1. **Java Introduction :-**

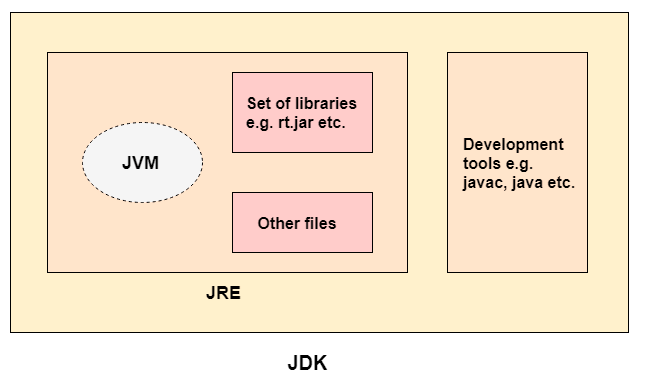
**What is Java?**

1. Java is a **class-based object-oriented programming language** for **building** **web** and **desktop** **applications**.
2. Java is **fast**, **secure**, and **reliable**, therefore it is widely used for developing Java applications in **laptops, data centres, game consoles, scientific supercomputers, cell phones, etc**.

**What is Java Platform?**

1. **Java Platform** is a **collection** **of programs** that help **programmers** **to** **develop and run Java programming applications efficiently**.
2. It includes an **execution engine**, **a compiler, and a set of libraries** in it. It is a **set of computer software and specifications.**
3. **James Gosling developed the Java platform at Sun Microsystems, and the Oracle Corporation later acquired it.**

**JDK, JRE, JVM?**



**JDK**

1. The **Java Development Kit (JDK)** is **a software development environment** which is used to develop **Java applications** and [**applets**](https://www.javatpoint.com/java-applet).
2. It is **physically** exists. It contains **JRE + development tools** (e.g. javac, java).
3. In order **to create, compile and run Java program** you would need JDK installed on your computer.

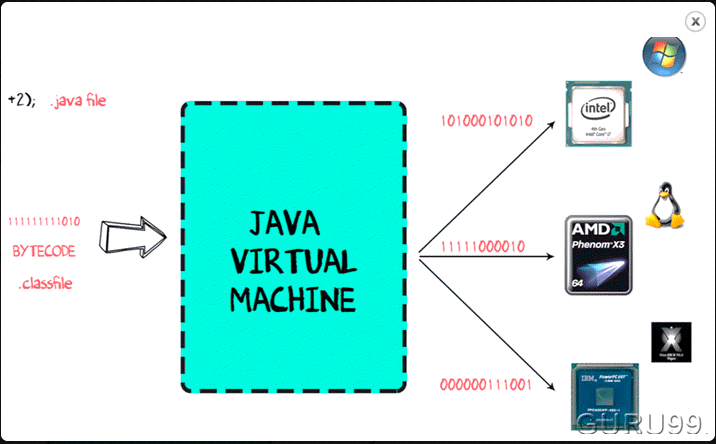
**JRE**

1. It is also written as **Java RTE**. **The JRE is a set of** **software tools which are used for developing Java applications.**
2. It is used to provide the **runtime environment**. It is the implementation of JVM.
3. It **physically** exists. It contains a **set of libraries + other files** that **JVM** uses at **runtime**.
4. In simple terms, **if you want to run a Java program**, you need **JRE**.

**JVM**

1. **JVM** (Java Virtual Machine) is an **abstract** **machine**. It is called a **virtual** **machine** because **it doesn't physically exist.**
2. It is a **specification** that provides a **runtime** **environment** in which **Java** **bytecode** **can be executed.**
3. It converts Java **bytecode** into **machine** **language**.

**How JVM works?**



1. The code to display the addition of two numbers is System.out.println(1+2), and saved as a .java file.
2. Using the **java compiler** the **Source** **code** is converted into an intermediate code called the **bytecode.** The output is a **.class file.**
3. This **byte** **code** is not understood by any platform, but only a virtual platform called the **Java Virtual Machine.**
4. This **Virtual** **Machine** resides in the **RAM** of your **Operating System**. When the **Virtual Machine is fed with this bytecode**, it identifies the platform it is working on and converts the **bytecode** into the **native** **machine** **code**.

## How is Java Platform Independent?

1. Like the C compiler, the **Java compiler does not produce native executable code for a particular machine**. Instead, Java produces a unique format called **bytecode**.
2. It executes according to the rules laid out in the virtual machine specification. Therefore, **Java is a platform-independent language.**
3. **Bytecode** is understandable to any **JVM** installed on any **OS**. In short, the java source code can run on all operating systems.

**JAVA Features:-**

1. **Simple**

Java is considered as one of simple language because it does not have complex features like Operator overloading, [multiple inheritance](https://beginnersbook.com/2013/05/java-multiple-inheritance/), pointers and explicit memory allocation.

1. **Robust**

* **Robust** means **reliable**. Java programming language is developed in a way that puts a lot of emphasis on early checking for possible errors, that’s why java compiler is able to detect errors that are not easy to detect in other programming languages.
* The main features of java that makes it robust are garbage collection, Exception Handling and memory allocation.

1. **Secure**

* We don’t have **pointers** and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java.
* That’s why **several security flaws** like **stack corruption** or **buffer overflow** is impossible to **exploit** in Java.

1. **Distributed**

Using java programming language we can create distributed applications. **RMI** (Remote Method Invocation) and **EJB** (Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one systems that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

1. **Multithreading**

Java supports [multithreading](https://beginnersbook.com/2013/03/multithreading-in-java/). Multithreading is a Java feature that **allows concurrent execution of two or more parts of a program for maximum utilization of CPU.**

1. **Portable**

* As discussed above, java code that is written on one machine can run on another machine.
* The platform independent byte code can be carried to any platform for execution that makes java code portable.

**2. What is OOPS?**

1. **Object-Oriented Programming System (OOPs)** is a **programming** **concept** that works on the **principles of** **abstraction**, **encapsulation**, **inheritance**, **and** **polymorphism**.
2. It allows users to **create** **objects** they want and **create** **methods** to **handle those objects**.
3. The basic concept of OOPs is to **create objects, re-use them throughout the program, and manipulate these objects to get results.**

**Class**

* **A** **class** is a **group of objects** which have **common** **properties**.
* It is a **template** **or** **blueprint** from which **objects** **are** **created**.
* It is a **logical** **entity**. It **can't** **be** **physical**.
* A class is a logical entity because we always use class to represents a category like a Fruit; fruit is logical, we cannot see or eat fruit but we can see or eat a mango or a banana which are physical entities and belongs to the fruit category.

A class in Java can contain:

* variables
* Methods
* Constructors
* Blocks
* Nested class and interface

**Object:**

1. **An object is an instance of a class.**
2. **An entity** that has **state**, **behaviour** and **Identity** is known as an **object**

* **State:** represents the **data (value) of an object.**
* **Behaviour:** represents the behaviour (functionality) of an object such as **deposit**, **withdraw**, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the **JVM to identify each object uniquely.**

**1. Abstraction**:

1. **Abstraction** is a process where you show only “**relevant**” **data** and “**hide**” **unnecessary details of an object from the user.**
2. For example, when you login to your bank account online, you enter your user\_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you.
3. There are two ways to achieve abstraction in java

* **Abstract class (0 to 100%)**
* **Interface (100%)**

1. **Abstract Class:**

* A class which is declared as abstract is known as an **abstract class**.
* It can have **abstract** and **non-abstract** methods. It needs to be extended and implements its methods.
* It **cannot** **be** **instantiated**.
* It can have [**constructors**](https://www.javatpoint.com/java-constructor) and **static** **methods** also. (constructor is instantiated using **child class object** i.e. **upcasting**)
* It can have **final methods** which will force the subclass not to change the body of the method.

abstract class Car

{

String model;

String brand;

public abstract void addFeatures();

//constructor

Car()

{

this.model = "X5";

this.brand = "BMW";

}

}

Class SportCar extends Car

{

boolean nitro;

@Override

public void addFeatures()

{

this.nitro = true;

}

public void showDetail()

{

System.out.println("Model :"+ model);

System.out.println("Brand :"+ brand);

System.out.println("Nitro :"+ nitro);

}

}

public class MainClass

{

public static void main (String arg[])

{

SportCar sportCar = new SportCar ();

sportCar.addFeatures ();

sportCar.showDetail ();

}

}

* Yes, an **abstract** **class** can have a **constructor** in Java. **To initialize the fields(variables) of abstract class.**
* The compiler automatically adds the **default** **constructor** in every class either it is an **abstract class or concrete class**.
* You can also provide a constructor to **abstract class explicitly**.
* When we use the constructor of the child class, the constructor of the **parent class invoked by child class constructor** either **implicitly** or **explicitly**. This is one of the reasons abstract class can have constructors in Java.

**Why can’t we create the object of an abstract class?**

* Because Abstract classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then what would happen? There would be no actual implementation of the method to invoke.
* Also because an **object** is **concrete**. An **abstract** **class** is like a **template**, so **you have to extend it and build on it before you can use it.**

1. **Interface:-**
   * The interface in Java is a mechanism to achieve [**abstraction**](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body.
   * It is used to achieve **abstraction** and **multiple** [**inheritance** in Java](https://www.javatpoint.com/inheritance-in-java).
   * Interface methods are by default **public** and **abstract.**
   * Interface variables are by default **public, static** and **final.**
   * Java Interface also **represents the IS-A relationship**.

**Note**:

* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have **default and static methods** in an interface.
* Since Java 9, we can have **private methods** in an interface.

#### **Why And When To Use Interfaces?**

1) **To achieve security** - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** **multiple** **interfaces**. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

interface FirstInterface {

public void myMethod(); // interface method

}

interface SecondInterface {

public void myOtherMethod(); // interface method

}

class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {

System.out.println("Some text..");

}

public void myOtherMethod() {

System.out.println("Some other text...");

}

}

class Main {

public static void main(String[] args) {

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

interface MyInterface

{

/\* compiler will treat them as:

\* public abstract void method1();

\* public abstract void method2();

\*/

public void method1();

public void method2();

}

class Demo implements MyInterface

{

/\* This class must have to implement both the abstract methods

\* else you will get compilation error

\*/

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public static void main(String arg[])

{

MyInterface obj = new Demo();

obj.method1();

}

}

**Output**:

implementation of method1

**2. Encapsulation:-**

1. **Encapsulation** in Java is a process of **wrapping** **code** **and** **data** **together into a single unit.(** The meaning of **Encapsulation**, is to make sure that "**sensitive**" data is hidden from users)
2. We can create a fully encapsulated class in Java by making **all the data members of the class private.**
3. Provide public **get** and **set** methods to **access** and **update** the value of a **private** variable.
4. It keeps the **data** and the **code** safe from **external** **interference**.
5. This way **data** can only be accessed by **public** **methods** thus making the **private** **fields** and their **implementation** **hidden** for **outside** **classes**. **That’s why** **encapsulation** **is** **known** **as** **data hiding.**
6. It is the process of **hiding** **information** **details** and **protecting** **data** and **behaviour** of the object.
7. A java **class** is the example of **encapsulation**. Java bean is the fully encapsulated class because all the data members are private here.

class EncapsulationDemo{

private int ssn;

private String empName;

private int empAge;

//Getter and Setter methods

public int getEmpSSN(){

return ssn;

}

public String getEmpName(){

return empName;

}

public int getEmpAge(){

return empAge;

}

public void setEmpAge(int newValue){

empAge = newValue;

}

public void setEmpName(String newValue){

empName = newValue;

}

public void setEmpSSN(int newValue){

ssn = newValue;

}

}

public class EncapsTest{

public static void main(String args[]){

EncapsulationDemo obj = new EncapsulationDemo();

obj.setEmpName("Mario");

obj.setEmpAge(32);

obj.setEmpSSN(112233);

System.out.println("Employee Name: " + obj.getEmpName());

System.out.println("Employee SSN: " + obj.getEmpSSN());

System.out.println("Employee Age: " + obj.getEmpAge());

}

}

**Output:**

Employee Name: Mario

Employee SSN: 112233

Employee Age: 32

## Abstraction vs. Encapsulation

Often encapsulation is misunderstood with Abstraction. Let’s study-

* Encapsulation is more about "How" to achieve a functionality
* Abstraction is more about "What" a class can do.

A simple example to understand this difference is a mobile phone. Where the complex logic in the circuit board is encapsulated in a touch screen, and the interface is provided to abstract it out.

1. **Inheritance:**
   1. Inheritance is a mechanism in **which one class acquires the property of another class**
   2. With **inheritance**, we can **reuse the fields and methods of the existing class.**
   3. Hence, **inheritance** **facilitates** **Reusability** and is an important concept of OOPs

**Why use inheritance in java?**

* For [**Method Overriding**](https://www.javatpoint.com/method-overriding-in-java) (so [**runtime polymorphism**](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For **Code Reusability.**

**Types:-**

* + 1. **Single Inheritance.**
    2. **Multi-level Inheritance.**
    3. **Hierarchical Inheritance.**
    4. **Multiple Inheritance (supported with only using implementing multiple interfaces)**

**Super Keyword:-**

1. The super keyword is similar to **"this"** keyword.
2. The keyword **“super”** can be used to access any **data** **member** or **methods** of the **parent** **class**.
3. Super keyword can be used at **variable**, **method** and **constructor** level.

