# OOPs (Object Oriented Programming System)

It is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

# Naming Convension

|  |  |  |
| --- | --- | --- |
| **Identifier Type** | **Rules for Naming** | Example |
| Packages | name is always written in **all-lowercase** | com.sun.eng |
| Classes | first letter in **Capital,** Internal words starts with **capital** | MyClass |
| Methods | first letter **lowercase,** Internal words starts with **capital** | myMethod |
| Variables | Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters. | I;j;k |
| Constants | all uppercase with words separated by underscores ("\_") | MIN\_WIDTH |

# Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

### Object Definitions:

* Object is a real world entity.
* Object is a run time entity.
* Object is an entity which has state (property) and behavior(method).
* Object is an instance of a class.

# Class

Collection of objects is called class. It is a logical entity.

## Constructor

In Java, constructor is a block of codes similar to method. It is called when an instance of object is created, and memory is allocated for the object.

### Rules for creating java constructor

There are basically two rules defined for the constructor.

1. ***Constructor name*** *must be same as its class name*
2. *Constructor must have* ***no explicit return type***
3. *Constructor cannot be*[***abstract***](https://www.javatpoint.com/abstract-class-in-java)*,*[***final***](https://www.javatpoint.com/final-keyword)*,*[***synchronized***](https://www.javatpoint.com/synchronization-in-java)*, and*[***static***](https://www.javatpoint.com/static-keyword-in-java)*. We cannot* ***override*** *a constructor.*

### Types of java constructors

There are two types of constructors in java:

### Default constructor

The default constructor is a constructor that the Java compiler adds to your code if no explicit constructor is available. The default constructor **invokes the super class constructor with no args first**.**if you want to call parent constructor (with and without argument) explicitly then that will be first statement.**

If you have added your own constructor (no matter whether it's without parameters or with parameters) the compiler will not add the default constructor in this case.

* if the class is declared public, then the default constructor is implicitly given the access modifier public;
* if the class is declared protected, then the default constructor is implicitly given the access modifier protected;

#### no-arg constructor

Default constructor is used to provide the default values to the object like 0, null etc. depending on the type.

**Default value set by default constructor**

|  |  |
| --- | --- |
| **int** | 0 |
| **float** | 0.0 |
| **String** | Null |
| **boolean** | FALSE |

Class **conscls**{

int a;

String b;

**conscls**(){

System.out.println("constructor is called : default value " + a +" and " +b);

}}

conscls obj=new conscls();

**Note**: **Default constructor and no-arg constructor are not same.**

#### Parameterized constructor

Class cons\_with\_para{

cons\_with\_para(inti, String b){

System.out.println("constructor with parameter is called : value " + i +" and "+b);

}}

Main()--

cons\_with\_para obj1 =new cons\_with\_para(111,"Hello 1");

#### Constructor Overloading in Java –(Default + parameterized constructer)

**Class** cons\_overload

{

cons\_overload(){

System.***out***.println("constructor is called without parameter");

}

cons\_overload(**int**i, String b){

System.***out***.println("constructor with parameter is called : value " + i +" and " +b);

}}

Main()--

cons\_overloadobj2=**new** cons\_overload();

cons\_overloadobj3=**new** cons\_overload(111,"Hello 1");

If there is no constructor in a class, compiler automatically creates a default constructor and access modifier will be same as class modifier.

#### Difference between constructor and method in java

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |
| constructors cannot be abstract, final, native, static, or synchronized. | Methos have all non-access modifier |
| constructors can have any of the access modifiers: public, protected, private, or none |  |

#### Copy Constructor

Here is no copy constructor in java. But, we can copy the values of one object to another. Same Class object is passed in constructor.

**Why**-That's helpful when we want to copy a complex object that has several fields, or when we want to make a deep copy of an existing object. There is also a condition, if we have made any changes in the copy it should not reflect in the original one and vice-versa.

**Class** copyclass

{

**Int** i;

String b;

copyclass(**int** i,String b){

this.i=i;

this.b=b;

}

copyclass(copyclass another){

this(another.i,another.b)

}}

Main()—

Copyclass obj4=**new**copyclass(11,"My world");

Copyclass copyobj4=**new** copyclass(obj4);

### Can we have private constructors ?

As you can easily guess, like any method we can provide access specifier to the constructor. If it’s made private, then it can only be accessed inside the class. If a method is private, it means that it can not be accessed from any class other than itself. When a class needs to prevent the caller from creating objects. Private constructors are suitable. 1` Objects can be constructed only internally.

### Do we need such private constructors?

1. Internal Constructor chaining

2. Singleton class design pattern

### What is a Singleton class?

As the name implies, a class is said to be singleton if it limits the number of objects of that class to **one**.

We can’t have more than a single object for such classes.

#### Why Singleton class is used

You use a singleton when you need to manage a shared resource. For instance a **printer spooler**. Your application should only have a single instance of the spooler in order to avoid conflicting request for the same resource.

Or a **database connection** or a **file manager** etc.

I would use it also to retrieve and store informations on external configuration files.

* + **Printer spooler**
  + **database connection**
  + **file manager**

#### ****Examples of Singleton class****

* **java.lang.Runtime**
* **java.awt.Desktop**
* **Logger**

#### How to create Singleton class?

1. Constructor must be private
2. There must be a static method which is accessed outside class without object and method must return **object**.
3. There must be **private** **static object** variable of same class type.(**why**:it should be used in static method)

**public** **class** Singleton {

**private** Singleton() {

}

**private** **static** Singleton *obj* = **null**;

**public** **static** Singleton getInstance() {

**if** (*obj* == **null**)

*obj* = **new** Singleton();

**return** *obj*;

} }

// Driver Class

class Main

{

public static void main(String args[])

{

MySingleton a = MySingleton.getInstance();

MySingleton b = MySingleton.getInstance();

a.x = a.x + 10;

System.out.println("Value of a.x = " + a.x);

System.out.println("Value of b.x = " + b.x);

}

}

We changed value of a.x, value of b.x also got updated because both ‘a’ and ‘b’ refer to same object, i.e., they are objects of a singleton class.

# Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism(method overriding).

Inheritance is the object oriented programming concept where an object is based on another object. Inheritance is the mechanism of code reuse. The object that is getting inherited is called **superclass** i.e **Parent** (**base** class**)** and the object that inherits the superclass is called **subclass** i.e **Child(derived** class, **extended** class).

### Why inheritance

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

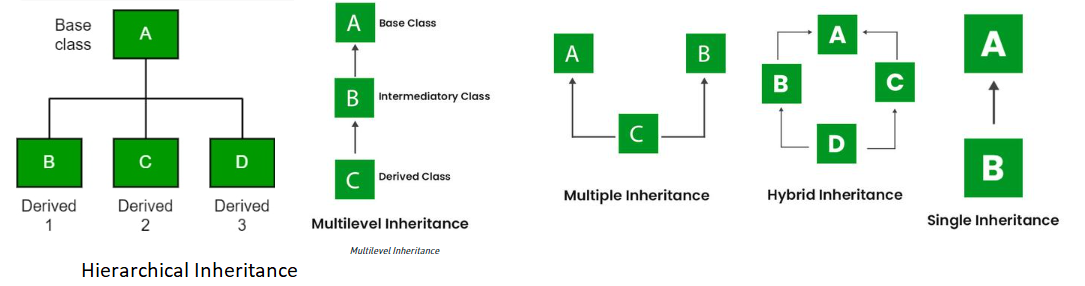
We use **extends** keyword in java to implement inheritance. Below is a simple example of inheritance in java.

## Types of Inheritance

1. Single
2. Multilevel
3. Hierarchical

**Multiple** inheritances is not supported in java through class. **it is possible using interface**.

**Hybrid** inheritance is a mix of two or more of the above types of inheritance. Since java doesn’t support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).



### Single Inheritance

In single inheritance, subclasses inherit the features of one superclass.

Class A {

public void foo(){

System.out.println("SuperClassA");

}}

Class B extends A{

public void bar(){

System.out.println("SubClassB");

}}

Main()--

B a = new B();

a.foo();

a.bar();

#### Inheritance and constructors in Java

* In Java, constructor of base class with no argument gets automatically called in derived class constructor.
* In case of constructor of base class with argument, you must implement this constructor. Base class constructor with argument call must be the first line in derived class constructor

**class** Derived extends Base {

Derived(int x) {

super(x);

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

}

* In case of both base constructor(without argument) and constructor(with argument), base class with no argument gets automatically called.

### Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.

**class** parent{

**void**p\_m(){System.***out***.println("This is parent");}

}

**class** child\_level1 **extends** parent{

**void** c1\_m() {System.***out***.println("this is child level1");}

}

**class** child\_level2 **extends** child\_level1{

**void** c2\_m() {System.***out***.println("this is child level2");}

}

Main()--

child\_level2 obj=**new** child\_level2();

obj.p\_m();

obj.c1\_m();

obj.c2\_m();

### Hirarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

**class** animal1{

**void** A() {System.***out***.println("This is Animal");}

}

**class** dog1 **extends** animal1{

**void** dog11(){ System.***out***.println("This is dog");}

}

**class** cat1 **extends** animal1

{**void** cat11(){System.***err***.println("This is cat");}

}

Main()—

cat1c=**new** cat1();

c.A();

dog1d=**new** dog1();

d.A();

### Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

Example:

Consider the below Java code. It shows error.

|  |
| --- |
| // First Parent class  classParent1  {      voidfun()      {          System.out.println("Parent1");      }  }    // Second Parent Class  classParent2  {      voidfun()      {          System.out.println("Parent2");      }  }    // Error : Test is inheriting from multiple  // classes  classTest extends **Parent1, Parent2**  {     publicstaticvoidmain(String args[])     {         Test t = newTest();         t.fun();     }  } |

## ****Important facts about inheritance in Java****

<https://www.geeksforgeeks.org/inheritance-in-java/>

* **Default superclass**: Except [**Object**](https://www.geeksforgeeks.org/object-class-in-java/) class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the [**Object**](https://www.geeksforgeeks.org/object-class-in-java/) class.
* **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. Although with interfaces, multiple inheritances are supported by java.
* Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. **Constructors are not members, so they are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass.
* **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

# Polymorphism

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

While Polymorphism means one name but many forms. For example, you have a smartphone for communication. The communication mode you choose could be anything. It can be a call, a text msg, a picture msg, mail, etc. So the goal is common that is communication, but their approach is different. This is calledPolymorphism**.**

In java, we use method overloading and method overriding to achieve polymorphism.

