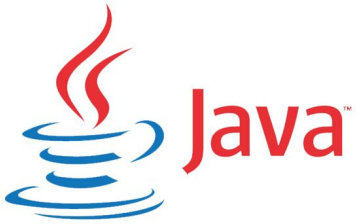


Java Referans Kitapçığı

Irmak Özonay

RMKOD Teknoloji

Bu referans kitapçığı burda bulunan Java'ya başlangıç videoları ile alakalı olarak hazırlanmıştır.

Java kodu nasıl çalışıyor?

Öncelikle kısaca terimlere bakarsak;

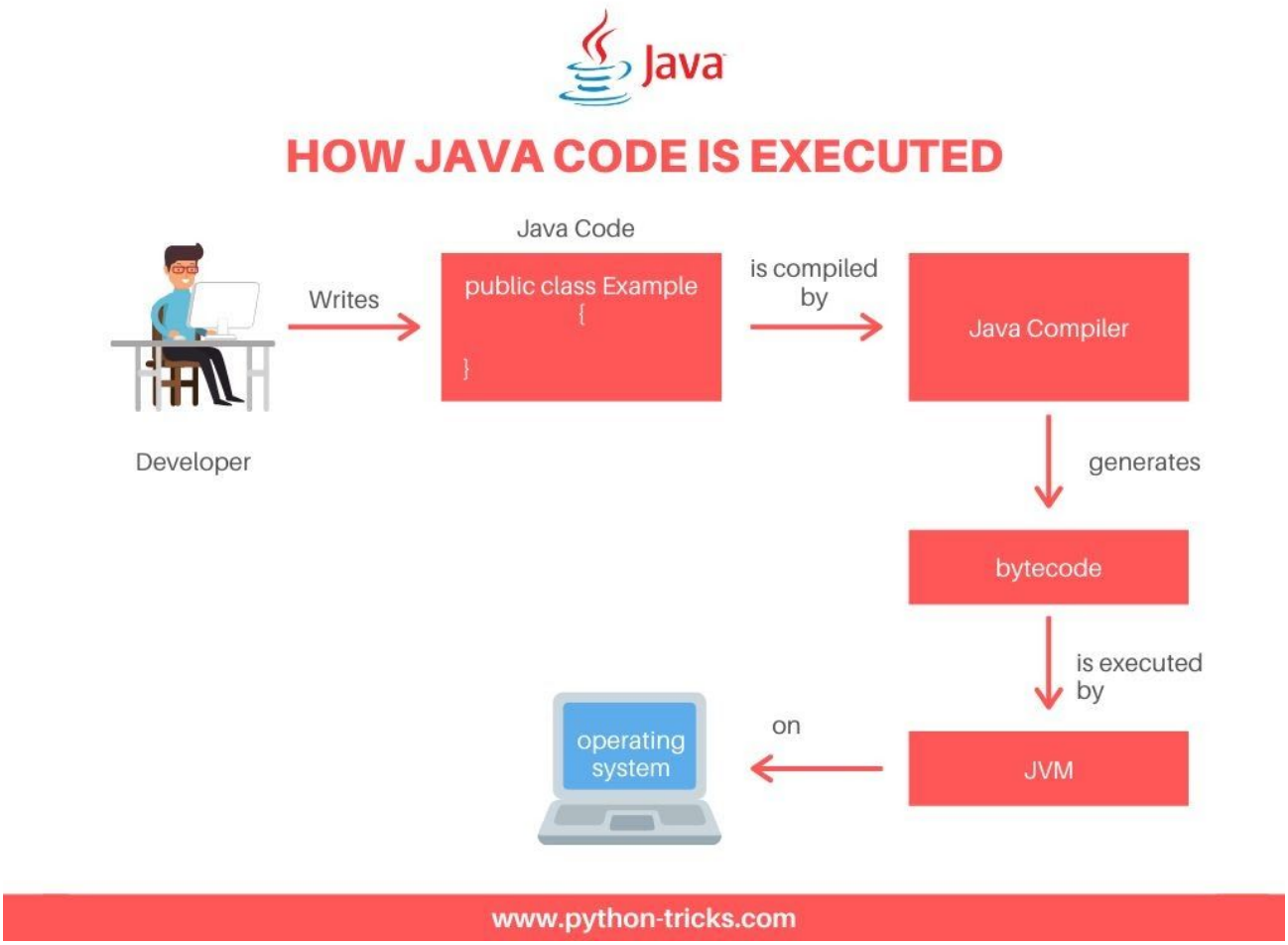
Java SE: Java Standard Edition

JDK: Java Development Kit - Java Geliştme Kiti

JRE: Java Runtime Environment - Java Çalıştırma Ortamı

JVM: Java Virtual Machine — Java Sanal Makinası

JDK'nın içinde JRE, geliştirme, hata ayıklama araçları, compiler (derleyici), baz Java kodları gibi geliştirme yaparken kullanacağımız araçlar bulunuyor. JRE'nin içinde de JVM bulunuyor.



https://python-tricks.com/how-java-works

1. Geliştirici tarafından Java kodu, yani .java uzantılı dosyalar oluşturulur.
2. Bu kod Java Compiler (derleyici) tarafından compile edilerek (derlenerek) bytecode'a yani .class dosyalarına dönüştürülür.
3. .class dosyası JVM tarafından çalışacağı işletim sistemine uygun hale getirilir. .class dosyaları, JRE bulunan tüm platformlarda çalışabilir.

Java'yı indirme

İndirme sayfası:

Open JDK: https://jdk.java.net

Oracle JDK: https://www.oracle.com/tr/java/technologies/javase-downloads.html

Bu iki JDK teknik olarak çok yüksek oranda aynı.

Farklar

Oracle JDK

Commercial yani üzerinden para kazanılan uygulamalar için ücretli

Uzun sürelerde güncelleme geliyor, bu sayede daha stabil

Performans olarak daha güçlü

Oracle tarafından geliştiriliyor

Open JDK

Ücretsiz

Sık güncelleme geliyor

Performansı Oracle JDK'ya kıyasla daha düşük, bu büyük kapsamlı uygulamalarda fark edilebilir

Geliştirme OpenJDK ekibi, Oracle ve komünite tarafından yapılıyor

Java Editörleri

Eclipse: https://www.eclipse.org/downloads

Netbeans: https://netbeans.org

VSCode: https://code.visualstudio.com/docs/languages/java

IntelliJ: https://www.jetbrains.com/idea

Mac Java kurulumu için https://youtu.be/3RSfQP6vyAs?t=142

Windows Java kurulumu için https://youtu.be/3RSfQP6vyAs?t=370

Bu seride verilen örnekler Open JDK ve Eclipse kullanılarak yapıldı.

Bir Java programının yapısı

Basit bir Java programının yapısı aşağıdaki gibidir:

```
package com.irmak;
public class Game {
   public static void main(String[] args) {
      greet("Irmak");
   }
   public static void greet(String name) {
      System.out.println(getMessage() + " " + name);
   }
}
```

Burda “com.irmak” paket ismi (package), “Game” java sınıfının ismi (class), “main” de metodun ismidir. (method).

Methodu kısaca, bir işlem gerçekleştiren kod bloğu olarak tanımlayabiliriz.

Classlar benzer mantıktaki, işlevdeki methodları içinde barındırırlar.

Packagelar da benzer mantıkta ve işlevdeki classları içinde barındırırlar.

- Class ve methodlar **küme parantezi** ile başlar ve yine küme parantezi ile kapanır, kod bu araya yazılır.
- **public** kelimesi, method ve classın diğer classlar tarafından erişime açık olduğunu belirtir. public bir “**access modifier**” yani erişim belirleyicidir. Object oriented programlamaya geçince bunları daha detaylı inceleyeceğiz.
- **Class**lar, access modifierdan sonra class kelimesi ve arından classın ismi yazılarak tanımlanır.
- **main** methodu, bu Java programını çalıştırılmasıyla birlikte ilk olarak çağırılan method ve bu programın başlangıç noktasıdır.
- Yukarıdaki örnekte **main** ve **greet** method isimleridir.
- **void** methodun “return type” yani dönüş tipidir. Return type, bir method, başka bir method tarafından çağırıldığında hangi data tipinde bir değerin döndürüleceğini belirtir. **void**, boşluk, geçersiz anlamındadır ve bu methodun bi değer döndürmeyeceği anlamına gelir. Farklı data tiplerini bir sonraki bölümde inceleyeceğiz.

