

08.01.2022

# PATİKA – INNOVA JAVA SPRİNG ODEV – 1

Bekir Gürkan GÜLDAŞ

# İçindekiler

1. Compiler .....	3
2. Interpreter .....	4
2.1. Compiler ile Interpreter Arasındaki Farklar .....	5
3. Pass By Value ve Pass By Reference .....	6
4. Primitive Type ve Wrapper Class .....	7
5. JDK, JRE, JVM, JIT Kavramları .....	8
6. Stack Hafıza ve Heap Hafıza .....	12
7. Serileştirme .....	14
8. Java 8 .....	15
9. Java 9 .....	21
10. SOLID Prensipleri .....	28
11. Model-View-Controller .....	29
12. Creational Design Pattern .....	30
12.1. Singleton Design Pattern .....	31
12.2. Builder Design Pattern .....	32

# 1. Compiler

Java, Python, C#, PHP, Pascal, Javascript gibi yüksek seviyeli programlama dilleriyle yazılmış kaynak kodunun makinenin mimarisine göre makine diline dönüştürülme işlemidir.

Üretilen makine kodunun tekrardan derlenmesine gerek kalmaksızın herhangi bir zamanda farklı girdilerle tekrar tekrar çalıştırılabilir.

Compiler, kaynak kodu derlemeden önce derlenecek tüm kodları kontrol eder. Herhangi bir yazım yanlışı veya yanlış kullanımlar mevcut ise bütün programın çeviri işlemi durur.

## 2. Interpreter

Interpreter, kaynak kodu satırlar halinde değerlendirir. Herhangi bir hata oluşana kadar kod değerlendirilmeye devam edilir. İlk hatanın görüldüğü yerde ise durur. Bu nedenle hata ayıklama işlemi daha kolaydır.

Interpreterlar standart bir çalıştırılabilir kod üretmezler. Derleyicilerin tersine kodun işlenmeyen satırları üzerinden hiç geçilmez ve buralardaki hatalar ile ilgilenilmez.

Genelde kaynak koddan, makine diline anlık olarak dönüşüm yaptıkları için, derleyicilere göre daha yavaş çalışırlar.

## 2.1. Compiler ile Interpreter Arasındaki Farklar

Interpreter	Compiler
Programı satır satır işler	Tüm programı tarar ve bir bütün olarak makine koduna çevirir
Kaynak kodu analiz etmekle zaman harcamaz. Ancak genel yürütme süresi daha yavaştır.	Kaynak kodun analizi için büyük zaman harcar. Ancak genel yürütme süresi daha hızlıdır
Herhangi bir hata olana kadar programı çalıştırır. İlk hata gördüğü yerde durur. Bu nedenle hata ayıklama kolaydır.	Tüm kaynak kodu taradıktan sonra hata mesajı üretir. Bu nedenle hata ayıklama nispeten zordur.

### 3. Pass By Value ve Pass By Reference

**Pass By Value:** Metotun içine aldığı parametrenin değeri, belleğin başka bir yerine kopyalanır. Metodun değişkenine erişmek veya değişkenini değiştirmek gerektiği zaman, yalnızca kopyaya erişilir/değiştirilir, orijinal değere dokunulmaz.

**Pass By Reference:** Değişkenin hafıza adresinin ilgili metoda iletildiği anlamına gelir. Yani hafızada ilgili değişkenin değerini saklayan bloğun adresi, metoda geçirilir.