The word “poly” means many and “morphs” means forms

## Types of polymorphism

1. compile time polymorphism -overloading
2. Runtime polymorphism. -overriding

## Compile time Polymorphism

#### What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

### Method overloading

Method overloading is nothing but having two or more methods in the same class with the same name but different **number/type** of arguments. **Method overloading is resolved using static binding at compile time**. To overload a method, you must either need to change number of argument, type of argument or order of argument if they are of different types.

Since return type is not part of method signature simply changing return type will not be considered as method overloading.

**class** overload{

**int** add(**int**x,**int**y) {**return**x+y;}

**double** add(**double**x,**int**y) {**return** (x+ y);}

}

Main()—

System.***out***.println(obj.add(1, 2));

System.***out***.println(obj.add(1.1, 2));

## Runtime Polymorphism

Dynamic polymorphism in Java is achieved by method overriding. Polymorphism allows you define one interface and have multiple implementations.

### Method Overriding

Method overriding is nothing but having two methods with the same name, same arguments, but different implementations. Method overriding can be done in sub class only. So, to override a method, you need to create a child class which extends parent class and create a method in child class with same name and same arguments. The return type can be same type as that of method in parent class or it can be a sub type of it. The method implementation to bind to the method call is resolved at run-time based on object using dynamic binding in Java.

class Parent {

void Print() {

System.out.println("parent class");

} }

class Child1 extends Parent {

void Print() {

System.out.println("Child1");

} }

class Child2 extends Parent {

void Print() {

System.out.println("Child2");

} }

class TestPolymorphism3 {

public static void main(String[] args)

{ Parent a;

a = new Child1();

a.Print();

a = new Child2();

a.Print();

**If type class and reference class is different**

Parent obj=new Child();

* Common method(overridden method) will be called from **child class**
* UnCommon method will be called from **parent class**

# [When do you use Java's @Override annotation and why?](https://stackoverflow.com/questions/94361/when-do-you-use-javas-override-annotation-and-why)

Use it every time you override a method for two benefits. Do it so that you can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does. Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten

# [When do you use Java's @SuppressWarnings annotation and why?](https://stackoverflow.com/questions/94361/when-do-you-use-javas-override-annotation-and-why)

@SuppressWarnings instruct the compiler to ignore or suppress, specified compiler warning in annotated element and all program elements inside that element. For example, if a class is annotated to suppress a particular warning, then a warning generated in a method inside that class will also be separated. You might have seen @SuppressWarnings("unchecked") and @SuppressWarnings("serial"), two of most popular examples of @SuppressWarnings annotation. Former is used to suppress warning generated due to unchecked casting while the later warning is used to remind about adding SerialVersionUID in a Serializable class.

**Can we Override static method? Explain with reasons ?**

**No**, we cannot override static method. Because static method is always bound to class whereas method overriding is associated with object i.e at runtime.

We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class (**Method Hiding**)

**Method Hiding**

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass. This mechanism happens because the static method is resolved at the **compile time**. Static method bind during the compile time using the **type of reference not a type of object**.

**If type class and reference class is different for static method**

Parent obj=new Child();

* Common method(overridden method) will be called from **Parent class**
* UnCommon method will be called from **parent class**

class Complex {

public static void f1()

{System.out.println("f1 method of the Complex class is executed.");

}}

class Sample extends Complex {

public static void f1()

{ System.out.println("f1 of the Sample class is executed.");

}}

public class Main {

public static void main(String args[])

{

Complex d1 = new Complex();

Complex d2 = new Sample();

d1.f1();// o/p: f1 method of the Complex class is executed.

d2.f1();// o/p: f1 method of the Complex class is executed.

}}

**Field Hiding**

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through **super**.

**Can we inherit static method/Variable?**

Yes, static method can be inherited in the sub class but they belong to the class in which they have been declared.

In subclass, static variable and method will be directly used without class Name or without creating object.

outside from super and subclass,static method/variable can be access using superclass or subclass.

public class Test2 {

public static void main(String[] args) {

System.out.println(staticExample2.name);

System.out.println(staticExample.name);

staticExample.print();

staticExample2.print();

}

}

class staticExample

{

static String name="khalid";

static void print()

{

System.out.println("this is khalid");

}

}

class staticExample2 extends staticExample

{

void print2()

{

System.out.println(name); //directly accessed

System.out.println(staticExample.name);

print();

staticExample.print();

}

}

**Can we overload static methods?**

The answer is ‘Yes’. We can have two ore more static methods with same name, but differences in input parameters.

**Can we overload methods that differ only by static keyword?**

We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same)

**An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.**

// Superclass

Class Base {

     // Static method in base class which will be hidden in subclass

    Public **static** void display() {

        System.out.println("Static or class method from Base");

    }

     // Non-static method which will be overridden in derived class

     Public void print()  {

         System.out.println("Non-static or Instance method from Base");

    }

}

In case of inheritance, either override or overload, compile error if anything different

// Subclass

Class Derived extends Base {

    // Static is removed here (Causes Compiler Error)

    Public void display() {

        System.out.println("Non-static method from Derived");

    }

    // Static is added here (Causes Compiler Error)

    Public static void print() {

        System.out.println("Static method from Derived");

   }

}

**In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.**

# Abstraction --(interface,abstract class/method)

Abstraction is the process of selecting important data sets for an Object in your software, and leaving out the insignificant ones.

Once you have modeled your object using Abstraction, the same set of data could be used in different applications.

**--Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use **abstract class** and **interface** to achieve abstraction.

1. Abstract class (0 to 100%)
2. Interface (100%)

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.

There are many ways to achieve abstraction in object oriented programming, such as encapsulation and inheritance.

--abstraction is the quality of dealing with ideas rather than events.

**Definition:**

Abstract class which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user

**Use of abstraction:**

they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them.The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

## 5.1 Abstract Method

Consider, a programmer trying to implement a functionality - displaying a menu on right click of a mouse button- and you might not know what all the items that should be displayed on the right click as its application specific i.e., the programmer knows how to display but not what to display. In these kind of scenarios you write an Abstract method (Abstract class) and forces everyone to inherit it.

--It is a process of hiding implementation details and showing only functionality to the user.

* A method without body (no implementation) is known as abstract method.
* A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well.
* A method must always be redefined in the subclass, thus making **overriding compulsory**
* Method **can not be private**

abstract void moveTo(double deltaX, double deltaY);

public class Abstract {

public static void main(String[] args) {

MouseEventevt=new Notepad();

evt.DisplayMenu();

//--------

MouseEvent evt1=new Wordpad();

evt1.DisplayMenu();}}

**abstract class** MouseEvent{

**abstract void** DisplayMenu();} //--Only signature is applicable

**class** Notepad **extends** MouseEvent{

**void** DisplayMenu(){ //abstract method must be implemented

System.***out***.println("list of menu in notepad"); } }

**Class** Wordpad **extends** MouseEvent{

**void**DisplayMenu(){

System.***out***.println("list of menu in Wordpad");

}}

## Abstract Classes (only inheritance)

* if a class has at least one pure virtual function(without body), then the class becomes abstract.
* an **instance** of an abstract class cannot be created, we can have references of abstract class type
* we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.
* Abstract classes can also have final methods (**methods that cannot be overridden**).
* An *abstract class* is a class that is declared abstract—it may or may not include abstract methods.
* If a class includes abstract methods, then the class itself *must* be declared abstract
* an abstract class can have parametrized constructors and default constructor is present in an abstract class.

Interface with abstract method—it will used only after implementation in a class

Abstract class with method implementation--—it will used only after inheritance in a class

Ex-1

public abstract class GraphicObject {

// declare fields

// declare nonabstract methods

abstract void draw();

}

### Abstract class with argument constructor

* We need to make sure that the class which is extending an abstract class have a constructor and it can call the superclass parameterized constructor.
* We can call the superclass parameterized constructor in a subclass by using super() call.
* If we are not placing super() call in the subclass constructor, a compile-time error will occur.

**public** **abstract** **class** Parent3 {

Parent3(**int** i)

{

System.***out***.println("Parent3 constructor is created");

}

}

**public** **class** Child3 **extends** Parent3{

Child3(**int** i) {

**super**(i);

// **TODO** Auto-generated constructor stub

}

## Can abstract method be protected?

Yes, except private, an abstract method can be public, protected or default (in interface all methods should be abstract and public).

# What is an Interface?

an interface is a reference type,implicitly abstract(no need to use the abstract keyword ) similar to a class, that can contain only

* constants --implicitly public, **static** and final
* abstract method --implicitly public and abstract (**no static**) (no need to use abstract keyword)
* default methods--implicitly public(no abstract) (**no abstract**)
* static methods--implicitly public (**no abstract**)
* nested interfaces--implicitely public and static.
* nested class – any access modifier
* An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

An interface can be defined as a container that stores the signatures of the methods to be implemented in the code segment. It improves the levels of Abstraction.

To use an interface in your class, append the keyword "**implements**" after your class name followed by the interface name.

class Dog implements Pet

interface RidableAnimal extends Animal, Vehicle

### ****Why do we use interface ?****

* Total Abstraction
* Multiple Inheritance
* Loose-Coupling
* An interface is 100% abstract class and has only abstract methods.
* Only public and default access modifier are used
* Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
* An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* An interface is a contract (or a protocol, or a common understanding) of what the classes can do.

**Multiple Inheritance**

Without Interface, the process of multiple inheritances is impossible as the conventional way of inheriting multiple parent classes results in profound ambiguity. This type of ambiguity is known as the Diamond problem. Interface resolves this issue.

**Loose Coupling**

The term Coupling describes the dependency of one class for the other. So, while using an interface, we define the method separately and the signature separately. This way, all the methods, and classes are entirely independent and archives Loose Coupling.

## [Why are interface variables static and final by default?](https://stackoverflow.com/questions/2430756/why-are-interface-variables-static-and-final-by-default)

(Only Variable… Not Method)

Since interface doesn't have a direct object, the only way to access them is by using a **class/interface name** and hence that is why if interface variable exists, it should be static otherwise it won’t be accessible at all to outside world. Now since it is static, it can hold only one value and any classes that implements it can change it and hence it will be all mess.

Hence if at all there is an interface variable, it will be implicitly **static, final and obviously public**

## Using an Interface as a Data Type

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

## Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

**Interface** MotorVehicle

{

**void** run();

**void** getFuel();

}

// My team mate complies and writes vehicle looking that way

**class** Car **implements** MotorVehicle

{

**int**fuel=10;

**publicvoid** run()

{System.***out***.println("Wrroooooooom"); }

**Publicvoid**getFuel()

{

//return this.fuel;

//return fuel;

System.***out***.println("Hello");

}

}

**Public class** A\_Interface {

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

MotorVehicle cc=**new** Car();

cc.run();

cc.getFuel();

}}

## Interface Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The extends keyword is used to **extend** an interface, and the child interface inherits the methods of the parent interface.