- Method isminden sonra parantez içinde yer alan değerler bu methodun dışarıdan alacağı **parametre**lerdir. Parantez içine “DataTipi parametreİsmi”, örneğin “String name” şeklinde yazılır. Eğer method bir parametre almıyorsa, parantezin içi boş bırakılır.

System.out.println(“Merhaba”), çıktı ekranına “Merhaba” yazar

System Java’nın kendi classlarından bir tanesi, **out** System classının fieldlarından (alanlarından) biri. **println** ise console’a yazı yazdırmak için tanımlanmış bir method. Parantez içinde methoda yollanan argüman (“Merhaba”).

Değişkenler

Değişkenler yani “variable”lar, bilgisayarın hafızasında geçici olarak değerler tutmamıza yarar. Bir değişken şu şekilde tanımlanır:

```
int secretNumber = 5;
```

Bu tanımlamada **int** tam sayıları tutmak için kullandığımız data tipi, **secretNumber** değişkenin ismi, **5** ise değişkenin atandığı değerdir.

İsimlendirme

Java’da isimlendirme yaparken aşağıdaki noktalara dikkat etmekte yarar var,

1. İsim biraz uzasa bile anlamlı isim vermek.
2. İlk kelimenin ilk harfini küçük, sonrasındaki her kelimenin ilk harfini büyük yapmak. Buna camelCase deniyor.
3. Method isimlerinin camelCase olması.
4. Class isimlerinde her kelimenin ilk harfinin büyük olması. (Ör: NumberGame)

Data Tipleri

Java’da data tipleri primitive ve reference olarak iki kategoriye ayrılır:

Primitive Type

Sayılar gibi basit değerleri tutmak için kullanılır.

Java’da 8 tane primitive bulunur, bunlar aşağıdaki gibidir:

Data Tipi	**Boyut**	**Açıklama**
byte	1 byte	-128 ile 127 arasındaki tam sayıları tutar
short	2 byte	-32.768 ile 32.767 arasındaki tam sayıları tutar
int	4 byte	-2.147.483.648 to 2.147.483.647 arasındaki tam sayıları tutar
long	8 byte	-9.223.372.036.854.775.808 to 9.223.372.036.854.775.807 arasındaki tam sayıları tutar
float	4 byte	Küsüratlı sayıları tutar. Noktadan sonra 6 veya 7 basamak alır

double	8 byte	Küsüratlı sayıları tutar. Noktadan sonra 15 basamak alır
boolean	1 bit	True (doğru) ve false (yanlış) değerlerini tutar
char	2 byte	Tek bir karakter, harf veya ASCII değeri tutar

Java Data Tipleri

Java'da neden byte ve short az kullanılır?

Byte ve short daha az hafıza kullanmasına rağmen Java'da sıklıkla int'in kullanıldığını göreceksiniz. Bunun sebebi integer ile yapılan işlemlerin Java'nın yapısı gereği daha hızlı ve performanslı yapılıyor olması. Örneğin aritmetik işlemler int üzerinden yapılır, short ve byte kullanılsa bile bunun sonradan int'e çevrilmesi gerekir. Java'nın hafıza kullanım yapısından ötürü belli durumlar haricinde short ya da byte kullandığımızda hafızadan da tasarruf etmiş olmuyoruz. short ve byte'ın gelen kullanım alanları, büyük arrayler, network işlemleri gibi örneklendirilebilir.

Reference Type

8 primitive type haricinde kalan tüm tipler reference (referans) tiptir. Bunlara referans tip denmesinin sebebi değişkenin içinde tutulan değerin, aslı değer değil de hafızadaki adres değeri olmasıdır.

Tarih gibi komplex objeleri tutmak için kullanılır.

```
Date startTime = new Date();
```

- Bu satırda **Date** data tipi, **startTime** ise değişken ismidir.
- **new** operatörü Date classından bir örnek oluşturarak, obje için hafızadan bir bölge ayırır (**allocate** eder) ve bu hafıza referansının değişkene atanmasını sağlar.
- new kelimesinden sonra örneğini yaratmak istediğimiz class (**Date()**) yazılır.

Object

Yukarıdaki satır ile Date classının startTime isimli bir **objesini** yaratmış olduk, yani bu classtan bir **örnek** yarattık.

- **Object** yani **nesne, bir classın örneği** olarak tanımlanır.
- Daha teknik ve global olarak söylemek gerekirse, **startTime objesi Date classının bir instance'ıdır.**
- Bir değişkene ilk değerinin atanması işlemine **initializing** denir.

Primitive typelardan farklı olarak reference typeların kendilerine ait methodları ve fieldları (alanları) olabilir.

Date objesinin bir methodunu çağırmak istersek bunu **dot operator (nokta operatörü)** ile aşağıdaki gibi yapabiliriz.

```
    startTime.getTime();
```

Reference ve primitive tiplerin hafızada nasıl tutulduğunu Stack Memory ve Heap Space bölümünden inceleyebilirsiniz.

String

String yazı tutmak için kullandığımız data tipidir. Reference tip olmasına rağmen new kelimesi ile initialize edilmesine gerek yoktur, aşağıdaki şekilde tanımlanabilir. String'in hafızada nasıl tutulduğunu String Pool bölümünden inceleyebilirsiniz.

```
    String welcomeMessage = "Sayi tahmini oyununa hosgeldin!";
```

Stringin de kendine ait methodları vardır, bazıları şu şekildedir:

```
  .toUpperCase() //stringin tüm harflerini büyük harfe dönüştürerek yeni bir string döndürür
  .toLowerCase() //stringin tüm harflerini küçük harfe dönüştürerek yeni bir string
döndürür
  .endsWith("!") //stringin ! ile bitip bitmediğini kontrol eder ve boolean (true veya
false) bir değer döndürür
  .startsWith("!") //stringin ! ile başlayıp başlamadığını kontrol eder ve boolean (true
veya false) bir değer döndürür
  .length() //stringing kaç karakter uzunluğunda olduğunu int olarak döndürür
  .indexOf("!") //parantez içinde belirtilen stringin, değişken içinde ilk görülme indeksini
döndürür
  .replaceAll("!", ".") //string değişkeni içinde yer alan tüm ! işaretlerini . ile
değiştirir ve yeni bir string döndürür
  .substring(0, 2) //stringin 0 ile 2 arasında bulunan string alt kümesini döndürür. Bu
örnek için döndürülen değer değişkenin ilk iki harfi olacaktır
```

Comment

Comment, yani yorumlar, compiler tarafından yok sayılır. Commentleri kodunuz ile ilgili notlar yazmak için kullanabilirsiniz.

Tek satırlık commentlerinizi **//** yazıp ardında eklebilirsiniz.

Birden çok satırlık bir yorum yazmak istiyorsanız **/**/** yazıp, yıldızların arasına yorum ekleyebilirsiniz.

```
//comment
/**
* comment
*/
```

Method chaining

Method chaining, yani method zincirleme, bir methoddan döndürülen değerin, bir değişkene atanmadan, sonuç üzerinden direk nokta operatörü ile başka bir method çağırılmasına denir.

```
value.substring(0, 1).toUpperCase()
```

Bu örnekte, substring ile value değerinin ilk karakteri alınıyor ve bu değer bir değişkene atanmadan, direk toUpperCase method çağrılıyor.

Sabit değerler

Constantlar yani sabit değerlerin değerleri tanımlandıktan sonra değiştirilemez. Bunları final kelimesi ile tanımlayabiliriz.

```
final String GAME_NAME = "Sayi Tahmini";
```

Arrayler

Arrayleri bir grup integerı, stringi, objeyi liste halinde tutmak için kullanıyoruz. Array içine her data tipi koyulabilir. Bir array aşağıdaki gibi yaratılıyor:

```
int[] guesses = new int[3];
```

- **int** data tipi
- **Köşeli parantez** bu tanımın bir array olduğunu belirtiyor
- **guesses** değişken ismi
- Arrayler reference type olduğu için **new** ile yaratılıyor
- new sonrasındaki köşeli parantezin içine de arrayin **boyutu**, yani arrayde kaç eleman tutulacağı yazılıyor

Arraye değer eklemek ise şu şekilde,

```
guesses[0] = 5;
guesses[1] = 8;
guesses[2] = 10;
```

Java'da arrayler sıfırdan başlar.

Arraye değer atarken değişken isminin yanına köşeli parantez içinde değer atanacak indeks yazılır.

Java'da arrayler neden sıfırdan başlar?