## 4. Primitive Type ve Wrapper Class

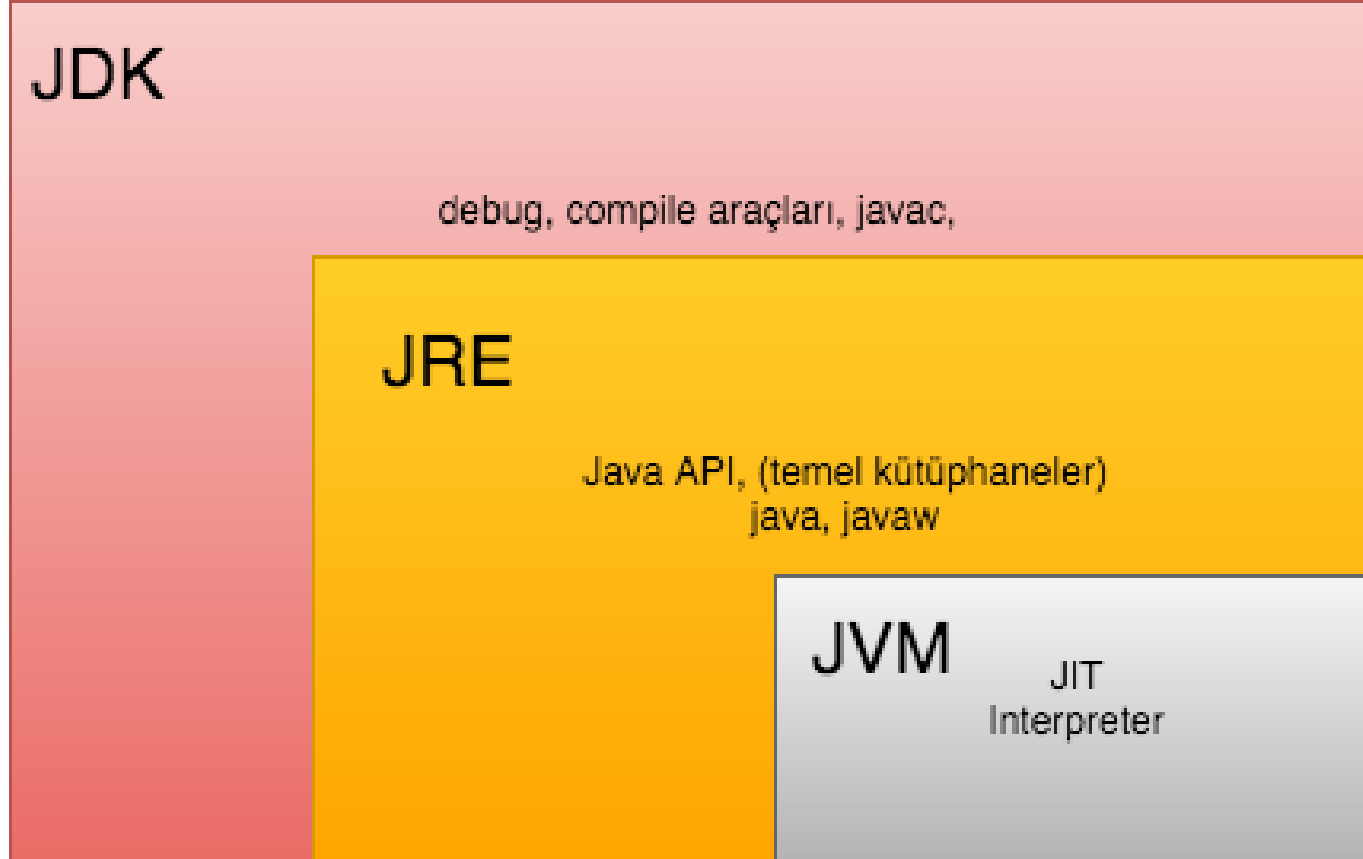
**Primitive türler** sabit bir boyutu sahiptir. Nesne olarak kabul edilmez ve doğrudan stackte depolanırlar. Java'da tanımlanan sekiz primitive tür mevcuttur.

**Wrapper sınıfları**, primitive türleri nesne olarak kullanmanın bir yolunu sağlar. Primitive türlerin varsayılan değerlerinin int için 0, char için `\u0000`, boolean için false gibi türlere bağlı olarak wrapper sınıfları için varsayılan değer null'dur.

Primitive Type Keyword

Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	approximately $\pm 3.40282347E+38F$ (6-7 significant decimal digits) Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)	0.0d
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'
boolean	Not precisely defined*	true or false	false

