( [koşul] )
{
      [döngüye girecek kodlar]
}
```

Burada belirtilen *koşul*, **boolean** bir ifadedir. Bu koşul **true** olduğu sürece döngü devam eder. Koşul **false** olursa döngü sonlanır.

Koşul ifadesi parantez içine yazılır. Eğer döngüye girecek kod tek satırdan oluşuyorsa blok açmaya gerek yoktur; fakat birden fazla satırdan oluşuyorsa mutlaka blok açılmalıdır.

```
System.out.println("10'dan geriye sayıyorum...");
int counter = 10;

while (counter >= 0)
{
    System.out.println(counter);
    counter--;
}
```

Yukarıdaki kod çalıştırıldığında çıktısı aşağıdaki gibi olur:

```
10'dan geriye sayıyorum...
10
9
8
7
6
5
4
3
2
1
0
```

Görüldüğü gibi koşul var olduğu sürece kodumuz çalışmış, sonra döngü sona ermiştir.

Döngülerin içeriği olmak zorunda değildir. Bazı durumlarda döngüye girecek kod olmasa bile döngüye girebiliriz. Aşağıdaki örneği inceleyelim:

```
int left = 100, right = 200;

while (++left < --right);

System.out.println("100 ile 200'ün ortası: " + left);
```

Bu algoritma, 100 ile 200'ün arasında tam ortada bulunan sayıyı bulmamızı sağlar. Kodu çalıştırdığımızda çıktısı şu şekilde olur:

```
100 ile 200'ün ortası: 150
```

Gördüğünüz üzere, içeriği olmasa bile bazı durumlarda döngüler fayda sağlayabilir.

do-while döngüsü

Yukarıda gördüğünüz gibi, while döngüsünde koşul döngünün başlangıcında test edilir. Eğer koşul sağlanmıyorsa (**false** ise) döngüye girilmez. Kısacası, yazdığınız programlarda while döngüsüne hiç girilmediği durumlar da yaşanabilir.

Fakat bazen, döngü koşulu sağlanmasa bile döngüye en az bir kere girilmesini isteyebiliriz. Eğer döngü koşulu döngünün başında değil, sonunda test edilirse, döngü en az bir kere işletilir. do-while döngüsü bunu sağlar. Bu döngünün while döngüsünden tek farkı, döngü koşulunun döngünün sonunda test edilmesidir.

Bu döngünün yapısı aşağıdaki şekildedir:

```
do
{
      [döngüye girecek kodlar]
} while ( [koşul] );
```

Gördüğünüz gibi, koşul döngünün sonunda test edilir. Bu durumda, koşul false olsa bile döngü en az bir kere çalıştırılmış olur. Bu şekilde döngü yazmak istediğiniz durumlarla karşılaşacaksınız. Aşağıdaki örneği inceleyelim:

```
int year = 2020;
do
{
      System.out.println(“Döngü işletiliyor..”);
      year++;
} while (year < 2020);
```

Bu kod çalıştırıldığında çıktısı aşağıdaki gibi olur:

```
Döngü işletiliyor..
```

Gördüğünüz gibi *year* değişkeni 2020'den küçük olmamasına rağmen döngü en az bir kere çalıştırılmıştır.

for döngüsü

Bu döngünün basit örneklerini daha önce görmüştük. Şimdi ayrıntılı inceleyelim.

2 farklı for döngüsü vardır. İlk for döngüsü, Java'nın ilk versiyonundan beri var olan geleneksel tarzdaki for döngüsüdür. Diğeri ise JDK 5 ile eklenen for-each döngüsüdür.

Geleneksel for döngüsünün yapısı şu şekildedir:

```
for ( [döngü başlangıcı] ; [döngü koşulu] ; [döngü adımı] )
{
      [döngüye girecek kodlar]
}
```

for döngüsü şu şekilde işler: İlk olarak döngüde sayaç işlevi görecek bir değişken oluşturulur. Bu değişkenin ilk değeri *[döngü başlangıcı]* ile belirtilen kısımda verilir. Bu değişken *[döngü adımı]* kısmında isteğe göre artırılır veya azaltılır. Döngünün hangi koşulda çalışacağı ise *[döngü koşulu]* kısmında boolean bir ifadeyle belirtilir. Aşağıdaki örneği inceleyelim:

```
for (int i = 1; i < 10; i++)
{
      System.out.println(i);
}
```

Bu örnekte int türünde *i* isminde bir döngü sayacı oluşturulur. Bu sayaç, döngünün her adımında *i++* ifadesiyle 1 artırılır. Bu döngü *i* sayacının değeri 10'dan küçük olduğu sürece çalışır. Bu döngü çalıştırıldığı zaman çıktısı aşağıdaki gibi olur:

```
1
2
3
4
5
6
7
8
9
```

for döngüsünün içindeki 3 kısmın çalıştırılma sırası şu şekildedir:

- İlk olarak *[döngü başlangıcı]* kısmı çalışır. Bu kısım sadece bir kez çalıştırılır, döngü boyunca bir daha çalıştırılmaz.
- Daha sonra *[döngü koşulu]* kısmı çalıştırılır. Eğer sonuç **true** ise döngüye girer. Bu kısım her döngünün başında tekrar kontrol edilir. Eğer sonuç **false** ise döngü sonlandırılır.
- Her döngünün sonunda *[döngü adımı]* kısmı çalıştırılır. Döngüde en son çalışan kısım burasıdır.

Şunu da belirtmek gerekir ki, *[döngü başlangıcı]* kısmında oluşturulan değişkenin kapsamı for döngüsüyle sınırlıdır. Bu değişkene for döngüsü dışında ulaşılamaz.

for döngüsü varyasyonları

Geleneksel for döngüsünün farklı varyasyonları olabilir. Döngünün içinde 3 farklı kısım olduğundan bahsetmiştik. Bu kısımların hiçbiri zorunlu değildir. Aşağıdaki örnekleri inceleyelim:

```
int i = 1;
for ( ; i < 10; i++)
{
      System.out.println(i);
}
```

Bu örnekte döngü sayacı, for döngüsünden önce oluşturulmuştur. Ayrıca *[döngü başlangıcı]* kısmının boş bırakıldığına dikkatinizi çekerim. Bu durumda döngü sayacı olarak döngünün dışında oluşturulan bir değişken kullanılır, ilk değeri de yine döngüden önce verilmiştir. Şunu da belirtelim ki, for döngüsünü bu örnekte olduğu gibi kullanırsanız, döngü sayacına for döngüsünden sonra da erişebilirsiniz.

```
int i = 1;
for ( ; i < 10; )
{
      System.out.println(i);
      i++;
}
```

Bu örnekte ise hem *[döngü başlangıcı]* kısmının hem de *[döngü adımı]* kısmının boş bırakıldığını görüyoruz. Döngü sayacını parantez içinde artırmak yerine, döngü kodlarının içinde artırıyoruz.

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

Bu örnekte ise bütün kısımların boş bırakıldığını görüyoruz. Bu tarz for döngüleri *sonsuz döngü* oluşturur. Sonsuz döngü, hiçbir koşulda sona ermeyen döngü demektir. Bu döngüde hiçbir koşul test edilmediği için döngü her zaman çalışır. Yani, eğer bu kodu çalıştırırsanız, bilgisayarınız sonsuza kadar bu döngüyü işletecektir.

break deyimi

Buraya kadar gördüğümüz bütün döngüler belirlediğimiz bir koşula göre kontrol ediliyor ve bu koşul sağlandığı sürece çalışıyordu. Bazı durumlarda, döngü koşulu sağlansa bile başka bir nedenden ötürü döngüyü sonlandırmak isteyebiliriz. Bu tarz durumlarda **break** deyimini kullanırız. Bu deyim, içinde kullanıldığı döngüyü anında sonlandırır. Aşağıdaki örneği inceleyelim:

```
boolean continueLoop = true;

for (int i = 2; i <= 100; i += 2)
{
	if (!continueLoop)
	{
		break;
	}

	System.out.println(i);

	if (i == 50)
	{
		continueLoop = false;
	}
}
```

Bu örnekte, 2'den 100'e kadar olan çift sayılar konsola yazdırılıyor. Fakat döngü ayrıca boolean bir değişkenle kontrol ediliyor. Döngü sayacı 50'ye ulaştığında bu değişken false olarak ayarlanıyor. Döngünün her adımında ise bu değişken kontrol ediliyor; eğer false ise döngü break deyimiyle sonlandırılıyor. Bu kod çalıştırıldığında, 2'den 50'ye kadar olan çift sayıların konsola yazdırıldığını görürüz. Çünkü 50'ye ulaştıktan sonra break deyimiyle döngü sonlandırılmıştır. Gördüğünüz gibi, aslında döngü koşulu sağlanmasına rağmen manuel olarak döngü sonlandırılmıştır.

continue deyimi

Bazı durumlarda döngünün sadece o adımının sonlanmasını, diğer adımların ise çalışmaya devam etmesini isteriz. Bu gibi durumlarda **continue** deyimini kullanırız. Aşağıdaki örneği inceleyelim:

```
for (int i = 0; i <= 10; i += 2)
{
	if (i == 4)
	{
		continue;
	}

	System.out.println(i);
}
```

Bu örnekte 0'dan 10'a kadar olan çift sayılar konsola yazdırılıyor. Fakat döngü sayacı 4 olduğunda continue deyimi çalıştırılıyor. continue deyimi kullanıldığında döngünün geri kalanı çalıştırılmaz; fakat diğer adımlar çalışmaya devam eder. Bu kod çalıştırıldığında çıktısı aşağıdaki gibi olur:

```
0
2
6
8
10
```

Gördüğünüz gibi, continue deyimi çalıştırıldığı için 4 konsola yazılmamış; fakat diğer değerler yazılmıştır.

for-each döngüsü

Çoğu zaman for döngüsünü, bir dizinin veya koleksiyonun içindeki elemanları gezmek için kullanırız. Aşağıdaki örneği inceleyelim:

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

for (int index = 0; index < numbers.length; index++)
{
      int number = numbers[index];
      System.out.println(number);
}
```

Bu örnekte 5 elemanlı bir dizi oluşturuluyor ve for döngüsü kullanarak bu dizinin elemanları konsola yazdırılıyor. Dizinin hiçbir elemanının atlanmadığına dikkat edelim.

Bu gibi durumlar (bir dizinin veya koleksiyonunun bütün elemanlarını gezmek) için **for-each** döngüsü geliştirilmiştir. Bu döngünün yapısı aşağıdaki gibidir:

```
for ( [elemanların türü] [değişken ismi] : [dizi veya koleksiyon] )
{
      [döngüye girecek kodlar]
}
```

İlk olarak elemanların türü belirtilir. Bu tür, dizinin veya koleksiyonun içindeki elemanların türüdür. Daha sonra bir değişken ismi verilir. Bu isim,

döngünün her adımında sıradaki elemanı belirtir. Sonra iki nokta konulur ve ardından dizinin veya koleksiyonun ismi yazılır.

Örneğin, yukarıdaki örneği for-each döngüsüyle aşağıdaki gibi yazabiliriz:

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

for (int number : numbers)
{
      System.out.println(number);
}
```

Sonucu aynı olmasına rağmen, bu kodun geleneksel for döngüsüne göre daha kısa olduğunu fark etmişsinizdir.

Bu döngüyle ilgili belirtmemiz gereken önemli bir nokta vardır. for-each içinde belirtilen döngü değişkeni **dolaylı olarak sabittir** (*effectively final*). Yani, döngü içinde bu değişkene atama yapamazsınız. Örneğin, aşağıdaki kod hata fırlatır:

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

for (int number : numbers)
{
      number = 10;   // Bu satır hataya neden olur.
      // for-each içinde tanımlanan değişkenlere değer atayamazsınız
}
```