Javascript, Python, C gibi diğer dillerde de arrayler benzer şekilde sıfırdan başlıyor. Her ne kadar şu anda, sıfırdan indeks olarak bahsetsek de, aslında ilk planlamada sıfır, arrayin hafızadaki başlangıç adresine olan uzaklık, yani ofset olarak tanımlanmış. Sıfıncı eleman yani ilk eleman ve öncesinde hiç bir eleman olmayan elemana array[0] olarak erişiliyor. Arrayler, her dilde sıfırdan başlamıyor ama şu an en popüler olan dillerde bu şekilde. Bu karar, dillerin tamamen kendi yapılarından ve performans optimize etme modellerinden kaynaklanıyor.

Array kısaca şu şekilde de initialize edilebilir:

```
int[] array = {5, 8, 10, 12, 15};
```

Arrayi yazdırma:

```
      System.out.println(Arrays.toString(guesses));
```

Aritmetik İşlemler

Java'da toplama(+), çıkartma(-), çarpma(*), bölme(/) ve kalan(%) işlemleri var.

```
      int value = 5 + 2; //7
      int value = 5 - 2; //3
      int value = 5 * 2; //10
      int value = 5 / 2; //2
      int value = 5 % 2; //3 (5'in 2'ye bölünmesinden kalan)
      int value = 5 * (7 + 2); //45 (Normal matematik işlemlerindeki gibi önce parantezin
içindeki işlem yapılıyor)
```

Bölme işlemi sonucunda almak istediğimiz değer küsuratlı ise, bölme işleminin iki küsuratlı sayı arasında gerçekleşmesi gerekir. Yukarıdaki örnekteki 5 ve 2 değerleri Java tarafından int olarak alınıyor ve tam sayılar üzerinden işlem yaparak sonucu 2.5 yerine 2 olarak veriyor.

Küsüratlı sonuç almak için aşağıdaki işlemlerden birini yapabiliriz:

```
      float value = 5f / 2f; //2.5 (f harfi sayının float olduğunu belirtmek için
kullanılır)
      double value = 5.0 / 2.0; //2.5 (sayının sonuna .0 eklenince sayı double alınır)
      double value = (double)5 / (double)2; //2.5 (int olan sayılar double a çevriliyor, bu
işleme casting deniyor)
```

Arttırma ve Azaltma İşlemleri

Bir sayıyı belli bir değer kadar artırmak veya azaltmak istediğimiz durumlarda aşağıdaki kısa yolları kullanabiliriz.

```
      int i = 0;
      i++; //i = i + 1
      i--; //i = i - 1
      i--; //i = i - 1
      i += 2; //i = i + 2
```

Control Flow

Control flow, yani kontrol akışı bir program çalışırken fonksiyonların çağırılma sırası, hangi kodun çalıştırılıp, çalışmayacağına karar verilmesi yani program akışına karar verilmesine denir.

Karşılaştırma Operatörleri

Karşılaştırma operatörlerini, değerleri birbirleri ile kıyaslamak için kullanıyoruz. Aşağıdaki işlemlerde, karşılaştırma sonucu doğru olduğu koşulda ifade true, yoksa false olacaktır.

```
int x = 2;
int y = 3;
x == y //eşit //false
x != y //eşit değil //true
x < y //küçüktür //true
x > y //büyüktür //false
x <= y //küçük eşit //true
x >= y //büyük eşit //false
```

If Statement (Eğer İfadesi)

Bir kod bloğunu belli bir koşul sağlandığında çalıştırmak için if kullanılır.

```
if (number > 50) {
     System.out.println("50'den buyuk");
 } else if (number < 20) {
     System.out.println("20'den kucuk");
 } else {
     System.out.println("20 ile 50 arasinda");
 }
```

If parantezinin içinde, boolean sonuç verecek bir ifade veya boolean bir değişken kullanılır. Parantez içine yazılan ifade true ise if küme parantezindeki kod bloğu çalışır.

If koşulu sağlanmadığında, varsa else if kontrol edilir, eğer else if koşulu da sağlanmıyorsa, else içindeki kod bloğu çalıştırılır.

Ternary Operator (?:)

Aşağıdaki gibi bir if-else kodunu ternary operatör ile tek satırda yazabiliriz.

```
String message = "";
if (age > 18) {
    message = "Araba kullanabilir";
} else {
    message = "Araba kullanamaz";
}
```

```
String message = age > 18 ? "Araba kullanabilir" : "Araba kullanamaz";
```

Ternary operator'da ? öncesindeki ifade/değişken true ise **?** sonrasındaki kısım; false ise **:** sonrasındaki kısım kullanılır.

Mantıksal Operatörler

Birden çok boolean değişkeni / ifadeyi bir arada kullanmak için mantıksal operatörler kullanılır.

AND (VE): val1 **&&** val2

val1 ve val2 değerlerinin ikisi de true ise sonuç true olur. AND operatöründe ikiden fazla ifade olabilir; sonucun true olması için sağlanan tüm koşulların true olması gerekir. İfadeler soldan sağa kontrol edilir, koşul false ise geriye kalan ifadeler kontrol edilmez, sonuç false olur.

```
if (money > 10 && age < 7) {
    System.out.println("Filme girebilir");
}
```

OR (VEYA): val1 **||** val2

val1 veya val2 değeri, ya da ikiside true ise sonuç true olur. OR operatöründe ikiden fazla ifade olabilir; sonucun true olması için bir koşulun true olması yeterlidir. İfadeler soldan sağa kontrol edilir, koşul true ise geriye kalan ifadeler kontrol edilmez, sonuç true olur.

```
if (money > 10 || hasCoupon) {
    System.out.println("Filme girebilir");
}
```

NOT (DEĞİL): !val1

boolean val1 değerinin zıttını alır. val1 true ise sonuç false, val1 false ise sonuç true olur.

```
if (!hasCoupon) {
    System.out.println("Filme giremez");
}
```

Switch Statement

Bir değişkenin farklı değerlerine göre, farklı kodlar çalıştırmamız gereken durumlarda switch kullanabiliriz.

```
switch (age) {
     case 1:
         System.out.println("Film A");
         break;
     case 2:
         System.out.println("Film B");
         break;
     case 3:
         System.out.println("Film C");
         break;
     default:
         System.out.println("Tüm Filmler");
         break;
 }
```

Switch case'de belli bir koşul sağlandıktan sonra, switch statementından break ile çıkmamız gerekiyor. break'e döngülerde bakacağız.

Döngüler (Loops)

Döngüler, belirtilen koşul sağlanana kadar bir kod bloğunu tekrarlamak için kullanılır.

For Loop

For döngüsü bir kodu belirli bir sayıda tekrarlamak için kullanılır, kodun tekrarlanacağı sayı belliyse ve kod içinde bir sayaç ihtiyacı duyuluyorsa tercih edilebilir.

```
for (int i = 0; i < 5; i++) {
    System.out.println("Merhaba " + i);
}
```

For loopta parantez içine 3 tane statement (ifade), girilir.

int i = 0: İlk statementta bir değişken tanımlıyoruz, bu sayaç görevi görecek.
i < 5: İkinci olarak bu loopun hangi koşulda devam edeceğinin kriterini giriyoruz.
i++: Son olarak da tanımlanan sayaç değişkeninin her döngüde nasıl değiştirileceğini giriyoruz.

Bu loop ekrana aşağıdaki sonucu yazacak

```
Merhaba 0
Merhaba 1
Merhaba 2
Merhaba 3
Merhaba 4
```

Loopta i değeri 0 ile başlıyor, her döngüde i değeri bir arttırılıyor ve i değeri 5'e ulaşınca i < 5 koşulu artık sağlanmadığı için loop son buluyor.

While Loop

While döngüsü bir kodu sayıdan bağımsız olarak, belli bir koşul sağlanana kadar tekrar etmek için kullanılır.

```
Scanner scanner = new Scanner(System.in);
int secretNumber = 1 + (int) (Math.random() * 10);
int guess = 0;
while (guess != secretNumber) {
    guess = scanner.nextInt();
}
```

Örnek olarak, yukarıdaki while döngüsünde, bilgisayarın yarattığı gizliNumara (secretNumber) ile, kullanıcıdan alınan sayı tahmini (guess) eşit olana kadar devam edecek.