## 5. JDK, JRE, JVM, JIT Kavramları





## 5.1. JDK (Java Development Kit)

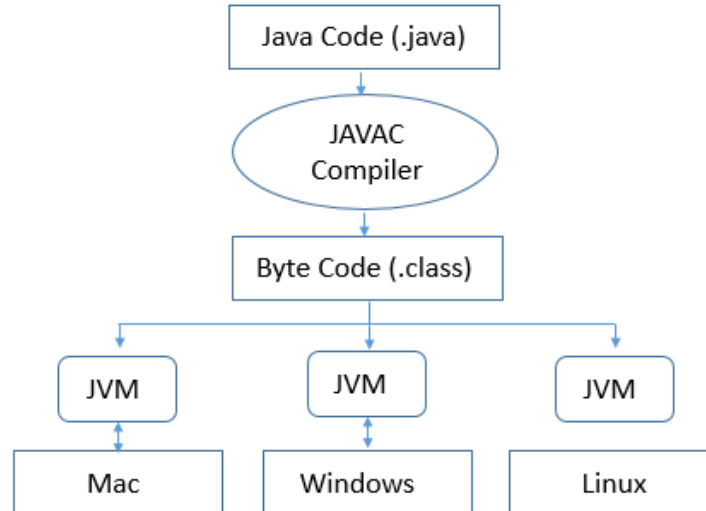
Java ile geliştirme yapmak için gerekli araçları içeren geliştirici paketidir. Herhangi bir Java uygulamasını çalıştırmak için Jdk'ya ihtiyaç yoktur, Jre programların çalışması için gerekli altyapıyı barındırır.

## 5.2. JRE (Java Runtime Environment)

Kullanıcıların Java programlarını çalıştırmaları için minimum gereksinimleri içeren, içerisinde JVM'yi ve Java platformu çekirdek dosyalarını bulunduran yazılımdır.

## 5.3. JVM (Java Virtual Machine)

Bir Java programında compile işlemi sonrası bytecode ismi verilen bir ara sürüm oluşur. Tek derleme işlemi Java sınıflarının bytekoda dönüştürülmesi esnasında yapılmaz. JVM bünyesinde de bytekodun makine koduna dönüştürüldüğü bir derleme gerçekleştirilir.

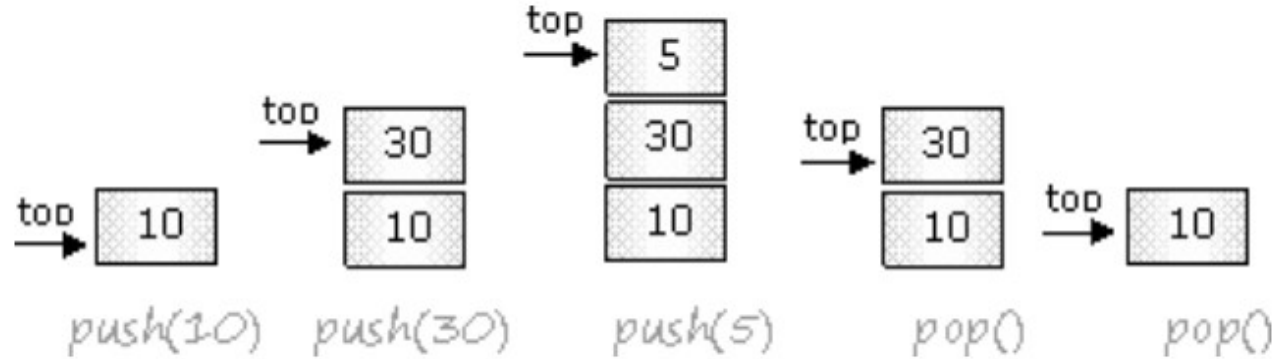


## 5.4. JIT (Just in Time Compiler)

JVM bünyesinde de bytekodun makine koduna dönüştürüldüğü bir derleme gerçekleştirilir. Bu işleme Just in time (JIT) compilation ismi verilmektedir. JIT ile derlenen uygulamalar AOT ile derlenen uygulamalara göre daha yavaş çalışır ancak JIT ile derlenen uygulamalar platform bağımsızdır.

## 6. Stack Hafıza

Programdaki aktif fonksiyonların Stack frame'lerini üst üste yığın halinde saklar, fonksiyon bitince de o fonksiyonun Stack frame'i Stack hafızadan çıkarılır (pop), Stack frame'i Fonksiyonun parametrelerinin ve local(yerel) değişkenlerinin tümünü içerir. Bu memory, geliştirici tarafından değil, compiler tarafından yönetilir. Stack'teki bilgiler kodunuzun derleme aşamasında, direk bellek içine yerleştirilir.