Public interface Sports

{

Public void setHomeTeam(String name);

publicvoidsetVisitingTeam(String name);

}

// Filename: Football.java

Public interface Football extends Sports{

Public void homeTeamScored(int points);

Public void visitingTeamScored(int points);

Public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

Public interface Hockey extends Sports{

Public void homeGoalScored();

Public void visitingGoalScored();

Public void endOfPeriod(int period);

Public void overtimePeriod(intot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

## Interface Extending Multiple Interfaces—Multiple Inheritance

A Java class can only extend one parent class.Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

**Example**

PublicinterfaceHockey **extends** Sports,Event

**what if two interface methods clash in implementation**

If a type implements two interfaces, and each interface define a method that has identical signature, then in effect there is only one method, and they are not distinguishable. If, say, the two methods have conflicting return types, then it will be a compilation error.

**Difference between Interface and Abstract class**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class**can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class**can be extended using keyword "extends". | An **interface class**can be implemented using keyword "implements". |
| 8) A Java**abstract class**can have class members like private, protected, etc. | Members of a Java interface are public by default. |

1. Abstract classes can have constants, members, method stubs (methods without a body) and defined methods, whereas interfaces can only have constants and methods stubs.
2. Methods and members of an abstract class can be defined with any visibility, whereas all methods of an interface must be defined as public (they are defined public by default).
3. When inheriting an abstract class, a concrete child class must define the abstract methods, whereas an abstract class can extend another abstract class and abstract methods from the parent class don't have to be defined.
4. Similarly, an interface extending another interface is not responsible for implementing methodsfrom the parent interface. This is because interfaces cannot define any implementation.
5. A child class can only extend a single class (abstract or concrete), whereas an interface can extend or a class can implement multiple other interfaces.

## When to use Interface or Abstract Class

I will give you an example first:

public interface LoginAuth{

public String encryptPassword(String pass);

public void checkDBforUser();

}

Now suppose you have 3 databases in your application. Then each and every implementation for that database needs to define the above 2 methods:

public class DBMySQL implements LoginAuth{

// Needs to implement both methods

}

public class DBOracle implements LoginAuth{

// Needs to implement both methods

}

public class DBAbc implements LoginAuth{

// Needs to implement both methods

}

But what if encryptPassword() is not database dependent, and it's the same for each class? Then the above would not be a good approach.

Instead, consider this approach:

public abstract class LoginAuth{

public String encryptPassword(String pass){

// Implement the same default behavior here

// that is shared by all subclasses.

}

// Each subclass needs to provide their own implementation of this only:

public abstract void checkDBforUser();

}

Now in each child class, we only need to implement one method - the method that is database dependent.

## Java 8 Interface Changes

1. Lambda expressions—related to anonymous class
2. Method references, —related to anonymous class
3. Functional interfaces —related to anonymous class
4. Stream API -- useful in collection
5. Default methods,
6. Base64 Encode Decode,
7. Static methods in interface,
8. Optional class,
9. Collectors class,
10. ForEach() method,
11. Parallel array sorting,
12. Nashorn JavaScript Engine,
13. Parallel Array Sorting,
14. Type and Repating Annotations,
15. IO Enhancements,
16. Concurrency Enhancements,
17. JDBC Enhancements etc.

### static method, default method

Consider an interface that you have developed called DoIt:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

}

Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

boolean didItWork(int i, double x, String s);

}

If you make this change, then all classes that implement the old DoIt interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly

you can define your new methods as default methods. The following example defines a default method named didItWork:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

default boolean didItWork(int i, double x, String s) {

// Method body

}

}

Note that you must provide an implementation for default methods. You could also define new static methods to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

### Accessing default Method in any class

InterfaceObj.DefaultMethodName()

### Accessing Static Method in any class

InterfaceName.StaticMethodName()

Deal with default method problem

We know that Java doesn’t allow us to extend multiple classes because it will result in the “Diamond Problem” where compiler can’t decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too. Because if a class is implementing both Interface1 and Interface2 and doesn’t implement the common default method, compiler can’t decide which one to chose.

So to make sure, this problem won’t occur in interfaces**,**

**it’s made mandatory to provide implementation for common default methods of interfaces.**

## if two interface have same static method/default method

1. Default methods must be overriden in implementing class, while static cannot.
2. It helps in extending interfaces without having the fear of breaking implementation classes.
3. Static method belongs only to Interface class, so you can only invoke static method by InterfaceName.staticMethodName, not on class implementing this Interface.
4. Common static method will not create problem as it is always accessed by interfaceName.static name which will always be unique.
5. Static methods are part of interface, we can’t use it for implementation class objects.
6. Both implementing class and interface can have static methods with same names, and neither overrides other

Example:

public class InterfaceStaticMethod implements Intf1,Intf2 {

public static void main(String[] args) {

Intf1 intf1=new InterfaceStaticMethod();

intf1.print();

Intf1.common();

Intf2.common();

}

@Override

public void print() {

Intf1.super.print();

Intf2.super.print();

System.out.println("this is common default method");

}

}

interface Intf2 {

void abc();

default void print()

{

System.out.println("This is interface 2");

}

static void common()

{

System.out.println("this is common method for intf2");

}

}

interface Intf1 {

void abc();

default void print()

{System.out.println("This is interface 1");

}

static void common()

{System.out.println("this is common method for intf1");

}

}

## if two interface and super class have same default method

## Instance methods are preferred over interface default methods.

public class Horse {

public String identifyMyself() {

return "I am a horse.";

}

}

public interface Flyer {

default public String identifyMyself() {

return "I am able to fly.";

}

}

public interface Mythical {

default public String identifyMyself() {

return "I am a mythical creature.";

}

}

public class Pegasus extends Horse implements Flyer, Mythical {

public static void main(String... args) {

Pegasus myApp = new Pegasus();

System.out.println(myApp.identifyMyself());

}

}

## Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

public interface Animal {

default public String identifyMyself() {

return "I am an animal.";

}

}

public interface EggLayer extends Animal {

default public String identifyMyself() {

return "I am able to lay eggs.";

}

}

public interface FireBreather extends Animal { }

public class Dragon implements EggLayer, FireBreather {

public static void main (String... args) {

Dragon myApp = new Dragon();

System.out.println(myApp.identifyMyself());

}

}

The method Dragon.identifyMyself returns the string I am able to lay eggs.

|  |  |  |
| --- | --- | --- |
| Defining a Method with the Same Signature as a Superclass's Method | | |
|  | **Superclass Instance Method** | **Superclass Static Method** |
| **Subclass Instance Method** | Overrides | Generates a compile-time error |
| **Subclass Static Method** | Generates a compile-time error | Hides |

References: <https://docs.oracle.com/javase/tutorial/java/IandI/override.html>

# Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines. Encapsulation is a mechanism to hide information from the client. The information may be data or implementation or algorithm. We achieve this **using access modifiers.**

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

For example, encapsulation in java is achieved using **private**, **protected** and **public** keywords.Itrestrict access to data at different levels.

### Data Hiding

It just provides a way to protect your data from the outside world. What it means is, lets say if I made my instance variable public, then anyone can change its state. But if we make our instance variable private/protected then actually we are restricting outside entities from making changes to it.

## How to achieve encapsulation in Java

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

public class EncapTest {

private int age;

public int getAge() {

return age;

}

public void setAge( int newAge) {

age = newAge;

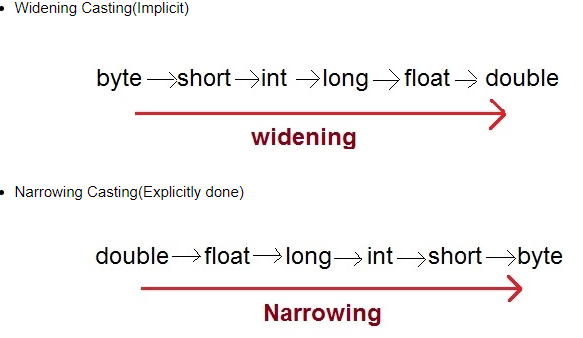
}}

## Benefits of Encapsulation

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.

# Type conversion in Java – Type Cast

Type casting in Java is to cast one **type**, a **class** or **interface**, into another type i.e. another class or interface. You can type cast both **primitive** and **reference type** in Java.  
When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.

Automatic conversion is supported from numeric type to char or Boolean. Also, char and Boolean are not compatible with each other.\

### How to do Explicit Conversion?

byte b;

int i = 257;

b = (byte) i;

--

double d = 100.04;

long l = (long)d;

--

byte b = 50;

b = (byte)(b \* 2);

### Rule for typecasting

* There is a rule in Java Language that classes or interface which shares the same type hierrachy only can be typecasted. If there is no relationship between then Java will throw **ClassCastException**.
* A reference variable can refer to an object if the object is of the same type as a variable or if it is a subtype
* Upcasting happens implicitly
* All Java objects are polymorphic and can be treated as objects of supertype due to upcasting
* only classes or interfaces from the same type hierarchy can be cast or converted into each other.

**ClassCastException**

If you try to cast two objects which don't share same type hierarchy, i.e. there is no parent-child relationship between them, you will get compile time error. On the other hand, if you typecast objects from same type hierarchy but the object which you are casting are not of the same type on which you are casting then it will throw ClassCastException in Java.

ClassCastException is quite common while using Java collection framework classes e.g. ArrayList, LinkedList or HashSet etc because they accept object of type java.lang.Object, which allows insertion of any object into collection.

**Why do we need type casting?**

you need type casting to get access to fields and methods declared on target type or class. You can not access them with any other type.

### Upcasting

**Parent** obj=new **child()**-- Parent obj can access only those **child** method which are common(override)

Casting from a subclass to a superclass is called upcasting. Typically, the upcasting is implicitly performed by the compiler.Upcasting is closely related to inheritance – another core concept in Java. It’s common to use reference variables to refer to a more specific type. And every time we do this, implicit upcasting takes place.

To demonstrate upcasting let’s define an Animal class:

public class Animal {

public void eat() {

// ...

}}

Now let’s extend Animal:

public class Cat extends Animal {

public void eat() {

// ...

}

public void meow() {

// ...

}}

public class Dog extends Animal {

public void eat() {

// ...