Do...While Loop

Do...While döngüsü while döngüsüne çok benzer, tek bir fark olarak while döngüsünden kod her çalıştırılma öncesinde koşul kontrolü yaparken, do while döngüsünde ilk çalışma öncesinde koşula bakılmadan kod çalıştırılır.

```
Scanner scanner = new Scanner(System.in);
int secretNumber = 1 + (int) (Math.random() * 10);
int guess = 0;
do {
    guess = scanner.nextInt();
} while (guess != secretNumber);
```

For-each Loop

For-each döngüleri, bir array veya collection içindeki değerler içinde dönmek için kullanışlıdır. Aşağıdaki formatta yazılır:

```
String[] shoppingList = { "elma", "armut", "muz" };
for (String item : shoppingList) {
    System.out.println(item);
}
```

For-each loopta parantez içinde öncelikle arrayin data tipi ve bir değişken ismi yazılır (**String item**) sonrasında **:** konarak içinde dönmek istediğimiz **arrayin** ismi yazılır.

Bu loop ekrana aşağıdaki sonucu yazacak

```
elma
armut
muz
```

Loop, arrayin içinde gezerek array içindeki her elemanı ekrana yazdırıyor.

break

break ifadesi, bir looptan çıkmayı sağlar.

Yukarıdaki for-each loop üzerinden bir örnek yapalım

```
String[] shoppingList = { "elma", "armut", "muz" };
for (String item : shoppingList) {
    System.out.println(item);
    break;
}
```

Bu loop ekrana aşağıdaki sonucu yazacak

```
elma
```

Arrayin ilk elemanı ekrana yazıldıktan sonra, kod break ifadesine geliyor ve arrayin diğer elemanları yazılmadan döngüden çıkılıyor.

continue

continue ifadesi, istenilen loop döngüsünün es geçilmesini sağlar.

For loop üzerinden bir örnek yapalım

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println("Merhaba " + i);
}
```

Bu loop ekrana aşağıdaki sonucu yazacak

```
Merhaba 0
Merhaba 1
Merhaba 3
Merhaba 4
```

i'nin değeri 3'e eşit olduğunda loop continue sonrasındaki kodları es geçerek bir sonraki döngüyle devam ediyor.

ArrayList

ArrayList'ler arrayler gibi bir grup değeri tutmayı sağlıyor. ArrayListlerde arrayler gibi initialize ederken bir kapasite vermek gerekmiyor. Bir ArrayList aşağıdaki şekilde tanımlanabilir:

```
ArrayList<String> shoppingList = new ArrayList<String>();
```

Arraylistin kendine ait methodları bulunmaktadır, sıkça kullandıklarımızdan bir kaçı şu şekilde:

```
shoppingList.add("elma"); //arrayin sonuna belirtilen elemani ekler
shoppingList.add(0, "erik"); //elemani belirtilen indekse ekler, varsa diger
elemanlari saga kaydirir
shoppingList.remove("elma"); //belirilen elemanin ilk gorulmesini listeden cikarir
shoppingList.indexOf("erik"); //belirilen elemanin ilk gorulme indeksini dondurur
shoppingList.contains("armut"); //belirtilen eleman listede varsa true dondurur
shoppingList.size(); //listedeki eleman sayisini dondurur
shoppingList.get(0); //belirtilen indeksteki elemani dondurur
shoppingList.clear(); //listedeki tum elemanlari siler
```

Örnekte dikkat ettiyseniz iki tane add methodu var, buna **method overloading** deniyor.
Arraylerde int, float, double gibi primitive typeları tutabilirken, ArrayListlerde primitive typelar tutulamıyor. Bunun için **wrapper class**ları kullanabiliyoruz.

Method Overloading

Method overloading farklı parametreler ile aynı isimdeki methodun farklı versiyonlarını oluşturmak olarak tanımlanabilir.

Wrapper Classes

Wrapper, sarıcı, saran anlamındadır. Wrapper classlar başka classların komponentlerini, fonksiyonelitelerini içinde barındırır. Örneğin int in wrapper classı Integer'dır, double'ınki Double'dır. Tanımlaması eşitleme kısa yolu ile yapılabilir.

```
Integer x = 5;
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

Autoboxing

int bir değer Integer bir arraye eklenebilir, bu esnada arka tarafta int değer otomatik olarak Integer a dönüştürülür, buna autoboxing denir.

Unboxing

Autoboxing işleminin zıttına yani wrapper classtan primitive typea geçişe unboxing denir.

Stack Memory ve Heap Space

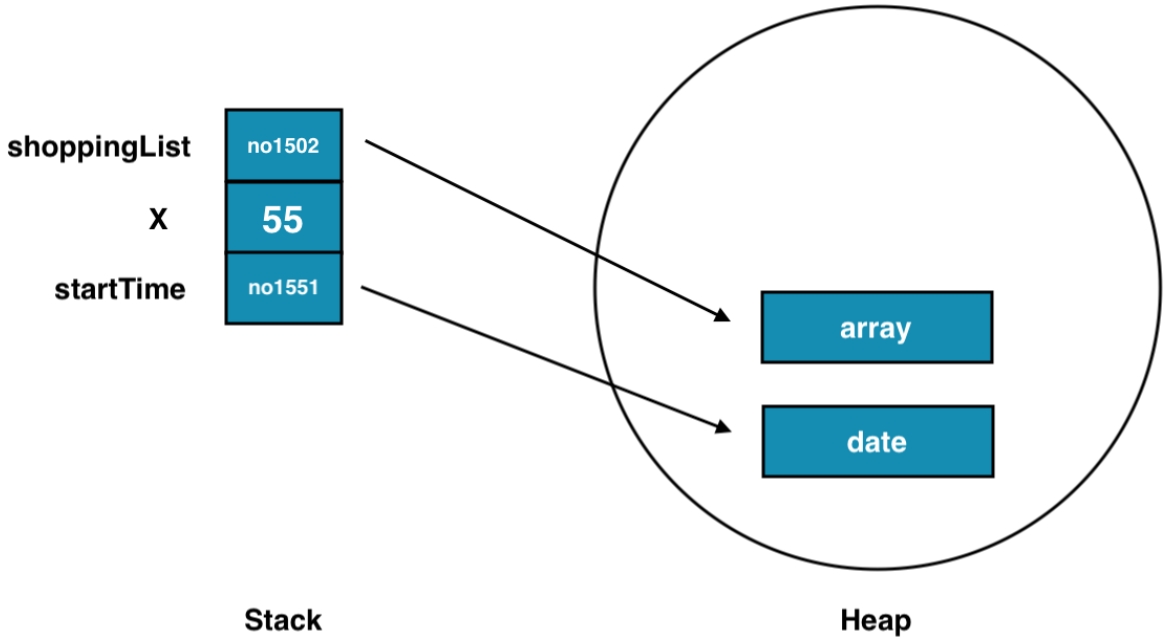
Javada değişkenler **stack memory** ve **heap space** denen iki hafızada tutuluyor.

- **Primitive** tipler **stack** memoryde tutuluyor
- **Reference** tipler **heap** spacede tutuluyor.
- Reference tiplerin sahip olduğu **adres, stack** memoryde saklanıyor.

Aşağıdaki örnekler bu yapıyı kafanızda daha iyi canlandıracaktır.

1.

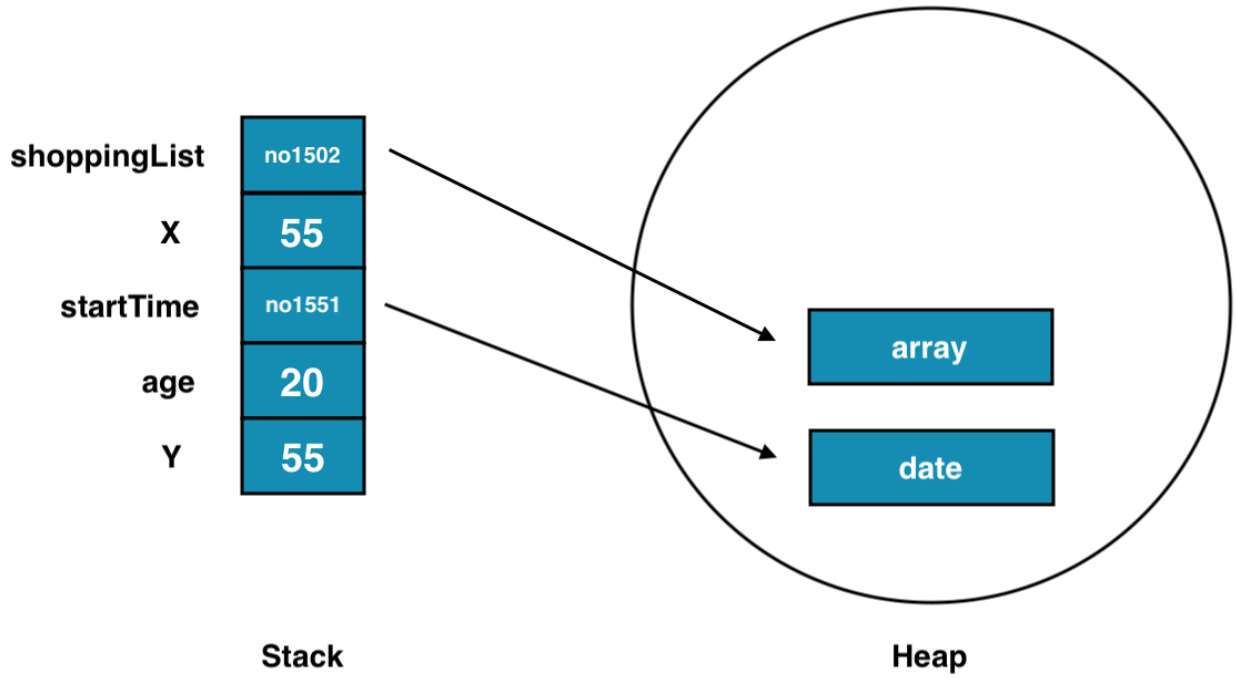
```
ArrayList<String> shoppingList = new ArrayList<String>(); //heap space
int x = 55; //stack memory
Date startTime = new Date(); //heap space
```