Stack veri yapısı çalışma şekli

## 6.1. Heap Hafıza

Bellek üzerinde yer tahsisi yapılan belli bir bölümdür. Boyut olarak herhangi bir kısıtlama yoktur. Heap de ayrılan veriye erişim kısıtlı değildir yani veri silinmediği takdirde istenilen yerde ayrılan verinin saklandığı yerin adresi ile o veriye erişebilir. Heap'teki bellek kullanımı compiler tarafından değil, geliştiriciler tarafından kontrol edilir.

## 7. Serileştirme

Serializable, bir değerin byte dizisine dönüştürülüp disk üzerinde hard copy olarak saklanabilmesidir. Nesnenin serileştirilmesi için Serializable interfacesini implement etmesi gerekir. Java serileştirmeyeyle beraber 2 temel sınıf sunar. ObjectInputStream ve ObjectOutputStream adı verilen bu iki sınıf ile, Serializable interfaceini uygulayan herhangi bir sınıf serileştirilebilir.

Bu iki sınıfdan ilki olan ObjectInputStream, ObjectInput interfaceini uygular ve serileştirilen nesneyi tekrar akışdan okumak için kullanılır. ObjectInputStream adındaki diğer sınıf, ObjectOutput interfaceini uygular ve herhangi bir nesneyi akışa yazdırmak için kullanılır.

# 8. Java 8

Java 8 ile gelen yenilikler genel olarak aşağıdaki şekilde listeleyebilir;

- Lambda expressions
- Functional interfaces
- Method references
- Stream API
- Optional Class
- Concurrency Improvement
- Default Methods
- JDBC Improvement

## 8.1. Lambda Expressions

Lambda expressionlar, herhangi bir class'a ait olmadan iş yapabilen fonksiyonlardır. Lambda sayesinde hem daha okunabilir kod üretimine, hem de kod tekrarının engellenmesi daha da kolaylaşmıştır. Bir lambda ifadesini tekrar tekrar kullanabilir, parametre olarak başka bir yere iletilebilir özelliklerine sahiptir..

```
1 package patika;
2
3 import java.util.ArrayList;
4
5 public class Program
6 {
7     public static void main(String[] args)
8     {
9         ArrayList<Integer> numbers = new ArrayList<Integer>();
10        numbers.add(1);
11        numbers.add(2);
12        numbers.add(3);
13        numbers.forEach( (n) -> { System.out.println(n); } );
14    }
15 }
```



## 8.2. Functional Interfaces

Tek bir abstract(soyut) methodu bulunan interface'ler için kullanılan tanımdır. Lambda ifadeleri ile sıkı bir ilişki içerisindeydir. Ayrıca Single Abstract Method Interfaces (SAM Interfaces) olarak da bilinir. Functional interface'ler default ve static methodlar içerebilir ancak tek bir tane abstract methodu olmalıdır. Bunun nedeni de lambda ifadeleri ile çalışabilmesini sağlamaktır.

```
1 package patika;
2
3 interface IMathOperation
4 {
5     int Add(int a, int b);
6 }
7
8 public class Program
9 {
10     public static void main(String[] args)
11     {
12         IMathOperation Sum = (a,b) -> {return a + b;};
13         IMathOperation Mult = (a,b) -> {return a * b;};
14
15         System.out.println("Sum : " + Sum.Add(5, 4) );
16         System.out.println("Mult: " + Mult.Add(5, 4));
17     }
18 }
```

## 8.3. Method References

Metodları, nesneler veya primitive değerlermiş gibi kullanılabilmesini ve bunları başka bir metoda parametre olarak gönderilmesini sağlar. :: söz dizimi aracılığıyla static methodlar class name ile, static olmayan methodlar ise instance objeleri ile referans verilebilir.

```
1 package patika;
2
3 interface IMathOperation
4 {
5     int Add(int a, int b);
6 }
7
8 public class Program
9 {
10     public static int Sum(int a, int b) { return a + b; }
11     public static int Mult(int a, int b) { return a * b; }
12
13     public static void main(String[] args)
14     {
15         IMathOperation Sum = Program::Sum;
16         IMathOperation Mult = Program::Mult;
17
18         System.out.println("Sum : " + Sum.Add(5, 4) );
19         System.out.println("Mult: " + Mult.Add(5, 4));
20     }
21 }
```