}}

Now we can create an object of Cat class and assign it to the reference variable of type Cat:

Cat cat = new Cat();

And we can also assign it to the reference variable of type Animal:

Animal animal = cat; //Parent~ Child

In the above assignment, implicit upcasting takes place. We could do it explicitly:

animal = (Animal) cat;

But there is no need to do explicit cast up the inheritance tree. The compiler knows that cat is an Animal and doesn’t display any errors.

Note, that reference can refer to any subtype of the declared type.

Using upcasting, we’ve restricted the number of methods available to Cat instance but haven’t changed the instance itself. Now we can’t do anything that is specific to Cat – we can’t invoke meow() on the animal variable.

Although Cat object remains Cat object, calling meow() would cause the compiler error:

// animal.meow(); The method meow() is undefined for the type Animal

To invoke meow() we need to downcast animal, and we’ll do this later.

But now we’ll describe what gives us the upcasting. Thanks to upcasting, we can take advantage of polymorphism.

For above; <https://www.baeldung.com/java-type-casting>

List<Animal> animals = new ArrayList<>();

animals.add(new Cat());

animals.add(new Dog());

We add cats and dogs and they are upcast to Animal type implicitly. Each Cat is an Animal and each Dog is an Animal. They’re polymorphic.

**By the way, all Java objects are polymorphic because each object is an Object at least. We can assign an instance of Animal to the reference variable of Object type and the compiler won’t complain:**

**Object object = new Animal(); //Object is parent class**

That’s why all Java objects we create already have Object specific methods, for example, toString().

Upcasting to an interface is also common.

### Why is Upcasting?

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype.

### Downcasting

Parent obj= new **child()**-- Parent obj can access only those method which are common

If you want to access extra method which are not in parent method then above code + below

( (**ChildClassName**) Parent\_obj )**.**childExtraMethod();

**ChildClassName** Obj=(**ChildClassName**) Parent\_obj

What if we want to use the variable of type Animal to invoke a method available only to Cat class? Here comes the downcasting. It’s the casting from a superclass to a subclass.

Animal animal = new Cat();

We know that animal variable refers to the instance of Cat. And we want to invoke Cat’s meow() method on the animal. But the compiler complains that meow() method doesn’t exist for the type Animal.

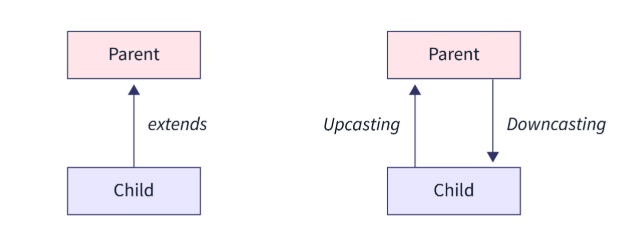
To call meow() we should **downcast** animal to Cat:

((Cat) animal).meow(); //After upcasting , downcasting is allowed

The inner parentheses and the type they contain are sometimes called the cast operator. Note that external parentheses are also needed to compile the code.

### Difference between Upcasting and Downcasting

* Upcasting is casting a subtype to a supertype , upward to the inheritance tree. Downcasting is casting to a subtype, downward to the inheritance tree.
* Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.
* Casting does not change the actual object type. Only the reference type gets changed.
* Upcasting is always safe and never fails.
* Downcasting can risk throwing a ClassCastException, so the **instanceof** operator is used to check type before casting.



### ClassCastException

ClassCastException’s always thrown at runtime if the type we downcast to doesn’t match the type of the real object. e.g

we are trying to convert an object which is an instance of Dog into a Cat instance.

Animal animal;

String s = (String) animal; //classCastException will be thrown

The compiler says “Cannot cast from Animal to String”.

For the code to compile, both types should be in the same **inheritance tree.**

**Example**

class Cat extends Animal

class Dog extends Animal

--

Animal anm1 = new Cat();

((Dog) anm1).bark(); // anm1 is refering cat obj so throw ClassCastException

### Down Casting properties

* Downcasting is necessary to gain access to members specific to subclass
* Downcasting is done using cast operator
* To downcast an object safely, we need instanceof operator
* If the real object doesn’t match the type we downcast to, then ClassCastException will be thrown at runtime

**Instanceof** keyword

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).It returns either true or false.

* If we apply the instanceof operator with any variable that has null value, it returns false.

class Simple1{

public static void main(String args[]){

Simple1 s=new Simple1();

System.out.println(s instanceof Simple1);//true

}

}

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

# Difference between JDK, JRE and JVM

JVM (Java Virtual Machine) is an abstract machine. The JVM performs following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

### JRE

JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. JVM + set of libraries

### JDK

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

# Static Keyword

In the Java programming language, the keyword static indicates that the particular member belongs to a type itself, rather than to an instance of that type.This means that **only one instance of that static member is created which is shared across all instances of the class.**

Staticmembers can only refer to static members.

* A static Method can’t access a non static variable nor can directly invoke non static Method (It can invoke or access Method or variable via instances).
* This and super cannot be used in static context.
* Static is a Non Access Modifier.
* **Static keywords can be used with variables and Methods**. It is not applicable to outer Class but inner class
* Variable or Methods marked static belong to the Class rather than to any particular Instance.
* Static Method or variable can be used without creating or referencing any instance of the Class.

## Applicable to

* Method
* Variable -- private static int ballCount;
* Class nested within another Class
* Initialization Block (Static block)

## Not Applicable to

The Static keyword cannot be applied to

* Class (Not Nested)
* Constructor
* Interfaces
* Method Local Inner Class(Difference then  nested class)
* Inner Class methods
* Instance Variables
* Local Variables
* **static method cannot be inherited in the sub class because they belong to the class in which they have been declared.**

## Why java doesn’t support static constructor?

Lets back to the point, since each constructor is being called by its subclass during creation of the object of its subclass, so if you mark constructor as static the **subclass** will not be able to access the constructor of its parent class because it is marked static and thus belong to the class only. This will violate the whole purpose of inheritance concept and that is reason why a constructor cannot be static.

### Rule

**When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.**

classTest

{

    // static method

    Static void m1()    {        System.out.println("from m1");    }

     Public static void main(String[] args)

    {

          // calling m1 without creatingany object of class Test

                m1();

    }

}

## can method return multiple values java?

You can return an **object of a Class** in Java. **List, Set, Map**

If you are returning more than 1 value that are related, then it makes sense to encapsulate them into a class and then return an object of that class.

If you want to return unrelated values, then you can use Java's built-in container classes like Map, List, Set etc.

## When to use static variables and methods?

Use the static variable for the property that is common to all objects. For example, in class Student, all students shares the same college name. Use static methods for changing static variables.

## Static Block

Static block is used for initializing the static variables.This block gets executed when the class is loaded in the memory. **A class can have multiple Static blocks**, which will execute in the same sequence in which they have been written into the program.

### Single Static Block

classJavaExample{

static int num;

static String mystr;

static{

num = 97;

mystr = "Static keyword in Java";

}}

### Multiple Static blocks

Lets see how multiple static blocks work in Java. They execute in the given order which means the first static block executes before second static block. That’s the reason, values initialized by first block are overwritten by second block.

classJavaExample2{

staticint num;

staticString mystr;

//First Static block

static{

System.out.println("Static Block 1");

num = 68;

mystr = "Block1";

}

//Second static block

static{

System.out.println("Static Block 2");

num = 98;

mystr = "Block2";

}

Public static void main(String args[])

{

System.out.println("Value of num: "+num);

System.out.println("Value of mystr: "+mystr);

}

}

## Instance Initialization Block (IIB) in Java

In a Java program, operations can be performed on methods, constructors and initialization blocks. IIBs are executed before constructors. They run each time when object of the class is created.

* Initialization blocks are executed whenever the class is initialized and before constructors are invoked.
* They are typically placed above the constructors within braces.
* It is not at all necessary to include them in your classes.

**Single initializing block**

classGfG

{

    // Instance Initialization Block

    {

        System.out.println("IIB block");

    }

    // Constructor of GfG class

    GfG()

    {

        System.out.println("Constructor Called");

    }

    publicstaticvoidmain(String[] args)

    {

        GfG a = newGfG();

    }

}

**Multiple Initializing block**

classGfG

{

    // Instance Initialization Block - 1

    {

        System.out.println("IIB1 block");

    }

    // Instance Initialization Block - 2

    {

        System.out.println("IIB2 block");

    }

    // Constructor of class GfG

    GfG()

    {

        System.out.println("Constructor Called");

    }

    // Instance Initialization Block - 3

    {

        System.out.println("IIB3 block");

    }

    // main function

    publicstaticvoidmain(String[] args)

    {

        GfG a = newGfG();

    }

}

# Variable Declaration and initialization

Variables are used to represent values that may be changed in the program. In Java, all variables must be declared before they can be used.

Declaration can be done only once but initialization can be done multiple times.

* Declaration means creating address in RAM
* Initialization means storing value in RAM

**int** age; // Declaration of variable age

age = 20; // initialization of the variable age

int age=20; // Declaration and initialization of the variable age

# This keyword

This is a **reference variable** that refers to the current object.Here is given the 6 usage of java this keyword.

1. this can be used to refer **current class instance variable**.
2. this can be used to invoke **current class method** (implicitly)
3. this() can be used to invoke **current class constructor**.
4. this can be passed as an **argument in the method** call.
5. this can be passed as **argument in the constructor** call.
6. this can be used to **return the current class instance from the method**.

**Rule: Call to this() must be the first statement in constructor.**

**This keyword can not be used inside static method (like main method).**

## This keyword with Instance Variable

**class** Student{

**int** rollno;

String name;

**Double** fee;

Student(**int** rollno,String name,**double** fee){

**this**.rollno=rollno; --rollno is local variable whereas this.rollno is instance variable.

**this**.name=name;

**this**.fee=fee;

}

void display(){System.out.println(rollno+" "+name+" "+fee);} }

Mani()--Student obj=**new** Student(111,"kk",203.3);obj.display();

Previous output--0 null 0.0

Current output-- 111 kk 203.3

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

## This keyword with method

**Class** cls\_method

{

**void** m() {

System.***out***.println("This is m");}

**void** n(){

m();

**this**.m();}}

Main()--cls\_method obj1=new cls\_method();obj1.n();

## This Keyword with constructor

**Class**currcons{

currcons(){System.***out***.println("this is default constructor");}

currcons(**int** a)

{

**this**();

System.***out***.println(a);}}

Main()-- curr\_cons obj2=**new**curr\_cons(10);

* Call to this() must be the first statement in constructor.
* Call to this() must be inside one of constructor method(no-arg OR arg constructor)
* You can not two times i.e this(2); this(2,3). Only single constructor call

## ****Calling parameterized constructor from default constructor****

**class** A{

A(){

**this**(5);

System.***out***.println("hello a");

}

A(**int** x){

This(2,3);

System.***out***.println(x);

}

A(**int** x, int y){

System.***out***.println(x); }

}

Main() --A obj3=**new** A();

## This keyword for inner class

As this key is used for current class, for accessing outer class, **OuterClassName**.**this** will be used.

public class ShadowTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

System.out.println("x = " + x);

System.out.println("this.x = " + this.x);

System.out.println("ShadowTest.**this**.x = " + ShadowTest.this.x);

}}

# Super Keyword

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by **super** keyword variable.