- x değeri (55) primitive type olduğu için direk stackte tutuluyor.
- shoppingList ve startTime objeleri heap spacede tutuluyor.
- shoppingList ve startTime objelerinin heapteki adres değerleri, stack memoryde tutuluyor.

2.

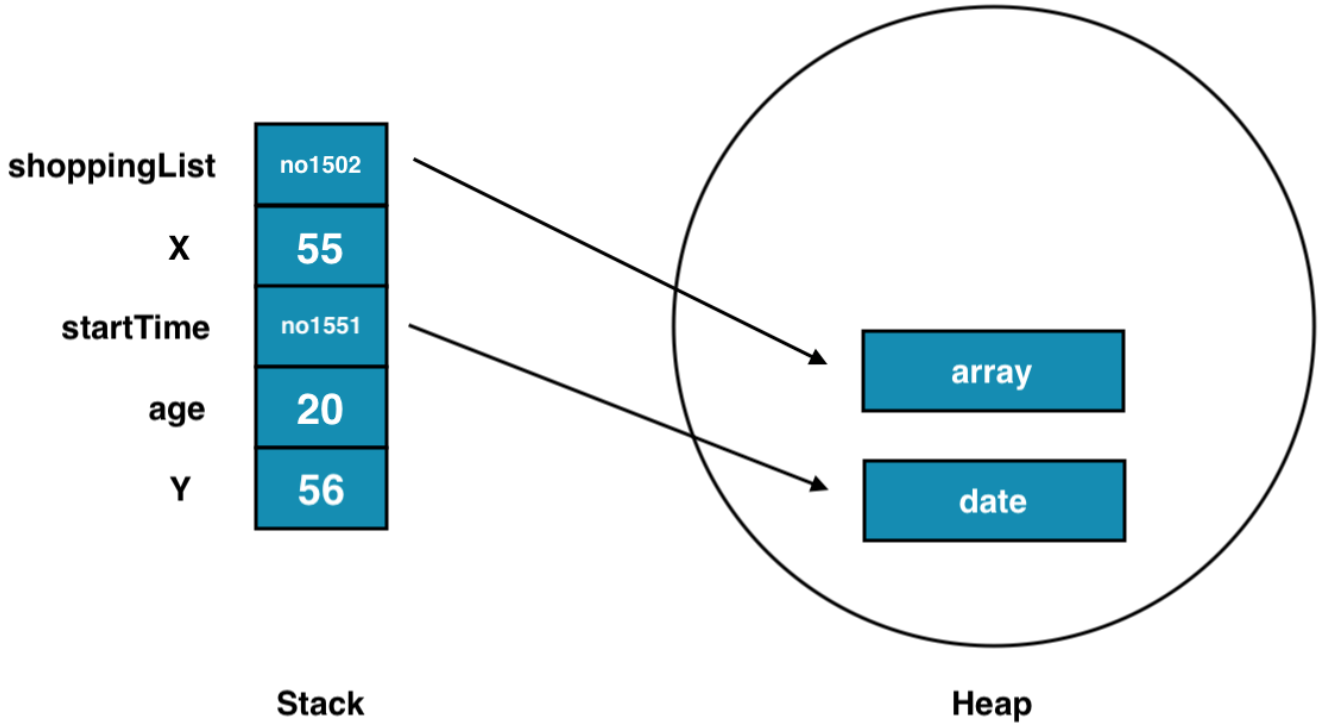
```
ArrayList<String> shoppingList = new ArrayList<String>(); //heap space
int x = 55; //stack memory
Date startTime = new Date(); //heap space
int age = 20; //stack memory
int y = x; //stack memory y = 55
```



- Stack'e age değeri 20 olarak eklendi.
- y, x ile aynı değere sahip olarak initialize edildi.
- y primitive type olduğu için direk stackte tutuluyor.

3.

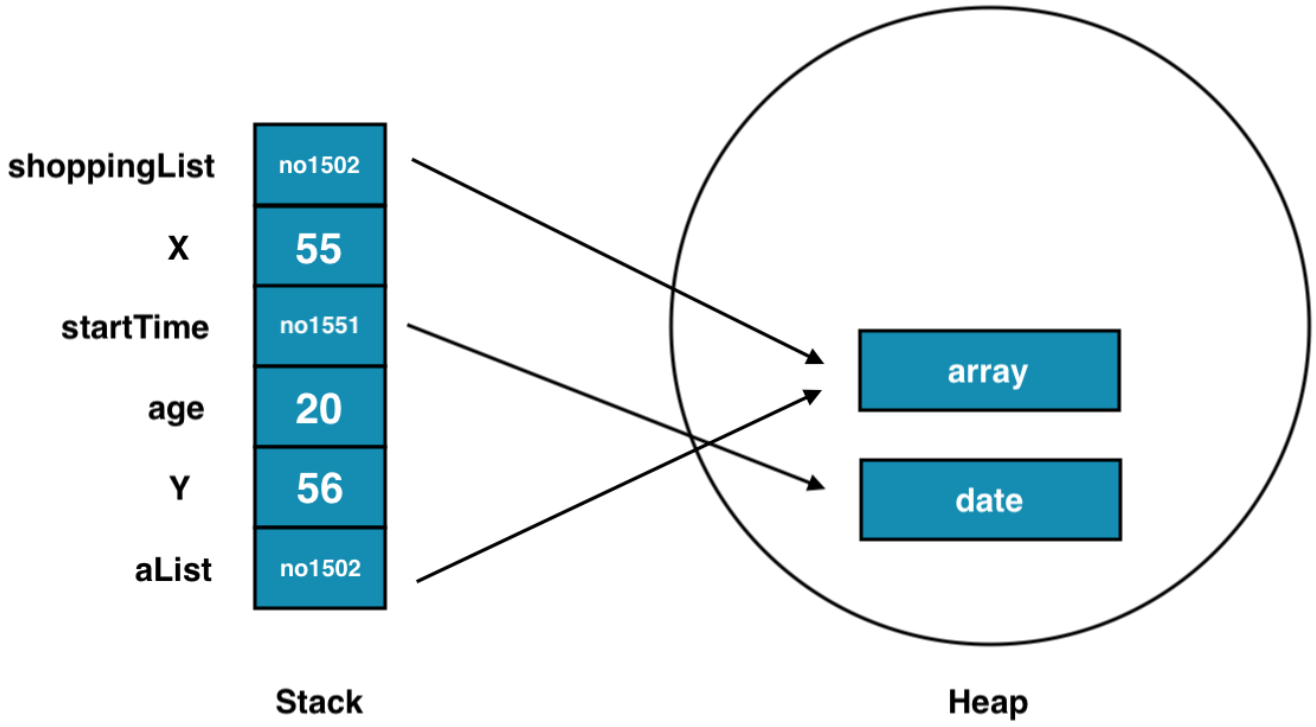
```
ArrayList<String> shoppingList = new ArrayList<String>(); //heap space
int x = 55; //stack memory
Date startTime = new Date(); //heap space
int age = 20; //stack memory
int y = x; //stack memory
y = 56; //stack memory y = 56
```



- y, x değeri ile initialize edilmiş olmasına rağmen, x'ten bağımsız olarak 56 ya eşitlenir, x değeri 55 olarak kalır.