## 8.4. Stream API

Stream API, Collection'lar üzerinde bazı işlemleri yapmayı kolaylaştıran bir yapıdır. Stream API sayesinde sık kullanılan çeşitli operasyonları yapabilir. Bunlardan birkaçını şöyle sıralanabilir;

- filter
- forEach
- map
- reduce
- distinct
- limit
- collect
- count
- min / max

```
1 package patika;
2
3 import java.util.ArrayList;
4
5 public class Program
6 {
7     public static void main(String[] args)
8     {
9         ArrayList<Integer> numbers = new ArrayList<Integer>();
10        numbers.add(1);
11        numbers.add(2);
12        numbers.add(3);
13        numbers.forEach( (n) -> { System.out.println(n); } );
14    }
15 }
```

## 8.5. Optional Class

Bir objenin kullanılmadan önce yapılan null check'lerin daha okunabilir ve kontrol edilebilir olmasını sağlayan Optional yapısıdır. Optional class ile daha güvenilir kodlar yazılabilir. Objeyi Optional ile wrap ederek eğer null değilse kullan, null ise başka birşey yap gibi koşullarda bulunulabilir.

```
1 package patika;
2
3 import java.util.Optional;
4
5 public class Program
6 {
7     public static void main(String[] args)
8     {
9         String[] str = new String[10];
10        Optional<String> checkNull = Optional.ofNullable(str[5]);
11        if(checkNull.isPresent()){ // check for value is present or not
12            String lowercaseString = str[5].toLowerCase();
13            System.out.print(lowercaseString);
14        }else
15            System.out.println("string value is not present");
16    }
17 }
```

# 9. Java 9

Java 9 ile gelen yenilikler genel olarak aşağıdaki şekilde listeleyebilir;

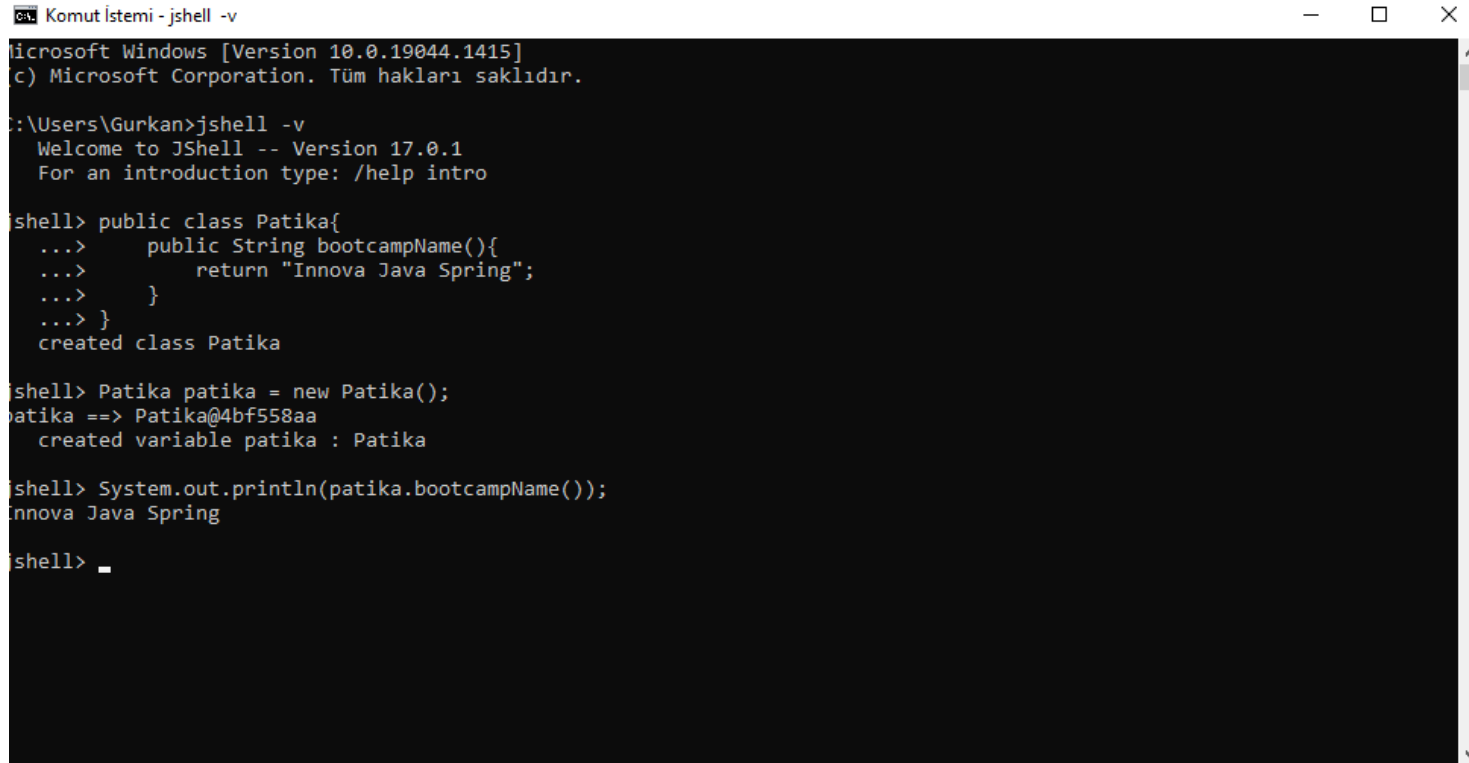
- Module System
- Java Shell Tool
- Private Interface Methods
- Anonymous Inner Classes
- Factory Methods
- Process API Improvement