The**super** keyword in java is a reference variable that is used to refer parent class objects. The keyword “super” came into the picture with the concept of Inheritance.

## Usage

1. super can be used to refer immediate parent class **instance variable**.
2. super can be used to invoke immediate parent class **method**.
3. super() can be used to invoke immediate parent class **constructor**.
4. Super() can be used only in constructor
5. Private methods of the super-class cannot be called.
6. Super keyword are not used in **static Method**.
7. super must be the first statement in a constructor

## super is used to refer immediate parent class instance variable.

This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

Example, janwarand kuttaboth classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

**class** janwar {

String color="white";

}

**class** kutta **extends** janwar{

String color="black";

**void** printColor(){

System.***out***.println(color);//prints color of kutta class

System.***out***.println(**super**.color);//prints color of Animal class

}}

Main()---kutta obj=**new** kutta();obj.printColor();

## ****Use of super in case of method overriding****

This is used when we want to call parent class method. So whenever a parent and child class have same methods then to resolve ambiguity we use super keyword.

/\* Base class Person \*/

classPerson

{

    Void message()

    {   System.out.println("This is person class");     }

}

/\* Subclass Student \*/

classStudent extends Person

{

    Void message()

    {

        System.out.println("This is student class");

    }

    // Note that display() is only in Student class

    Void display()

    {

        // will invoke or call current class message() method

        message();

        // will invoke or call parent class message() method

        super.message();

    }

}

### Use of Super if child class is not overriding any method

When child class doesn’t override the parent class method then we don’t need to use the super keyword to call the parent class method. This is because in this case we have only one version of each method and child class has access to the parent class methods so we can directly call the methods of parent class without using super.

## ****Use of super with parent classconstructors****

super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parameter as well as non parametric constructors depending upon the situation.

Super();

Super(1,”hello”);

**Note**: Invocation of a superclass constructor must be the first line in the subclass constructor.

## Super key: Calling default method of interface

interface myi {

int y = 200;

void abc();

default void defMathod()

{

System.out.println("this is default method");

}

static void interfaceStaticMathod()

{

System.out.println("this is interfaceStaticMathod method");

}

}

--

public class SuperExample extends ParentForSuper implements myi {

SuperExample() {

myi.super.defMathod();

myi.*interfaceStaticMathod*();

}

## Super key: Inner class inheritance

public class SuperExample2 extends OuterClass.innerClass {

SuperExample2() { new OuterClass().super();

}

public static void main(String[] args) {

SuperExample2 obj = new SuperExample2();

obj.innerClassMethod();

System.out.println(obj.x);

}

}

class OuterClass {

class innerClass {

int x = 10;

void innerClassMethod() {

System.out.println("This is innerClassMethod");

}}

}

# Final Keyword

final is a [non-access modifier](https://www.geeksforgeeks.org/access-and-non-access-modifiers-in-java/) applicable **only to a variable, a method or a class**.

Final Variable 🡪 To create **constant variable**

Final Method 🡪 **Prevent Method overriding**

Final Class 🡪 **Prevent Inheritance**.

**Final Variable**

The only difference between a normal variable and a final variable is that we can **re-assign value** to a normal variable but we cannot change the value of a final variable once assigned. Hence, final variables must be used only for the values that we want to remain constant throughout the execution of program.This also means that you must initialize a final variable.

Final int speedlimit = 5;

1. As discussed above, a final variable cannot be reassign
2. Instance variable can be initialized in constructor, initializing block, static block(if static final)
3. **Reference final variable ( object type is final)**

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed.

final StringBuilder sb = new StringBuilder("Geeks");

sb.append("ForGeeks");

final object of class can not be assigned to another object.

final int arr[] = {1, 2, 3, 4, 5}; // Note: arr is final

for (int i = 0; i < arr.length; i++)

{

arr[i] = arr[i]\*10;

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

}

The array arr is declared as final, but the elements of array are changed without any problem. Arrays are objects and object variables are always references in Java.

1. When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created.

public static void main(String args[])

{

// local final variable

final int i;

i = 20;

System.out.println(i);

}

**Final methods**

When a method is declared with final keyword, it is called a final method. A final method cannot be [overridden](https://www.geeksforgeeks.org/overriding-in-java/). The [Object](https://www.geeksforgeeks.org/object-class-in-java/) class does this—a number of its methods are final.We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes.

class A

{

final void m1()

{

System.out.println("This is a final method.");

}

}

class B extends A

{

void m1()

{

// COMPILE-ERROR! Can't override.

System.out.println("Illegal!");

}

}

**Final Class**

If you make any class as final, you cannot extend it.For example, all [Wrapper Classes](https://www.geeksforgeeks.org/wrapper-classes-java/) like [Integer](https://www.geeksforgeeks.org/java-lang-integer-class-java/),[Float](https://www.geeksforgeeks.org/java-lang-float-class-in-java/) etc. are final classes.

**Final class** Bike {}

**class** Honda1 **extends** Bike // it gives compilation error

{

**void** run(){System.***out***.println("running safely with 100kmph");} }

**Advantage**

* To prevent the inheritance of class
* **To make an Immutable class**

**immutable class**

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like String, Boolean, Byte, Short,Integer, Character) and String class is immutable.

The other use of **final** with classes is to [create an immutable class](https://www.geeksforgeeks.org/create-immutable-class-java/) like the predefined [String](https://www.geeksforgeeks.org/string-class-in-java/) class.You can not make a class immutable without making it **final**.

Following are the requirements to create immutable class  
• Class must be declared as **final** (So that child classes can’t be created)  
• Data members in the class must be declared as **final** (So that we can’t change the value of it after object creation)  
• A parameterized constructor to set initial value  
• Getter method for all the variables in it  
• No setters(To not have option to change the value of the instance variable)

// An immutable class

public final class Student

{

final String name;

final int regNo;

public Student(String name, int regNo) {

this.name = name;

this.regNo = regNo;

}

public String getName() {

return name;

}

public int getRegNo() {

return regNo; }

}

### Property of Immutable class

Immutable means that once the constructor for an object has completed execution that instance can't be altered.

This is useful as it means you can pass references to the object around, without worrying that someone else is going to change its contents. Especially when dealing with concurrency, there are no locking issues with objects that never change.

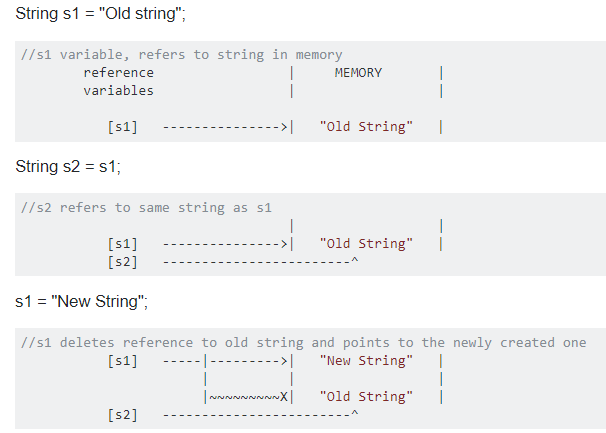
An immutable object is an object where the internal fields cannot be changed.

### Advantages

**Concurrency**: If the internal structure of an immutable object is valid, it will always be valid. There's no chance that different threads can create an invalid state within that object. Hence, immutable objects are Thread Safe.

**Garbage collection**: It's much easier for the garbage collector to make logical decisions about immutable objects.

**HashCode:**Immutable objects will always have the same Hash code, so they can be used as the keys in a HashMap (or similar). If the hash code of an element in a hash table was to change, the table entry would then effectively be lost, since attempts to find it in the table would end up looking in the wrong place. This is the main reason that String objects are immutable - they are frequently used as HashMap keys.



**Is final method inherited?**

Ans) **Yes**, final method is inherited but you cannot override it.

**Can we declare a constructor final?**

**No**, because constructor is never inherited.

**Can a java abstract method be final and abstract both?**

No, an abstract class method cannot be final and abstract both in java.

Reason is that, by making a method final in abstract class, we are stopping derived class not to override and implement it. Whereas, by making it abstract, we are forcing derived class to override it and implement it. So, making a method both final and abstract is contradictory. Hence, using both final and abstract method does not make any sense.  
Secondly, final method must have a body whereas abstract method can have only declaration. So, it is also contradictory.

**Difference between final, finally and finalize in Java???**

The java.lang.Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation.

Since Object class contains finalize method hence finalize method is available for every java class since Object is superclass of all java classes. Since it is available for every java class hence Garbage Collector can call finalize method on **any java object**

FinalizeExample cal = **new** FinalizeExample();

// finalize cal

System.***out***.println("Finalizing...");

cal.finalize();

System.***out***.println("Finalized.");

# Short Cut Key

You can view all shortcut of eclipse Windows>>preference>>(Search Key)

Ctrl + shift + / ---Collapse all code

Ctrl + shift + \* --Expanse all code

Ctrl + shift + L ----gives us list of shortcut key name and its functionality

Ctrl + shift + R—searching file within work space

Ctrl + shift + T—searching class,function and others within work space

Ctrl+shift+F—format your code

Alt + UP/Down arrow –move selected line/lines up and down

F4---to view class or function hirarchy

F3 – go to defintion/implemention of method

Ctrl+M—maximize main java file and remove all other views like explorer,problem,outline etc

Ctr + T –To view class or function hirarchy with less information

Ctr + D – delete line without selecting line or delete selected line/lines

Ctr + L – go to specfic line number

Ctr + Q – will take you to last edited line even though you were at different position

Ctr + O – outlione of current calss

Alt + Shift + X—

Alt+Shift +M --- would extract method based on certain selected functionality

Alt+Shift+I –(inline)would delete extracted method to previous one

Alt+Shift+R—select keyword and press these keyword and change name, it would replace all occurance

Alt+shift+L---list of expression would be replaced with new variable and expression is put.