4.

```
ArrayList<String> shoppingList = new ArrayList<String>(); //heap space
int x = 55; //stack memory
Date startTime = new Date(); //heap space
int age = 20; //stack memory
int y = x; //stack memory
y = 56; //stack memory y = 56
ArrayList<String> aList = shoppingList; //heap space
```



- aList değişkeni shoppingList'e eşitlendiğinde, shoppingList'in stack'teki adres değeri aList içine kopyalanmış olur.
- aList ve shoppingList değerleri heap'teki aynı değere işaret ederler.
- İki arrayden birine eleman ekleme gibi bir işlem yapılırsa, işlem heap'teki değer üzerinden yapılır. Bu yüzden, değişken yazdırılınca eklenen yeni eleman iki değişkende de gözükecektir.

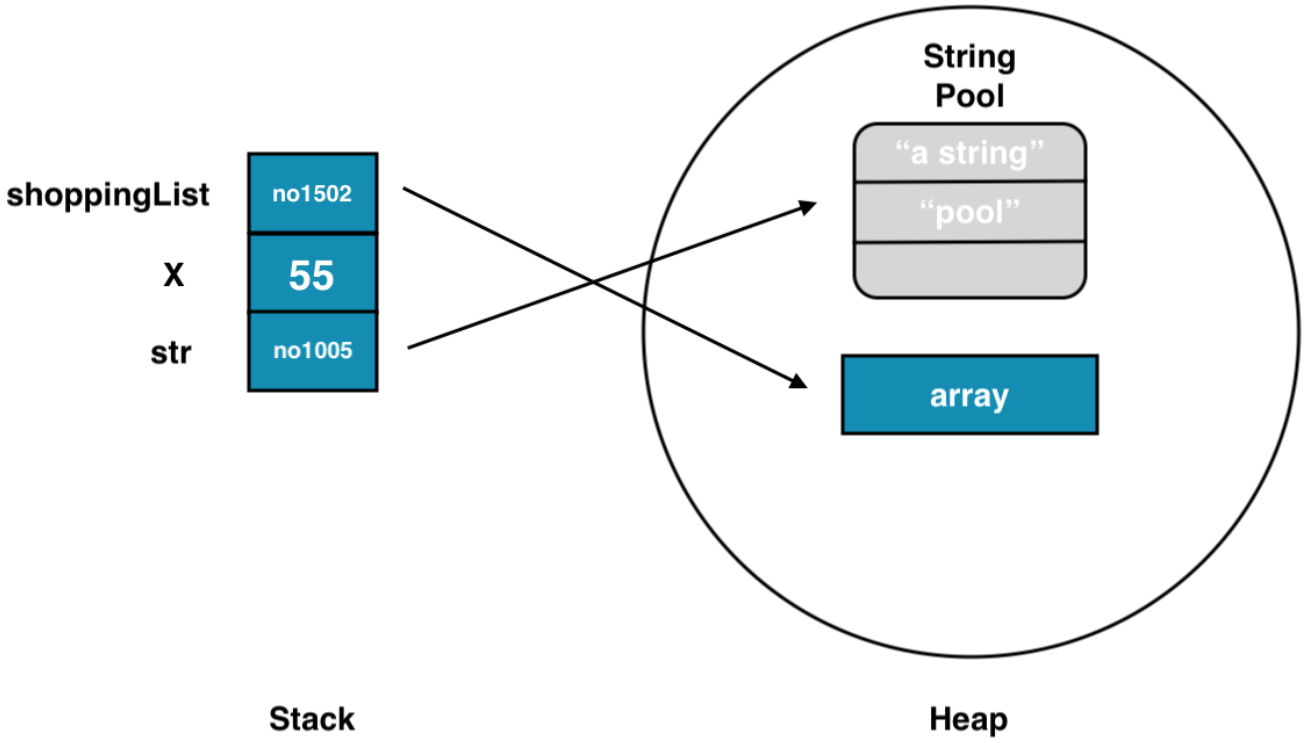
String Pool

Java'da tırnak işareti içinde ("") oluşturduğumuz String değerler, yani **string literal**'lar, performansın daha iyi olması heap space içinde yer alan **string pool**'da tutulur.

Aşağıda **string pool** yapısının örnekleri bulunuyor:

1.

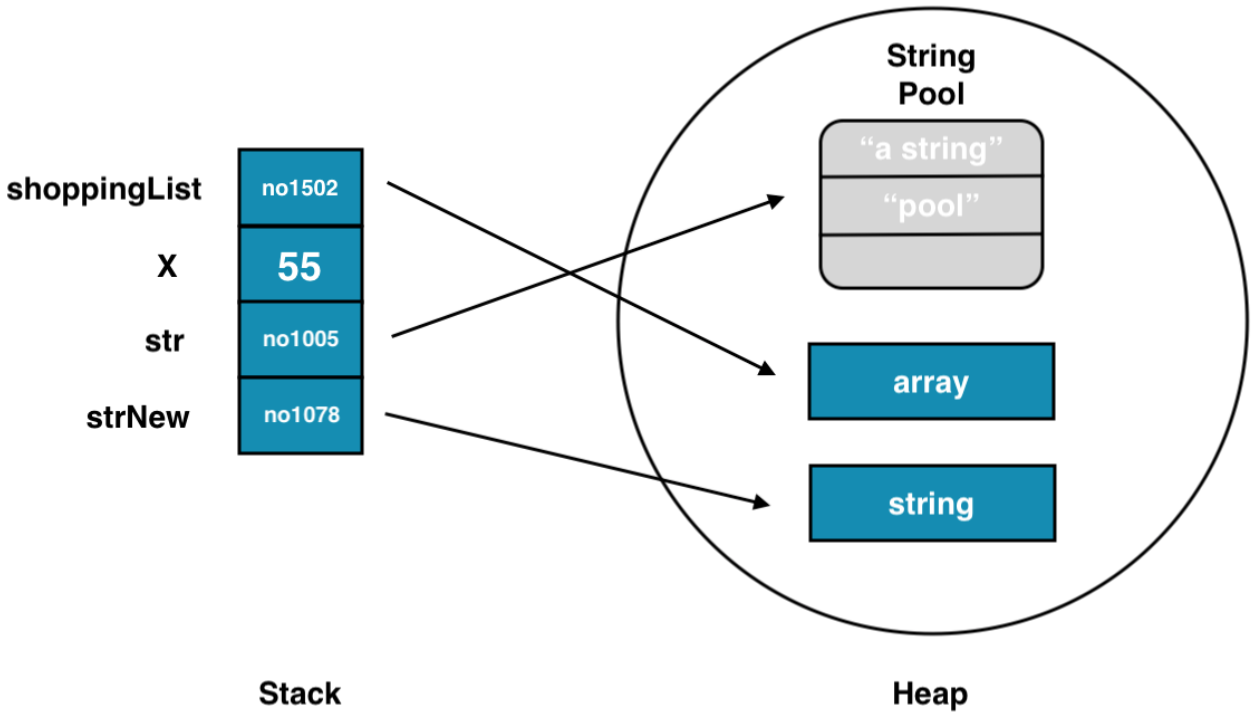
```
ArrayList<String> shoppingList = new ArrayList<String>();
int x = 55;
String str = "pool";
```



- String bir değer, tırnak işareti ile initialize edildiğinde, öncelikle string pool içindeki değerler kontrol edilir.
- Tanımlanan str değişkeninin değeri, string pool içinde olmadığı için yeni bir değer yaratılarak, stack adresi bu değere işaret edecek şekilde doldurulur.

2.