## 9.1. Module System

Java 9 öncesi bütün Java kodları tek bir `rt.jar` dosyasında derlenmesi kullanılmayan kütüphanelerin de mecburen projeye eklemesine neden oluyorken Java 9 ile beraber Modüler bir sisteme geçildi. Bütün Java Standart Edition kodları buna uyumlu hale getirildi ve kodlar kendi içinde gruplara ayrıldı. Java, 71 tane modüle ayrılarak `module-info.java` dosyası ile modülün hangi paketleri dışarı açacağı ve hangi başka modülleri kullanacağı tanımlamaları yapılabilir oldu.

## 9.2. Java Shell Tool

JShell ile terminal üzerinde Java kodu yazıp çalıştırılabilir.



```
Komut İstemi - jshell -v
Microsoft Windows [Version 10.0.19044.1415]
(c) Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\Gurkan>jshell -v
Welcome to JShell -- Version 17.0.1
For an introduction type: /help intro

jshell> public class Patika{
...>     public String bootcampName(){
...>         return "Innova Java Spring";
...>     }
...> }
created class Patika

jshell> Patika patika = new Patika();
patika ==> Patika@4bf558aa
created variable patika : Patika

jshell> System.out.println(patika.bootcampName());
Innova Java Spring

jshell> _
```

## 9.3. Private Interface Methods

Java 9, interfacelerde private metotlar kullanılmasına izin vermektedir. Bu sayede interfacelere gövdeli metotlar oluşturulabilmektedir.

```
1 package patika;
2
3 interface MathOperation
4 {
5     default int Sum(int a, int b)
6     {
7         return sumOperation(a, b);
8     }
9
10    default int Mult(int a, int b)
11    {
12        return multOperation(a, b);
13    }
14
15    private int sumOperation(int a, int b)
16    {
17        return a+b;
18    }
19
20    private int multOperation(int a, int b)
21    {
22        return a*b;
23    }
24 }
25
26 public class Program implements MathOperation
27 {
28     public static void main(String[] args)
29     {
30         MathOperation mathOperation = new Program();
31
32         System.out.println("Sum : "+mathOperation.Sum(5, 4));
33         System.out.println("Mult: "+mathOperation.Mult(5, 4));
34     }
35 }
```