**Syntax**:

**super.<method-name>();**

**4. Polymorphism:-**

* 1. It is the ability of an **object** to take **many** **forms**.
  2. Inheritance lets users inherit **attributes** and **methods**, and polymorphism uses these **methods** **to perform different tasks.**
  3. Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

1. **Method overloading**
2. If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but **different in no. of Parameters, type of parameters and order of parameters it is known as Method Overloading.**
3. **Method overloading** is an example of [**Static Polymorphism**](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)**,** **Static** Polymorphism is also known as **compile time binding or early binding.**
4. [**Static binding**](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) happens at **compile time**. Method overloading is an example of static binding **where binding of method call to its definition happens at Compile time.**

**Method Overloading and Type Promotion**

* When a **data** **type** **of** **smaller** **size** is promoted to the **data** **type** **of** **bigger** **size** than this is called **type** **promotion**.
* For example: **byte** data type can be promoted to **short**, a **short** data type can be promoted to **int**, **long**, **double** etc.

**What it has to do with method overloading?**  
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.  
Let’s take an example to see what I am talking here:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* I am passing float value as a second argument but

\* it got promoted to the type double, because there

\* wasn't any method having arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the **double** **type** as there wasn’t any method with argument list as (int, float)

1. **Method** **Overriding**
   1. If **subclass** (child class) has the same method as declared in the **parent** **class**, it is known as **method overriding in Java**.
   2. In other words, if a **subclass** provides the **specific** **implementation** of the method that has been declared by one of its **parent** class, it is known as **method** **overriding**.
   3. It is an example of **Runtime Polymorphism.**

### **Usage of Java Method Overriding**

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

class Human{

//Overridden method

public void eat()

{

System.out.println("Human is eating");

}

}

class Boy extends Human{

//Overriding method

public void eat(){

System.out.println("Boy is eating");

}

public static void main( String args[]) {

Boy obj = new Boy();

//This will call the child class version of eat()

obj.eat();

}

}

**Output:**

Boy is eating

**Method Overriding and Dynamic Method Dispatch**

1. Method Overriding is an example of [**Runtime** **polymorphism**](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/). When a **parent** **class** **reference** **points** to the **child** **class** **object** then the call to the **overridden** **method** **is** **determined** **at** **runtime**.
2. **Because during method call which method (parent class or child class) is to be executed is determined by the type of object.**
3. This process in which call to the **overridden** **method** is resolved at **runtime** is known as **dynamic** **method** **dispatch**.

class ABC{

//Overridden method

public void disp()

{

System.out.println("disp() method of parent class");

}

}

class Demo extends ABC{

//Overriding method

public void disp(){

System.out.println("disp() method of Child class");

}

public void newMethod(){

System.out.println("new method of child class");

}

public static void main( String args[]) {

/\* When Parent class reference refers to the parent class object

\* then in this case overridden method (the method of parent class)

\* is called.

\*/

ABC obj = new ABC();

obj.disp();

/\* When parent class reference refers to the child class object

\* then the overriding method (method of child class) is called.

\* This is called dynamic method dispatch and runtime polymorphism

\*/

ABC obj2 = new Demo();

obj2.disp();

}

}

Output:

disp() method of parent class

disp() method of Child class

#### **Rules for Java Method Overriding**

1. The method must have the **same** **name** as in the **parent class**
2. The method must have the **same** **parameter** as in the **parent class.**
3. [**Access Modifier**](https://beginnersbook.com/2013/05/java-access-modifiers/) of the **overriding** **method** (method of subclass) cannot be more **restrictive** than the **overridden** **method** of parent class.**( For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method ) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.)**
4. There must be an **IS-A relationship** (inheritance).

* **Checked Exception Rule:**

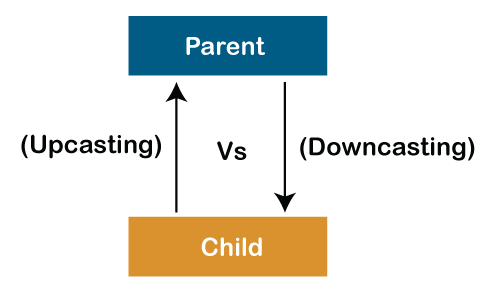
1. **Overriding** **method** may or may not declare checked exception.
2. **Overriding** **method** can declare **same** **checked** **exception** or **sub** **type** declared by **overridden** **method**.
3. **Overriding** **method** can’t declare **checked** **exception** which is not declared by **Overridden** **method**.

* **Important**:-

1. In **Runtime** **polymorphism** when **parent** **class** **refers** to the **child** **class** **object** then **object** can call **all** **overridden** **methods** of **child** **class** and **non**-**overridden** methods of **parent** **class**.
2. But it can’t call the newly declared methods of child class.

**Upcasting and Downcasting:**

1. A process of converting **one data type** to **another** is known as **Typecasting** and **Upcasting** and **Downcasting** is **the type of object typecasting**.
2. In Java, the **object** can also be **type** **casted** like the **datatypes**.
3. **Parent** and **Child** **objects** are **two types of objects**. So, there are two types of typecasting possible for an **object**, i.e., **Parent to Child** and **Child to Parent** or can say **Upcasting** and **Downcasting**.



**Upcasting**

1. **Upcasting** is a type of **object** **typecasting** in which a **child object** **is type casted to a** **parent class object**.
2. By using the **Upcasting**, we can easily access the **variables** and **methods** of the **parent class to the child class.**
3. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. For instance, we can access the **overridden** **methods**.
4. **Upcasting** is also known as **Generalization** and **Widening**.
5. Java compiler will do the implicit casting.

###### **Whenever upcasting happens always remember**

**1) Parent's all variables will be accessed**

**2) Child's methods(overridden methods if overriding happened else inherited methods as it is from parent)**

**will be called.**

But same is not applicable to variables because **variables decision happens at a compile time**, so always class A’s variables (not child’s inherited variables) will be accessed.

**class** Parent{  
 **void** PrintData() {  
 System.***out***.println(**"method of parent class"**);  
 }  
}  
  
**class** Child **extends** Parent {  
 **void** PrintData() {  
 System.***out***.println(**"method of child class"**);  
 }  
}  
**class** UpcastingExample{  
 **public static void** main(String args[]) {  
  
 Parent obj1 = (Parent) **new** Child();  
 Parent obj2 = (Parent) **new** Child();  
 obj1.PrintData();  
 obj2.PrintData();  
 }  
}

**output:**

method of child class

method of child class

**Downcasting:**

1. Similarly, down casting means the typecasting of a **parent object to a child object**.
2. **Downcasting** cannot be implicit. It has to be done explicitly.
3. **Downcasting** is also known as **Specialization** and **Narrowing**.

**// gives compile time error**

**Child c = new Parent();**

**// gives class cast exception**

**Child c = (Child) new Parent();**

**class** Parent {  
  
 String **name**;  
  
 *// A method which prints the data of the parent class* **void** showMessage()  
 {  
 System.***out***.println(**"Parent method is called"**);  
 }  
}  
  
*// Child class***class** Child **extends** Parent {  
  
 **int age**;  
  
 *// Performing overriding* @Override  
 **void** showMessage()  
 {  
 System.***out***.println(**"Child method is called"**);  
 }  
}  
  
**class** Downcasting{  
  
 **public static void** main(String[] args)  
 {  
 Parent p = **new** Child();  
 p.**name** = **"Shubham"**;  
  
 *// Performing Downcasting Implicitly  
 //Child c = new Parent(); // it gives compile-time error  
  
 // Performing Downcasting Explicitly* Child c = (Child)p;  
  
 c.**age** = 18;  
 System.***out***.println(c.**name**);  
 System.***out***.println(c.**age**);  
 c.showMessage();  
 }  
}

**Output:**

Shubham

18

Child method is called

**It is like Child child = new Child () but using parent class reference so that’s why we can access all members of parent and child class using inheritance characteristics.**

**3. Java Variables:-**

1. A **variable** is a **container** which holds the value while the [Java program](https://www.javatpoint.com/simple-program-of-java) is executed.
2. A variable is assigned with a **data** **type**.
3. A **variable** is the **name of a** **reserved** **area** **allocated** **in** **memory**.
4. **Variable** is a name of **memory** **location**. There are three types of **variables** in java: **Instance, Local** and **Static**.

**1). Instance Variable:**

* **A variable which declared inside the class but outside the method.**
* It is called an instance variable because its **value is instance-specific** and is **not shared among instance.**

**2). Local Variable:**

* A **variable** declared inside the **body of the method** is called **local variable.**
* You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
* A **local** **variable** cannot be defined with **"static"** keyword.

**3). Static Variable:**

* A variable that is declared as **static** **keyword** is called a **static** **variable**.
* It cannot be **local**.
* You can create a **single** **copy** of the **static** **variable** and **share it among all the** **instances of the class.**
* **Memory** **allocation** for **static** **variables** happens only once when the **class is loaded** **in the memory.**

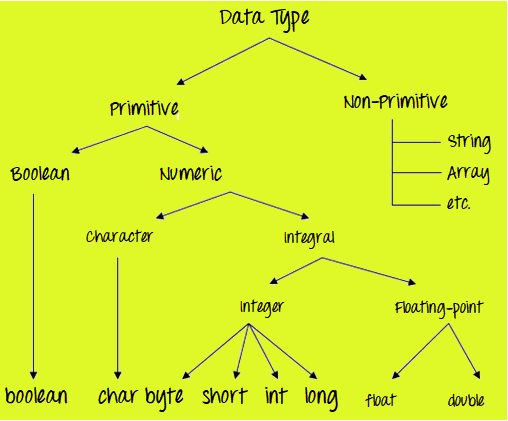
**Data Types in Java:**

**Data** **types** specify the different **sizes** and **values** that can be stored in the **variable**. There are two types of data types in Java:

1. **Primitive data types:**

* In Java language**, primitive data types are the building blocks of data manipulation**
* The primitive data types include **Boolean, char, byte, short, int, long, float and double.**

1. **Non-primitive data types:** The non-primitive data types include [**Classes**](https://www.javatpoint.com/object-and-class-in-java)**,**[**Interfaces**](https://www.javatpoint.com/interface-in-java)**, and**[**Arrays**](https://www.javatpoint.com/array-in-java)**.**



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| boolean | false | 1 bit |
| char | '\u0000' | 2 byte |
| byte | 0 | 1 byte |
| short | 0 | 2 byte |
| int | 0 | 4 byte |
| long | 0L | 8 byte |
| float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

* **Why char uses 2 byte in java and what is \u0000?**

It is because java uses **Unicode** **system** not ASCII code system. The \**u0000** is the **lowest** **range** of **Unicode** **system**.

* Smallest value: \u0000
* Largest value: \uFFFF

**Type Casting:**

* In **Java**, **type casting** is a method or process that converts a **data** **type** **into** **another** **data** **type** in both ways **manually** and **automatically**.
* The **automatic** **conversion** is done by the **compiler** and **manual** **conversion** **performed** by the **programmer**.

1. **Automatically:**

* Data type of **smaller** **capacity** is assigned to **larger** **capacity**.
* Also called **widening**
* It is safe because there is no chance to lose data.

double d;

Int i = 10;

d = i;

**It takes place when:**

* Both data types must be compatible with each other.
* The target type must be larger than the source type.

**byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

1. **Manually:**

* Data type of **larger** **capacity** is assigned to **smaller** **capacity**.
* Also called **Narrowing**.
* Here explicite type casting is required;

Double d = 10;

Int i;

i = (int) d;

**double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

**4. Array:**

* Java array is an **object** which contains **elements of a** **similar** **data** **type**.
* The elements of an array are stored in a **contiguous** **memory** **location**.
* It is a data structure where we store similar elements.
* **We can store only a fixed set of elements in a Java array.**

### **Advantages**

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an **index** **position**.

### **Disadvantages**

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

**class** Testarray{

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

**int** a[]=**new** **int**[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//traversing array

**for**(**int** i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

**Output:**

10

20

70

40

50

**int** a[]={33,3,4,5}; //declaration, instantiation and initialization

**5. String:**

* String is basically an **object** that **represents** **sequence** **of** **character** **values**.
* String is an **immutable class.**
* An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

**char**[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=**new** String(ch);

is same as:

String s="javatpoint";

* **The** **java.lang.String** **class** **implements** **Serializable**, **Comparable** **and** **CharSequence** **interfaces**.

### **How to create a string object?**

There are two ways to create String object:

1. **By string literal**
2. **By new keyword**
3. **String Literal:**

Java **String** **literal** is created by using **double** **quotes**. For Example:

String s="welcome";

This **string** **object** will be created in **string** **pool**.

**String Pool:-**

* 1. It is a **special** **memory** reserved in **heap memory** to store the **string objects** created using the **string literal.**
  2. **String** **pool** doesn’t allow **two string objects** with **same values.**
  3. Each time you create a string literal, the **JVM** checks the "**string** **constant** **pool**" first. If the string already exists in the pool, a reference to the pooled instance is returned.
  4. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:
  5. It is preferred to use **String** **literals** as it allows **JVM** to **optimize memory allocation.**
  6. In earlier versions of Java up to JDK 6 String pool was located inside **PermGen(Permanent Generation) space**. But in **JDK 7** it is moved to the main **heap area**.