Alt+Shift+C—To change method signature

Alt+Shift+M—to move method into other class

Alt+shift+T—select some expression and see list of actions to be performed

 first one gives to highrachy of a prticular class /method. Select and check for more details

Right one gives you all keyword in class for selected word

**Code Generation**

Alt+Shift+S -- Select Private variable and select “Generate Getter and Setter”

## [Replace String in all files in Eclipse](https://stackoverflow.com/questions/6800799/replace-string-in-all-files-in-eclipse)

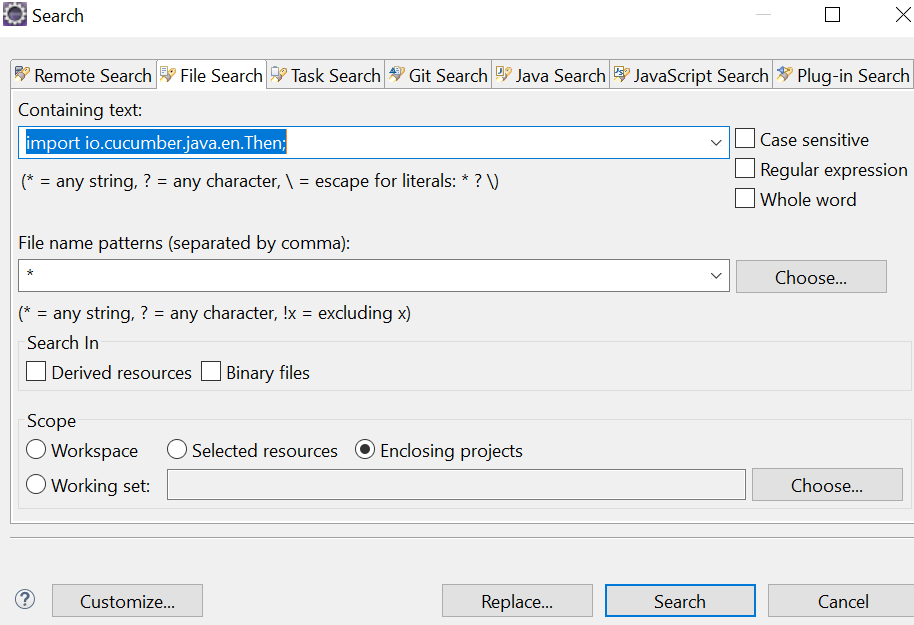
<http://www.avajava.com/tutorials/lessons/how-do-i-do-a-find-and-replace-in-multiple-files-in-eclipse.html>

Step1: Click on “Search” and select “File”

Step2: Enter Search text in section “Containing text:”

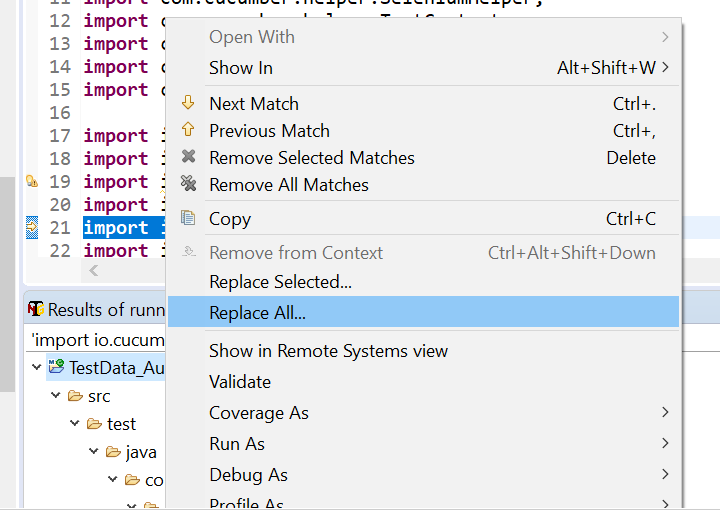
Step3: choose “Enclosing project” in section “Scope”

Step4: Click on “Search” button



Step5: Select project directory and “search” result and right click and select all

Step6: Enter replacing string and click Ok



# Why java main method is static?

|  |
| --- |
| Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation. What exactly is an instance in Java? class A {  int x=0;  public static void main(String [] args)  {  int y=0;  y=y+1;  x=x+1;  } }  Let us compile and run this code.  step 1: javac A.class (.class file is generated which is byte code)  step 2: java A (.class file is converted into executable code)  During the step 2,The main method and the static elements are loaded into the RAM for execution. In the above scenario, No issue until the line y=y+1. But whenever x=x+1 is executed, the run time error will be thrown as the **JVM does not know what the x is which is declared outside the main method(non-static).**  So If by some means the content of .class file is available in the memory for CPU to execute, there is no more issue.  This is done through creating the Object and the keyword **NEW** does this Job.  **The concept of reserving memory in the RAM for the contents of hard disk (here .class file) at runtime is called Instance**  The Object is also called the instance of the class. |

# Variables and Data Types in Java

There are three types of variables in java:

1. **Local variable**:
   * It is declared inside the method is called local variable.
   * You can use the variable only within that method.
   * Other methods in the class aren’t even aware that the variable exists.
   * You can not use static on a declaration for a local variable.
   * You can not use access modifier for local variable.
   * Unlike class and instance variables (0 for int,NULL for String),a local variable is not initialized by default.it must be initialized before first use.
   * int x = 5, y = 10,z; Multiple variable declartion separated by comma(,)
2. **Instance variable**: It is declared inside the class but outside the method, is called instance variable. It is not declared as static.
3. **Static variable**: A variable that is declared as static is called static variable. It cannot be local.
4. **Class Variable:** In object-oriented programming with classes, a class variable is a variable defined in a class of which a single copy exists, regardless of how many instances of the class exist.

**Static variable** and **Static method** is class variable.

public class SoccerPlayer

{

private int soccer\_skill =100; //instance variable

private String name="Bob"; //instance variable

public static String specie= "Human"; //class variable

Leg right = new Leg(80); // reference varaible

Leg left = right; // reference varaible

}

class Leg

{

public int shootingSkill; //instance varaible

public Leg(int skill)

{

shootingSkill = skill;

}}

1. **Reference variable: class object**

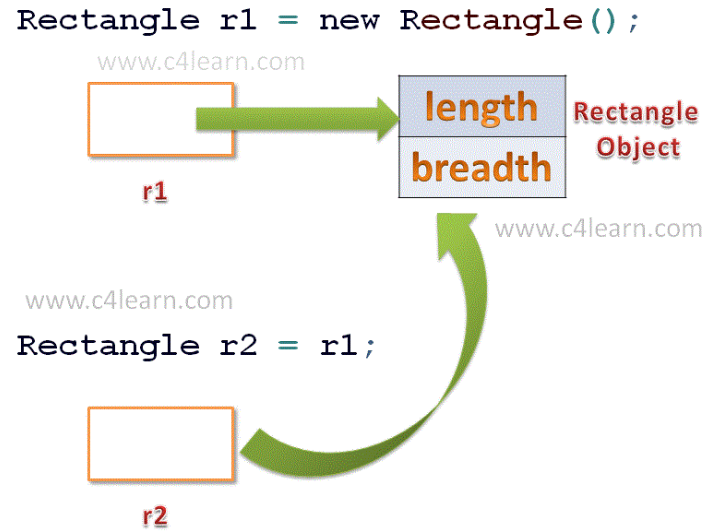
The only way you can access an object is through a reference variable. A reference varaible, points to an object. A reference variable is declared to be of a specific type and that type can never be changed. Reference variables can be declared as

1. static variables
2. instance variables
3. method parameters
4. local variables.

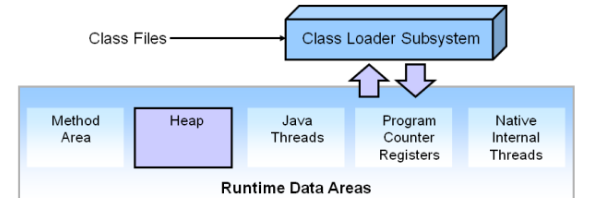
A reference variable that is declared as final can’t never be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed.

### Example :

This , super



# Memory Architecture Of JVM



Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So Java objects reside in an area called the heap.When the heap becomes full, garbage is collected. During the garbage collection objects that are no longer used are cleared, thus making space for new objects.

**HEAP** (**young space+Old Space)**

Note that the JVM uses more memory than just the heap. For example Java methods, thread stacks and native handles are allocated in memory separate from the heap, as well as JVM internal data structures.

The heap is sometimes divided into two areas (or generations) called the nursery (or young space) and the old space. The nursery is a part of the heap reserved for allocation of new objects. When the nursery becomes full, garbage is collected by running a special young collection, where all objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. When the old space becomes full garbage is collected there, a process called an old collection.

When your java file is compiled by JVM then first static method and variables are store in heap Area of JVM and then objects are allocated.

**Stack**

Stack is generated with each thread created by program.Each Thread has its own stack. local variables & function calls are stored in stack. It’s life depends upon Thread’s life as thread will be alive it will also and vice-versa.

Garbage Collection

Garbage collection is the process of freeing space in the heap or the nursery for allocation of new objects.

**PC Register**

It is also associated by its thread. It basically is a address of current instruction is being executed. Since each thread some sets of method which is going to be executed depends upon PC Register.

**Method Area**

It is memory which is shared among all Threads like Heap. It is created on Java Virtual Machine startup. It contains the code actually a compiled code, methods and its data and fields.

**Native Method Stack**

Native methods are those which are written in languages other than java. JVM implementations cannot load native methods and can’t rely on conventional stacks . It is also associated with each thread. In short it same as stack but it is used for native methods.

## Understanding Locks

When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the synchronized keyword, or with wait and notify. Synchronization is achieved by the use of locks, each of which is associated with an object by the JVM. For a thread to work on an object, it must have control over the lock associated with it, it must “hold” the lock. Only one thread can hold a lock at a time. If a thread tries to take a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is so called “contention” for the lock.

## Lock Chains

Several threads can be tied up in what is called lock chains. Although they appear somewhat complex, lock chains are fairly straightforward. They can be defined as follows:

* Threads A and B form a lock chain if thread A holds a lock that thread B is trying to take. If A is not trying to take a lock, then the lock chain is “open.”
* If A->B is a lock chain, and B->C is a lock chain, then A->B->C is a more complete lock chain.
* If there is no additional thread waiting for a lock held by C, then A->B->C is a complete and open lock chain.

