

Bansel Somergül

07.07.2023

Istanbul

12i Systems

Summer Internship



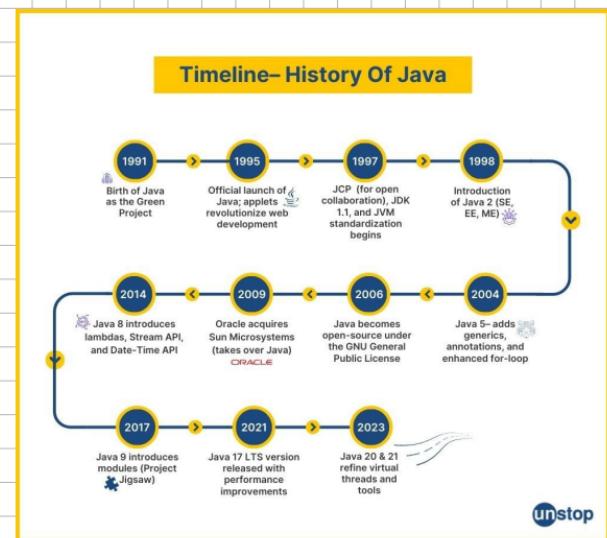
Java™

Java is a high-level, general-purpose, memory-safe, object-oriented programming language. It is intended to let programmers write once, run anywhere (WORA),^[18] meaning that compiled Java code can run on all platforms that support Java without the need to recompile.^[19] Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages.

Java gained popularity shortly after its release, and has been a popular programming language since then.^[20] Java was the third most popular programming language in 2022 according to GitHub.^[21] Although still widely popular, there has been a gradual decline in use of Java in recent years with other languages using JVM gaining popularity.^[22]

Java was designed by James Gosling at Sun Microsystems. It was released in May 1995 as a core component of Sun's Java platform. The original and reference implementation Java compilers, virtual machines, and class libraries were released by Sun under proprietary licenses. As of May 2007, in compliance with the specifications of the Java Community Process, Sun had relicensed most of its Java technologies under the GPL-2.0-only license. Oracle, which bought Sun in 2010, offers its own HotSpot Java Virtual Machine. However, the official

Java	
Paradigm	Multi-paradigm: generic, object-oriented (class-based), functional, imperative, reflective, concurrent
Designed by	James Gosling
Developer	Oracle Corporation
First appeared	May 23, 1995; 30 years ago ^[1]
Stable release	Java SE 24 ^{[2][3]} ✓ / 18 March 2025; 3 months ago
Typing discipline	Static, strong, safe, normative, manifest
Memory management	Automatic garbage collection
Filename extensions	.java, .class, .jar, .jmod, .war
Website	oracle.com/java/ ↗ java.com ↗ dev.java/ ↗
Influenced by	GLU ^[4] Simula ^[5] Lisp ^[4] Smalltalk ^[4] Ada 83, C++, ^[15] C# ^[4] Eiffel ^[7] Mesa ^[6] Modula-3, ^[9] Oberon, ^[10] Objective-C, ^[11] UCSD Pascal ^[12] , ^[13] Object Pascal ^[14]
Influenced	Ada 2005, ARKTS, BeanShell, C#, Chapel, ^[15] Clojure, ECMAScript, Fantom, Gambas, ^[16] Groovy, Hack, ^[17] Haxe, J#, JavaScript, JS++, Kotlin, PHP, Python, Scala, Seed7, Vals
Java Programming at Wikibooks	



Hangi IDE'ye tutturancağız

3'üncü de tutturancağız cuih
ide'ler belli tip uygulalar
geliştirmek için çok işe yarılır

IntelliJ Idea

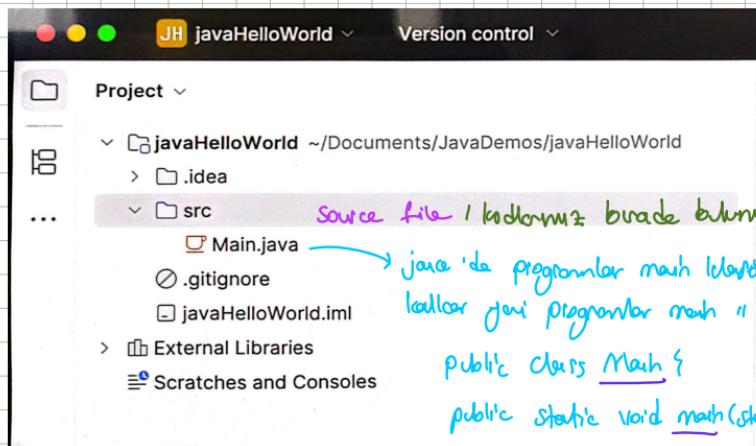
• Kullan gelistirme

Netbeans

• Swing gelistirme
cuih en iyi ide

Eclipse

• Kurucu uygulalar
cuih



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

intellisense

```
}
```

main structure
for Java

programlama yapmayı
yapmak için bu.
isaret tutturır

```
System.out.println("Hello Java");
```

→ etenice bir sententile
Java yazıncaı doğrular

line denote

→ Java, case sensitive'dır. Jan buyrun - Jancale farklı bir dildir.

II Variables

Degiskenler data tuttugunu işgorlardır. Bu işgorlарın sağesinde bellekte tutulan değişkenlerin adresleri bilinçde㈠ gerek. İsteklenen bu değişkenler ile ulaşılabilirlik şartı.

Ayrıca değişkenler sağesinde, bir dosya programının bir yerinde bir değişkenin bir kere atayıp sonradan o dosya isteklenirken kullanılabilirliğine

- Değişkenlerinizi bilgilere **reusability** sağlar
- dataları çağıştırılmış lokallara tek bir tek bir kullanılabilirlik için değişkenleri kullanırız
ama ana nedenlik bellekten fazla rafya yer almaz için kullanımızıza

II Data Types

- Değişken türmlerken, değişkenin tuttuğu verinin tipini belirtmeniniz gerekmektedir.
Cümlə: Java, type sensitivity bir dildir.

Primitive Types

type name	kind of value	memory used	size range
boolean	true or false	1 byte	not applicable
char	single character (Unicode)	2 bytes	all Unicode characters
byte	integer	1 byte	-128 to 127
short	integer	2 bytes	-32768 to 32767
int	integer	4 bytes	-2147483648 to 2147483647
long	integer	8 bytes	-9223372036854775808 to 9223372036854775807
float	floating-point number	4 bytes	$-3.40282347 \times 10^{38}$ to $-1.40239846 \times 10^{-45}$
double	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

→ modern long data tipi var icin en çok kullanılır olanlar, bit neden
int, byte, short vb. ile yaygın mı? Her segi long türündedirken short
mi?

- Cihazın ve donanım açısından bir sonraki önemdeki en önemli
farkı "memory size" kılavuzunda ortaya çıkar:

byte	1 byte
short	2 byte
int	4 byte
long	8 byte

geneldeki üzere $\times 2$ 'lik
bir artışı mevcuttur

// Switch - Case

- genelde ifi kod yapısını telif hukmünü sınırlamak için kullanılır
- program dallandırma yapıları

switch (condition) {

case : _____
 break;

default : _____

}

- logikte hatalı yazıldığınız
larda direkt male bulun

ctrl + shift + alt + 1

// Loops

Bir programda, birbirine benzeyen ifadeler / genel ifadeleme teknikleri kullanılarak
yazılan işlemleri döngüler denir.

→ for → while → do-while → foreach

for (int baslangic ; bitis ; artim)

{

 // kodlar burası

}

while (dengesi saglanma)

{

 // kodlar burası

 // koşul artırmayı unutma

}

do {

 // kodlar burası

 // + kere koşul olurken döngü

} while (şart burası)

// Arrays

Benzersiz türdeki verilen bir arada tutmak istenilen eserlerin topluluğudur.

Data Type [] ArrayName = new Data Type [ArraySize]

for (String ogrenci : ogrenciler)
{

 // kodlar burası

}

II Multidimensional Arrays

Cüde boyutlu dizi ləndir.

`DataType[][] ArrayName = new DataType[size][size]`

II Stringler ile Çalışma

Stringler, vəzində karakter dizi ləndir. Buranın dələyi vətəndə belənən qeydiyi məntilişlər kəndə de gələnlərdir (indi's rəb'i).

- String mesaj = "Hello World!"
- mesaj.length → elemən sayıını verir
- mesaj.charAt() → girilen indexdəki eleməni verir
- mesaj.concat("Metin") → metin birləştirmə işləni yapar
çıktı: Hello World!Metin


→ boşluğda da bir karakterlərdür.

* buradakı concat işləni sonuncuda orjinal string olun mesaj stringi iəməgi degişməz

mesaj.concat("Metin") → Yeni bir string olur, kəskinələk olaraq başqa deyiklərə atrafında gələcək

→ mesaj. startsWith(.) → parentet içindeki ifade ile başlıyor
bu ifadede oronacık ifade
" " ile oronur, '' ile değil | cuihü bir string ile
nw dyle barkar

bu ifadede oronacık ifade | cuihü bir karakter ile
" " ile oronur, '' ile değil | baslıyor durumunu sağlanır

→ mesaj. endsWith(.) → birne durumunu içindeler, mantık gibi

asagidaki ifade destination obdu, konuları daşıyıcı iken burası传染病

char [] karakterler = new char [5]

→ mesaj. getChars(srcBegin=0, srcEnd=5, karakterler, destBegin=0)

string içindelki belli
bir kuruş atmamız
baglar

source
başlangıç
nolması

source
bitiş
nolması

dest
nolma

destination
başlangıç
nolması

5 dörtl
editmet 0,1,2,3,4
5 tane

→ mesaj. indexOf(' ') : parentet içinde yazılanın kaçinci
(" ") karakter oldugunu bulmaya yarar
indis bilgisini verir

→ mesaj. indexOf(' ') : istedeki ile beraber mantık one orone
en sağdan başlar
(" ")

* her iliskide, oronan ifadesi bulduğu ilk end sonucu verir
ve oronayı bıralar.

→ mesaj. replace(" ", " ") : parentet içindelki ifadesi mesaj
değişkenin eski degeri ile değiştirir

oldchar: eski ifade veya target:

newchar: yeni ifade

replacement:

bu islemlerin sonucu
yeni metin verir
original ifade degismez

↳ substring: gelen metin icinden bir parca almak istir

mesaj.substring(begIndex: 3, endIndex: 8) 3-4-5-6-7
bu indexler olur

↳ mesaj.split(regex: "-"): orjinal metni çift tırnak içindeki
dunne gibi parçalar

Geçerli döşemeler, jaziklerde
için for kullandıkları

↳ mesaj.toLowerCase(): mesajı küçük harfe çevirir

mesaj.toUpperCase(): " " büyük " "

↳ mesaj.trim(): stringin başındaki - sonundaki boşlukları siler

```
① Main.java ×
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Scanner input = new Scanner(System.in); // Scanner object
7         System.out.print("Please enter a number: ");
8         int number = input.nextInt(); // To get a number
9
10
11         boolean isPrime = true;
12
13         // Prime numbers bigger than 1
14         if (number == 1) {
15             isPrime = false;
16         }
17
18         // The only even prime number is 2
19         else if (number == 2) {
20             isPrime = true;
21         }
22
23         else {
24             for (int i = 2; i <= Math.sqrt(number); i++) {
25                 if (number % i == 0) {
26                     isPrime = false;
27                     break;
28                 }
29             }
30
31             if (isPrime) {
32                 System.out.println("Prime number");
33             } else {
34                 System.out.println("Not Prime number");
35             }
36
37             input.close(); // Closing scanner object
38         }
39     }
40 }
```

əzələcəyi mi deyil mi?