# **Java String intern () :-**

* The **java string intern ()** method returns the interned string. It returns the canonical representation of string.
* It can be used to return string from memory, if it is created by **new** keyword. It **creates exact copy of heap string object in string constant pool.**

1. **New Keyword:**

String s=**new** String("Welcome");//creates two objects and one reference variable

* In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the **literal** "**Welcome**" will be placed in the **string** **constant** **pool**.
* The variable **s** will refer to the object in a heap (non-pool).

# **Immutable String in Java**

* In java, **string objects are immutable**. Immutable simply means **unmodifiable** or **unchangeable**.
* Once string object is created its **data** or **state** can't be changed but a new **string** **object is created.**

Let's try to understand the immutability concept by the example given below:

**class** Testimmutablestring{

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

 String s="Sachin";

  s.concat(" Tendulkar");//concat() method appends the string at the end

  System.out.println(s);//will print Sachin because strings are immutable objects

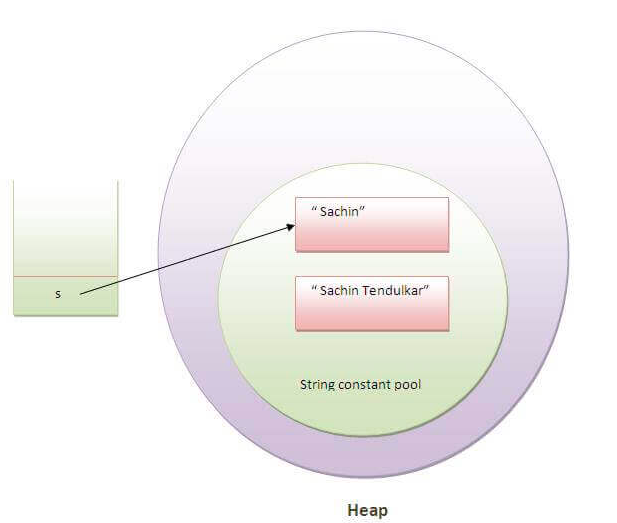
 }

}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring)

Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with **Sachin Tendulkar**. That is why string is known as **immutable**.



* As you can see in the above figure that two objects are created but s reference variable still refers to **"Sachin**" not to **"Sachin Tendulkar".**
* But if we explicitly assign it to the reference variable, it will refer to **"Sachin Tendulkar"** object. For example:

**class** Testimmutablestring1{

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

   String s="Sachin";

   s=s.concat(" Tendulkar");

   System.out.println(s);

 }

}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1)

Output:Sachin Tendulkar

**Why string objects are immutable in java?**

* Because java uses the concept of **string** **literal**. Suppose there are 5 reference variables, all refers to one object **"Sachin".**
* If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why **string** **objects** are **immutable** in java.

**Why did the String pool move from PermGen to**the **normal heap area?**

* **PermGen** **space** is limited, the **default** **size is just 64 MB**. it was a problem with creating and storing too many string objects in PermGen space.
* That’s why the String pool was moved to a larger heap area. To make Java more memory efficient, the concept of string literal is used.
* By the use of the ‘new’ keyword, The JVM will create a new string object in the normal **heap area** even if the same string object is present in the **string pool**.

For ex-

*String a=new String(“Bhubaneswar”)*

Let’s have a look at the concept with a java program and visualize the actual JVM memory structure:

Program:

|  |
| --- |
| class StringStorage {      public static void main(String args[])      {          String s1 = "TAT";          String s2 = "TAT";          String s3 = new String("TAT");          String s4 = new String("TAT");          System.out.println(s1);          System.out.println(s2);          System.out.println(s3);          System.out.println(s4);      }  } |

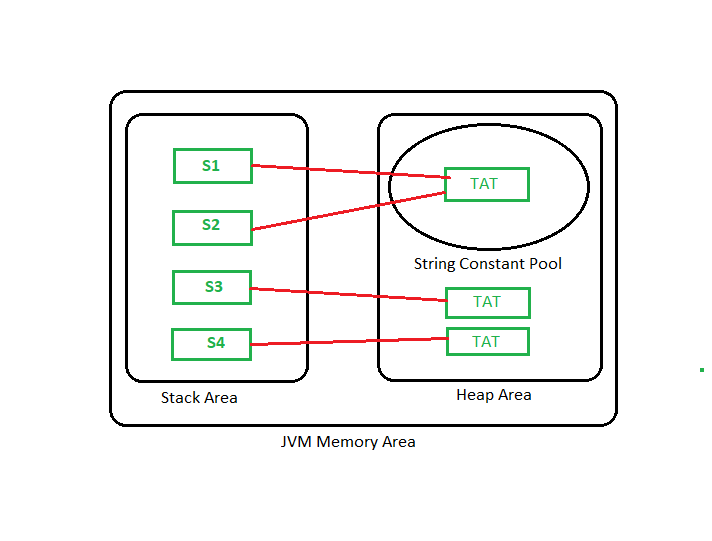
**Output**

TAT

TAT

TAT

TAT



**Note:** All objects in Java are stored in a heap. The reference variable is to the object stored in the stack area or they can be contained in other objects which puts them in the heap area also.

# **Java StringBuffer Class**

* Java **StringBuffer** class is used to create **mutable** **(modifiable**) **String objects.**
* The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

#### **Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | It creates an empty String buffer with the initial capacity of 16. |
| StringBuffer(String str) | It creates a String buffer with the specified string.. |
| StringBuffer(int capacity) | It creates an empty String buffer with the specified capacity as length. |

**class** StringBufferExample{

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

StringBuffer sb=**new** StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**Output:**

Hello Java

# **Java StringBuilder Class**

* Java StringBuilder class is used to create **mutable** (modifiable) String. The Java **StringBuilder** class is same as **StringBuffer** class except that it is **non**-**synchronized**.
* It is available since JDK 1.5.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuilder() | It creates an empty String Builder with the initial capacity of 16. |
| StringBuilder(String str) | It creates a String Builder with the specified string. |
| StringBuilder(int length) | It creates an empty String Builder with the specified capacity as length. |

**class** StringBuilderExample{

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

StringBuilder sb=**new** StringBuilder("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

A list of differences between **StringBuffer** and **StringBuilder** are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | **StringBuffer** is ***synchronized*** i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | **StringBuilder** is ***non-synchronized*** i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | **StringBuffer** **is *less efficient*** **than** **StringBuilder**. | **StringBuilder** **is *more efficient*** **than** **StringBuffer**. |

**How to create Immutable class?**

* There are many immutable classes like **String, Boolean, Byte, Short, Integer, Long, Float, Double etc.**
* In short, all the **wrapper** **classes** and **String** **class** is **immutable**. We can also create **immutable** **class** by creating **final** **class** that have **final data members** as the example given below:

### **Example to create Immutable class**

|  |
| --- |
| In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method. |

**public** **final** **class** Employee{

**final** String pancardNumber;

**public** Employee(String pancardNumber){

**this**.pancardNumber=pancardNumber;

}

**public** String getPancardNumber(){

**return** pancardNumber;

}

}

**The above class is immutable because:**

* The **instance** **variable** of the class is **final** i.e. we cannot change the value of it after creating an object.
* The **class** is **final** so we **cannot create the subclass.**
* There is no **setter** **methods** i.e. we have no option to change the value of the **instance** **variable**.

These points makes this class as immutable.

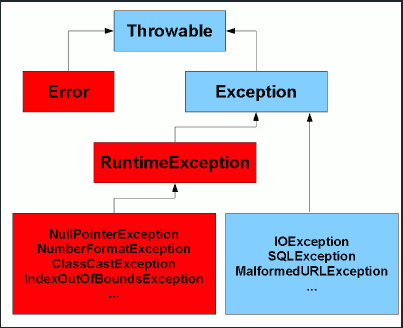
**5. Exception Handling:-**

**What is Exception in Java?**

* **Exception in Java** is an **event** that **interrupts the** **execution** **of** **program** **instructions** and **disturbs the normal flow of program execution**.
* It is an **object** that **wraps an error event information that occurred within a method** and it is passed to the **runtime** **system**.

**Exception handling:-**

The Exception Handling in Java is one of the powerful mechanism to handle the **runtime** **errors** **so that normal flow of the application can be maintained.**



1. **Checked Exception : -**

* **Checked** **exceptions** are checked at **compile-time**.
* It means if a method is throwing a checked exception then it should **handle the exception using**[**try-catch block**](https://beginnersbook.com/2013/04/try-catch-in-java/) **or it should declare the exception using** [**throws** **keyword**](https://beginnersbook.com/2013/04/java-throws/), **otherwise the program will give a compilation error.**
* **Checked** **exceptions** are propagated using **throws** keyword.

**class** FilenotFound\_Demo {  
  
 **public static void** main(String args[]) **throws** FileNotFoundException {  
 File file = **new** File(**"E://file.txt"**);  
 FileReader fr = **new** FileReader(file);  
 }  
}

1. **Runtime Exception :-**

* **Unchecked** **exceptions** are not checked at **compile** **time**. The unchecked exceptions are those exceptions that occur during the execution of the program. Hence they are also referred to as **Runtime exceptions**.
* It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error.
* **Most of the times these exception occurs due to the bad data provided by user during the user-program interaction.**
* It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately.
* All **Unchecked** **exceptions** are **direct** **sub** **classes** of **Runtime Exception** class.
* Unchecked exceptions are propagated automatically.

**public class** JavaExceptionExample {  
 **public static void** main(String args[]) {  
 **try** {  
 *//code that may raise exception* **int** data = 100 / 0;  
 } **catch** (ArithmeticException e) {  
 System.out.println(e);  
 }  
 *//rest code of the program* System.out.println(**"rest of the code..."**);  
 }  
}

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

1. **Error**

* The error can be defined as an abnormal condition that indicates something has gone wrong with the execution of the program. These are not handled by Java programs.
* Error is irrecoverable e.g. **OutOfMemoryError**, **VirtualMachineError**, **AssertionError** etc

## Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

**Throw Keyword:**

* The Java throw keyword is used to **explicitly** throw an exception.
* We can throw either **checked** or **unchecked** exception in java by **throw** **keyword**.
* The **throw** **keyword** is **mainly used to throw custom exception.** We will see custom exceptions later.
* The syntax of java throw keyword is given below.

**class** Throw {  
 **public void** p(**int** n) {  
 **try** {  
 **if** (n < 10) {  
 **throw new** ArithmeticException(**"Invalid No."**);  
 } **else** {  
 System.***out***.println(**"No. is valid"**);  
 }  
 } **catch** (Exception e) {  
 System.***out***.println(e.getMessage());  
 }  
 }  
  
 **public static void** main(String args[]) {  
 Throw obj = **new** Throw();  
 obj.p(5);  
 System.***out***.println(**"normal flow..."**);  
 }  
}

**Output:**

**Invalid No.**

**normal flow...**

**Throws Keyword:**

* The **Java throws keyword** is used **to declare an exception.**
* It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

**import** java.io.IOException;  
  
**class** Testthrows1 {  
 **void** m() **throws** IOException {  
 **throw new** IOException(**"device error"**);*//checked exception* }  
  
 **void** n() **throws** IOException {  
 m();  
 }  
  
 **void** p() {  
 **try** {  
 n();  
 } **catch** (Exception e) {  
 System.***out***.println(**"exception handled"**);  
 }  
 }  
  
 **public static void** main(String args[]) {  
 Testthrows1 obj = **new** Testthrows1();  
 obj.p();  
 System.***out***.println(**"normal flow..."**);  
 }  
}

exception handled

normal flow...

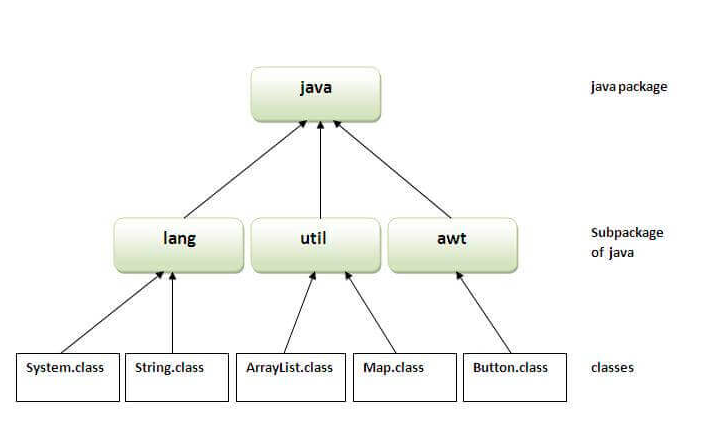
**Difference between throw and throws in Java**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException, SQLException. |

**6.Java Package**

* A **java package** is a group of similar types of **classes**, **interfaces** and **sub**-**packages**.
* Package in java can be categorized in two form, built-in package and user-defined package.
* There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
* Here, we will have the detailed learning of creating and using user-defined packages.



**Advantage of Java Package**

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2. Java package provides access protection.
3. Java package removes naming collision.