# Operators in java

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | *expr*++ *expr*-- |
| prefix | ++*expr* --*expr* +*expr* -*expr*~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | <<>>>>> |
| Relational | comparison | <><= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & --The bitwise & operator always checks both conditions whether first condition is true or false |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | Ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

# Conditional Statement

**Int**marks=65;

**if**(marks<50){

System.***out***.println("fail");

}

**elseif**(marks>=50 &&marks<60){

System.***out***.println("D grade");

}

**else**{

System.***out***.println("else condition");

}

# Ternary Operator ?

Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is

condition ? if true : if false

The above statement means that if the condition evaluates to true, then execute the statements after the ‘?’ else execute the statements after the ‘:’.

System.out.println((i==5) ? "i=5":((i==10) ? "i=10":"i is not equal to 5 or 10"));

# Java Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement.

**Int**x=2;

**switch**(x)

{

**case** 1:

**int**y=10;

System.***out***.println(y);

**break**;

**case** 2:

**int**z=10;

System.***out***.println(z);

**break**; //if break is not applied then it will execute remaining code without case check

**case** 3:

**int**zz=10;

System.***out***.println(zz);

**default**:

System.***out***.println("no value matched");

**break**;

}

Stringaa= "hello";

**switch**(aa)

{

**case**"hello 1":

System.***out***.println("hello 1");

**break**;

**case**"Hello":

System.***out***.println("Hello");

**break**;

**case**"hello":

System.***out***.println("hello");

**break**;

**default**:

System.***out***.println("default");

**break**;

}

**Important Notes for Switch**

* The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.Otherwise Compile Error
* When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
* When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
* A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

# Java For-Each Loop

The **for-each** loop in Java (also called the enhanced for loop) was introduced in Java 5 to simplify iteration over **arrays** and **collections**. It is cleaner and more readable than the traditional for loop and is commonly used when the exact index of an element is not required.

**Syntax**

for (type var : **array** **or** **Collection**) {

statements using var;

}

Parameters:

* type: The data type of the elements in the array or collection.
* var: The variable that holds the current element during each iteration.
* array: The array or collection being iterated over.

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

for (int e : arr) {

System.out.print(e + " "); }

# Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

**Syntax:**

for(initialization;condition;incr/decr){

//code to be executed

}

## Infinitive loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

for(;;){

//code to be executed

}

**int**ar[]=**newint**[10];

**int**ar2[]=**newint**[5];

**int**ar1[]={ 22,33,44,55,66,77,88};

**for** (**int**i=0; i<ar.length; i++ )

{ar[i]+= i;

System.***out***.println(ar[i]); }

**for** (**int**i=0; i<ar1.length; i++ )

{ar[i]+= i;

System.***out***.println(ar1[i]); }

for(int i=0;i>10;i++)

{System.out.println("hello");}

It will first initialize (int i=0) then check condition (i>10). If condition is true then it enters inside curly brace other exit from for loop. In above case print statement will not be executed.

for(int i=0;i>10;i++);

Adding semi-colon for-loop the line is ended.above is complete for loop statement

https://www.javatpoint.com/java-while-loop

# Java While Loop

The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

while(condition){

//code to be executed

}

If you pass **true** in the while loop, it will be infinitive while loop.

while(true){

 System.out.println("infinitive while loop");

 }

int j = 1;

while(j>10);

Above is complete while loop statement.

# Java do-while Loop

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java do-while loop is executed at least once because condition is checked after loop body.

do{

//code to be executed

}while(condition);

# Java Break Statement

Suppose you are working with loops. It is sometimes desirable to skip some statements inside the loop[Continue] or terminate the loop immediately without checking the test expression[break].

The Java break is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

The break statement terminates the loop immediately, and the control of the program moves to the next statement following the loop.It is almost always used with decision making statements (if...else Statement).

**Note:** Break cannot be outside of loop or switch statement

# Labeled break Statement

The break statement we have discussed till now is unlabeled form of break statement, which terminates the innermost for, while, do..while and switch statement. There is another form of break statement, labeled break, that can be used to terminate the outer loop.

**How to use label?**

Just before for or While loop

mylabel :

**for** (**int**j = 0; j< 5; j++) {

**for** (**int**i = 0; i< 10; i++) {

System.***out***.println(i);

**if** (i == 4)

**break** mylabel;

}

System.***out***.println("Inner loop exited");

}

# Java Continue Statement

The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

**Note:** Continue cannot be outside of loop

**Query: If both are used.Break and Continue**

**if** (i>= 4)

**continue**;

**break**;

System.***out***.println(i);

**if** (i>= 4)

**break**;

**continue**;

System.***out***.println(i);

In above case print statement throw compile time error

# Java Single Line Comment

The single line comment is used to comment only one line.

//This is single line comment

/\*

This  is

multi line

comment

\*/

# Variable

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location.

There are three types of variables in java:

## local variable

A variable which is declared inside the method is called local variable.

## instance variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

## static variable

A variable that is declared as static is called static variable. It cannot be local.

class A{

int data=50;//instance variable

static int m=100;//static variable

void method(){

int n=90;//local variable

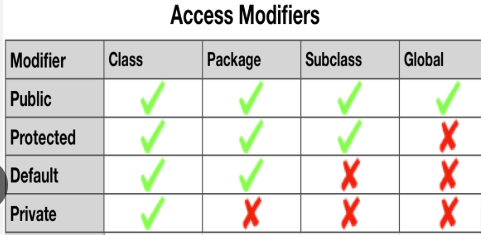
}

}//end of class

# Visibility of Variables and Methods

Access modifiers is applicable for

1. Class
2. Interface : Fields and methods in interfaces are implicitly declared public if you leave out an access modifier
3. its constructors
4. fields
5. methods.



## Access Modifier

* Private
* Default
* Protected
* public

## Default –accessible within package

The default Java access modifier is declared by not writing any access modifier at all.

By default, the variables and methods of a class are accessible to members of the class itself and to other classes in the same package. Within package

The default Java access modifier is declared by not writing any access modifier at all. The default access modifier means that code inside the class itself as well as code inside classes in the same package as this class, can access the class, field, constructor or method which the default access modifier is assigned to. Therefore, the default access modifier is also sometimes referred to as the package access modifier.

## Private—accessible within class

private are accessible only within their class.

## Public—accessible within project

At the other end of the spectrum, members declared as public are accessible from any class in any package, provided the class itself can be seen. (The class that contains the methods must also be public to be seen outside of its package)

Within project

## Protected (Slightly less restrictive than Default)

protected is slightly less restrictive than the default level of accessibility.

protected members are visible to subclasses of the class, even if they are defined in a different package.

The protected access modifier provides the same access as the default access modifier, with the addition that subclasses can access protected methods and member variables (fields) of the superclass. This is true even if the subclass is not located in the same package as the superclass.

|  |  |
| --- | --- |
| **Modifier** | **Visibility outside the class** |
| private | None |
| No modifier (default) | Classes in the package |
| protected | Classes in package and subclasses inside or outside the package |
| public | All classes |

# Visibility of class: Access Modifier

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers private and protected cannot be assigned to a class. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

Only default,public -- allowed

**private**,**protected** -- Not allowed

**public stati classc**, **static class** --- not allowed

## Access Modifiers and Inheritance

When you create a subclass of some class, the methods in the subclass cannot have less accessible access modifiers assigned to them than they had in the superclass. For instance, if a method in the superclass is public then it must be public in the subclass too, in case the subclass overrides the method. If a method in the superclass is protected then it must be either protected or public in the subclass.Not applicable to variable.

While it is not allowed to decrease accessibility of an overridden method, it is allowed to expand accessibility of an overridden method. For instance, if a method is assigned the default access modifier in the superclass, then it is allowed to assign the overridden method in the subclass the public access modifier.

**Order of access modifier**

Public > protected > default > private

## Interface Access Modifiers

Java interfaces are meant to specify fields and methods that are publicly available in classes that implement the interfaces. Therefore you cannot use the private and protected access modifiers in interfaces. Fields and methods in interfaces are implicitly declared public if you leave out an access modifier, so you cannot use the default access modifier either (no access modifier).

## Non-Access Modifier

* static
* final
* abstract
* synchronized
* volatile
* Transient

<https://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm>

# DATA TYPE in JAVA



## Data type size

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** | **Range** |
| boolean | False | 1 bit | False True |
| byte | 0 | 1 byte | -128 to 127— 2^7 |
| short | 0 | 2 byte | -32,768 to 32,767 – 2^15 |
| int | 0 | 4 byte | -2^31 to 2^31-1 .Default Type |
| long | 0L | 8 byte | -263 to 263-1.have to mention L for long |
| float | 0.0f | 4 byte | have to mention f for float |
| double | 0.0d | 8 byte | Suffix is not required. default type |
| Char | G | 2 Byte | Single character |

\*\*A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

**Example**

int I,j,k;

i=10

---OR

int i=10;

A data type that is primitive, such as the long variable, actually stores the value. If we give a value to the partNumber value, for example 4030023, that is what Java stores.

**Int vs double**

int i=200;

double d=i;

System.out.println(d); --200.0

## Reference Data Type or Non-primitive Or reference variable

Non-primitive, or reference data types, are the more sophisticated members of the data type family. They don't store the value, but store a reference to that value. Instead of partNumber 4030023, Java keeps the reference, also called address, to that value, not the value itself.

Reference types can be a

* Class
* Interface
* array variable
* String

### Arrays

An array is a group of variables that share the same data type, and are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.

\*declaring array don’t create object. Need to use **new**keyword to create object.