Main.java

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Scanner input = new Scanner(System.in);
7
8         System.out.print("Please Enter a Character = ");
9         char character = input.next().charAt(0);
10
11         // I want to keep code simple. So i didn't check other characters like i,i etc.
12         // And also i didn't use try-catch to keep code simple and just learn essentials
13         if (character == 'a' || character == 'e' || character == 'o' || character == 'u') {
14             System.out.println("Girilen karakter sessli harftir");
15         }
16
17         else{
18             System.out.println("Girilen karakter sessiz harftir");
19         }
20
21         input.close();
22     }
23 }
24 }
```

Main.java

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         // In number theory, a perfect number is a positive integer that is equal to the sum of its positive proper divisors,
7         // that is, divisors excluding the number itself. For instance, 6 has proper divisors 1, 2 and 3, and 1 + 2 + 3 = 6, so 6 is a perfect number.
8
9         Scanner input = new Scanner(System.in);
10        System.out.print("Please enter a number = ");
11        int number = input.nextInt();
12
13        int sum = 0;
14
15        for (int i = 1; i <= (number / 2); i++) {
16            if (number % i == 0) {
17                sum += i;
18            }
19        }
20
21        if (sum == number) {
22            System.out.println("The number is a perfect number ");
23        }
24
25        else {
26            System.out.println("The number is not a perfect number ");
27        }
28    }
29 }
```

```
1 import java.util.Scanner;
2 
3 public class Main {
4     public static void main(String[] args) {
5 
6         // Two numbers whose positive integer divisors, excluding themselves, are equal are called amicable numbers.
7 
8         Scanner input = new Scanner(System.in);
9         System.out.print("Please Enter First Number = ");
10        int firstNumber = input.nextInt();
11        System.out.print("Please Enter Second Number = ");
12        int secondNumber = input.nextInt();
13 
14        int sum1 = 0;
15        int sum2 = 0;
16 
17        for (int i = 1; i <= (firstNumber/2); i++) {
18            if (firstNumber % i == 0) {
19                sum1 += i;
20            }
21        }
22 
23        for (int i = 1; i <= (secondNumber/2); i++) {
24            if (secondNumber % i == 0) {
25                sum2 += i;
26            }
27        }
28 
29        if (sum1 == secondNumber && sum2 == firstNumber) {
30            System.out.println(firstNumber + " and " + secondNumber + " are friend number.");
31        }
32 
33        else {
34            System.out.println(firstNumber + " and " + secondNumber + " are not friend number.");
35        }
36 
37        input.close();
38    }
39 }
```

```
1 import java.util.Scanner;
2 
3 public class Main {
4     public static void main(String[] args) {
5 
6         int[] sayilar = new int[6];
7 
8         Scanner input = new Scanner(System.in);
9 
10        for (int i = 0; i < sayilar.length; i++) {
11            System.out.print("Please Enter " + (i+1) + ". Number: ");
12            sayilar[i] = input.nextInt();
13        }
14 
15        int aranacakSayi;
16        System.out.print("Please Enter Searching Number: ");
17        aranacakSayi = input.nextInt();
18 
19        boolean bulduduMu = false;
20 
21        for (int i = 0; i < sayilar.length ; i++) {
22            if (sayilar[i] == aranacakSayi) {
23                System.out.println("The Number is at " + i + ". index");
24                bulduduMu = true;
25            }
26        }
27 
28        if (!bulduduMu) {
29            System.out.println("The Number is not in the array");
30        }
31 
32        input.close();
33 
34    }
35 }
```

II Metodlarla Çalışma

Programlamada "Don't Repeat Yourself" prensibi mercattir. Yani

'Kendini tekrar etme' denilir. Yani, sonlu sayıda kod + kere
kullanmak istiyacını oluşturmak yerine!

Bu nedenle da "fonksiyonel programlama" mantığı derneğe gider. Bu
bir fonksiyonun görevi de aynı dikkat etmektedir.

→ Bir fonksiyonun görevi bir işlemi yapmak şeklinde yazılmak
gerekir. Cunku bir fonksiyon içerişinde birden fazla işlem olursa ve
bu işlerden sadexe bir tanesini kullanmak istesek tüm parçası
kullanmamız gereklidir. Sadexe o kod parçasını kullanma səməniz

Oluşur. Yani kodu ve öz olsalı:

Bir fonksiyon → bir işlem

* main yapınız da bir method'dur ve jax'a de ille bize matn
(yazılı) verir.

Main.java x

```
1 import java.util.Scanner;  
2  
3 public class Main {  
4     public static void main(String[] args) {  
5         sayiBulmaca(); → soyi Bulmaca adında  
6         bir fonksiyon  
7     }  
8 }
```

10 public static void sayiBulmaca() { 1 usage }

fonsiyonun scope'u içinde de
kodlar mercattir

* soyi Bulmaca fonsiyonu
1 kere yazıldı. Article
istedigimiz zaman her
defter istedigimiz
zaman aqroabilidir.

→ main'den aqroabilen fonsiyon

fonsiyon istenildiğinde
icin carnel case
notasyonu =

public static void mesajVer(int aranacakSayi, int[] sayilar) {

fonksiyon özellikleri

fonksiyon
adı

Fonksiyon'a gelecek
olarak parametreler

Main.java ×

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         // Call the function
7         sayiBulmaca();
8
9     }
10
11     // sayiBulmaca function
12     public static void sayiBulmaca() { 1 usage
13
14         int[] sayilar = new int[6];
15
16         Scanner input = new Scanner(System.in);
17
18         for (int i = 0; i < sayilar.length; i++) {
19             System.out.print("Please Enter " + (i+1) + ". Number: ");
20             sayilar[i] = input.nextInt();
21         }
22
23         int aranacakSayi;
24         System.out.print("Please Enter Searching Number: ");
25         aranacakSayi = input.nextInt();
26
27         mesajVer(aranacakSayi, sayilar);
28
29         input.close();
30     }
31
32     @
33     public static void mesajVer(int aranacakSayi, int[] sayilar) { 1 usage
34         boolean bulunduMu = false;
35
36         for (int i = 0; i < sayilar.length ; i++) {
37             if (sayilar[i] == aranacakSayi) {
38                 System.out.println("The Number is at " + i + ". index");
39                 bulunduMu = true;
40             }
41
42             if (!bulduMu) {
43                 System.out.println("The Number is not in the array");
44             }
45         }
46     }
}
```

* void metodlar, genye deger donduran jepibirdir

void metodlar, islen gereklilikte icin, bir enin genye getirmek icin vb. islen bozlu kullanilar.

→ void denebili fonksiyonlar genye deger donduran fonksiyonlardır. Bu fonksiyonlar icin return ifadesi kullanılır.

```
public static void guncelle() { no usages  
    System.out.println("Güncellendi");  
}
```

```
// a return function  
public static int topla(int a, int b) { 1 usage  
    return a + b;  
}
```

parametrelər
fonksiyon tipi ile return tipi
ayri olmala zənəbdür

Variable Arguments

fonksiyonlarda parametre olmali istedigimiz kəndar parametre göndərmək üçün variable arguments kullanılır

```
public static int topla( int... sayilar )
```

variable arguments füsmi,
bir nevi int array göndərilməsi gibi olur

```
// a return function  
public static int topla(int a, int b) { 1 usage  
    return a + b;  
}  
  
// variable arguments  
public static int topla(int... sayilar) { no usages  
    int toplam = 0;  
    for (int sayı : sayilar) {  
        toplam += sayı;  
    }  
    return toplam;  
}
```

ayrı 'simde ora forlu
iñi fonksiyon cunus'

döni təzələmə forlu
parametre sayilar forlu

↳ variable arguments yapısı, parametre olarak ne gönderdiğimde kullanımda poteri kodu okuyabiliyor olsun gibi için test edilebilirliği oluşturmakta sor

II class'lar ile Çalışma

Java, C#, C++ gibi dillerde nesne oluşturmak için bir sınıf ile örneklendirilebilir. Nesne oluşturmak için sınıfın içindeki metodları kullanır.

Sınırlı tutarlılığı: herhangi bir sınıfın içindeki metodları kullanımda sınırlı tutarlılığı var.

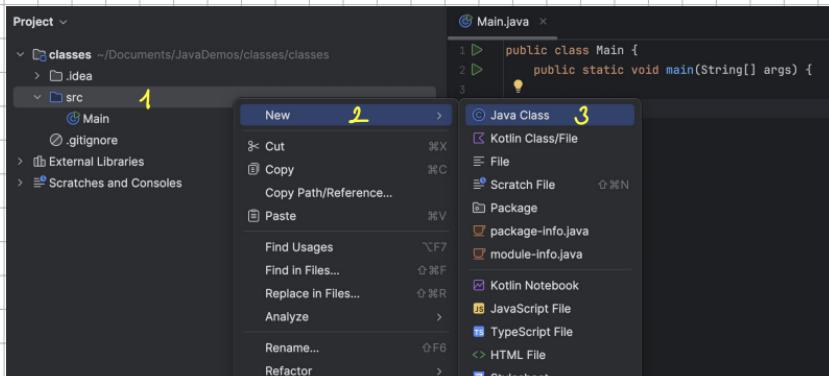
JAVA'da var olan sınırları, genelde metodları da kullanmakta sınırlı tutarlılığı koruyor.

• Bize en çok yardım eden bir class içinde kullanabileceğimiz

```
public class Main { → main class'i içinde kullanabileceğimiz
    public static void main (String[] args) {
        }
    }
```

• JAVADA her class 'tir / nesne 'dir

1. class'ların ilk ve temel özelliğinin genel olduğu



```

Main.java x CustomerManager.java → tımladığınız class
1 public class Main {
2     public static void main(String[] args) {
3
4         CustomerManager customerManager = new CustomerManager(); → class'ından object
5         bu class'tan bu isimde new short üretmektedir
6     }
7 }

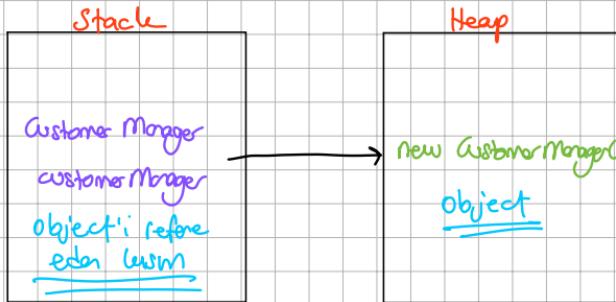
```

Yani burada class'ının bir örneğinin üretmektedir. Yani class'ının
bir object üretmektedir.

Artık bu nesneyi kullanıp class'ının özellikleri kullanabiliriz.

→ class'lar referans tipleridir

- bilgişayar belleği ibi ana bükümü oluşturur:



```

Main.java x CustomerManager.java
1 public class Main {
2     public static void main(String[] args) {
3
4         // reference type
5         CustomerManager customerManager = new CustomerManager();
6         customerManager.Add();
7         customerManager.Remove();
8         customerManager.Update();
9
10
11         CustomerManager customerManager2 = new CustomerManager();
12         customerManager = customerManager2; // both object show same thing
13
14     }
15

```

Hedef referansı oluyan
Atesneler Garbage Collector
Jepisi sefesinde
silmek

* Programlarda kullanığınız dört tipi iki şekilde dir.

1. Value type

- Stack'te leşmekte
- byte, short, int, long
- float, double
- char
- boolean

2. Reference type

- Stack'ta heap'te leşmekte
- string
- scanner, arraylist
- object
- interface
- array, enum

► Java'da veriler bellekte saklanma ve değişkenlere atama şekillerine göre temel olarak ikiye ayrılır: değer tipleri (value types) ve referans tipleri (reference types). Bu ayrılm, değişkenlerin nasıl davranışlarını ve bellekte nasıl yönetildiğini anlamak için çok önemlidir.

Değer Tipleri (Value Types) / İlkel Veri Tipleri (Primitive Data Types)

Java'da değer tipleri genellikle **ilkel veri tipleri (primitive data types)** olarak bilinir. Bu tipler, verinin kendisini doğrudan saklar. Yani bir değişken bir ilkel türde sahip olduğunda, o değişkenin bellekteki konumu doğrudan verinin değerini içerir.

Özellikleri:

- **Bellekte Doğrudan Değer Saklar:** Değişkenin bellekteki yeri, doğrudan o değerin kendisidir.
- **Sabit Boyut:** Her ilkel tipin bellekte kapladığı yer sabittir.
- **Stack Bellekte Saklanır:** Genellikle **stack (yığın) bellek** alanında saklanırlar. Bu, daha hızlı erişim sağlar.
- **null Değer Alamazlar:** İlkel tipler `null` (boş) değer alamazlar. Her zaman varsayılan bir başlangıç değerine sahiptirler (örneğin, sayılar için `0`, boolean için `false`).
- **Atama Yaptığınızda Kopyalama Olur:** Bir ilkel tip değişkenini başka bir ilkel tip değişkenine atadığınızda, değerin bir kopyası oluşturulur. Yani orijinal değişkenle yeni değişkenin bağımsız kopyaları olur. Birini değiştirmek diğerini etkilemez.

Java'daki İlkel Veri Tipleri (8 Tane):

1. Tam Sayılar:

- `byte` (8-bit)
- `short` (16-bit)
- `int` (32-bit, en sık kullanılan)
- `long` (64-bit)

2. Ondalık Sayılar (Kayan Noktalı):

- `float` (32-bit)
- `double` (64-bit, varsayılan ve daha yaygın)

3. Karakterler:

- `char` (16-bit Unicode karakter)

4. Mantıksal Değerler:

- `boolean` (`true` veya `false`)

Örnek:

```
Java

int sayi1 = 10;
int sayi2 = sayi1; // sayi1'in değeri olan 10, sayi2'ye kopulanır

sayi2 = 20; // sayi2'yı değiştirmek sayi1'i etkilemez

System.out.println("sayi1: " + sayi1); // Çıktı: sayi1: 10
System.out.println("sayi2: " + sayi2); // Çıktı: sayi2: 20
```

Referans Tipleri (Reference Types)

Referans tipleri, verinin kendisini doğrudan saklamazlar. Bunun yerine, verinin (yani nesnenin) bellekteki konumuna bir **referans (bellek adresi)** tutarlar. Asıl veri (nesne) **heap (yükün) bellek** alanında saklanır.

Özellikleri:

- **Bellekte Referans Saklar:** Değişkenin kendisi, objenin bellekteki adresini (referansını) tutar. Objenin verileri ise o adreste bulunur.

- **Heap Bellekte Saklanır:** Referans tiplerinin asıl verileri (objeleri) **heap bellek** alanında oluşturulur ve saklanır. Java'nın çöp toplayıcısı (Garbage Collector) bu alanı yönetir.
- **null Değer Alabilirler:** Bir referans tipi değişken, hiçbir objeyi işaret etmiyorsa `null` değerini alabilir.
- **Atama Yaptığınızda Referans Kopyalanır:** Bir referans tipi değişkeni başka bir referans tipi değişkenine atadığınızda, objenin kendisi kopyalanmaz; sadece objeye olan **referans (bellek adresi)** kopyalanır. Bu durumda, her iki değişken de bellekteki **aynı objeyi** işaret eder. Bir değişken üzerinden objede yapılan değişiklikler, diğer değişken üzerinden de görünür olur.

Java'daki Referans Tipleri:

- **Sınıflar (Classes):** `String`, `Scanner`, `ArrayList`, `Object` ve kullanıcı tanımlı tüm sınıflar (`Person`, `Car` vb.).
- **Arayüzler (Interfaces):**
- **Diziler (Arrays):** lüklü tipleri veya referans tipleri barındıran tüm diziler (`int[]`, `String[]`, `Person[]` vb.).
- **Enum'lar (Enumerations):**

Örnek:

```
Java

String metin1 = "Merhaba"; // metin1, "Merhaba" objesinin referansını tutar
String metin2 = metin1; // metin2 de aynı "Merhaba" objesinin referansını tutar

// Not: Stringler Java'da 'immutable' (değişmez) olduğu için bu örnek biraz yanlıltır.
// String bir değer ataması aslında yeni bir obje yaratır.
// Daha iyi bir örnek için kendi sınıfımızı kullanalım:

class Kutup {
    int x;
    Kutup(int x) { this.x = x; }
}
```

```

public class Main {
    public static void main(String[] args) {
        Kutup nokta1 = new Kutup(10); // nokta1, heap'te yeni bir Kutup objesine referans eder
        Kutup nokta2 = nokta1; // nokta2 de AYNI Kutup objesine referans eder

        nokta2.x = 20; // nokta2 üzerinden objenin x değerini değiştirdik

        System.out.println("nokta1.x: " + nokta1.x); // Çıktı: nokta1.x: 20 (Çünkü
        System.out.println("nokta2.x: " + nokta2.x); // Çıktı: nokta2.x: 20
    }
}

```

Temel Fark Özeti:

Özellik	Değer Tipleri (İlkel Tipler)	Referans Tipleri
Ne Saklar?	Verinin kendisini	Verinin (objenin) bellek adresini
Bellek Alanı	Genellikle Stack	Heap
Boyut	Sabit	Değişkenye göre değişir (objenin boyutu)
null Değer?	Alamaz	Alabilir
Atama Davranışı	Değer kopyalama (bağımsız kopyalar)	Referans kopyalama (aynı objeyi işaret eder)
Örnekler	int, boolean, double, char vb.	String, Array, Sınıf objeleri, Scanner vb.

Export to Sheets

Encapsulation

• Field ve Attribute

Class'lar ortak操作larını takip etmesinin yanı sıra bir diğer özelliti de "özellik" formasyonu. Bu özellitler 'field' ve 'attribute' olarak isimlendirilir

• data tutan yapıllerdir
• Variables'lerdir bir nesne

* Yazılım geliştiricilerin SOLID prensibi önemlidir. Bu prenziye iliskide deghizilebilir

```

Main.java  ProductManager.java  Product.java
1 public class Main {
2     public static void main(String[] args) {
3         Product product = new Product();
4         product.name = "Laptop";
5         product.price = 36980;
6         product.description = "Asus ROG Strix 166";
7         product.id = 1;
8         product.stockAmount = 3;
9         System.out.println(product.name);
10
11     ProductManager productManager = new ProductManager();
12     productManager.Add(product);
13 }
14 }
```

```

Main.java  ProductManager.java  Product.java
1 public class ProductManager { no usages
2     public void Add(Product product) { // sending product object as a parameter no usages
3         // JDBC
4         System.out.println("Product Added " + product.name);
5     }
6 }
```

```

Main.java  ProductManager.java  Product.java
1 public class Product { no usages
2     // attributes veya field
3     int id; no usages
4     String name; no usages
5     String description; no usages
6     double price; no usages
7     int stockAmount; no usages
8 }
```

Java'da Encapsulation (Kapsülleme) Nedir?

Encapsulation (Kapsülleme), nesne yönelimli programmanın (OOP) dört temel prensibinden biridir. Temel olarak, veriyi (nitelikler/özellikler) ve o veri üzerinde işlem yapan metodları (davranıslar) bir araya getirme ve bu verİYE dışardan doğrudan erişimi kısıtlama sürecidir. En basit ifadeyle, bir nesnenin iç çalışma mekanizmasını dış dünyadan gizlemek ve sadece belirlenmiş arayüzler (metotlar) aracılığıyla erişim sağlamak.

Bir hapi veya kapsülü düşünebilirsiniz. Kapsülü içinde farklı maddeler bulunur ama siz sadece dışındaki kapsülü görür ve yutarsınız. İçindeki maddelerin ne olduğunu, nasıl çalıştığını bilmenize veya doğrudan onlara dokunmanız gereklidir. Encapsulation da benzer bir mantıkla çalışır.

Encapsulation'ın Temel Bileşenleri ve Nasıl Uygulanır?

Java'da encapsulation genellikle şu şekilde uygulanır:

1. Nitelikleri (`private`) Yapmak:

- Bir sınıfın verilerini (fields/attributes) doğrudan dışardan erişilemez hale getirmek için `private` erişim belirleyicisi kullanılır. `private` olarak tanımlanan bir niteliğe yalnızca aynı sınıfından erişilebilir. Bu, dışardan doğrudan müdahaleyi engeller ve verinin bütünlüğünü korur.

2. `public Getirici (Getter)` ve `Ayarlayıcı (Setter)` Metotları Kullanmak:

- `private` olarak tanımlanmış verilere dışardan kontrollü bir şekilde erişim sağlamak ve onları değiştirmek için `public` metotlar (genellikle getter ve setter metotları olarak adlandırılır) yazarısınız.
- `Getter Metotları (get...)`: Bir niteliğin değerini okumak (almak) için kullanılır.
- `Setter Metotları (set...)`: Bir niteliğin değerini değiştirmek (ayarlamak) için kullanılır. Bu metotlar içinde veri doğrulama (validation) ve iş kuralları gibi mantıklar da eklenebilir.

- Encapsulation, kullanıcıyı herhangi bir alanı kullanmayı kısıtlıyor
- Bir object'in iç contente ulaşırırmamızı dış dünyadan gizlemek ve sadece belirlenmiş metodlar aracılığıyla erişim sağlayacaktır.

Neden Encapsulation Kullanılır? (Faydalari)

1. Veri Gizleme (Data Hiding):

- Nesnenin iç durumunun dış dünyadan gizlenmesini sağlar. Kullanıcılar (diğer kod parçaları), bir nesnenin iç detaylarını bilmek sorunda kalmazlar, sadece o nesnenin sunduğu arayüzü (metotları) kullanırlar.

2. Veri Bütünlüğü ve Güvenliği:

- Niteliklere doğrudan erişimi engellediği için, verilerin yanlış veya geçersiz bir duruma düşmesini onler. Setter metotları aracılığıyla veri girişi kontrol edilebilir ve doğrulanabilir. Örneğin, bir yaş değerinin negatif olmasına engellebilirsiniz.

3. Esneklik ve Bakım Kolaylığı:

- Bir sınıfın iç yapısını değiştirdiğinizde (örneğin, bir niteliğin adını veya veri tipini değiştirdiğinizde), bu değişiklik dış kodu etkilemez. Çünkü dış kod sadexe getter/setter metodlarını kullanır, niteliğin kendisine bağımlı değildir. Bu da kodun bakımını ve evrimini kolaylaştırır.

4. Kullanım Kolaylığı (Usability):

- Sınıfın karmaşık iç yapısını dışarıya açmak yerine, sadece anlaşılır ve amaca yönelik metotlar sunar. Bu, sınıfı kullanan geliştiriciler için daha basit ve hata yapmaya daha az açık bir arayüz sağlar.