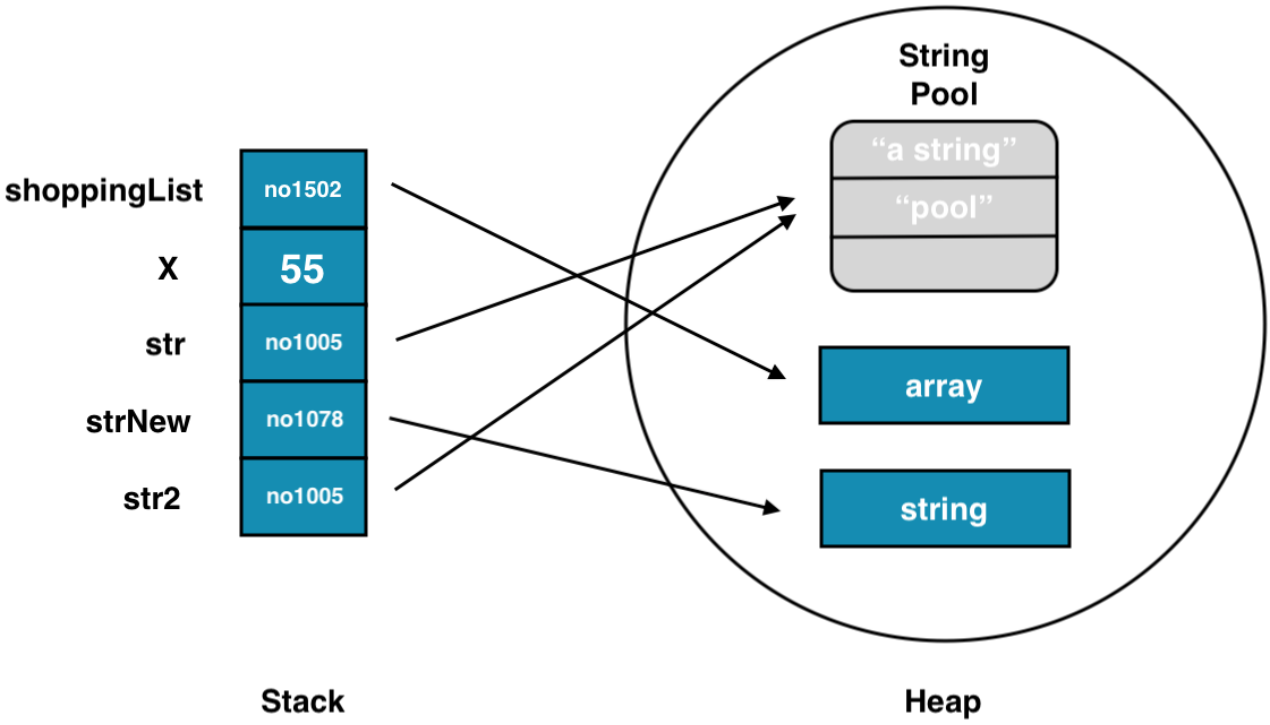
```
ArrayList<String> shoppingList = new ArrayList<String>();
int x = 55;
String str = "pool";
String strNew = new String("pool");
```



- String new kelime ise yaratılırsa, bu değer string pool içinde yaratılmaz, heap içinde yaratılır.

3.

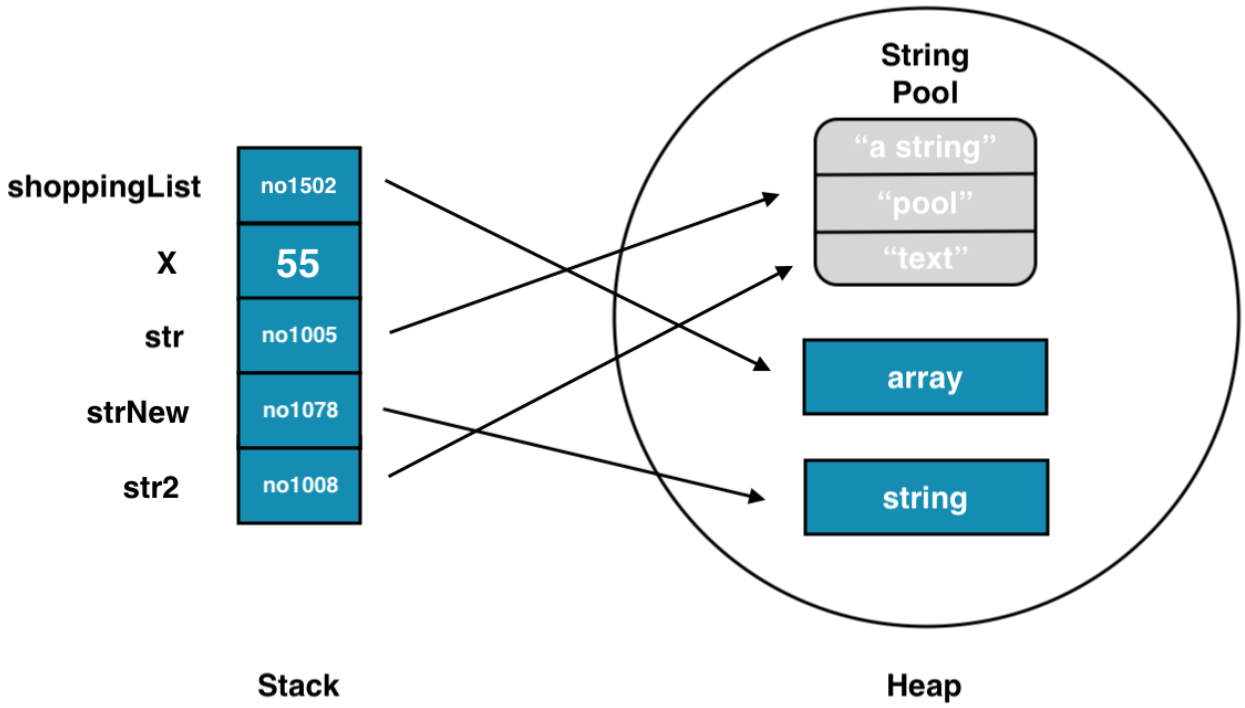
```
ArrayList<String> shoppingList = new ArrayList<String>();
int x = 55;
String str = "pool";
String strNew = new String("pool");
String str2 = "pool";
```



- Bu örnekte str2 değişkeni literal olarak, yani tırnak işareti ile yaratıldığı için, string pool kontrol ediliyor.
- str2'nin değeri string pool'da önceden bulunduğu için, varolan adres str2 değişkenine atanır.

4.

```
ArrayList<String> shoppingList = new ArrayList<String>();
int x = 55;
String str = "pool";
String strNew = new String("pool");
String str2 = "test";
```

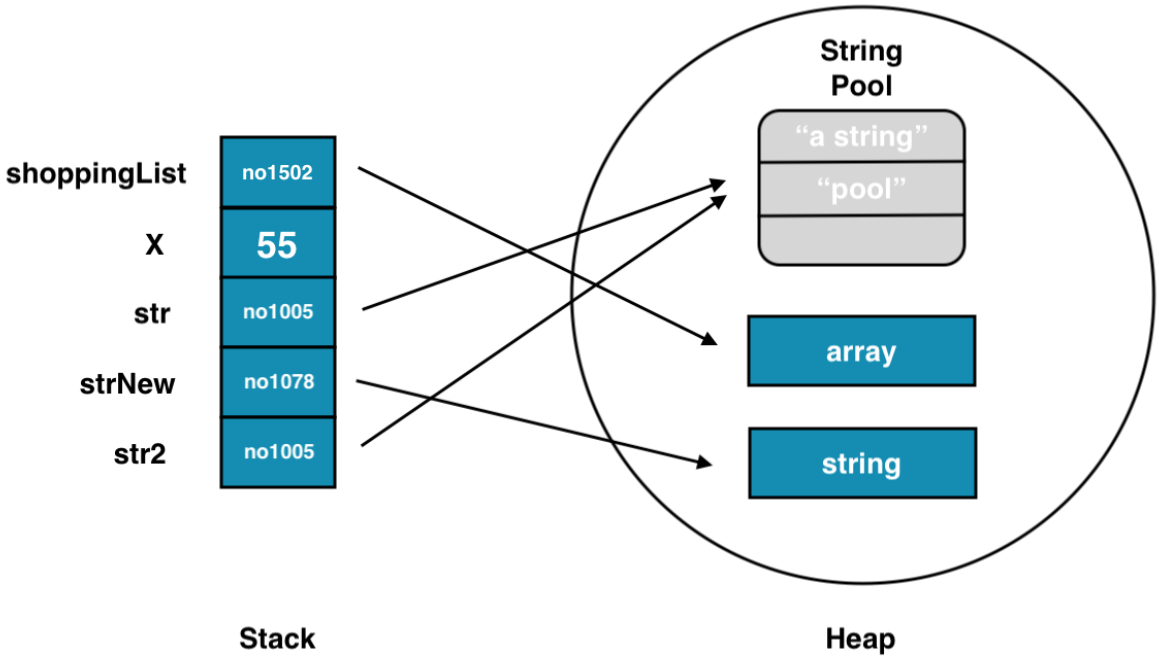


- str2 değiştirilirse str'nin değeri değişmez.
- String pool'da yeni bir değer oluşturulur.
- Bunun sebebi stringlerin **immutable** yani değişmez olmasıdır.