**Access Specifier:-**

* There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.
* The **access** **modifiers** in Java **specifies the accessibility or scope of a field, method, constructor, or class.**
* We can change the access level of **fields**, **constructors**, **methods**, and **class** by applying the **access** **modifier** on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a **private** **modifier** is only within the class. It cannot be accessed from outside the class.
2. **Default**: If you do not specify any access level, it will be the **default**. The access level of a **default** **modifier** is only within the package. It cannot be accessed from outside the package.
3. **Protected**: The access level of a **protected** **modifier** is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a **public** **modifier** is everywhere. It can be accessed from **within the** **class, outside the class, within the package and outside the package**.

**7. Cloning:-**

* The **object cloning** is a way to **create exact copy of an object**. The **clone**() method of **Object** **class** is used to **clone an object.**
* The **java.lang.Cloneable interface** must be **implemented by the class whose object clone we want to create.**
* If we don't implement **Cloneable** **interface**, clone() method

generates **CloneNotSupportedException**.

* The **clone() method** is defined in the **Object** **class**. Syntax of the clone() method is as follows:
* **Clonable is a marker interface means it doesn’t have any abstract method.**

**protected** Object clone() **throws** CloneNotSupportedException

### **Why use clone() method ?**

* The **clone() method** saves the extra processing task **for creating the exact copy of an object.**
* If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use **object** **cloning**.

**class** Student18 **implements** Cloneable {  
 **private int rollNo**;  
 **private** String **name**;  
  
 Student18(**int** rollNo, String name) {  
 **this**.**rollNo** = rollNo;  
 **this**.**name** = name;  
 }  
  
 **public** Object clone() **throws** CloneNotSupportedException {  
 **return super**.clone();  
 }  
  
 **public static void** main(String args[]) {  
 **try** {  
 Student18 s1 = **new** Student18(101, **"amit"**);

// creates instance of object type then we need type cast to required object type.  
 Student18 s2 = (Student18) s1.clone();  
  
 System.***out***.println(s1.**rollNo** + **" "** + s1.**name**);  
 System.***out***.println(s2.**rollNo** + **" "** + s2.**name**);  
  
 } **catch** (CloneNotSupportedException c) {  
 }  
 }  
}

**Output:**

101 amit

101 amit

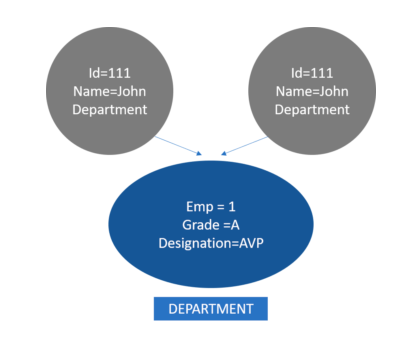
**Shallow Copy**

1. The **default implementation** of the **clone method** creates a **shallow copy** of the source object, it means a **new instance of type Object is created.**
2. **It copies all the fields** to a **new** **instance** and **returns a new object of type** **‘Object’**. This **Object** **explicitly** **needs** to be **typecast** in **object type of source object.**
3. This object will have an exact copy of all the fields of source object including the **primitive type** and **object references**.
4. **If the source object contains any references to other objects in field then in the new instance will have only references to those objects, a copy of those objects is not created.**
5. **This means if we make changes in shallow copy then changes will get reflected in the source object. Both instances are not independent.**
6. The clone method in Object class is protected in nature, so not all classes can use the clone () method. You need to implement Cloneable interface and override the clone method. If the Cloneable interface is not implemented then you will get **CloneNotSupportedException.super.clone()** will return shallow copy as per implementation in Object class.

**package** test;  
  
**class** Department {  
  
 String **empId**;  
 String **grade**;  
 String **designation**;  
  
 **public** Department(String empId, String grade, String designation) {  
 **this**.**empId** = empId;  
 **this**.**grade** = grade;  
 **this**.**designation** = designation;  
 }  
}  
  
**class** Employee **implements** Cloneable {  
  
 **int id**;  
 String **name**;  
 Department **dept**;  
  
 **public** Employee(**int** id, String name, Department dept) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 **this**.**dept** = dept;  
 }  
  
 *// Default version of clone() method. It creates shallow copy of an object.* **protected** Object clone() **throws** CloneNotSupportedException {  
 **return super**.clone();  
 }  
}  
  
**class** ShallowCopyInJava {  
  
 **public static void** main(String[] args) {  
  
 Department dept1 = **new** Department(**"1"**, **"A"**, **"AVP"**);  
 Employee emp1 = **new** Employee(111, **"John"**, dept1);  
 Employee emp2 = **null**;  
  
 **try** {  
 *// Creating a clone of emp1 and assigning it to emp2* emp2 = (Employee) emp1.clone();  
 } **catch** (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
  
 *// Printing the designation of 'emp1'* System.***out***.println(emp1.**dept**.**designation**); *// Output : AVP  
  
 // Changing the designation of 'emp2'* emp2.**dept**.**designation** = **"Director"**;  
  
 *// This change will be reflected in original Employee 'emp1'* System.***out***.println(emp1.**dept**.**designation**); *// Output : Director* }  
}

**AVP**

**Director**



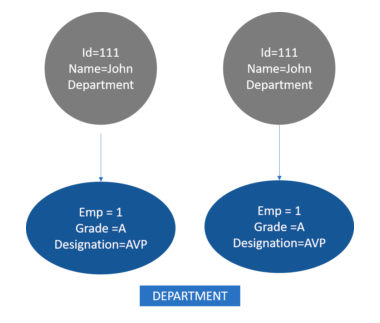
**Deep Copy:**

1. The deep copy of an object will have an exact copy of all the fields of source object like a shallow copy,
2. But unlike shallow copy if the source **object has any reference to object as fields, then a replica of the object is created by calling clone method.**
3. This means that both **source** and **destination** **objects are independent of each other.**
4. Any change made in the cloned object will not impact the source object.

**class** Department **implements** Cloneable {  
 String **empId**;  
 String **grade**;  
 String **designation**;  
  
 **public** Department(String empId, String grade, String designation) {  
 **this**.**empId** = empId;  
 **this**.**grade** = grade;  
 **this**.**designation** = designation;  
 }  
  
 *//Default version of clone() method.* **protected** Object clone() **throws** CloneNotSupportedException {  
 **return super**.clone();  
 }  
}  
  
**class** Employee **implements** Cloneable {  
 **int id**;  
 String **name**;  
 Department **dept**;  
  
 **public** Employee(**int** id, String name, Department dept) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 **this**.**dept** = dept;  
 }  
  
 *// Overriding clone() method to create a deep copy of an object.* **protected** Object clone() **throws** CloneNotSupportedException {  
 Employee emp = (Employee) **super**.clone();  
 emp.**dept** = (Department) **dept**.clone();  
 **return** emp;  
 }  
}  
  
**class** DeepCopyInJava {  
 **public static void** main(String[] args) {  
  
 Department dept1 = **new** Department(**"1"**, **"A"**, **"AVP"**);  
 Employee emp1 = **new** Employee(111, **"John"**, dept1);  
 Employee emp2 = **null**;  
  
 **try** {  
 *// Creating a clone of emp1 and assigning it to emp2* emp2 = (Employee) emp1.clone();  
 } **catch** (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
  
 *// Printing the designation of 'emp1'* System.***out***.println(**"Emp1 : "** + emp1.**dept**.**designation**); *// Output : AVP  
  
 // Changing the designation of 'emp2'* emp2.**dept**.**designation** = **"Director"**;  
  
 *// This change will be reflected in original Employee 'emp1'* System.***out***.println(**"Emp1 : "** + emp1.**dept**.**designation**); *// Output : AVP* }  
}

**Emp1 : AVP**

**Emp1 : AVP**



**Difference between Shallow Copy and Deep Copy**

|  |  |
| --- | --- |
| **Shallow Copy** | **Deep Copy** |
| Cloned object and source object are not disjoint completely | Cloned objects and source objects are completely independent of each other. |
| Changes made in the cloned instance will impact the reference variable of the source object | Changes made in the cloned instance will not impact the reference variable of the source object. |
| The default version of the clone is the shallow copy | To create deep copy we need to override the clone method of Object class. |
| Shallow copy is preferred if class variables of the object are only primitive type as fields | A deep copy is preferred if the object’s class variables have references to other objects as fields. |
| It is relatively fast | It is relatively slow. |

**8. Java Inner Classes:-**

1. **Java inner class** or **nested class** is a class that is declared **inside the class or interface.**
2. We use **inner** **classes** **to logically group classes and interfaces in one place to be more readable and maintainable.**
3. **Additionally, it can access all the members of the outer class, including private data members and methods.**
4. **Outer class can access data member of inner class using outer class object only.**

### **Advantage of Java inner classes**

There are three advantages of inner classes in Java. They are as follows:

1. **Nested** **classes** represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including **private**.
2. **Nested classes are** used **to develop more readable and maintainable code because it logically group classes and interfaces in one place only.**
3. **Code Optimization**: It requires less code to write.

**1. Java Member Inner class**

**2. Anonymous inner class**

**3. Java local inner class**

**4. Java static nested class**

# **Java Member Inner class**

* 1. A non-static class that **is created inside a class but outside a method is called member inner class**.
  2. It is also known as a **regular inner class**.
  3. It can be declared **with access modifiers like public, default, private, and protected.**

**class** Outer {  
  
 **private int data** = 30;  
  
 **class** Inner {  
 **void** msg() {  
 System.***out***.println(**"data is "** + **data**);  
 }  
 }  
  
 **public static void** main(String args[]) {  
  
 Outer outer = **new** Outer();  
 Outer.Inner in = outer.**new** Inner();  
 in.msg();  
   
 }  
}

**data is 30**

1. **Anonymous Inner class :**

* A **class** that has **no** **name** is known as an **anonymous** **inner** **class** in Java.
* It should be used if you have **to override a method of class or interface**. Java Anonymous inner class can be created in two ways:

**abstract class** Person {  
 **abstract void** eat();  
}  
  
**class** TestAnonymousInner {  
  
 **public static void** main(String args[]) {  
  
 Person p = **new** Person() {  
 **void** eat() {  
 System.***out***.println(**"nice fruits"**);  
 }  
 };  
 p.eat();  
   
 }  
}

**Output:**Hello Java Program for Beginners

nice fruits

## Internal working of given code

Person p=**new** Person(){  
 **void** eat(){System.out.println(**"nice fruits"**);}  
 };

1. **A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.**
2. **An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.**
3. **Local Inner class :**

* **A** **class** i.e., **created inside a method**, is called **local inner class in java.**
* **Local** **Inner** **Classes** are the **inner** **classes** that are defined inside a block. Generally, this **block is a method body.**
* **If you want to invoke the methods of the** **local** **inner** **class**, **you must** **instantiate** **this class inside the method**.

**class** localInner1 {  
 **private int data** = 30;*//instance variable* **void** display() {  
 **class** Local {  
 **void** msg() {  
 System.***out***.println(**data**);  
 }  
 }  
 Local l = **new** Local();  
 l.msg();  
 }  
  
 **public static void** main(String args[]) {  
 localInner1 obj = **new** localInner1();  
 obj.display();  
 }  
}

**30**

# **Java static nested class**

* **A** **static** **class** is a class that is **created** **inside** **a class,** is called a **static nested class in Java.**
* It cannot access **non-static data members** **and** **methods**. **It can be accessed by Outer class name.**
* It can access **static data members of the outer class, including private**.
* The static nested class cannot access **non-static (instance) data members or methods**

**class** Outer {  
  
 **static int** *data* = 30;  
  
 **static class** Inner {  
 **void** msg() {  
 System.***out***.println(**"data is "** + *data*);  
 }  
 }  
  
 **public static void** main(String args[]) {  
 Outer.Inner outer = **new** Outer.Inner();  
 outer.msg();  
 }  
}

**data is 30**

**9. Multithreading:-**

* **Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing **multiple** **threads** **simultaneously** **for** **maximum utilization of CPU.**
* A thread is a **lightweight** **sub-process**, the **smallest** **unit** of **processing**. **Multiprocessing** and **multithreading**, both are used to achieve multitasking.
* **However**, we use **multithreading** than **multiprocessing** because **threads use a** **shared memory area.**
* They **don't allocate separate memory** area so saves memory, and **context-switching between the threads takes less time than process.**

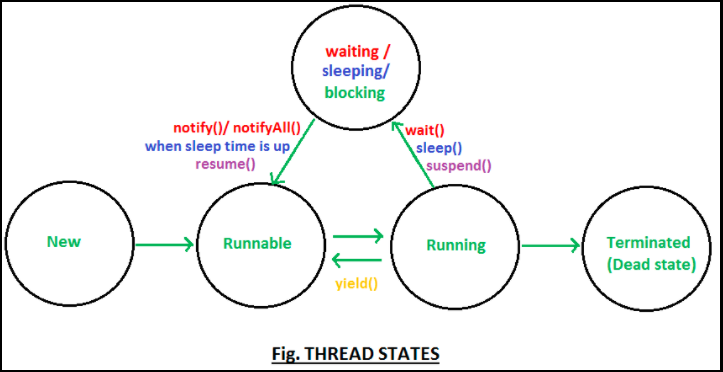
### **Advantages of Java Multithreading**

* 1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
  2. You **can perform many operations together, so it saves time**.
  3. Threads are **independent**, **so it doesn't affect other threads if an exception occurs in a single thread.**

## Java Thread class

* Java provides **Thread class** to achieve **thread** **programming**.
* Thread class provides [**constructors**](https://www.javatpoint.com/java-constructor) and **methods** to **create and perform operations on a thread.**
* **Thread** **class** extends [**Object class**](https://www.javatpoint.com/object-class) and **implements Runnable interface.**
* **Methods are**

**Start (), run (), sleep(), setName(), getName() etc.**



|  |
| --- |
| **1) New**  The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

**2) Runnable**

The thread is in runnable state after invocation of start() method, but the **thread scheduler** has not selected it to be the running thread.