```
public class Ogrenci {  
    // 1. Nitelikler 'private' olarak tanımlanır (veri gizleme)  
    private String ad;  
    private int yas;  
    private String ogrenciNo;  
  
    // Kurucu metod (Constructor) - Nesne oluşturulurken başlangıç değerlerini ayarlar  
    public Ogrenci(String ad, int yas, String ogrenciNo) {  
        this.ad = ad;  
        // Setter metodu ile yaşı doğrulaması yapılmıştır  
        setYasi(yas); // Constructor içinde de setter çağırılabilir  
        this.ogrenciNo = ogrenciNo;  
    }  
  
    // 2. 'public' Getter Metotları - Niteliklerin değerlerini okumak için  
    public String getAd() {  
        return ad;  
    }  
  
    public int getYas() {  
        return yas;  
    }  
  
    public String getOgrenciNo() {  
        return ogrenciNo;  
    }  
  
    // 3. 'public' Setter Metotları - Niteliklerin değerlerini güvenli bir şekilde değiştirmek için  
    public void setAd(String ad) {  
        // İsteğe bağlı olarak burada veri doğrulama yapılabilir  
        if (ad != null && ad.trim().isEmpty()) {  
            this.ad = ad;  
        } else {  
            System.out.println("Hata: Ad boş olamaz!");  
        }  
    }  
  
    public void setYasi(int yas) {  
        // Yaş için veri doğrulama Örneği: Yaş 0 ile 120 arasında olmalı  
        if (yas > 0 && yas <= 120) {  
            this.yas = yas;  
        } else {  
            System.out.println("Hata: Yaş 0 ile 120 arasında olmalıdır!");  
        }  
    }  
  
    // OgrenciNo genellikle değişmez, bu yüzden sadece getter ekleyebiliriz (setter eklemeyebiliriz)  
    // Eğer bir niteliğin sadece okunmasını istiyorsak, sadece getter ekleriz.  
    // Eğer dışarıdan erişimini tamamen engellemek istiyorsak, ne getter ne de setter eklemeyiz.  
}
```

Kullanım Örneği:

```
Java

public class Main {
    public static void main(String[] args) {
        Ogrenci ogrenci1 = new Ogrenci("Aysel Yilmaz", 20, "12345");

        // Veriye doğrudan erişim yok (çünkü 'private')
        // System.out.println(ogrenci1.ad); // Hata verir!

        // Getter metotları ile verilere güvenli erişim
        System.out.println("Ogrenci Adı: " + ogrenci1.getAd());
        System.out.println("Ogrenci Yasi: " + ogrenci1.getYas());

        // Setter metotları ile veriyi güvenli bir şekilde değiştirme
        ogrenci1.setYas(22); // Geçerli bir yaşı
        System.out.println("Yeni Yaşı: " + ogrenci1.getYas());

        ogrenci1.setYas(-5); // Geçersiz yaşı, setter hata mesajı verir ve yaşı deg
        System.out.println("Son Yaşı: " + ogrenci1.getYas()); // Hala 22
    }
}
```

Encapsulation, kodunuzu daha modüler, güvenli ve bakımı kolay hale getiren temel bir OOP prensibidir. Java'da genellikle getter ve setter metotları aracılığıyla uygulanır.

Main.java ProductManager.java Product.java

```
1  public class Product { 3 usages
2  	// attributes veya field
3  	// tüm bu attribute'ler public'tır ve her yerden erişilebilir
4  	// ama encapsulation gereği bunlar private yapılmıştır
5  	// int id;
6  	// String name;
7  	// String description;
8  	// double price;
9  	// int stockAmount;
10
11  // private sadece tanımlandığı sınıfta geçerlidir
12  private boolean onSale; no usages
13
14  // field'ları - işte böyle
15  private int _id; 1
16  private String _name;
17  private String _description;
18  private double _price;
19  private int _stockAmount;
20
21  public int getId() {
22      return _id;
23  }
24
25  // set edilecek de
26  public void setId(int id) {
27      _id = id; // türkçe注释
28      // yani burada
29  }
30
31 }
```

Show Context Actions

- 1 Paste ⌘V
- Copy / Paste Special ⌘C
- Column Selection Mode ⌘⌘B
- Find Usages ⌘F7
- Go To ⌘G
- Folding ⌘M
- Analyze ⌘A
- Rename... ⌘F6
- Refactor 1 >
- Generate... ⌘N
- Toggle Field Watchpoint
- Open In >
- Local History >
- Git >
- Compare with Clipboard
- Diagrams >
- Create Gist...

2 Encapsulate Fields...

- Migrate Packages and Classes >
- Invert Boolean...

16:23 LF UI

→ field'ları
encapsulation
metodlarını böyle
jepole kümle bu
genellikle kullanılır

• Constructor •

Japıca metodlardır. Normalde biz jayzzale deki her class'in arkasında otomatik olusuturulan bir constructor methodu vardır. Eğer bu methodu biz kendimiz jayzzale bu methodu özellestirebiliriz.

public class ile formülün \rightarrow class'tan nesne üretildiği esnabla
constructor fonsiyonu cuişer

3

parametresiz constructor'dır

Java'da Constructor Nedir?

Java'da constructor (yapıcı metot), bir sınıfın yeni bir obje (nesne) oluşturulduğunda otomatik olarak çağrılan özel bir metot türüdür. Temel görevi, objenin niteliklerini (alanlarını) başlangıç değerleriyle başlatmak ve objenin kullanıma hazır olmasını sağlamak.

Constructor'lar, bir evin inşaat planındaki temel atma töreni gibidir. Bir ev inşa edilmeden önce temelinin sağlam atılması gerekti gibi; bir obje de kullanılmadan önce doğru şekilde başlatılmışmalıdır.

Constructor'ların Temel Özellikleri

- Sınıfı Aynı Ada Sahip Olmalıdır:** Constructor'ın adı, ait olduğu sınıfın adıyla tamamen aynı olmak zorundadır (büyük/küçük harf duyarlılığı dahil).
- Geri Dönüş Tipi Yoktur:** Constructor'ların `void`, `int`, `String` gibi hiçbir geri dönüş tipi olmaz. Bu, onları normal metodlardan ayıran en önemli özelliklerden biridir.
- new Anahtar Kelimesiyle Çağırılır:** Bir constructor'u doğrudan çağrılmazsınız. Bunun yerine, bir sınıfın yeni bir objesini oluşturmak için `new` anahtar kelimesini kullandığınızda, Java sanal makinesi (JVM) otomatik olarak uygun constructor'u çağırır.

```
Java
SinifAdi objeAdi = new SinifAdi(); // Burada constructor çağrılır
```

- Aynı Kişiye (Overloading) Yapılabilir:** Bir sınıfta birden fazla constructor tanımlayabilirsiniz, ancak her birinin farklı parametre listelerine (farklı sayıda veya farklı tipte parametreler) sahip olması gereklidir. Bu **constructor overloading** denir. Bu, objeleri farklı yollarla başlatma esnekliğini sağlar.

- Varsayılan Constructor (Default Constructor):** Eğer bir sınıfta hiçbir constructor tanımlanmasa da, Java derleyicisi otomatik olarak **parametresiz (no-arg)** bir **varsayılan constructor** oluşturur. Bu constructor'un hiçbir parametresi yoktur ve içinde herhangi bir kod bulunmaz (objenin niteliklerine varsayılan değerlerini atar). Ancak siz herhangi bir constructor tanımladığınız anda, derleyici varsayılan constructor'u otomatik olarak oluşturmayı durdurur.

Constructor Kullanım Amaçları

- Objeyi Başlatma:** Bir objenin alanlarına başlangıç değerlerini atamak.
- Objenin Durumunu Ayarlama:** Objeyi mantıksal olarak geçerli bir duruma getirmek. Örneğin, bir `Hesap` objesi oluşturulurken bakiye `0` olarak ayarlanabilir.
- Bağımlılıkları Enjekte Etmeye:** Bir objenin çalışması için ihtiyaç duyduğu diğer objeleri (bağımlılıkları) constructor aracılığıyla almak.

Constructor Örnekleri

Aşağıda, bir `Kitap` sınıfı için farklı constructor türlerinin nasıl tanımlandığını görebilirsiniz:

```

public class Kitap {
    String baslik;
    String yazar;
    int sayfaSayisi;
    boolean yayindaMi; // Yeni eklandı

    // 1. Parametresiz Constructor (No-arg Constructor)
    // Eğer başka hiçbir constructor yazılmazsa, Java bunu otomatik ekler.
    // Ancak parametreli constructor yazıldığında, bunu yazmak gereklidir.
    public Kitap() {
        this.baslik = "Başlıksız";
        this.yazar = "Bilinmiyor";
        this.sayfaSayisi = 0;
        this.yayindaMi = false; // Varsayılan değer
        System.out.println("Parametresiz Kitap constructor'ı çağrıldı.");
    }

    // 2. Parametreli Constructor (Tüm nitelikleri alan)
    public Kitap(String baslik, String yazar, int sayfaSayisi) {
        this.baslik = baslik; // Gelen değeri niteliğe ata
        this.yazar = yazar;
        this.sayfaSayisi = sayfaSayisi;
        this.yayindaMi = true; // Varsayılan olarak yayinlandığını varsayıyalım
        System.out.println("3 parametreli Kitap constructor'ı çağrıldı.");
    }

    // 3. Parametreli Constructor (Başlık ve yazarı alan - Constructor Overloading)
    public Kitap(String baslik, String yazar) {
        this.baslik = baslik;
        this.yazar = yazar;
        this.sayfaSayisi = 0; // Varsayılan değer
        this.yayindaMi = false;
        System.out.println("2 parametreli Kitap constructor'ı çağrıldı.");
    }

    // Kitap bilgilerini yazdırma için bir metot
    public void kitapBilgileriniYazdir() {
        System.out.println("Başlık: " + baslik + ", Yazar: " + yazar + ", Sayfa Sayısı: " + sayfaSayisi + ", Yayınlı mı: " + yayindaMi);
    }
}

```

Constructor'ları Kullanma

Main metodunda bu constructor'ları nasıl çağrıcağıza bakalım:

```

Java

public class Main {
    public static void main(String[] args) {
        // Parametresiz constructor ile obje oluşturma
        Kitap kitap1 = new Kitap();
        kitap1.kitapBilgileriniYazdir();
        // Çıktı: Parametresiz Kitap constructor'ı çağrıldı.
        // Başlık: Başlıksız, Yazar: Bilinmiyor, Sayfa Sayısı: 0, Yayınlı mı: false