Neden String eşitliği kontrol edilirken .equals kullanılır?

Aşağıdaki örneği tekrar incelersek,

```
String str = "pool";
String strNew = new String("pool");
String str2 = "pool";
str == strNew // false
str.equals(strNew); // true
str == str2; // true
str.equals(str2); // true
strNew == str2; // false
strNew.equals(str2); // true
```



- İki string değişkeni **== operatörü** ile kıyaslandığında, değişkenlerin stack içindeki değerleri kıyaslanır.
- İki string değişkeni **.equals** ile kıyaslandığında, değişkenlerin sahip oldukları değerler kıyaslanır.
- Aynı değere sahip literal değişkenler, str ve str2, stack içinde aynı adrese sahip oldukları için, **==** ile kıyaslandığında sonuç **true** olur. Bu iki değişken **.equals** ile kıyaslandığında da sonuç **true** olur.
- str ve strNew aynı değere sahip olmalarına rağmen **==** ile karşılaştırıldıklarında, stack değerleri farklı olduğu için sonuç **false** olur. Bu iki değişken **.equals** ile karşılaştırıldığında, sahip oldukları değerler aynı olduğu için sonuç **true** olur.

Exception

Exception hata bilgisi içen bir objedir. Kod çalışırken bir noktada hata oluşursa exception objesi oluşturulur ve bu runtime sisteme aktarılır. Bu aktarma işlemine **exception throwing** yani istisna fırmatma deniyor. Bir kaç exception örneğine bakarsak:

IndexOutOfBoundsException: Bir arrayin uzunluğundan büyük bir indeksteki elemana erişilmeye çalışırsa atılır.

```
ArrayList<String> shoppingList = new ArrayList<String>();
 shoppingList.add("elma");
 shoppingList.get(8);
```

NullPointerException: Bir obje gereken noktada null bir değere ulaşılırsa atılır.

```
String name = null;
System.out.println(name.substring(0,1));
```

StringIndexOutOfBoundsException: Bir stringin uzunluğundan büyük bir indeksteki elemana erişilmeye çalışırsa atılır.

```
String name = "elma";
System.out.println(name.substring(7,8));
```

Bu tarz exceptionların testler esnasında tespit edilmesi veya kodlama esnasında potansiyel olarak exception oluşabilecek yerlerin belirlenip önlemlerin alınması gerekir.

Stack Trace

Stack trace programda bir exception oluştuğu esnada programın çağırdığı methodlar listesidir. Bu örnekte, methodlar aşağıdan yukarı doğru çağırılırkan, Game classının 79. satırında hata oluşmuştur.

```
İsmin nedir?
Exception in thread "main" java.lang.NullPointerException
        at com.irmak.Game.firstLetterUppercase(Game.java:79)
        at com.irmak.Game.getUsersName(Game.java:40)
        at com.irmak.Game.startGame(Game.java:31)
        at com.irmak.Game.greet(Game.java:24)
        at com.irmak.Game.main(Game.java:9)
```

Exception Handling

Exception handling, Türkçe'ye istisna ele alma olarak çevrilse de, daha çok hata yakalama olarak duyabilirsiniz.

Bir exception oluşturulup runtime'a aktarıldıktan sonra, çağırılan tüm methodlar, exception ele alınmış mı diye kontol edilir. Eğer hiç bir methodda exception yakalanmamışsa, program durdurulur.

Try-catch

Bir koddaki hatayı yakalamak için, kodu try-catch içine alabiliriz. Try, dene, catch ise yakala anlamımda. Catch yazdıktan sonra parantez içine yakalamak istediğimiz exception tipini (IndexOutOfBoundsException) ve ardından exceptionın atanacağı ismi (ex) yazıyoruz.

```
try {
    shoppingList.add("elma");
    shoppingList.get(8);
    System.out.println("Tamamlandi");
} catch (IndexOutOfBoundsException ex) {
    System.out.println("IndexOutOfBounds yakalandi");
}
```

Burda exception 3. satırda, arrayin 8. elemanına erişmeye çalışılırken atılacak. Bir hata atılınca, hata atılan satırdan sonraki kod satırları çalışmaz. Yani bu örnekte "Tamamlandi" kelimesi ekrana yazmayacak, hatadan sonra sonra, kod catch'e düşecek ve ekrana "IndexOutOfBounds yakalandi" yazılacak.

Yukarıdaki örnekte IndexOutOfBoundsException harince başka bir exception daha oluşabileceğini düşünüyorsak, başka bir catch ile bunu da ele alabiliriz.

```
try {
    shoppingList.add("elma");
    shoppingList.get(8);
    System.out.println("Tamamlandi");
} catch (IndexOutOfBoundsException ex) {
    System.out.println("IndexOutOfBounds yakalandi");
```

```
    } catch (NullPointerException ex) {
        System.out.println("NullPointer yakalandi");
    }
```

Bu kodda oluşabilecek tüm exceptionları yakalamak ve özel bir exception tipi belirtmek istemiyorsak catch içine aşağıdaki gibi sadece Exception da yazabiliriz, bu tüm Exception tiplerini yakalamamızı sağlar.

```
try {
    shoppingList.add("elma");
    shoppingList.get(8);
    System.out.println("Tamamlandi");
} catch (Exception ex) {
    System.out.println("Hata yakalandi");
}
```

Error

Exceptionlardan farklı olarak Java'da bir de **error**'lar var. Error ve exception kavramlarının ikisini de Türkçe'da hata olarak duyacaksınızdır ancak bu iki kavram birbirinden farklıdır. JVM bir error oluşturduğunda bu programın toparlayamadığı, ciddi bir durum oluşmuş olması anlamına gelir. Bunlar hafıza yetersizliği, stack taşması, donanımsal hatalar gibi sebeplerden kaynaklanabilir. Eğer geçerli bir sebep yoksa errorların catch bloğu içinde yakalanmaması, hatanın kaynağının bulunup çözülmesi gerekir. Bir kaç error örneğine bakarsak:

StackOverflowError: Stack'in taşması, kullanılacak memory kalmamasında oluşur. Recursion kaynaklı olabilir. **Recursion** bir methodun kendini çağırmasına denir.

```
public static void startGame() {
    startGame();
}
```

OutOfMemoryError: Heap space'de yer kalmaması durumdan oluşur.

IOError: Donanımsal hatalarda oluşur.

Try-catch-finally

Sistemsel kaynaklara, veritabanlarına erişirken, işlem bittikten sonra bu kaynakların kapatılması gerek. Aksi halde hafıza sızıntısı veya veritabanında bağlantıların tükenmesi gibi sorunlar ile karşılaşabiliriz. Bu kapatma işlemini kaynak ile işimiz bittiğinde yapmalıyız ancak kaynağa erişme ile kapatma arasında gerçekleşen işlemler esnasında bir hata oluşursa, hata sonrasındaki kod çalıştırılmayacağı için kaynak erişimi kapatılmadan kod catch'e düşebilir.

Bu durumda try-catch in sonuna bir finally bloğunu ekliyor ve kaynağı burda kapatıyoruz. Finally son bir işlem yapmak istediğiniz her try-catch'in sonuna eklenebilir. Finally bloğu, hata atılsa da atılmasa da çalışır. Bir örneğe bakarsak;

```
Scanner scanner = null;
try {
     scanner = new Scanner(System.in);
     String answer = scanner.nextLine();
     System.out.println(answer);
 } catch (Exception ex) {
     System.out.println("hata");
 } finally {
     if (scanner != null) {
         scanner.close();
     }
 }
```

Burda, finally bloğunun içindeki kaynağı kapatan .close() methodu her koşulda çağırılacak. scannerin değerinin null olmadığına emin olmak için de kapamadan önce null kontrolü yapılıyor.

Try with resources

Kaynak kapatma işlemi yaptığımız finally bloklarında, aşağıda örneğini verdiğim try with resources yapısını da kullanabiliriz.

```
 try (var scanner = new Scanner(System.in);) {
     String answer = scanner.nextLine();
     System.out.println(answer);
```

```
        } catch (Exception ex) {
            System.out.println("hata");
        }
```

Burda, erişeceğimiz kaynağı try'ın parantezlerinin içinde tanımlıyoruz. Bu tanımlama, işlevsel olarak finallyde yaptığımız null kontrolü ve close işlemlerinin aynısını yapıyor ama kod satırı olarak hem daha az satır kullanmış oluyoruz hem de daha temiz bir görüntü oluşuyor.