**3) Running**

The thread is in running state if the **thread scheduler** has selected it.

**4) Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run.

**5) Terminated**

A thread is in terminated or dead state when its **run()** method exits.

# **Thread Scheduler in Java**

* **Thread scheduler** in java is the **part of the JVM that decides which thread should run.**
* There is **no** **guarantee** **that which runnable thread will be chosen to run by the thread scheduler.**
* **Only one thread at a time can run in a single process.**
* The **thread scheduler** mainly **uses pre-emptive** **or** **time slicing scheduling** **to** **schedule the threads.**

# **How to create thread**

There are two ways to create a thread:

1. **By extending Thread class**
2. **By implementing Runnable interface**

### **Thread class:**

|  |
| --- |
| * Thread class provide **constructors** and **methods** to create and perform operations on a thread. * **Thread** **class** **extends** **Object class** **and** **implements** **Runnable interface.** |

### **Commonly used Constructors of Thread class:**

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

1. **Extending Thread Class**

**class** Multi **extends** Thread {  
   
 **public void** run() {  
 System.***out***.println(**"thread is running..."**);  
 }  
  
 **public static void** main(String args[]) {  
 Multi t1 = **new** Multi();  
 t1.start();  
 }  
}

1. **Implementing Runnable Interface**

**class** Multi3 **implements** Runnable {  
 **public void** run() {  
 System.***out***.println(**"thread is running..."**);  
 }  
  
 **public static void** main(String args[]) {  
 Multi3 m1 = **new** Multi3();  
 Thread t1 = **new** Thread(m1);  
 t1.start();  
 }  
}

If you are not extending the **Thread class**, your class object would not be treated as a **thread object**. So you need to explicitly create **Thread class object**. We are passing the object of your class that implements Runnable so that your class **run ()** method may execute.

# **Thread Scheduler in Java**

* **Thread scheduler** in java is the part of the JVM that decides which thread should run.
* **There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.**
* **Only one thread at a time can run in a single process.**
* **The thread scheduler mainly** uses **pre-emptive** **or** **time slicing scheduling** to schedule the threads.

# **Synchronization in Java**

* **Synchronization** **in java is the capability to control the access of multiple threads to any shared resource.**
* **Java Synchronization** is better option where we want to allow **only one thread to access the shared resource.**

### **Why use Synchronization**

**The synchronization is mainly used to**

* 1. **To prevent thread interference.**
  2. **To prevent consistency problem.**

### **Concept of Lock in Java**

1. **Synchronization** is built around **an internal entity known as the lock or monitor.**
2. **Every** **object** **has a** **lock** **associated with it**.
3. A thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
4. From Java 5 the package **java.util.concurrent.locks** contains several lock implementations.
5. **Synchronized Method :**
6. **Synchronized method is used to lock an object for any shared resource**.
7. **When a thread invokes a synchronized** **method**, **it** **automatically acquires the lock for that object and releases it when the thread completes its task.**

**class** Table {  
  
 *//synchronized method* **synchronized void** printTable(**int** n) {  
 **for** (**int** i = 1; i <= 5; i++) {  
 System.***out***.println(n \* i);  
 **try** {  
 Thread.*sleep*(400);  
 } **catch** (Exception e) {  
 System.***out***.println(e);  
 }  
 }  
  
 }  
}  
  
**class** MyThread1 **extends** Thread {  
 Table **t**;  
  
 MyThread1(Table t) {  
 **this**.**t** = t;  
 }  
  
 **public void** run() {  
 **t**.printTable(5);  
 }  
  
}  
  
**class** MyThread2 **extends** Thread {  
 Table **t**;  
  
 MyThread2(Table t) {  
 **this**.**t** = t;  
 }  
  
 **public void** run() {  
 **t**.printTable(100);  
 }  
}  
  
**class** TestSynchronization2 {  
 **public static void** main(String args[]) {

***//only one object***Table t = **new** Table();  
  
 MyThread1 t1 = **new** MyThread1(t);  
 MyThread2 t2 = **new** MyThread2(t);  
  
 t1.start();  
 t2.start();  
 }  
}

5

10

15

20

25

100

200

300

400

500

# **Synchronized Block in Java**

1. **Synchronized** **block** **can be used to perform** **synchronization** **on** **any** **specific** **resource of the method.**
2. **Suppose you have 50 lines of code in your method**, but you want to **synchronize only 5 lines**, **you can** **use synchronized block.**
3. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

**Points to remember for Synchronized block**

1. **Synchronized block is used to lock an object for any shared resource.**
2. **Scope of synchronized block is smaller than the synchronized method.**

**class** Table {  
  
 **void** printTable(**int** n) {  
  
 *//synchronized block* **synchronized** (**this**) {  
 **for** (**int** i = 1; i <= 5; i++) {  
 System.***out***.println(n \* i);  
 **try** {  
 Thread.*sleep*(400);  
 } **catch** (Exception e) {  
 System.***out***.println(e);  
 }  
 }  
 }  
 }*//end of the method*}  
  
**class** MyThread1 **extends** Thread {  
 Table **t**;  
  
 MyThread1(Table t) {  
 **this**.**t** = t;  
 }  
  
 **public void** run() {  
 **t**.printTable(5);  
 }  
  
}  
  
**class** MyThread2 **extends** Thread {  
 Table **t**;  
  
 MyThread2(Table t) {  
 **this**.**t** = t;  
 }  
  
 **public void** run() {  
 **t**.printTable(100);  
 }  
}  
  
**class** TestSynchronizedBlock1 {  
 **public static void** main(String args[]) {  
   
 *//only one object* Table obj = **new** Table();  
  
 MyThread1 t1 = **new** MyThread1(obj);  
 MyThread2 t2 = **new** MyThread2(obj);

t1.start();  
 t2.start();  
 }  
}

**100**

**200**

**300**

**400**

**500**

**5**

**10**

**15**

**20**

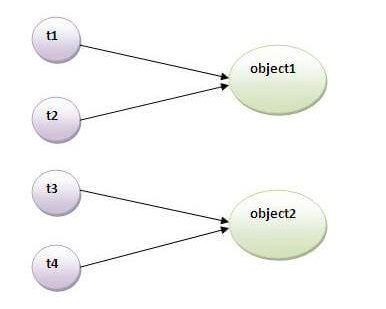
**25**

1. **Static Synchronized method:-**

**Class Level Lock in Java**

* 1. **Every** **class in Java has a unique lock which is nothing but** [**class level lock**](https://www.geeksforgeeks.org/object-level-class-level-lock-java/).
  2. If a thread wants to execute a **static** **synchronized** **method**, **then the thread requires a class level lock.**
  3. **Class level lock prevents multiple threads to enter a synchronized block in any of all available instances of the class on runtime.**
  4. This means if in runtime there are 10 instances of a class, only one thread will be able to access only one method or block of any one instance at a time.

### **Problem without static synchronization**



* Suppose there are **two objects of a shared class** (e.g. Table) named **object1** **and** **object2**.
* In case of **synchronized** **method** **and** **synchronized** **block** there cannot be interference between **t1** **and** **t2** or **t3** **and** **t4** because **t1** and **t2** both refers to a **common object that have a single lock.**
* But there can be **interference** **between** **t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock**.
* I want no interference between **t1** and **t3** or **t2** and **t4**.**Static** **synchronization** **solves this problem.**

**package** test;  
  
**class** Table {  
  
 *//static synchronized method* **synchronized static void** printTable(**int** n) {  
 **for** (**int** i = 1; i <= 10; i++) {  
 System.***out***.println(n \* i);  
 **try** {  
 Thread.*sleep*(400);  
 } **catch** (Exception e) {  
 }  
 }  
 }  
}  
  
**class** MyThread1 **extends** Thread {  
 **public void** run() {  
 Table.*printTable*(1);  
 }  
}  
  
**class** MyThread2 **extends** Thread {  
 **public void** run() {  
 Table.*printTable*(10);  
 }  
}  
  
**class** MyThread3 **extends** Thread {  
 **public void** run() {  
 Table.*printTable*(100);  
 }  
}  
  
  
**class** MyThread4 **extends** Thread {  
 **public void** run() {  
 Table.*printTable*(1000);  
 }  
}  
  
**class** TestSynchronization4 {  
 **public static void** main(String t[]) {  
  
 MyThread1 t1 = **new** MyThread1();  
 MyThread2 t2 = **new** MyThread2();  
 MyThread3 t3 = **new** MyThread3();  
 MyThread4 t4 = **new** MyThread4();  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 }  
}

1

2

3

4

5

6

7

8

9

10

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

10

20

30

40

50

60

70

80

90

100

* **The join() method**

The **join () method** waits for a thread to die. In other words, it causes the currently running threads to stop executing **until the thread it joins with completes its task.**

### **Syntax:**

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

**class** TestJoinMethod1 **extends** Thread {  
 **public void** run() {  
 **for** (**int** i = 1; i <= 5; i++) {  
 **try** {  
 Thread.*sleep*(500);  
 } **catch** (Exception e) {  
 System.***out***.println(e);  
 }  
 System.***out***.println(i);  
 }  
 }  
  
 **public static void** main(String args[]) {  
 TestJoinMethod1 t1 = **new** TestJoinMethod1();  
 TestJoinMethod1 t2 = **new** TestJoinMethod1();  
 TestJoinMethod1 t3 = **new** TestJoinMethod1();  
 t1.start();  
 **try** {  
 t1.join();  
 } **catch** (Exception e) {  
 System.***out***.println(e);  
 }  
  
 t2.start();  
 t3.start();  
 }  
}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

# As you can see in the above example, when **t1 completes its task then t2 and t3 starts executing.**

# **Inter-thread communication in Java**

1. **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
2. Cooperation (Inter-thread communication) is a mechanism in **which a thread is paused running in its critical section** and **another thread is allowed to enter (or lock) in the same critical section to be executed.**
3. it is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### **1) wait() method**

* Causes **current** **thread** **to release the lock** and **wait** until either another thread invokes the **notify()** method or the **notifyAll()** method for **this** **object**, or a **specified amount of time has elapsed.**
* The current thread must own this object's monitor, so it must be called from the **synchronized method only** otherwise it will throw exception.

### **2) notify() method**

* **Wakes up a single thread that is waiting on this object's monitor/lock**.
* If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

**3) notifyAll() method**

* **Wakes up all threads that are waiting** **on** **this** **object's monitor/lock.**

Syntax:

public final void notifyAll()

**Generics:**

* 1. **The** **Java Generics** programming is introduced **in J2SE 5** to deal with **type-safe** **objects**. It makes the code stable by detecting the **bugs at compile time.**
  2. Before **generics**, **we could store any type of objects in the collection**, i.e., **non-generic**.
  3. Now **generics** force the java programmer to **store a specific type of objects.**

List li = new Arraylist ();

li.add(5);

li.add(“abc”);

* 1. because **add()** method parameter used to be Object type **add(Object o)**

**While reading**

**String str = (String) li.get(1);**

* 1. **Get ()** return type used to be an Object so we needed to cast explicitly.

At 1 position there is string then it is ok. But if integer occurs then there will be **class cast exception.**

* 1. **Java 5 onwards List<String> list = new ArrayList<String>(); now compiler will force to store only String type of objects in list otherwise it will give compilation error.**
  2. **And while reading data no type casting is required.**

## Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn’t allow to store other objects. Without Generics, we can store any type of objects.

**List list = new ArrayList();**

**list.add(10);**

**list.add("10");**

**With Generics, it is required to specify the type of object we need to store.**

**List<Integer> list = new ArrayList<Integer>();**

**list.add(10);**

**list.add("10");// compile-time error**

**2) Type casting is not required:** There is no need to typecast the object.Before Generics, we need to type cast.

**List list = new ArrayList();**

**list.add("hello");**

**String s = (String) list.get(0);//typecasting**

**After Generics, we don't need to typecast the object.**

**List<String> list = new ArrayList<String>();**

**list.add("hello");**

**String s = list.get(0);**

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

**List<String> list = new ArrayList<String>();**

**list.add("hello");**

**list.add(32);//Compile Time Error**

**Generic Class:-**

* + 1. **A class that can refer to any type is known as a generic class**. Here, we are using the **T type parameter** to create the **generic class of specific type.**
    2. The **T** type indicates that it can refer to any type **(like String, Integer, and Employee)**.
    3. **The type you specify for the class** **will be used to** **store and retrieve the data.**

class MyGen<T> {  
  
 T obj;  
  
 public T getObj() {  
 return obj;  
 }  
  
 public void setObj(T obj) {  
 this.obj = obj;  
 }  
}  
  
class Test {  
 public static void main(String args[]) {  
 MyGen<String> myGen = new MyGen<>();  
 myGen.setObj("ABC");  
 System.*out*.println("Generic class example :"+myGen.getObj());  
  
 }  
}

**Generic class example : ABC**

## Generic Method

1. Like the generic class, we can create a generic method that **can accept any type of arguments.**
2. **Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.**
3. Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

class TestGenerics4 {  
  
 public static <E> void printArray(E[] elements) {  
 for (E element : elements) {  
 System.*out*.println(element);  
 }  
 System.*out*.println();  
 }  
  
 public static void main(String args[]) {  
 Integer[] intArray = {10, 20, 30, 40, 50};  
 Character[] charArray = {'J', 'A', 'V', 'A', 'T', 'P', 'O', 'I', 'N', 'T'};  
  
 System.*out*.println("Printing Integer Array");  
 *printArray*(intArray);  
  
 System.*out*.println("Printing Character Array");  
 *printArray*(charArray);  
 }  
}

**Collections in Java**

1. **The Collection in Java is a framework that provides an architecture** **to** **store and manipulate the group of objects.**
2. **The Java Collections Framework is a collection of** **interfaces** and **classes** which helps in **storing** and **processing** the **data efficiently.**
3. **Java** **Collections** **can** **achieve** **all** **the** **operations** that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion.**

**What is a framework in Java?**

* It provides readymade architecture.
* It represents a **set of** **classes** and **interfaces**.
* It is **optional**.

**Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |
|  |

**Methods of Iterator interface**

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

**Iterable Interface**

* The **Iterable** interface is the **root interface for all the collection classes**.
* The **Collection** **interface** extends the **Iterable** **interface** and therefore all the subclasses of **Collection** **interface** also implement the **Iterable** **interface**.

It contains only one abstract method. i.e.

**Iterator<T> iterator ()**

It returns the iterator over the elements of type

**Collection Interface**

* The Collection interface is the interface which is implemented by all the **classes in the collection framework.**
* It declares the methods that every collection will have.
* In other words, we can say that the **Collection** **interface** **builds the foundation on which the collection framework depends.**

1. **List Interface :-**

* **Ordered collection of data**
* **Duplicates and null** values are allowed.
* **Maintains the insertion** **order**.
* It allows **positional** **access** and **insertion** of elements.
* **Lists are further classified into the following:**

1. **ArrayList**
2. **LinkedList**
3. **CopyOnWrite ArrayList**
4. **Vectors -> stack**
5. **Set Interface :-**

* **Unordered** **collection** of data.
* It **doesn’t** **allow** **duplicate** **elements**.
* At most **one** **null** value **allowed**.
* **Set are further classified into the following:**

1. **Hash Set**
2. **Sorted set (interface) -> Tree set**
3. **Linked Hash Set**
4. **Map Interface :-**

* Allows **key-value** storage.
* **Key must be unique and values can be null.**
* **Map are further classified into the following:**

1. **Hash map :**

* **Only 1 null key is allowed and any no. of null values**
* **Not thread safe.**
* **The HashMap class does not preserve the order of insertion of entries into the map.**
* **Default capacity 16 and load factor 0.75**

1. **Hash Table :**

* **No null key** **and values are allowed.**
* **Thread** **safe. Lock will be on entire table.**
* **Default capacity 11 and load factor 0.75**

1. **Concurrent HashMap :**

* **By default 16 lock are there i.e. Thread can acquire 16 lock parallel.**
* **Lock will not be on entire HashMap.**

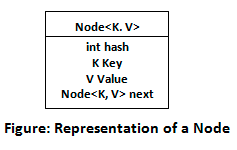
**Working of Hash Map:-**

**What is Hashing?**

* It is the process of converting an **object** into an **integer** **value**.
* The **integer** **value** helps in **indexing** and **faster** **searches**.

**What is HashMap?**

* **HashMap** is a part of **the Java collection framework. It implements the map interface.**
* It uses a technique called **Hashing**. **it** **internally** **uses** **Hash Table implementation**
* It stores the data in the pair of **Key** and **Value**. **HashMap** contains an **array of the nodes**, and the **node is represented as a class.**
* HashMap contains an **array of the nodes**, and the **node** is represented as a **class**.
* It uses an **array** and **Linked List** **data structure internally for storing Key and Value.**



Before understanding the internal working of HashMap, you must be aware of **hashCode () and equals () method.**

1. **equals ():**

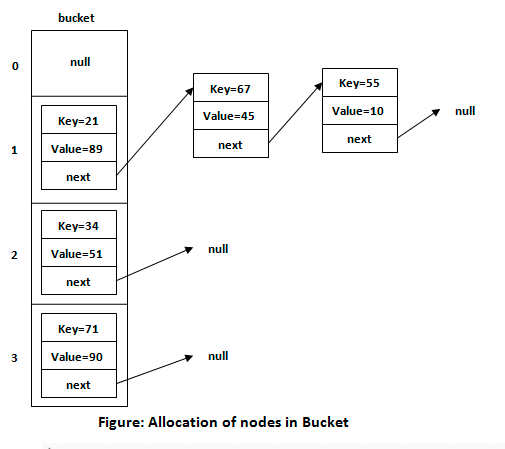
* **It checks the equality of two objects**. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden.
* If you override the **equals** () method, then it is mandatory to override the **hashCode ()** method.

1. **hashCode ():**

* This is the method of the object class. It returns the memory reference of the object in integer form.
* The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. **Hash code of null Key is 0.**

1. **Buckets:**

* **Array** **of the node is called buckets**.
* **Each** **node has a data structure like** a **Linked List**.
* **More than one node can share the same bucket. It may be different in capacity.**



**Calculating index:-**

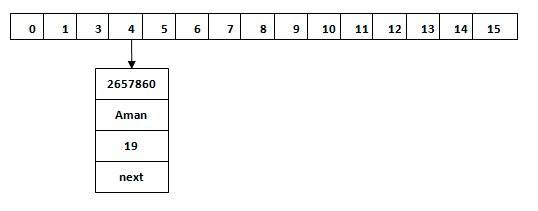
Index minimizes the size of the array. The Formula for calculating the index is:

**Index = hashcode(Key) & (n-1)**

**Where n is the size of the array. Hence the index value for "Aman" is:**

**Index = 2657860 & (16-1) = 4**

The value 4 is the computed index value where the **Key and value will store in HashMap.**



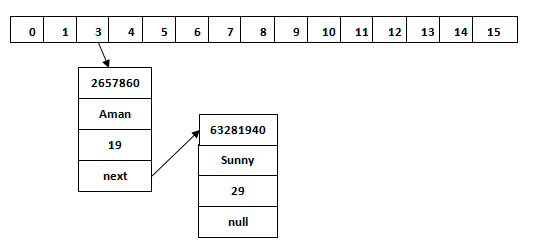
**Hash Collision:-**

* **This is the case when the calculated index value is the same for** **two or more Keys.** Let's calculate the hash code for another Key "**Sunny."**
* Suppose the hash code for "Sunny" is **63281940**. To store the Key in the memory, we have to calculate index by using the index formula.

**Hashing function:**

Index=63281940 & (16-1) = 4

* The **value** **4** is the **computed** **index** value where the **Key will be stored in HashMap.**
* **Then hashCode is used to check if there is already a key with the same hashCode or not in the bucket(singly linked list).**
* **If there already exists a key with the same hashCode,** In this case, **equals()** method check that both Keys are equal or not.
* If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the Linked List.
* Hence both Keys will be stored at index 4.



**get() method in HashMap**

* get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key.
* When get(K Key) method is called, it calculates the **hash** **code** of the **Key**.
* Suppose we have to fetch the Key "Aman." The following method will be called.

**map.get(new Key("Aman"));**

* It generates the hash code as **2657860**. Now **calculate the index value of 2657860 by using index formula.** The index value will be 4, as we have calculated above.
* get() method search for the index value 4. It compares the **first element Key** with **the given Key.**
* **If both keys are equal, then it returns the value else check for the next element in the node if it exists.**
* In our scenario, it is found as the first element of the node and return the value 19.

**Let's fetch another Key "Sunny."**

* The hash code of the Key "**Sunny**" is **63281940**. The calculated index value of **63281940** **is** **4**, as we have calculated for **put ()** method.
* Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29.
* It returns null if the next of the node is null.

**Re-Hashing:**

* Whenever the number of entries in the **HashMap** crosses the threshold value then **the bucket size of the HashMap is doubled and rehashing is performed and all already existing entries (nodes) of the map are copied and new entries are added to this increased HashMap.**
* Threshold value = Bucket size \* Load factor
* Eg. If bucket size is **16** and the load factor is **0.75** then the threshold value is **12**.

**Comparable and Comparator Interface:**

**1. Comparable Interface:-**

1. Java **Comparable** interface is used to **order/sort list of user-defined class objects.**
2. This interface is found in **java.lang package** and contains only **one method named compareTo(Object).**
3. It provides a **single** **sorting** **sequence** **only**, i.e., you can sort the elements on the basis of **single data member only**. For example, it may be roll no, name, age or anything else.

### **compareTo(Object obj) method**

**public int compareTo(Object obj):**

It is used to compare the **current object** with the **specified object**. It returns

* **positive integer, if the current object is greater than the specified object.**
* **negative integer, if the current object is less than the specified object.**
* **zero, if the current object is equal to the specified object.**

**We can sort the elements of:**

1. **String objects** (by default implement comparable interface)
2. **Wrapper class objects** (by default implement comparable interface)
3. **User-defined class objects**

**Note:**

1. **String class and Wrapper classes** implement the **Comparable interface by default.**
2. So if you store the objects of **string** **or wrapper classes** in a **list**, **set** or **map**, it will be **Comparable by default**.

* **Example with String with List :**

**public class** Test{  
  
 **public static void** main(String args[]){  
  
 List<String> list = **new** ArrayList<>();  
 list.add(**"B"**);  
 list.add(**"A"**);  
  
 Collections.*sort*(list);  
  
 **for**(String s : list){  
 System.***out***.println(**" values : "**+ s);  
 }  
 }  
}

**Output :**

Values : A

Values : B

* **Example with int array :**

class TestGenerics4 {  
  
  
 public static void main(String args[]) {  
  
 int[ ] arr = {1,5,2,6};  
  
 Arrays.*sort*(arr);  
 System.*out*.println("Array --> "+ Arrays.*toString*(arr));  
 }  
}

**output:**

Array --> [1, 2, 5, 6]

* **With user defined class object stored in List :**

**class** Student **implements Comparable<Student**> {  
  
 **int rollNo**;  
 String **name**;  
 **int age**;  
  
 Student(**int** rollNo, String name, **int** age) {  
 **this**.**rollNo** = rollNo;  
 **this**.**name** = name;  
 **this**.**age** = age;  
 }  
  
 @Override  
 **public int** compareTo(Student st) {  
 **if** (**age** == st.**age**)  
 **return** 0;  
 **else if** (**age** > st.**age**)  
 **return** 1;  
 **else  
 return** -1;  
 }  
}  
  
**class** TestSort1 {  
 **public static void** main(String args[]) {  
  
 ArrayList<Student> al = **new** ArrayList<Student>();

al.add(**new** Student(101, **"Vijay"**, 23));  
 al.add(**new** Student(106, **"Ajay"**, 27));  
 al.add(**new** Student(105, **"Jai"**, 21));  
  
 Collections.*sort*(al);  
  
 **for** (Student st : al) {  
 System.***out***.println(st.**rollNo** + **" "** + st.**name** + **" "** + st.**age**);  
 }  
 }  
}

**Output :**

105 Jai 21

101 Vijay 23

106 Ajay 27

**Comparator Interface:**-

1. Java **Comparator** **interface** is used **to order the objects of a user-defined class**.
2. This interface is found in **java.util package** and contains 2 methods

**compare(Object obj1,Object obj2)**

**equals(Object element).**

1. It provides **multiple** **sorting** sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int compare(Object obj1, Object obj2) | It compares the first object with the second object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |

## Collections class

* 1. **Collections** class provides static methods for sorting the elements of a collection. If **collection elements are of Set or Map, we can use TreeSet or TreeMap**.
  2. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

#### **Method of Collections class for sorting List elements**

#### **public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

**Example :**

**class** Student {  
 **int rollno**;  
 String **name**;  
 **int age**;  
  
 Student(**int** rollno, String name, **int** age) {  
 **this**.**rollno** = rollno;  
 **this**.**name** = name;  
 **this**.**age** = age;  
 }  
}

**class** AgeComparator **implements** Comparator<Student> {  
  
 @Override  
 **public int** compare(Student s1, Student s2) {  
 **if** (s1.**age** == s2.**age**)  
 **return** 0;  
 **else if** (s1.**age** > s2.**age**)  
 **return** 1;  
 **else  
 return** -1;  
 }  
}  
  
**class** NameComparator **implements** Comparator<Student> {  
  
 @Override  
 **public int** compare(Student s1, Student s2) {  
 **return** s1.**name**.compareTo(s2.**name**);  
 }  
}  
  
  
**class** Simple {  
 **public static void** main(String args[]) {  
  
 ArrayList<Student> al = **new** ArrayList<Student>();  
 al.add(**new** Student(101, **"Vijay"**, 23));  
 al.add(**new** Student(106, **"Ajay"**, 27));  
 al.add(**new** Student(105, **"Jai"**, 21));  
  
 System.***out***.println(**"\*\*\*\* Sorting by Name \*\*\*\*"**);  
  
 Collections.*sort*(al, **new** NameComparator());  
  