        System.out.println("----");

        // 3 parametreli constructor ile obje oluşturma
        Kitap kitap2 = new Kitap("Java Programlama", "Ali Can", 500);
        kitap2.kitapBilgileriniYazdir();
        // Çıktı: 3 parametreli Kitap constructor'ı çağrıldı.
        // Başlık: Java Programlama, Yazar: Ali Can, Sayfa Sayısı: 500, Yayınlı mı: true

        System.out.println("----");

        // 2 parametreli constructor ile obje oluşturma
        Kitap kitap3 = new Kitap("Veri Yapıları", "Ayşe Demir");
        kitap3.kitapBilgileriniYazdir();
        // Çıktı: 2 parametreli Kitap constructor'ı çağrıldı.
        // Başlık: Veri Yapıları, Yazar: Ayşe Demir, Sayfa Sayısı: 0, Yayınlı mı: false
    }
}

```

this() Anahtar Kelimesi

Bir constructor'dan, aynı sınıftaki başka bir constructor'ı çağırma için `this()` anahtar kelimesini kullanabilirsiniz. Bu, kodun temiz constructor'lar yazmaya yardımcı olur.

Java

```
public class Kitap {  
    String baslik;  
    String yazar;  
    int sayfaSayisi;  
    boolean yayindaMi;  
  
    public Kitap() {  
        this("Bogusluk", "Bilinmiyor", 0); // Diğer constructor'ı çağır  
        this.yayindaMi = false; // Farklı olan niteliği burada ayırala  
        System.out.println("Parametresiz Kitap constructor'ı çağrıldı.");  
    }  
  
    public Kitap(String baslik, String yazar, int sayfaSayisi) {  
        this.baslik = baslik;  
        this.yazar = yazar;  
        this.sayfaSayisi = sayfaSayisi;  
        this.yayindaMi = true; // Varsayılan olarak yayında  
        System.out.println("3 parametreli Kitap constructor'ı çağrıldı.");  
    }  
  
    public Kitap(String baslik, String yazar) {  
        this(baslik, yazar, 0); // 3 parametreli constructor'ı çağır  
        System.out.println("2 parametreli Kitap constructor'ı çağrıldı.");  
    }  
  
    // ... (kitapBilgileriniYazdır metodunu aynı kalır)  
}
```

Yukarıdaki örnekte, parametresiz constructor ve 2 parametreli constructor, işin çoğunu yapan 3 parametreli constructor'ı çağrıyor. Bu, constructor mantığını merkezi hale getirerek kodun daha düzenli olmasını sağlar.

Constructor'lar, Java'da nesne oluşturanın ve başlatmanın ayrılmaz bir parçasıdır. Objelerinizin her zaman geçerli bir durumda olmasını sağlamak için çok önemlidirler.

Main.java x Dortslem.java

```
1 ▷ public class Main {  
2 ▷     public static void main(String[] args) {  
3         Dortslem dortslem = new Dortslem();  
4  
5         // method overloading  
6         1System.out.println(dortslem.topla(3,5));  
7         2System.out.println(dortslem.topla(3, 5.5));  
8         3System.out.println(dortslem.topla(3,5,4));  
9     }  
10
```

```
>Main.java @ Dortslem.java x  
1 public class Dortslem { 2 usages  
2     1 public int topla(int sayi1, int sayi2){ 1 usage  
3         return sayi1 + sayi2;  
4     }  
5  
6     2 public double topla(int sayi1, double sayi2){ 1 usage  
7         return sayi1 + sayi2;  
8     }  
9  
10    3 public int topla(int sayi1, int sayi2, int sayi3){ 1 usage  
11        return sayi1 + sayi2;  
12    }
```

→ overloading mantığı

Inheritance

Miras / Kalıtım absolütür Terelli sınıfların kalıtımıne dayanır. **Classların,**

object'lerin birbirinden kalıtım altosunu sağlar.

public class Customer extends Person {

3
Customer class'ı Person class'ından miras/kalıtım alır. Jni → Customer : ata'dan miras alan sınıf
→ Person : ata sınıfı

extends anahtar kelimesi ile inheritance kullanılır

Java'da Inheritance (Miras Alma) Nedir?

Java'da Inheritance (miras alma), nesne yönelimli programlanmanın (OOP) temel direktelerinden biridir. Bir sınıfın başka bir sınıfın özelliklerini (niteliklerini) ve davranışlarını (metotlarını) almasını sağlayan bir mekanizmadır. Bu, kodun yeniden kullanılabilirliğini (reusability) artırır ve sınıflar arasında hiyerarşik bir ilişkii kurar.

Kısaca, "bir şeyin, başka bir şeyin genişletilmiş hali olması" olarak düşünebilirsiniz. Örneğin, bir "Araba" sınıfı bir "Taşıt" sınıfından miras alabilir, çünkü araba bir taşıttır ve taşıtan genel özelliklerine (hız, tekerlek sayısı gibi) ek olarak kendine özgü özelliklere (marka, model gibi) sahiptir.

Inheritance'in Temel Kavramları

1. Ana Sınıf (Superclass / Parent Class / Base Class):

- Özelliklerini ve metodlarını diğer sınıflara veren sınıfır.
- 2. Alt Sınıf (Subclass / Child Class / Derived Class):
 - Ana sınıfın özellikler ve metodları alan sınıfır. Alt sınıf, ana sınıfın tüm `public` ve `protected` nitelik ve metodlarına erişebilir (veya onları kendi bünyesine dahil eder). `private` niteliklere doğrudan erişemez ama genellikle `public` getter/setter metodları aracılığıyla dolaylı olarak erişebilir.
 - Alt sınıf, miras aldığı özelliklere ek olarak kendi benzerlik özelliklerini ve metodlarını da tanımlayabilir.

Inheritance Nasıl Uygulanır? (`extends` Anahtar Kelimesi)

Java'da inheritance, `extends` anahtar kelimesi kullanılarak gerçekleştirilir.

```
Java

class AnaSınıf {
    // Ana sınıfın özellikleri ve metodları
}

class AltSınıf extends AnaSınıf {
    // Alt sınıfın kendine özgü özellikleri ve metodları
    // Aynı zamanda AnaSınıf'ın public/protected özelliklerine ve metodlarına da erişebilir
}
```

Inheritance'in Faydaları

1. Kod Tekrarını Azaltma (Code Reusability):

- Ortak özellikler ve davranışları bir ana sınıfta tanımlarsınız ve bu özelliklerin paylaşılan tüm alt sınıfları onları miras alır. Bu, aynı kodu defalarca yazmaktan kurtarır.
- Örnek: Tüm hayvanların bir `yemeKategori()` metodlu varsa, bunu `Hayvani` ana sınıfında tanımlayıp `Kedi`, `Köpek`, `Kuş` gibi alt sınıflar miras alması sağlanabilirsiniz.

2. Genişletilebilirlik (Extensibility):

- Mevcut kod yapısını değiştirmeden yeni özellikler eklemeye veya davranışları değiştirmeye kolaylaştırır. Yeni bir alt sınıf oluşturarak mevcut ana sınıfın işlevsellğini genileyebilirsiniz.

3. Polymorphism'ı Müküm Klar:

- Inheritance, polymorphism (çok biçimlilik) için temel oluşturur. Bir ana sınıf referansı, alt sınıfların objelerini tutabılır ve bu, esnek ve modüler kod yazmaya olanak tanır.

4. Bakım Kolaylığı:

- Ortak kod tek yerde (ana sınıfda) bulunduğu için, değişiklikler veya hata düzeltmeleri tek bir yerden yapılabilir ve tüm alt sınıflara yansır.

Inheritance Orneği

Bir `Hayvan` sınıfı oluşturulmuş ve ondan miras alan `Kedi` ve `Köpek` sınıflarını tanımlayalım:

```
// Ana Sınıf (Superclass)
class Hayvan {
    String isim; // Ortak özellik

    public Hayvan(String isim) {
        this.isim = isim;
    }

    public void sesCikar() { // Ortak davranış
        System.out.println("Hayvan ses çıkarır.");
    }

    public void yemekYe() { // Ortak davranış
        System.out.println(isim + " yemek yiyor.");
    }
}

// Alt Sınıf (Subclass) - Hayvan sınıfından miras alır
class Kedi extends Hayvan {
    public Kedi(String isim) {
        super(isim); // Ana sınıfın constructor'ını çağırır
    }

    @Override // Metodu geçersiz kılma (override)
    public void sesCikar() {
        System.out.println(isim + " Miyavlar.");
    }

    public void tirmala() { // Kediye özgü davranış
        System.out.println(isim + " tirmalıyor.");
    }
}

// Alt Sınıf (Subclass) - Hayvan sınıfından miras alır
class Köpek extends Hayvan {
    public Köpek(String isim) {
        super(isim); // Ana sınıfın constructor'ını çağırır
    }
}
```

```
@Override // Metodu geçersiz kılma (override)
public void sesCikar() {
    System.out.println(isim + " Hayvan.");
}

public void kuyrukSalla() { // Köpeğe özgü davranış
    System.out.println(isim + " kuyruk sallıyor.");
}
```

Kullanım Örneği:

```
Java

public class Main {
    public static void main(String[] args) {
        Hayvan genelHayvan = new Hayvan("Genel Hayvan");
        genelHayvan.sesCikar(); // Çıktı: Hayvan ses çıkarır.
        genelHayvan.yemekYe(); // Çıktı: Genel Hayvan yemek yiyor.

        System.out.println("----");

        Kedi tekir = new Kedi("Tekir");
        tekir.sesCikar(); // Çıktı: Tekir Miyavlar. (Override edilmiş metot)
        tekir.yemekYe(); // Çıktı: Kediye özgü davranış. (Miras alınan metot)
        tekir.tirmala(); // Çıktı: Tekir tirmalıyor. (Kediye özgü metot)

        System.out.println("----");