## 9.4. Anonymous Inner Classes

Anonymous class başka bir class içinde tanımlanmış bir class'tır. Anonymous class'ların bir ismi yoktur bu yönden inner class'lardan ayrılır. İsimlerinin olmaması da bu classlardan bir object yaratılma ihtiyacı olmamasından dolayıdır. Object yaratırken bu object'in bir veya birden fazla metodu override edilmesi istenebilir ve sadece oluşturulan bu object'te bu metotların override edilmesini isteyebilir. Bu durumlarda anonymous class kullanılır.

```
1 package patika;
2
3 class MathOperation
4 {
5     public int Sum(int a, int b)
6     {
7         return a+b;
8     }
9     public int Mult(int a, int b)
10    {
11        return a*b;
12    }
13 }
14
15 public class Program
16 {
17     public static void main(String[] args)
18     {
19         MathOperation mathOperation = new MathOperation()
20         {
21             @Override
22             public int Sum(int a, int b)
23             {
24                 return 2*(a+b);
25             }
26
27             @Override
28             public int Mult(int a, int b)
29             {
30                 return (a+b)*(a+b);
31             }
32         };
33
34         System.out.println("Sum : "+mathOperation.Sum(5, 4));
35         System.out.println("Mult: "+mathOperation.Mult(5, 4));
36     }
37 }
```

## 9.5. Factory Methods

Koleksiyon kitaplığı List, Set ve Map arabirimi için statik fabrika yöntemlerini içerir. Bu yöntemler, az sayıda koleksiyon oluşturmak için kullanışlıdır.

```
1 package patika;
2
3 import java.util.List;
4 import java.util.Map;
5
6 public class Program
7 {
8     public static void main(String[] args)
9     {
10         //Create List with List.of()
11         List<String> userNames = List.of("Bekir","Gurkan","Hamit");
12         //Create Map with Map.of()
13         Map<String, String> userNamesAndBranch = Map.of("Bekir","Student","Gurkan","Student","Hamit","Teacher");
14
15         userNames.forEach( (n) -> System.out.println(n));
16         userNamesAndBranch.forEach( (n,b) -> System.out.println(n+" "+b));|
17     }
18 }
```

## 9.6. Process API Improvement

İşletim sistemi süreçlerini yönetmeye ve kontrol etmeye yardımcı olur. ProcessHandle, süreçleri işlemeye ve kontrol etmeye yardımcı olur. Süreçleri izleyebilir, childleri listeleyebilir, bilgi alabilir. Bu arabirim, değer tabanlı, değişmez ve iş parçacığı için güvenli örnekler döndüren statik fabrika yöntemleri içerir.

```
1 package patika;
2
3 public class Program {
4     public static void main(String[] args)
5     {
6         ProcessHandle currentProcess = ProcessHandle.current(); // Current processhandle
7         System.out.println("Process Id: " + currentProcess.pid()); // Process id
8         System.out.println("Direct children: " + currentProcess.children()); // Direct children of the process
9         System.out.println("Class name: " + currentProcess.getClass()); // Class name
10        System.out.println("All processes: " + ProcessHandle.allProcesses()); // All current processes
11        System.out.println("Process info: " + currentProcess.info()); // Process info
12        System.out.println("Is process alive: " + currentProcess.isAlive());
13        System.out.println("Process's parent " + currentProcess.parent()); // Parent of the process
14    }
15 }
```

# 10. SOLID Prensipleri

## **S — Single-responsibility principle**

- Bir sınıfın veya metodun yapması gereken yalnızca bir işi olması gerekir.

## **O — Open-closed principle**

- Bir sınıf ya da metod halihazırda var olan özellikleri korumalı ve değişikliğe izin vermemelidir. Yani davranışını değiştirmiyor olmalı ve yeni özellikler kazanabiliyor olmalıdır.

## **L — Liskov substitution principle**

- Kodlarda herhangi bir değişiklik yapmaya gerek duymadan alt sınıfları, türedikleri(üst) sınıflar yerine kullanılabilmelidir.

## **I — Interface segregation principle**

- Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturulmalıdır.

## **D — Dependency Inversion Principle**

- Sınıflar arası bağımlılıklar olabildiğince az olmalıdır özellikle üst seviye sınıflar alt seviye sınıflara bağımlı olmamalıdır.

# 11. Model-View-Controller

Projenin farklı amaçlara hizmet eden yapılarını birbirinden ayırarak, projeyi daha rahat geliştirilebilir ve test edilebilir duruma getirilmiş olur.

## **Model**

Veri tabanına erişim, veri tabanı ilişkileri gibi data ile ilgili işlemlerle birlikte Hibernate gibi frameworkleri içerisinde bulunduran katmandır yani data(veri) işlemleri bu katmanda gerçekleşir.

## **View**

Bu katman kullanıcının ekranda gördüğü katman olarak adlandırılır. Bu kısımda arayüz teknolojileri kullanılır.