 **for** (Student st : al) {  
 System.***out***.println(st.**rollno** + **" "** + st.**name** + **" "** + st.**age**);  
 }  
  
 System.***out***.println();  
  
 System.***out***.println(**"\*\*\*\* Sorting by age \*\*\*\*"**);  
  
 Collections.*sort*(al, **new** AgeComparator());  
 **for** (Student st : al) {  
 System.***out***.println(st.**rollno** + **" "** + st.**name** + **" "** + st.**age**);  
 }  
 }  
}

**Output:**

\*\*\*\* Sorting by Name \*\*\*\*

106 Ajay 27

105 Jai 21

101 Vijay 23

\*\*\*\* Sorting by age \*\*\*\*

105 Jai 21

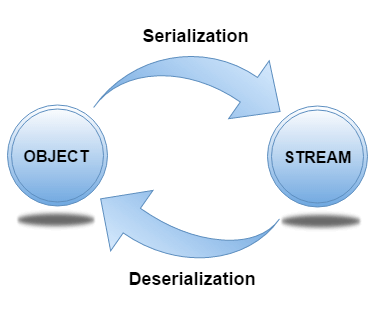
101 Vijay 23

106 Ajay 27

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| 2) Comparable **affects the original class**, i.e., the actual class is modified. | Comparator **doesn't affect the original class**, i.e., the actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is present in **java.lang** package. | A Comparator is present in the **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method. |

# **Serialization and Deserialization in Java**

1. **Serialization in Java** is a mechanism of **writing the state of an object into a byte-stream**. It is mainly used in **Hibernate, RMI, JPA, EJB and JMS technologies.**
2. The reverse operation of serialization is called **deserialization** **where byte-stream is converted into an object.**
3. The **serialization** and **deserialization** process is **platform-independent**, it means you can **serialize an object on one platform** and **deserialize it on a different platform**.
4. For **serializing** the object, we call the **writeObject()** method of **ObjectOutputStream** class For **deserialization** we call the **readObject()** method of **ObjectInputStream** class.
5. We must have to implement the **Serializable** **interface** **for serializing the object.**



## java.io.Serializable interface

1. **Serializable** is a marker interface (has no data member and method). It is used to "mark" Java classes so that the **objects of these classes may get a certain capability**.
2. The **Clonable** and **Remote** are also marker interfaces.
3. The **Serializable** interface must be implemented by the **class whose object needs to be persisted.**
4. **The String class** and all the **wrapper classes** implement the **java.io.Serializable** **interface** **by default.**

**Java 8 Features:-**

1. **Lambda Expressions:-**

* It provides a **clear** and **concise** **way** to represent **one method interface** **using an** **expression**.
* **It is very useful in collection library**. It helps to **iterate**, **filter** and **extract** data from **collection**.

**Implementation to Functional interface.**

**interface** Draw {  
 String draw();  
}  
  
  
**class** Test {  
  
  
 **public static void** main(String arge[]) {  
  
 Draw d = () -> {  
 **return "Drawing"**;  
 };  
  
 System.***out***.println(d.draw());  
  
 }  
}

**Extract data from collection.**

**class** Test {  
  
 **public static void** main(String args[]) {  
   
 List<String> list = **new** ArrayList<>();  
  
 list.add(**"A"**);  
 list.add(**"B"**);  
  
  
 list.forEach**((item) -> {  
  
 if(item.equals("A")){  
 System.*out*.println("Found A");  
 }  
  
 }**);  
   
 }  
}

1. **Functional Interface :**

* An Interface that contains exactly **one** **abstract** **method** is known as **functional interface.**
* It can have any number of **default**, **static** **methods** but can **contain only one abstract** **method**.
* It can contain any number of **Object** **class** **methods**.
* Functional Interface is also known as **Single Abstract Method Interfaces** **or SAM Interfaces.**
* It is a new feature in Java, which helps to achieve functional programming approach.

1. **Optional Class**

* It is a **public** **final** **class** and used to deal with **NullPointerException** **in Java application.**
* You must import **java.util package** to use this class.
* It provides methods which are **used to check the presence of value for particular variable.**

1. **forEach**

* Java provides a new method **forEach()** **to iterate the elements**. It is defined in **Iterable** and **Stream** **interfaces**.
* It is a **default** **method** defined in the **Iterable** **interface**. **Collection** **classes** which extends **Iterable** **interface** can use forEach() method to iterate elements.
* **This method takes a** **single** **parameter** **which** **is** **a** **functional** **interface**. **So, you** **can pass lambda expression as an argument.**

1. **Default and Static method in interface:**

* Java provides a facility to create **default methods inside the interface**.
* Methods which are defined inside the interface and tagged with default are known as default methods. **These methods are non-abstract methods**.
* The concept of default method is used to define a method with default implementation.
* You can override default method also to provide more specific implementation for the method.
* You can also define **static** **methods** inside the **interface**. **Static** **methods** are used to **define utility methods**.

**interface** Sayable {  
   
 *// default method* **default void** say() {  
 System.***out***.println(**"Hello, this is default method"**);  
 }  
  
 *// Abstract method* **void** sayMore(String msg);  
  
 *// static method* **static void** sayLouder(String msg) {  
 System.***out***.println(msg);  
 }  
}  
  
**class** DefaultMethods **implements** Sayable {  
  
 *// implementing abstract method* **public void** sayMore(String msg) {   
 System.***out***.println(msg);  
 }  
  
 **public static void** main(String[] args) {  
 DefaultMethods dm = **new** DefaultMethods();  
 dm.say(); *// calling default method* dm.sayMore(**"Work is worship"**); *// calling abstract method* Sayable.*sayLouder*(**"Helloooo..."**); *// calling static method* }  
}

**Why default methods provided?**

* The reason why default methods were included in the Java 8 release is pretty obvious.
* In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.
* Default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Thus, there's no need to modify the implementing classes.

**Why static methods provided?**

* The idea behind static interface methods is to provide **a simple mechanism that allows us to increase the degree of cohesion of a design by putting related methods together in one single place without having to create an object.**
* Furthermore, static methods in interfaces make possible to group related utility methods.

1. **Collectors**

* **Collectors is a final class that extends Object class.**
* **It provides reduction operations, such as** **accumulating elements into collections**, **summarizing elements according to various criteria, etc.**

1. **Stream API :**
2. The **Stream** **API** is used to process **collections** **of** **objects**.
3. A **stream** is **a sequence of objects that supports various methods which can be pipelined to produce the desired result.**
4. A **stream** is not a **data** **structure** instead it takes input from the **Collections, Arrays or I/O channels.**
5. Streams don’t change the **original** **data** **structure**, they only **provide the** **result as per the pipelined methods.**
6. **Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined**.
7. **Terminal operations mark the end of the stream and return the result.**

**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

**Example :**

**class** Person {  
  
 **private** String **name**;  
 **private int age**;  
  
 Person(String name, **int** age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public int** getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(**int** age) {  
 **this**.**age** = age;  
 }  
}  
  
**class** Test {  
 **public static void** main(String args[]) {  
  
 List<Person> list = **new** ArrayList<>();  
 list.add(**new** Person(**"Ajay"**, 20));  
 list.add(**new** Person(**"Vijay"**, 25));  
 list.add(**new** Person(**"Joesph"**, 35));  
  
  
 List<String> found = list.stream()  
 .filter(person -> person.getAge() > 20)  
 .map(person -> person.getName() + **" Yo"**)  
 .collect(Collectors.*toList*());  
  
 System.***out***.println(found);  
  
 }  
}

## Stream Filter

* Java stream provides a method **filter()** to filter stream elements on the basis of given predicate.
* Suppose, you want to get only even elements of your list, you can do this easily with the help of filter() method.
* This method takes **predicate** as an **argument** and returns a stream of resulted elements.

**Spring MVC :-**

The Spring Web **model-view-controller (MVC) framework** is designed around a ***DispatcherServlet*** that handles all the **HTTP requests and responses**. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram −



Following is the sequence of events corresponding to an incoming HTTP request to **DispatcherServlet** −

1. After receiving an HTTP request, **DispatcherServlet** consults the **HandlerMapping** to call the appropriate **Controller**.
2. The **Controller** takes the request and calls the **appropriate service methods based on used GET or POST method.** The service method will set model data based on defined business logic and returns **view name** to the **DispatcherServlet**.
3. The **DispatcherServlet** will take help from **ViewResolver** to pick up the **defined view for the request**.
4. Once view is finalized, The **DispatcherServlet** passes the model data to the **view** which is finally **rendered on the browser.**

**Spring Framework**

1. **Spring** is the most popular **application development framework for enterprise Java.**
2. Millions of developers around the world use Spring Framework to create high **performing**, **easily** **testable**, and **reusable** **code**.
3. **Spring framework is an open source Java platform.**
4. It was initially written by **Rod Johnson** and was **first released under the Apache 2.0 license in June 2003.**
5. **Spring** is a **lightweight** framework. It can be thought of as a framework of frameworks because **it provides support to various frameworks such as**[**Struts**](https://www.javatpoint.com/struts-2-tutorial)**,**[**Hibernate**](https://www.javatpoint.com/hibernate-tutorial)**, Tapestry,**[**EJB**](https://www.javatpoint.com/ejb-tutorial)**,**[**JSF**](https://www.javatpoint.com/jsf-tutorial), etc
6. The Spring framework comprises several modules such as **IOC, AOP, DAO, Context, ORM, WEB MVC etc**. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first

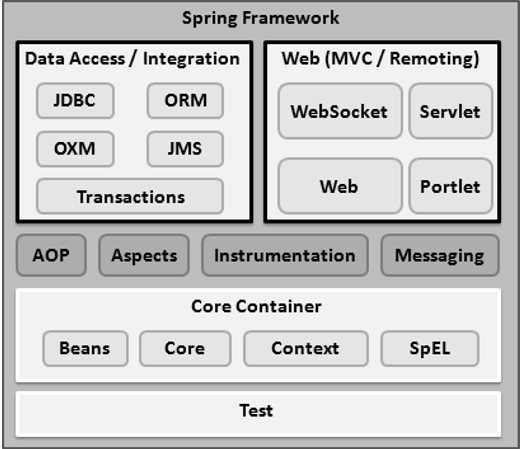
**Dependency Injection** and **AOP** are main concepts in spring framework.

**Dependency Injection**

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the **IoC**.

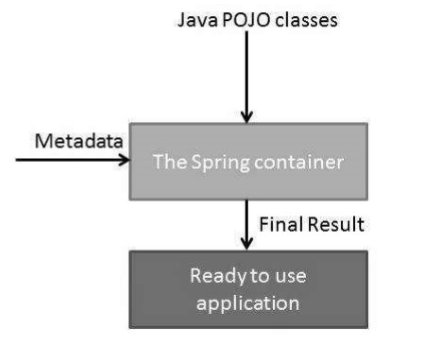
**AOP**

One of the key components of spring is the **Aspect Oriented Programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.



**Spring IOC Container:**

1. The **Spring** **IoC** **container** is at the **core of the Spring Framework**. The container will **create the objects**, **wire them together**, **configure them**, and **manage their complete life cycle from creation till destruction.**
2. The spring container uses DI to manage the components that make up an application. These **objects** are called **Spring** **Beans.**
3. It **injects** **dependencies** when a **bean is created** and **manages the bean life cycle** **during execution**.
4. The container gets its **instructions** on **what objects to instantiate**, **configure**, **and** **assemble by reading the configuration metadata provided**.
5. The **configuration metadata** can be represented either **by XML, Java annotations, or Java code**
6. The Spring IoC container makes use of **Java POJO classes** **and configuration metadata** to **produce a fully configured and executable system or application**.



1. The fundamental tasks of Spring IoC are:

* **Instantiating**
* **Configuring, and**
* **Assembling Bean**

1. Spring provides two types of IOC containers:
   1. **Bean Factory**
   2. **Application Context**

The ***ApplicationContext* container** includes all functionality of the ***BeanFactory*container,** so it is generally recommended over ***BeanFactory***.

**BeanFactory** can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

**Beans:**

* 1. The **objects** that **form the backbone of your application** and that are **managed by the Spring IoC container** are called **beans**.
  2. A **bean** is an **object** that is **instantiated**, **assembled**, and **otherwise managed by a Spring IoC container.**
  3. **These** **beans** **are created** with the **configuration** **metadata that you supply to the** **container**.

**Dependency Injection types:-**

1. Dependency Injection (DI) is a **design** **pattern** used to implement IoC.
2. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways.
3. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class **(dependent class)** is a class which depends on the service class
2. **Service Class:** The service class **(dependency)** is a class that provides service to the client class.
3. **Injector Class:** The injector class injects the service class object into the client class.

**Constructor Injection:** In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

**Setter Injection:** In the property injection the injector supplies the dependency through a public property of the client class.

**Method Injection:** In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

**Spring Boot:-**

1. **Spring** **Boot** is a project that is built on the top of the **Spring** **Framework**. It provides an easier and faster way to set up, configure, and run **both simple and web-based applications.**
2. **Spring** **Boot** is a spring module that provides the **RAD** (Rapid Application Development) feature to the **spring framework.**

**@SpringBootApplication**

* 1. If we're writing a Spring Boot application, it is helpful to know that **@SpringBootApplication** is a composed annotation that includes **@ComponentScan**.
  2. As long as our **@SpringBootApplication** class is at the root of our project, it will scan every **@Component** we define by default.