        Köpek karabas = new Köpek("Karabas");
        karabas.sesCikar(); // Çıktı: Karabas Hayvan. (Override edilmiş metot)
        karabas.yemekYe(); // Çıktı: Karabas yemek yiyor. (Miras alınan metot)
        karabas.kuyrukSalla(); // Çıktı: Karabas kuyruk sallıyor. (Köpeğe özgü metot)
    }
}
```

• Jerarhon geliştirmeli en önemli oğu değişirdir

Bizler bir insan projesi gibi gerekli her şeyi olası olsun da doğrularız. İşler yapmamızdağız.

Bire gebze şartının doğrulanması gereklidir.

- **Tekli Miras (Single Inheritance):** Java'da bir sınıf yalnızca tek bir sınıfın miras alabilir.
- Yani bir sınıfın birden fazla doğrudan ana sınıfı olamaz.

- **Coklu Miras (Multiple Inheritance)** Yok: Java, "elmas problemi" gibi karmaşıklıklar onlemek için doğrudan çoklu miras deseti sumuz. Ancak bu ekşilik, arayüzler (**Interfaces**) aracılığıyla giderilebilir.
- **final** Sınıflar: Bir sınıf **final** olarak tanımlanırsa, bu sınıftan başka hiçbir sınıf miras alamaz.
- **Constructor Mirası Yok:** Constructor'lar miras almaz. Alt sınıfın, kendi constructor'ları içinde **super()** anahtar kelimesi ile ana sınıfın constructor'ını çağrılabılır.
- **Object** Sınıfı: Java'daki tüm sınıflar doğrudan veya dolaylı olarak **java.lang.Object** sınıfından miras alır. Bu, **Object** sınıflının en üst seviye hierarşik sınıf olduğu anlaşılmıştır.

Inheritance, kodunuzu daha modüler, genisletilebilir ve yeniden kullanabilir hale getirerek büyük ve karmaşık uygulamaların yönetimini kolaylaştırır.

// Polymorphism

Java'da Polymorphism (Çok Biçimlilik) Nedir?

Java'da Polymorphism (Çok Biçimlilik), Nesne Yönelimli Programmanın (OOP) dört temel prensibinden biridir. Kelime anlamı olarak "çok biçimli olma" demektir. Programlamada ise, bir objenin farklı formlarda davranışabilme yeteneğini veya bir eylemin farklı şekillerde gerçekleştirilebilmesini ifade eder.

Basitçe anlatmak gerekirse, aynı isimde bir metodun farklı sınıflarda farklı davranışlar sergilemesidir. Bu sayede, aynı tipte bir referans değişkeni (genellikle ana sınıf veya arayüz tipi), farklı alt sınıfların objelerini tutabılır ve bu objeler üzerinde aynı metod çağırıldığında, objenin gerçek tipine göre farklı bir davranış gözlemlersiniz.

Polymorphism'in Temel Prensipleri

Polymorphism iki ana mekanizma ile sağlanır:

1. Metot Geçersiz Kılma (Method Overriding - Runtime Polymorphism):

- Bu, bir alt sınıfın, ana sınıfından miras aldığı bir metodу kendi ihtiyaçına göre yeniden tanımlamasıdır. Yani, metod imzası (adi, parametreleri, dönüşüm tipi) aynı kalır, ancak metod içeriği (gövdesi) değişir.
- Çalışma zamanında (runtime) hangi metotun çağrılacagına objenin **gerçek tipi** (instance tipi) karar verir. Bu yüzden buna **Runtime Polymorphism** denir.
- Örneğin, bir `Hayvan` sınıfında `sesCikar()` metodunu varken, `Kedi` sınıfı bunu "Miyavlar" olarak, `Köpek` sınıfı ise "Havalar" olarak geçersiz kılabılır.

2. Metot Aşırı Yüklenme (Method Overloading - Compile-time Polymorphism):

- Bu, aynı sınıf içinde, **aynı isme sahip ancak farklı parametre listelerine sahip** birden fazla metod tanımlanmasıdır. Farklı parametre listesi, farklı sayıda parametre, farklı tipte parametreler veya parametrelerin farklı sırası anlamına gelebilir.

- Derleme zamanında (compile-time) hangi metotun çağrılacagına parametre listesine bakılarak karar verilir. Bu yüzden buna **Compile-time Polymorphism** veya **Static Polymorphism** denir.
- Örneğin, bir `Toplama` sınıfında `topla(int a, int b)` ve `topla(double a, double b)` gibi iki farklı metod olabilir.

Polymorphism'in Faydaları

1. Esneklik ve Genişletilebilirlik:

- Yeni alt sınıflar ekleyerek kodunuzu kolayca genişletebilirsiniz. Mevcut kodu değiştirmeden yeni davranışlar tanımlayabilirsiniz.

2. Kod Tekrarını Azaltma:

- Ortak davranışları ana sınıfı tanımlanır ve alt sınıflar tarafından özelleştirilebilir. Bu, gereksiz kod tekrarını önlüyor.

3. Daha Okunabilir ve Anlaşılır Kod:

- Genel bir arayüz kullanarak (örneğin, `Hayvan` tipi bir değişkenle) farklı objeler üzerinde işlem yapabilirsiniz. Bu, kodunuzu daha modüler ve anlaşılır hale getirir.

4. Bakım Kolaylığı:

- Bir ana sınıfın veya arayüzünün davranışında yapılan bir değişiklik, ondan türetilen tüm sınıfları etkileyebilir. Bu, kodun bakımını kolaylaştırır.

Runtime Polymorphism (Metot Geçersiz Kılma) Örneği:

Daha önceki Inheritance örneğimizi polymorphism bağlamında inceleyelim:

. Bir sınıfı başta bir class içinde new ile oluşturmak, o ibi class'ı birbirine bağlı hale getirmek onolutor.

. Javi'de default olane child class, parent class'i ezbilir jori override olgi default olane vor.

Eger bunu ölümden istesek jori methodumuzu herkes için gen jgoip bunu override edilgece koyduktan istesek final anahtar kelimesini Tercih ediyiz

(final) var → override jole ise

→ method overriding

Method overriding

```
// Ana Sınıf (Superclass)
class Hayvan {
    String isim;

    public Hayvan(String isim) {
        this.isim = isim;
    }

    public void sesCikar() { // Ortak davranış
        System.out.println("Hayvan ses çıkarır.");
    }
}

// Alt Sınıf (Subclass) - Hayvan sınıfından miras alır
class Kedi extends Hayvan {
    public Kedi(String isim) {
        super(isim);
    }

    @Override // sesCikar metodunu geçersiz kılıyoruz
    public void sesCikar() {
        System.out.println(isim + " Miyavlar.");
    }
}

// Alt Sınıf (Subclass) - Hayvan sınıfından miras alır
class Kopek extends Hayvan {
    public Kopek(String isim) {
        super(isim);
    }

    @Override // sesCikar metodunu geçersiz kılıyoruz
    public void System.out.println("Merhaba, ben AI.");
    }

    public void sesCikar() {
        System.out.println(isim + " Havalar.");
    }
}
```

Polymorphism Kullanımı:

Java

```
public class Main {
    public static void main(String[] args) {
        // Ana sınıf tipinde referans değişkenleri oluşturuyoruz,
        // ancak bunlara alt sınıf objeleri atıyoruz.
        Hayvan hayvan1 = new Kedi("Pamuk");
        Hayvan hayvan2 = new Kopek("Kont");
        Hayvan hayvan3 = new Hayvan("Leo"); // Ana sınıfın kendisi de olabilir

        // Aynı metodu (sesCikar()) çağrıryoruz,
        // ancak objenin gerçek tipine göre farklı davranışlar alıyor.
        hayvan1.sesCikar(); // Çıktı: Pamuk Miyavlar.
        hayvan2.sesCikar(); // Çıktı: Kont Havalar.
        hayvan3.sesCikar(); // Çıktı: Hayvan ses çıkarır.

        System.out.println("----");

        // Bir dizi veya koleksiyon içinde polymorphism kullanımı:
        Hayvan[] hayvanlar = new Hayvan[3];
        hayvanlar[0] = new Kedi("Minnoş");
        hayvanlar[1] = new Kopek("Zeytin");
        hayvanlar[2] = new Hayvan("Fino");

        for (Hayvan h : hayvanlar) {
            h.sesCikar(); // Her objenin kendi sesCikar() metodu çağrılır
        }
        // Çıktı:
        // Minnoş Miyavlar.
        // Zeytin Havalar.
        // Hayvan ses çıkarır.
    }
}
```

Compile-time Polymorphism (Metot Aşırı Yükleme) Örneği

Java

method over loading

```
class Hesaplayici {  
    // 1. İki int sayısını toplama  
    public int topla(int a, int b) {  
        System.out.println("int + int toplama yapıldı.");  
        return a + b;  
    }  
  
    // 2. İki double sayısını toplama (aynı metot adı, farklı parametre tipleri)  
    public double topla(double a, double b) {  
        System.out.println("double + double toplama yapıldı.");  
        return a + b;  
    }  
  
    // 3. Üç int sayısını toplama (aynı metot adı, farklı sayıda parametre)  
    public int topla(int a, int b, int c) {  
        System.out.println("int + int + int toplama yapıldı.");  
        return a + b + c;  
    }  
}  
  
public class MainOverloading {  
    public static void main(String[] args) {  
        Hesaplayici h = new Hesaplayici();  
  
        // Derleyici, parametre tiplerine ve sayısına göre hangi 'topla' metodunu  
        // seçtiğini yazdırıyor.  
        System.out.println(h.topla(5, 10));           // Çıktı: int + int toplama yapıldı.  
        System.out.println(h.topla(5.5, 10.5));       // Çıktı: double + double toplama yapıldı.  
        System.out.println(h.topla(1, 2, 3));         // Çıktı: int + int + int toplama yapıldı.  
    }  
}
```

• bir sınıf hem normal bir sınıfı handsome abstract bir sınıfı miras alır

// Abstract class

Java'da Abstract Class (Soyut Sınıf) Nedir?

Java'da **abstract class** (soyut sınıf), doğrudan nesnesi oluşturulamayan, diğer sınıflar tarafından miras alınmak (genişletilmek) üzere tasarlanmış özel bir sınıfır. Abstract sınıflar, genellikle ilişkili sınıflar için ortak bir temel sağlar ve belirli metodların alt sınıflar tarafından mutlaka uygulanmasını (implemente edilmesini) zorunlu kılar.

Bir abstract sınıfı, bir plan veya taslağı gibi düşünebilirsiniz; bu planın kendisi tam bir ürün değildir, ancak ondan türetilen ürünler için bir çerçeveye çizer.

Abstract Class'ların Temel Özellikleri

1. **abstract Anahtar Kelimesi:** Bir sınıfı soyut yapmak için `class` anahtar kelimesinin önüne `abstract` anahtar kelimesi eklenir:

Java

method over loading

```
public abstract class BenimSoyutSinifim {  
    // ...  
}
```

2. **Nesnesi Oluşturulamaz:** Abstract bir sınıfın doğrudan bir objesini (`new BenimSoyutSinifim()`) oluşturamazsınız. Bir abstract sınıfı kullanmak için, ondan miras alan (extending) somut (concrete) bir alt sınıf oluşturmanız ve bu alt sınıfın objesini yaratmanız gereklidir.

3. Hem Abstract Hem de Somut (Concrete) Metotlar içerebilir:

- Abstract Metotlar: Gövdesi olmayan (implementasyonu olmayan) metotlardır. Sadece imzaları (adi, parametreleri, dönüş tipi) bulunur ve sonuna noktalı virgül (;) konur. Bu metotlar, `abstract` anahtar kelimesiyle tanımlanır.

Java

```
public abstract void soyutMetot(); // Gövdesi yok
```

Bu tür metotlar, alt sınıflar tarafından **zorunlu olarak override edilmeli** ve uygulanmalıdır.

- Somut Metotlar: Normal metotlar gibi gövdesi olan (implementasyonu olan) metotlardır.

Java