## **Controller**

Kullanıcıların view üzerinden verdiği komutların Controller aracılığı ile model katmanının işlenmesini sağladığı katmandır, yani view ve model arasında kalan katmandır. Metotlar ve fonksiyonlar bu katmanda çağırılarak kullanılır.

## 12. Creational Design Pattern

Yaratımsal tasarım kalıpları, yazılım nesnelerinin nasıl yaratılacağı hakkında genel olarak öneriler sunarak kullandığı esnek yapı sayesinde daha önceden belirlenen durumlara bağlı olarak gerekli nesneleri yaratır. Yaratımsal desenler, hangi nesnenin çağırılması gerektiğini izlemeden sistemin uygun nesneyi çağırmasını sağlayan tasarım kalıplarıdır. Nesnelerin yaratılması gerektiği durumlarda uygulamaya farkedilebilir bir esneklik katar.

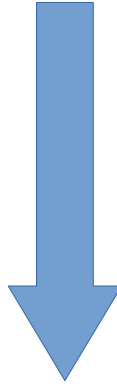
## 12.1. Singleton Design Pattern

Nesnenin sadece bir defa oluşturulmasını öngören bir mekanizma kurulmak istenildiğinde etkin bir biçimde kullanılabilen bir tasarım desendir. Oluşturulan bir sınıftan sadece bir nesne yaratılacak şekilde bir kısıtlama yapabilme olanağı sağlar ve nesneye ilk kez ihtiyaç duyulana kadar yaratılmayabilir.

```
1 package patika;
2
3 class SingletonExample
4 {
5     private static SingletonExample instance;
6
7     private SingletonExample() {}
8
9     public static SingletonExample getInstance()
10    {
11        return instance == null ? instance = new SingletonExample() : instance;
12    }
13 }
14
15 public class Program {
16     public static void main(String[] args)
17     {
18         SingletonExample singleton1 = SingletonExample.getInstance();
19         SingletonExample singleton2 = SingletonExample.getInstance();
20
21         System.out.println("Singleton1: "+singleton1); // Reference of Singleton1
22         System.out.println("Singleton2: "+singleton2); // Reference of Singleton2
23
24         System.out.println(singleton1 == singleton2); // True
25     }
26 }
```

## 12.2. Builder Design Pattern

Tek ara yüz kullanarak karmaşık bir nesne grubundan gerektiğinde parça yaratılmasını sağlar. Nesne grubu kullanıldıkça istenilen şekilde yapılır ve bu sayede kullanılmayan parçaların gereksiz yere yaratılarak kaynak harcama durumu ortadan kaldırılmış olur.





```

1 package patika;
2
3 public class Car {
4
5     private String modelName, color;
6     private int productionYear, price;
7
8     public Car(Builder builder)
9     {
10         this.modelName = builder.modelName;
11         this.color = builder.color;
12         this.productionYear = builder.productionYear;
13         this.price = builder.price;
14     }
15     public static class Builder
16     {
17         private String modelName, color;
18         private int productionYear, price;
19
20         public Builder() {}
21
22         public Builder modelName(String modelName)
23         {
24             this.modelName = modelName;
25             return this;
26         }
27
28         public Builder color(String color)
29         {
30             this.color = color;
31             return this;
32         }
33
34         public Builder productionYear(int productionYear)
35         {
36             this.productionYear = productionYear;
37             return this;
38         }
39
40         public Builder price(int price)
41         {
42             this.price = price;
43             return this;
44         }
45
46         public Car build()
47         {
48             return new Car(this);
49         }
50     }
51     @Override
52     public String toString() {
53         String car = "Model Name:      " + this.modelName + "\n"
54             + "Color:          " + this.color + "\n"
55             + "Year of Production: " + this.productionYear + "\n"
56             + "Price:         " + this.price;
57         return car;
58     }
59 }

```

```

1 package patika;
2
3
4 public class Program {
5     public static void main(String[] args)
6     {
7         Car Bmw = new Car.Builder().modelName("iX3").color("Black").productionYear(2021).price(1646500).build();
8         Car Renault = new Car.Builder().color("Gray").productionYear(2013).price(150000).modelName("Fluence").build();
9
10        System.out.println(Bmw.toString());
11        System.out.println(Renault.toString());
12
13    }
14 }
15

```