**@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Configuration**

**@ComponentScan**

1. **@ComponentScan** ensures that the classes decorated with **@Component** **are found and** **registered as spring beans**.
2. **@ComponentScan** is **automatically** included with **@SpringBootApplication**

**@EnableAutoConfiguration**

1. Spring Boot **auto-configuration automatically** configures the **spring application based on the jar dependencies that we have added.**
2. For example, if the H2 database Jar is present in the class path and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project.
3. We can **enable the auto-configuration feature by using the annotation** **@EnableAutoConfiguration**. But this annotation does not use because it is wrapped inside the **@SpringBootApplication** annotation.

**@springbootconfiguration**

1. When we mark a class with **@SpringBootConfiguration**, it means that the class provides **@Bean definition methods**.
2. **The spring container processes the** **configuration** **class** **to instantiate and configure beans for our application.**

### **Configuration vs @SpringBootConfiguration:**

As per the spring documentation, @SpringBootConfiguration is just an alternative to the spring standard @Configuration annotation. The only**difference between the two is that the @SpringBootConfiguration allows the configuration to be found automatically.**

**@Component**

1. **@Component** is the most generic **spring** **annotation**. A class annoted with **@Component** is found during **class path** **scanning and registered in the context as a spring bean.**
2. **@Service, @Repository, and @Controller are specializations of @Component,** **which are used for more specific cases.**

**@Configuration**

**Use @Configuration annotation on top of any class to declare that this class provides one or more @Bean methods** **and may be processed by the spring container to generate bean definitions** **and service requests for those beans at runtime.**

**@Bean**

1. **Spring @Bean annotation** tells that a **method** **produces** **a** **bean** **to** **be managed by the spring container.**
2. **It is a** **method**-**level** **annotation**. **During** **Java** **configuration** **(@Configuration),** **the** **method is executed and its return value is registered as a bean within a Bean Factory**.
3. **@Bean** serves a similar purpose as **@Component**. It is not auto detected. **Methods decorated with @Bean produce a bean to be managed by the spring container during configuration stage**.
4. The @**Bean** annotation is used to **explicitly declare a bean creation.**

**Bean Scopes:-**

1. **Singleton** –

* Only one instance of the spring bean will be created for the spring container.
* This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
* During the application context initialization a single bean will be created even if we don’t request for it.

@Service  
**public class** Fifa **implements** Game {  
  
 @Override  
 **public void** Running() {  
 System.***out***.println(**"Fifa is running"**);  
 }  
}

@SpringBootApplication  
**public class** SpringTestApplication {  
  
 **public static void** main(String[] args) {  
 ApplicationContext applicationContext = SpringApplication.*run*(SpringTestApplication.**class**, args);  
  
 Fifa fifa1 = (Fifa) applicationContext.getBean(**"fifa"**);  
 System.***out***.println(**"fifa1 hashcode : "** + fifa1.hashCode());  
  
 Fifa fifa2 = (Fifa) applicationContext.getBean(**"fifa"**);  
 System.***out***.println(**"fifa2 hashcode : "** + fifa2.hashCode());  
 }  
  
}

fifa1 hashcode : 1503614751

fifa2 hashcode : 1503614751

1. **Prototype** –

* A **new** **instance** will be created every time the **bean** is requested from the **spring container.**
* Object is created only when we request from a **spring** **container** otherwise no object will be created in container.

@Service  
@Scope(**"prototype"**)  
**public class** Fifa **implements** Game {  
  
 @Override  
 **public void** Running() {  
 System.***out***.println(**"Fifa is running"**);  
 }  
}

@SpringBootApplication  
**public class** SpringTestApplication {  
  
 **public static void** main(String[] args) {  
 ApplicationContext applicationContext = SpringApplication.*run*(SpringTestApplication.**class**, args);  
  
 Fifa fifa1 = (Fifa) applicationContext.getBean(**"fifa"**);  
 System.***out***.println(**"fifa1 hashcode : "** + fifa1.hashCode());  
  
 Fifa fifa2 = (Fifa) applicationContext.getBean(**"fifa"**);  
 System.***out***.println(**"fifa2 hashcode : "** + fifa2.hashCode());  
 }  
  
}

fifa1 hashcode : 1402333753

fifa2 hashcode : 353891891

1. **Request** – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each **HTTP** **request**.
2. **Session** – A new bean will be created for each **HTTP** **session** by the container.
3. **global-session** – This is used to create global session beans for Portlet applications

**ORM Framework:**

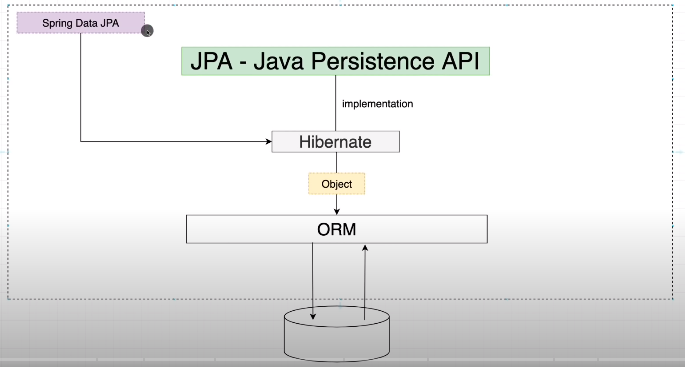
1. **Object** **Relational** **Mapping** (ORM) is a functionality which is used to **develop** and **maintain a relationship between an object and relational database by mapping an object state to database column.**
2. It is capable to handle various database operations easily such as **inserting**, **updating**, **deleting** etc.

**Following are the various frameworks that function on ORM mechanism: -**

* Hibernate
* TopLink
* ORMLite
* iBATIS
* JPOX

**JPA**

1. **The Java Persistence API (JPA) is a specification of Java. It is used to persist data** between **Java object** and **relational database.**
2. **JPA** acts as a **bridge** between **object-oriented domain models** **and** **relational database systems.**
3. **As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation.**
4. **So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.**
5. **Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools.**
6. The **javax.persistence** package contains the JPA classes and interfaces.



Why should we use JPA?

1. JPA is **simpler**, **cleaner**, and less labour-intensive than JDBC, SQL, and hand-written mapping.
2. **JPA** is suitable **for non-performance oriented complex applications.**
3. The main advantage of JPA over JDBC is that, in JPA, **data** is represented by **objects** and **classes** while in **JDBC** **data** is represented by **tables** and **records**.
4. It uses POJO to represent persistent data that simplifies database programming. There are some other advantages of JPA:

* JPA avoids writing DDL in a database-specific dialect of SQL. Instead of this, it allows mapping in XML or using Java annotations.
* JPA allows us to avoid writing DML in the database-specific dialect of SQL.
* JPA allows us to save and load Java objects and graphs without any DML language at all.
* When we need to perform queries JPQL, it allows us to express the queries in terms of **Java entities** **rather than the (native) SQL table and columns.**

## ORM Tool

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



## Hibernate Framework

1. **Hibernate** is a **Java** **framework** that **simplifies the development of Java application to interact with the database.**
2. It is an **open** **source**, **lightweight**, **ORM** (Object Relational Mapping) tool.
3. **Hibernate implements the specifications of JPA (Java Persistence API) for data persistence**

JPA Mapping Directions

Mapping Directions are divided into two parts: -

1. **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owning side that specifies how an update can be made in the database.
2. **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

Following are the various **ORM** mappings: -

* **One-to-one -** This **association** is represented by **@OneToOne** annotation. Here, **instance of each entity is related to a single instance of another entity.**
* **One-to-many -** This **association** is represented by **@OneToMany** annotation. In this relationship, **an instance of one entity can be related to more than one instance of another entity.**
* **Many-to-one -** This mapping is defined by **@ManyToOne** annotation. In this relationship, **multiple instances of an entity can be related to single instance of another entity.**
* **Many-to-many -** This **association** is represented by **@ManyToMany** annotation. **Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side**.

**JPA Entity:**

1. **Entities** in **JPA** are nothing but **POJOs** **representing data that can be persisted to the database.**
2. **An entity represents a table stored in a database**.
3. **Every instance (objects) of an entity represents a row in the table.**

# **JPA JPQL Introduction**

1. **The** **JPQL** **(Java Persistence Query Language) is** **an** **object-oriented query language** which is used to **perform database operations on persistent entities.**
2. **Instead** **of** **database** **table**, **JPQL** **uses** **entity** **object** **model** **to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL.**
3. Thus, it provides an easy platform for developers to **handle SQL tasks.**
4. **JPQL** is an extension of Entity JavaBeans Query Language (EJBQL), adding the following important features to it: -

* It can perform join operations.
* It can update and delete data in a bulk.
* It can perform aggregate function with sorting and grouping clauses.
* Single and multiple value result types.

## JPQL Features

* It is a platform-independent query language.
* It is simple and robust.
* It can be used with any type of database such as MySQL, Oracle.
* JPQL queries can be declared statically into metadata or can also be dynamically built in code.

**JPA Cascading Operations :-**

* 1. In JPA, if any operation is applied on an entity then it will perform on that particular entity only. These operations will not be applicable to the other entities that are related to it.
  2. To establish a dependency between related entities, JPA provides javax.persistence.CascadeType enumerated types that define the cascade operations.
  3. These cascading operations can be defined with any type of mapping i.e. **One-to-One, One-to-Many, Many-to-One, Many-to-Many.**

|  |  |
| --- | --- |
| **Cascade Operations** | **Description** |
| PERSIST | In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted. |
| MERGE | In this cascade operation, if the parent entity is merged then all its related entity will also be merged. |
| DETACH | In this cascade operation, if the parent entity is detached then all its related entity will also be detached. |
| REFRESH | In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed. |
| REMOVE | In this cascade operation, if the parent entity is removed then all its related entity will also be removed. |
| ALL | In this case, all the above cascade operations can be applied to the entities related to parent entity. |

**What is API?**

1. It is **(Application Programming Interface)** interface which allows **digital** **devices**, **software** **applications**, and **data** **servers** to **talk with each other**
2. APIs are a set of **functions** and **procedures** that allow for the creation of applications that **access data** and **features** of other **applications**, **services**, or **operating systems.**
3. It is code written by someone else and we are using it in our application.
4. To avoid the miss use of **API**, **API** **key** can be applied to make it secure.
5. Sometimes we create a utility code (JAR file) and use it another code for reusability this is also an API.

**Example**:

* Zomato calling **Google** **map** **API** which **then talks to the google server** to show the live location of delivery valet to customers.
* We can implement **sign up with google** **functionality** in our application by simply calling **google signup API**.

All the **web** **services** are **API** but all the **APIs** are not **web services.**

**Web Services:**

* It is a **client-server application** or **application** **component** for **communication**.
* The method of communication between two devices over the network.
* It is a software system for the **interoperable** **machine to machine communication.**
* It is a **collection** of **standards** and **protocols** for exchanging information between **two system and application.**
* Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet

**There are two types:**

1. SOAP web Service (XML).
2. REST web service (JSON).

Web Service Components

There are three major web service components.

1. SOAP
2. WSDL
3. UDDI

SOAP

1. SOAP is an acronym for **Simple Object Access Protocol.**
2. SOAP is a XML-based protocol for accessing web services.
3. SOAP is a W3C recommendation for communication between applications.
4. SOAP is XML based, so it is platform independent and language independent. In other words, it can be used with Java, .Net or PHP language on any platform.

WSDL

1. WSDL is an acronym for Web Services Description Language.
2. WSDL is a xml document containing information about web services such as method name, method parameter and how to access it.
3. WSDL is a part of UDDI. It acts as a interface between web service applications.
4. WSDL is pronounced as **wiz-dull.**

UDDI

1. UDDI is an acronym for Universal Description, Discovery and Integration.
2. UDDI is a XML based framework for describing, discovering and integrating web services.
3. UDDI is a directory of web service interfaces described by WSDL, containing information about web services.

SOAP Web Services

1. **SOAP** stands for **Simple Object Access Protocol**. It is a **XML-based protocol** for **accessing web services.**
2. SOAP is a W3C recommendation for communication between two applications.
3. SOAP is XML based protocol. It is platform independent and language independent.
4. By using SOAP, you will be able to interact with other programming language applications.

**Advantages of Soap Web Services**

* **WS Security**: SOAP defines its own security known as WS Security.
* **Language and Platform independent**: SOAP web services can be written in any programming language and executed in any platform.

**Disadvantages of Soap Web Services**

* **Slow**: SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So it is slow and consumes more bandwidth and resource.
* **WSDL dependent**: SOAP uses WSDL and doesn't have any other mechanism to discover the service.

**RESTful Web Services (REST API)**

1. REST stands for **Representational State Transfer**.
2. REST is an **architectural** **style** not a **protocol**.

**Advantages of RESTful Web Services**

* **Fast**: RESTful Web Services are fast because there is no strict specification like SOAP. It consumes less bandwidth and resource.
* **Language and Platform independent**: RESTful web services can be written in any programming language and executed in any platform.
* OOPs Concepts in Java
* **Can use SOAP**: RESTful web services can use SOAP web services as the implementation.
* **Permits different data format**: RESTful web service permits different data format such as Plain Text, HTML, XML and JSON.

**Micro Services:-**