```
public void somutMetot() {  
    System.out.println("Bu somut bir metottur.");  
}
```

Bu metotlar alt sınıflar tarafından miras alır ve isteğe bağlı olarak override edilebilir.

4. Nitelikler (Fields) ve Constructor'lar içerebilir:

- Abstract sınıflar niteliklere (değişkenlere) ve constructor'lara sahip olabilirler. Constructor'lar abstract sınıfın kendisinin nesnesi oluşturulmadığı için doğrudan çağrılamaz, ancak alt sınıfların constructor'larından `super()` anahtar kelimesi ile çağrılırlar ana sınıf niteliklerinin başlatılmasına kullanılır.

5. `final` Metotlar içerebilir: Abstract bir sınıf, `final` metotlar içerebilir. `final` metotlar, alt sınıflar tarafından geçersiz kılınamaz (`override` edilemez).

6. `private` Metotlar içerebilir: Abstract bir sınıf, `private` metotlar içerebilir. Bu metotlar sadece kendi sınıfı içinde kullanılabilir ve alt sınıflar tarafından erişilemez veya override edilemez.

Neden Abstract Class Kullanılır? (Kullanım Amaçları)

1. Ortak Davranışları Tanımlama:

- Bir grup ilişkili sınıf için ortak olan nitelikleri ve somut metodları tek bir yerde tanımlayarak kod tekrarını önlüyor.

• Örnək: `Sekil` adında bir abstract sınıf, `alanHesapla()` adında abstract bir metod ve `isimGetir()` adında somut bir metod içerebilir. Tüm şekillerin bir alanı ve ismi vardır, ancak alan hesaplama her şekli için farklıdır.

2. Şablon Metot Oluşturma:

- Belli bir algoritmanın adımlarını (şablonunu) tanımlamak için kullanılır. Bazı adımlar somut (tüm alt sınıflarda aynı), bazıları ise soyut (alt sınıflar tarafından uygulanması gereken) olabilir. Bu, Tasarım Desenleri'ndeki "Şablon Metot (Template Method)" deseninin temelidir.

3. Alt Sınıflara Uygulama Zorunluluğu Getirme:

- Abstract metodlar sayesinde, bu metodları miras alan tüm somut alt sınıfların, bu metodları mutlaka implement etmesini zorunlu kılar. Bu, belirli bir davranışın tüm türəv sınıflarda mevcut olmasını garanti eder.

4. Polymorphism (Çok Biçimlilik) İçin Temel Sağlama:

- Abstract sınıfın, polymorphism için güçlü bir temel sağlar. Bir abstract sınıf tipinde bir referans değişkeni kullanarak, ondan türetilen farklı alt sınıf objelerini tutabilir ve aynı metodları çağırarak farklı davranışlar elde edebilirsiniz.

Abstract Class Örneği

Bir `Calisan` (çalışan) abstract sınıfı oluşturulalım. Tüm çalışanların bir adı ve maaş hesaplama metodu olsun, ancak maaş hesaplama şekli çalışan türüne göre değişim.

Java

```
// Abstract Sınıf
public abstract class Calisan {
    private String ad;
    private int id; // Çalışan kimliği

    public Calisan(String ad, int id) {
        this.ad = ad;
        this.id = id;
    }

    // Somut metot: Tüm çalışanlar için aynı
    public String getAd() {
        return ad;
    }

    public int getId() {
        return id;
    }

    // Abstract metot: Maas hesaplama her çalışan türü için farklı olacağı için so
    // Alt sınıflar bu metodun zorunlu olarak implement etmeli.
    public abstract double maasHesapla();

    // Somut bir metot da:
    public void bilgilerGoster() {
        System.out.println("Çalışan Adı: " + ad);
        System.out.println("Çalışan ID: " + id);
    }
}

// Somut Alt Sınıf 1
class MaaşlıCalisan extends Calisan {
    private double saatlikUcret;
    private int calismaSaatı;

    public MaaşlıCalisan(String ad, int id, double saatlikUcret, int calismaSaatı)
        super(ad, id); // Ana sınıfın constructor'ını çağır
    this.saatlikUcret = saatlikUcret;
    this.calismaSaatı = calismaSaatı;
}

@Override // Abstract metodu implement etmek zorunlu
public double maasHesapla() {
    return saatlikUcret * calismaSaatı;
}
```

```
// Somut Alt Sınıf 2
class Yonetici extends Calisan {
    private double aylıkMaas;
    private double bonus;

    public Yonetici(String ad, int id, double aylıkMaas, double bonus) {
        super(ad, id); // Ana sınıfın constructor'ını çağır
        this/aylıkMaas = aylıkMaas;
        this.bonus = bonus;
    }

    @Override // Abstract metodu implement etmek zorunlu
    public double maasHesapla() {
        return aylıkMaas + bonus;
    }
}
```

Kullanım Örneği:

→ abstract class

→ extended from an Abstract class

Kullanım Örneği:

Java

```
public class Main {
    public static void main(String[] args) {
        // Calisan abstract sınıfından doğrudan nesne oluşturamayız:
        // Calisan c = new Calisan("Test", 1); // Hata verir!

        // Alt sınıfların nesnelerini oluşturuyoruz
        MaaşlıCalisan calisan = new MaaşlıCalisan("Ahmet Yılmaz", 101, 50.0, 160);
        Yonetici yonetici = new Yonetici("Ayşe Demir", 201, 7000.0, 1500.0);

        // Polymorphism kullanarak abstract sınıf referansı ile objeleri tutabiliyor:
        Calisan[] tumCalisanlar = new Calisan[2];
        tumCalisanlar[0] = calisan;
        tumCalisanlar[1] = yonetici;

        for (Calisan c : tumCalisanlar) {
            c.bilgilerGoster(); // Somut metot çağırıldı
            System.out.println("Maas: " + c.maasHesapla()); // Polymorphic metod çıktı
            System.out.println("----");
        }
        /* Çıktı:
        Çalışan Adı: Ahmet Yılmaz
        Çalışan ID: 101
        Maas: 8000.0
        ----
        Çalışan Adı: Ayşe Demir
        Çalışan ID: 201
        Maas: 8500.0
        ----
        */
    }
}
```

Abstract Class ve Interface Arasındaki Fark

Hem abstract class'lar hem de interface'ler (arayüzler) soyutlama sağlasa da, önemli farkları vardır:

Özellik	Abstract Class	Interface
Nesne Oluşturma	Doğrudan nesnesi oluşturulamaz.	Doğrudan nesnesi oluşturulamaz.
Metotlar	Hem soyut (<code>abstract</code>) hem de somut (<code>concrete</code>) metotlar içerebilir.	Java 8 öncesi sadece soyut metotlar. Java 8+ ile <code>default</code> ve <code>static</code> somut metotlar içerebilir.
Nitelikler	Normal nitelikler içerebilir. (<code>private</code> , <code>protected</code> , <code>public</code> gibi erişim belirleyicileryle)	Sadece <code>public static final</code> nitelikler içerebilir (sabitler).
<code>extends / implements</code>	Bir sınıf sadece tek bir abstract sınıf <code>extends</code> edebilir.	Bir sınıf birden fazla interface'i <code>implements</code> edebilir.
Constructor	Constructor'u olabilir.	Constructor'u olamaz.
Erişim Belirleyiciler	Üyeler için herhangi bir erişim belirleyici kullanabilir.	Varsayılan olarak tüm metotları <code>public abstract</code> , tüm alanları <code>public static final</code> dir.

[Export to Sheets](#)

Kısacası, bir sınıf hiyerarşisinde ortak davranış ve temel implementasyon sağlamak istiyorsanız **abstract class** kullanırsınız. Tamamen bir "sözleşme" veya "davranışsal yetenek" tanımlamak istiyorsanız **interface** kullanırsınız.

```
④ GameCalculator.java × ⑤ ManGameCalculator.java ⑥ WomanGameCalculator.java
1 @  public abstract class GameCalculator { 3 usages 3 inheritors new* 3 related problems
2
3     // the class which get inheritance from GameCalculator, must use
4     // this function and must override this abstract method
5     public abstract void hesapla(); no usages new*
6
7     // final --> i don't want override
8     // so all class must use this if get inheritance from GameCalculator
9     public final void gameOver() { no usages new*
10         System.out.println("Game Over!");
11     }
12 }
13
```

// Interface

- class yapısı degildir
- inheritance olurak kabul edilmekter, implementation olurak kabul edilirler
- Interface istenilenlerinde istenilenme basına İ harfi getirilir

I Customer

Java'da Interface'ler (Arayüzler) Nedir?

Java'da **Interface (Arayüz)**, tamamen soyut metotlardan (Java 8 öncesi için) oluşan veya bir sınıfın davranışsal bir sözleşmesini tanımlayan bir referans tipidir. Bir interface, bir sınıfın ne yapması gerektiğini tanımlar, **nasıl yapacağını değil**. Yani, bir interface'i implement eden (uygulayan) bir sınıfın belirli metotları sağlamak sorundanız olduğunu garanti eder.

Bir interface'i, bir cihazın (örneğin bir televizyon) uzaktan kumandası gibi düşünebilirsiniz. Kumanda üzerindeki düğmeler ("ac/kapa", "kanal değiştir", "ses aç/kapa") televizyonun ne tür işlemleri olduğunu (davranışlarını) gösterir. Ancak bu düğmelerin arkasında televizyon içinde bu işlemlerin nasıl çalıştığını dair bir detay yoktur. Kumanda sadece bir "sözleşme"dir; bu düğmeler basıldığında bir şeyler olacağını garanti eder.

Interface'lerin Temel Özellikleri

- Anahtar Kelimesi:** Bir arayüzü tanımlamak için `interface` anahtar kelimesi kullanılır:

```
Java  
  
public interface BenimArayuzum {  
    // ...  
}
```

- Tamamen Soyutту (Java 8 Öncesi):**

- Java 8'den önce, bir interface'deki tüm metotlar otomatik olarak `public abstract` idi ve herhangi bir metot gövdesi (implementasyon) içeremezdi. Sadece metot imzalanır.

- Java 8 ve Sonrası:** `default` ve `static` metotlar ile interface'lere somut (implementasyonu olan) metotlar eklenebilir. Bu, geriye dönük uyumluluğu bozmadan interface'lere yeni işlevsellik eklemeye olanak tanır.
- private Metotlar (Java 9 ve Sonrası):** Java 9 ile birlikte `private` metotlar da interface'lere eklenebilir hale geldi. Bunlar, `default` metotların içinde kod tekrarını azaltmak için kullanılabilir.

- Tüm Nitelikler `public static final` dir:**

- Bir interface içinde tanımlanan tüm nitelikler (değişkenler) otomatik olarak `public static final`'dır (yani **sabitlerdir**). Bu anahtar kelimeleri açıkça yazmasanız bile, Java derleyicisi bunları otomatik olarak ekler. Sabitler, genellikle büyük harflerle yazırlar.

- Nesnesi Oluşturulamaz:** Abstract sınıflar gibi, interface'lerin doğrudan bir objesi (`new BenimArayuzum()`) oluşturulamaz.

- implements Anahtar Kelimesi:** Bir sınıf, bir veya daha fazla interface'i uygulamak (implement etmek) için `implements` anahtar kelimesini kullanır. Bir sınıf bir interface'i implement ediyorsa, interface'deki tüm **soyut metotları** override etmek ve kendi implementasyonunu sağlamak sorundadır.

Java

```
class SinifAdi implements ArayuzAdi {  
    // ArayuzAdi'daki tüm abstract metodları implement etmeli  
}
```

- Çoklu Miras Desteği:** Java'da bir sınıf sadece tek bir sınıfın miras alabilenken (tekli miras), **bir sınıf birden fazla interface'i implement edebilir**. Bu, Java'nın sınıflar için çoklu mirasın getirdiği "elmas problemi"ni aşma yoludur.

public class OracleCustomerBal
implements ICustomerBal

↓

ICustomerBal customerBal =

new OracleCustomerBal();

↓

Yeni interface, onu implemente eden class'in referansını tutabılır

• polymorphism'in en çok uygulandığı alanlardan bir tanesi de interface'lerdir

Neden Interface Kullanılır? (Kullanım Amaçları)

- Tam Soyutlama Sağlama:**

- Bir interface, bir grup sınıf için tamamen soyut bir davranış kümesi tanımlamanıza olanak tanır. Alt sınıfların davranışlarını kendi başlarına implement etmesini zorunlu kılar.

- Çoklu Davranış Mirası (Çoklu Mirasın Yerine):**

- Java'da sınıfın çoklu miras olmadığı için, bir sınıfın birden fazla kaynaktan davranış miras olmasını sağlamak için interface'ler kullanılır. Bu sayede bir sınıf birden fazla "türden" olabilir (örneğin, hem **Üzübilir** hem de **Yüzebilir** gibi).

- Gevşek Bağılılık (Loose Coupling):**

- Interface'ler, kod parçaları arasındaki bağımlılığı azaltır. Bir sınıf bir interface üzerinden kullanırsanız, o sınıfın iç implementasyon detaylarına bağlı olmazsınız. Bu, daha esnek ve bakımı kolay kod yazmaya yardımcı olur.

- Polymorphism (Çok Bireşimlilik) İçin Güçlü Bir Temel:**

- Bir interface tipinde referans değişkenleri tanımlayabilir ve bu referanslara interface'i implement eden farklı sınıfları objelerini atayabilirsiniz. Bu, kodu daha genel ve genişletilebilir hale getirir.

- API Tanımlama:**

- Genellikle, kütüphanelerin veya çerçevelerin (frameworks) API'leri (Uygulama Programı Arayüzleri) interface'ler kullanılarak tanımlanır. Bu, kullanıcıları belirli davranışları implement etmeleri için bir "sözleşme" sunar.

Interface Örneği

Bir **HareketEdebilir** interface'i tanımlayalım ve bunu **Araba** ve **Ucuk** sınıflarına implement edelim:

```
1 // Interface Tanımı:
2 public interface HareketEdebilir {
3     // Tüm metodlar (Java 8 öncesi) varsayılan olarak public abstract'tır.
4     // Explicit olarak yazmanız gerek yok.
5     void ilerle();
6     void dur();
7
8     // Java 8 sonrası: default metodlar
9     default void hızlan() {
10         System.out.println("Cısim hızlanıyor...");
11     }
12
13     // Java 8 sonrası: static metodlar
14     static void tümHareketEdenleriSelamlal() {
15         System.out.println("Tüm hareket eden varlıklar selamlanır!");
16     }
17 }
18
19 // Sınıf, HareketEdebilir arayüzüni implement eder
20 class Araba implements HareketEdebilir {
21     private String marka;
22
23     public Araba(String marka) {
24         this.marka = marka;
25     }
26
27     @Override // Interface'deki abstract metodları implement etmek zorunlu
28     public void ilerle() {
29         System.out.println(marka + " araba ileri gidiyor.");
30     }
31
32     @Override // Interface'deki abstract metodları implement etmek zorunlu
33     public void dur() {
34         System.out.println(marka + " araba durdu.");
35     }
36
37     // default metodu isteğe bağlı olarak override edebiliriz
38     @Override
39     public void hızlan() {
40         System.out.println(marka + " araba gaza basıyor ve hızlanıyor!");
41     }
42 }
43
44 // Başka bir sınıf, HareketEdebilir arayüzünü implement eder
45 class Uçak implements HareketEdebilir {
46     private String model;
47
48     public Uçak(String model) {
49         this.model = model;
50     }
51
52     @Override // Interface'deki abstract metodları implement etmek zorunlu
53     public void ilerle() {
54         System.out.println(model + " uçak havalandıyor.");
55     }
56
57     @Override // Interface'deki abstract metodları implement etmek zorunlu
58     public void dur() {
59         System.out.println(model + " uçak iniş yapıyor.");
60     }
61 }
```

```

1 public class Main {
2     public static void main(String[] args) {
3         // Interface tipinde referans değişkenleri oluşturuyoruz,
4         // bunlara implement eden sınıfların objelerini atıyoruz.
5         HareketEdebilir nesne1 = new Araba("Mercedes");
6         HareketEdebilir nesne2 = new Uçak("Boeing 747");
7
8         nesne1.ilerle(); // Çıktı: Mercedes araba ileri gidiyor,
9         nesne1.hizlan(); // Çıktı: Mercedes araba gaza basıyor ve hızlanıyor! (Override edilmiş default metot)
10        nesne1.dur(); // Çıktı: Mercedes araba durdu.
11
12        System.out.println("—");
13
14        nesne2.ilerle(); // Çıktı: Boeing 747 uçak havalandı.
15        nesne2.hizlan(); // Çıktı: Cisim hızlanıyor... (Default metot kullanıldı)
16        nesne2.dur(); // Çıktı: Boeing 747 uçak iniş yapıyor.
17
18        System.out.println("—");
19
20        // Static metodу interface üzerinden çağırabiliriz
21        HareketEdebilir.tumHareketEdenlerIseSelamlama(); // Çıktı: Tüm hareket eden varlıklar selamlanır!
22
23        System.out.println("—");
24
25        // Polymorphism'ın gücü:
26        HareketEdebilir[] hareketliler = new HareketEdebilir[2];
27        hareketliler[0] = new Araba("BMW");
28        hareketliler[1] = new Uçak("Airbus A380");
29
30        for (HareketEdebilir h : hareketliler) {
31            h.ilerle();
32            h.dur();
33            System.out.println("—");
34        }
35        /* Çıktı:
36        BMW araba ileri gidiyor,
37        BMW araba durdu.
38
39        Airbus A380 uçak havalandı.
40        Airbus A380 uçak iniş yapıyor.
41
42    */
43 }

```

Abstract Class ve Interface Arasındaki Temel Farklar (Önemli!)

Her ikisi de soyutlama sağlamasına rağmen, kullanım senaryoları ve yetenekleri farklıdır:

Özellik	Abstract Class	Interface
Implementasyon	Hem soyut (<code>abstract</code>) hem de somut (<code>concrete</code>) metodlar içerebilir.	Java 8 öncesi sadece soyut metodlar. Java 8+ ile <code>default</code> ve <code>static</code> somut metodlar içerebilir. Java 9+ ile <code>private</code> metodlar da.
Nitelikler	Normal nitelikler içerebilir (değişkenler).	Sadece <code>public static final</code> nitelikler (sabitler) içerebilir.
Constructor	Constructor'u olabilir.	Constructor'u olamaz.
Kalitim	Bir sınıf sadece tek bir abstract sınıfı <code>extends</code> edebilir.	Bir sınıf birden fazla interface'i <code>implements</code> edebilir.
Erişim Belirleyiciler	Üyeler için herhangi bir erişim belirleyici kullanabilir.	Varsayılan olarak tüm metodları <code>public abstract</code> , tüm alanları <code>public static final</code> dir.
Kullanım Amacı	Ortak bir temel sınıf sağlamak ve alt sınıflar arasında "bir-tür-dır (is-a)" ilişkisi kurmak.	Davranışsal bir sözleşme tanımlamak ve bir sınıfın birden fazla "yapabili" (can-do)" yeteneğini ifade etmek.

[Export to Sheets](#)

Özetle, bir grup sınıf arasında **ortak implementasyon** ve **ozellikler** paylaşmak ve hiyerarşik bir yapı kurmak istediğinizde **abstract class** kullanınız. Bir sınıfın **belirli davranışları** **sağlamasını** **zorunlu kılmak** ve kodlar arasında gevşek bağıllılık (loose coupling) sağlamak istediğinizde ise **interface** kullanınız.

Elmas Problemi Nedir?

Elmas Problemi (Diamond Problem), çoklu miras (multiple inheritance) destekleyen programlama dillerinde ortaya çıkan bir belirsizlik sorunudur. Java gibi bazı diller, bu problemi önlemek için doğrudan çoklu mirası sınıflar için desteklemezler.

Problemin adı, sınıf hiyerarşisinin diyagramı çizildiğinde bir elmas şeklini andırmamasından gelir:



Burada:

- A bir üst (ana) sınıfıtır.
- B ve C sınıfları A sınıfından miras alır.
- D sınıfı ise hem B hem de C sınıflarından miras alır.

Problemin Ortaya Çıkışı

Diyelim ki A sınıfında birMetot() adında bir metot var. B ve C sınıfları bu birMetot() 'u miras alır ve kendi içlerinde bu metodu geçersiz kılarlar (override ederler). Yani B 'nin birMetot() versiyonu farklı, C 'nin birMetot() versiyonu farklı bir iş yapar.

Şimdi D sınıfına gelelim. D, hem B 'den hem de C 'den miras aldığı için, D sınıfının bir objesi (Dobjesi) oluşturulduğunda ve Dobjesi.birMetot() çağrıldığında, derleyici bir ikilemle karşılaşır:

- D sınıfı, B sınıfının birMetot() versyonunu mu kullanmalı?
- Yoksa C sınıfının birMetot() versyonunu mu kullanmalı?

Bu belirsizlik, derleyicinin hangi versiyonu seçeceğini karar verememesine ve dolayısıyla bir "elmas problemi"ne yol açar.

II Composition - Inner class and static

static yapımlar basla yerden new anahtar kelimesi ile çağrılabilir
yukarı değildir

```
1 public class ProductValidator { 1 usage new *
2
3     // to use "ProductValidator.isValid(product)", we should make this method static
4     // as static class, we can make an object from ProductValidator class
5     // but the difference is this: we can only make one object
6     public static boolean isValid(Product product) { 1 usage new *
7         if (product.price > 0 && !product.name.isEmpty())
8             return true;
9     }
10    else {
11        return false;
12    }
13 }
14
15 }                                     1 public class ProductManager { 2 usages new *
16
17     public void add(Product product) { new *
18
19         // ProductValidator validator = new ProductValidator();
20
21         if (ProductValidator.isValid(product)) {
22             System.out.println("Added");
23         }
24
25         else {
26             System.out.println("Product not valid");
27         }
28     }
29 }
```

burada ProductValidator classinden
bir tane nesne oluştur re onu

ProductValidator. ile çağrımıza
harcas kullanıblır.

2

yeni bir method
static yapımları
zaten class
isni ile direkt
çağrılar //

- static türündeki object'ler yaratıldıkları sınıfta bellekteki
atılmışları garbage collector tarafından yinelemeyecektir

* Java'da static constructor yapısı sınıfın new'landığı zaman çalışır
yani static bir içinde static constructor çalışır.

C#'da ise durum farklıdır. Constructor hem static yapıda hem
de new'landığı zaman çalışır

→ static'te de çalışması için constructor'in de static olması gereklidir

Java'da ana class static olamaz.

ama **inner class** static olabilir

→ ana class'in içindeki başka
bir class

public class MainClass {

public static class InnerClass {

}

}

* Anna inner class yapısı SOLID principle'ye uygun olduğu için kriterimiz
öncelikliyor.
→ bir class bir iş yapar