Exception ae[] = new Exception[3];

Object aao[][] = new Exception[2][3];

int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };

int a[]= new a[4];

char ac[] = { 'n', 'o', 't', ' ', 'a', ' ','S', 't', 'r', 'i', 'n', 'g' };

String[] aas = { "array", "of", "String", };

Int[] arr={1,2,3,4,5,6,7};

Int[][] arr={{1,2,3},{11,22,33},{111,222,333}};

Int[][] arr=new int[2][1]

Arr={{1},{2}};

byte[] rowvector, colvector, matrix[];

byterowvector[], colvector[], matrix[][];// this is equivalent to above

### Two Dimensional Array

int[][] arr = new int[2][3];

for (int i = 0; i < arr.length; i++) {

for (int j = 0; j < arr[i].length; j++) {

arr[i][j] = j;

System.out.print(arr[i][j] + " ");

}

System.out.println("");

}

**Array of Array**

In java there is no concept of two dimensional array. A two dimensional array in java is just an array of array. So below image correctly defines two dimensional array structures in java.

int[][] arr = new int[2][3];

arr[0]=new int[3];

int[][] myArray = { {0,1,2,3}, {3,2,1,0}, {3,5,6,1}, {3,8,3,4} };

**non-symmetric size array**

Now if two dimensional arrays in java is an array-of-array, then it should also support non-symmetric sizes as shown in below image.

int[][] arrMulti = new int[2][]; // yes it's valid

arrMulti[0] = new int[2];

arrMulti[1] = new int[3];

int[][] arr={

{1,2},

{11,22,33}

};

#### How to Find Multiple Missing Integers in Given Array of Numbers with Duplicates in Java?

You have given an integer array of size N. Array contains numbers from 1 to N-1 but a couple of numbers are missing in an array which also contains duplicates.

we need to use a different approach, something like a roll-call you would have seen in your school.The teacher has a register with names of all students, he goes through the list and mark absences on red. We can use the same approach to find all the missing numbers in the list.

We can use an array as register and it's an index as names of the numbers.and tick marking all the numbers which are present by storing one of their respective indices.

int[] input = { 4, 4, 2, 4, 5, 5, 7, 3, 3, 3 };

int[] register = new int[input.length];

for (int i : input) {

register[i] = 1;

}

for (int i = 1; i < register.length; i++) {

if (register[i] == 0) {

System.out.println(i);

}

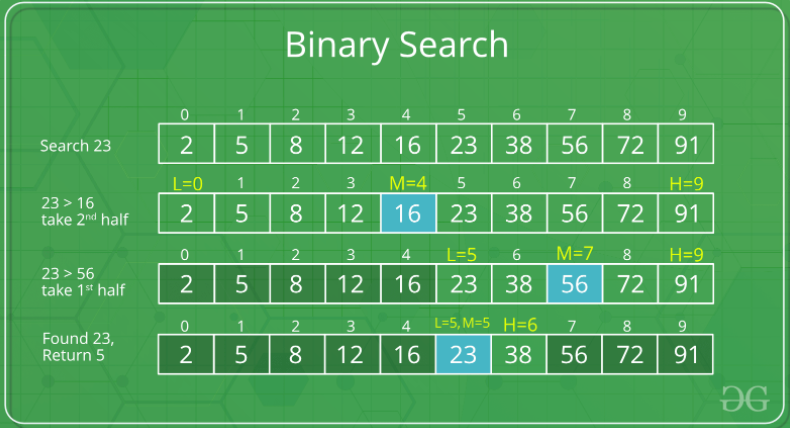
}

#### Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do a **linear search**.

**Binary Search**: Search a sorted array by repeatedly dividing the search interval in half.  Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Program

**public** **static** **int** search(**int**[] arr, **int** value) {

**int** start, length, mid;

length = arr.length - 1;

start = 0;

**while** (start <= length) {

mid = start + (length - start) / 2;

**if** (value == arr[mid])

**return** mid;

**else** **if** (value > arr[mid])

start = mid + 1;

**else**

length = mid - 1;

}

**return** -1;

}

### String

### String Literal

String Literals are always enclosed in double quotes and are implemented using the java.lang.String class. Enclosing a character string within double quotes will automatically create a new String object. For example,

String s = "this is a string";

String objects are immutable, which means that once created, their values cannot be changed.It is store in string constant pool(special area reseved by JVM) so it is highly recommendable.

### String Object

String obj=new String(“This is string”);

## Separators

Separators are symbols that indicate the division and arrangement of groups of code. The structure and function of code is generally defined by the separators. The separators used in Java are as follows:

### parentheses ( )

Used to define precedence in expressions, to enclose parameters in method definitions, and enclosing cast types.

### braces { }

Used to define a block of code and to hold the values of arrays.

### brackets [ ]

Used to declare array types.

### semicolon ;

Used to separate statements.

### comma ,

Used to separate identifiers in a variable declaration and in the for statement.

### period .

Used to separate package names from classes and subclasses and to separate a variable or a method from a reference variable.

## Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

**Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

## Wrapper class in Java

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object , an address of a system can be seen as an object , an image can be treated as an object and a simple data type can be converted into an object (with wrapper classes).

**Wrapper class in java** provides the mechanism to convert primitive into object and object into primitive.The eight classes of java.lang package are known as wrapper classes in java.

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it.

 Integer i=new Integer(10);

This is known as Boxing, converting a primitive type into an object.  
     Now to get that integer value back,  
 int a=i.intValue();  
This operation is known as Unboxing converting the value of a wrapper class object into a primitive type.

The list of eight wrapper classes are given below:

|  |  |  |
| --- | --- | --- |
| **SN** | **Primitive Type** | **Wrapper class** |
| 1 | boolean | Boolean |
| 2 | char | Character |
| 3 | byte | Byte |
| 4 | short | Short |
| 5 | int | Integer |
| 6 | long | Long |
| 7 | float | Float |
| 8 | double | Double |

## Importance of Wrapper classes

There are mainly two uses with wrapper classes.

1. To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

String str="35";

inta=Integer.parseInt(str)\*2;

**Autoboxing and Unboxing**

**Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

char ch = 'a';

       // Autoboxing- primitive to Character object conversion

        Character a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects

        arrayList.add(25);

**Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

Character ch = 'a';

      // unboxing - Character object to primitive conversion

        char a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        arrayList.add(24);

        // unboxing because get method returns an Integer object

        int num = arrayList.get(0);

        // printing the values from primitive data types

        System.out.println(num);

# Many Ways to create object of class

1. className obj=new className();
2. className obj=className.class.getInstance();

# Class instance

There is one java class which name is java.lang.**C**lass

className.class gives object of class name “Class”.

Class **forName(“Fully Qualified Name”) :** complete package name.className e.g com.mysql.jdbc.Driver

return Class object which helps to find all method of this class(mentioned in argument)

Test3 obj = **new** Test3();

Class cls = obj.getClass();

System.***out***.println(cls);

Class cls1=Class.*forName*("reflection.Test3");

System.***out***.println(cls1);

System.***out***.println(Test3.**class**);

System.***out***.println(**this**.getClass());

## Method list of class “Class”

Method[]getMethods() – return array of java.lang.reflect.Method (reflection class)

Method[]getDeclaredMethods() – return array of java.lang.reflect.Method (reflection class)

# Java Reflection

Reflection is an API which is used to examine or modify the behavior of **methods**, **classes**, **interfaces** at runtime.

Package is java.lang.reflect

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods (including wait,equals,toString) of the class to which an objects belongs.getDeclaredMethods() gives only declared methods to that class.
4. **Fields** the getDeclaredField() is used to get the Fields of class
5. **getName()** gives name of either class,method accordingly.

**public** **class** Test2 {

// creating a private field

**private** String s;

// creating a public constructor

**public** Test2() {

s = "GeeksforGeeks";

}

// Creating a public method with no arguments

**public** **void** method1() {

System.***out***.println("The string is " + s);

}

// Creating a public method with int as argument

**public** **void** method2(**int** n) {

System.***out***.println("The number is " + n);

}

// creating a private method

**private** **void** method3() {

System.***out***.println("Private method invoked");

}

@Test

**public** **void** abc(){

// Creating object whose property is to be checked

Test2 obj = **new** Test2();

// Creating class object from the object using

// getclass method

Class cls = obj.getClass();

System.***out***.println("The name of class is " + cls.getName());

// Getting the constructor of the class through the

// object of the class

Constructor constructor = cls.getConstructor();

System.***out***.println("The name of constructor is " + constructor.getName());

System.***out***.println("The public methods of class are : ");

// Getting methods of the class through the object

// of the class by using getMethods

Method[] methods = cls.getMethods();

// Printing method names

**for** (Method method : methods)

System.***out***.println(method.getName());

// creates object of desired method by providing the

// method name and parameter class as arguments to

// the getDeclaredMethod

Method methodcall1 = cls.getDeclaredMethod("method2", **int**.**class**);

// invokes the method at runtime

methodcall1.invoke(obj, 19);

// creates object of the desired field by providing

// the name of field as argument to the

// getDeclaredField method

Field field = cls.getDeclaredField("s");

// allows the object to access the field irrespective

// of the access specifier used with the field

field.setAccessible(**true**);

// takes object and the new value to be assigned

// to the field as arguments

field.set(obj, "JAVA");

// Creates object of desired method by providing the

// method name as argument to the getDeclaredMethod

Method methodcall2 = cls.getDeclaredMethod("method1");

// invokes the method at runtime

methodcall2.invoke(obj);

// Creates object of the desired method by providing

// the name of method as argument to the

// getDeclaredMethod method

Method methodcall3 = cls.getDeclaredMethod("method3");

// allows the object to access the method irrespective

// of the access specifier used with the method

methodcall3.setAccessible(**true**);

// invokes the method at runtime

methodcall3.invoke(obj);

}

## Method Class

The java.lang.reflect.Method class provides information about, and access to, a single method on a class or interface. The reflected method may be a class method or an instance method (including an abstract method).

The Method[] array will have one Method instance for each **public method** declared in the class. Below code will list all public method of class “cls\_Method”.

Class cls=cls\_Method.class;

Method[] mthds=cls.getMethods();

for (int i=0;i<mthds.length; i++)

System.out.println(mthds[i].getName());

# Java ClassLoader getResourceAsStream() Method

The **Java ClassLoader getResourceAsStream()**method returns an input stream for reading the specified resource.

public InputStream getResourceAsStream(String name)

**Return Value**

This method returns an input stream for reading the resource, or null if the resource could not be found.

**Configs** folder are available under *src/test/java/resources/*

import org.apache.commons.io.IOUtils;

List<String> files= IOUtils.readLines(BaseClass.class.getClassLoader().getResourceAsStream("configs/"));

files.forEach(p ->

{

try {

prop.load(BaseClass.class.getClassLoader().getResourceAsStream("configs/" + p));

} catch (IOException e) {

throw new RuntimeException(e);

}

}

);

# Java DateTime class

import java.time.LocalDateTime;

import java.time.LocalTime;

System.*out*.println(LocalDateTime.*now*()); // 2024-12-31T11:54:54.923925500

System.*out*.println(LocalDate.*now*()); // 2024-12-31

System.*out*.println(LocalTime.*now*()); // 11:54:54.925002300

# Arrays -- Utility Class

The **Arrays** class in **java.util package** is a part of the **Java Collection Framework**.

## ****Arrays.asList()****

* Returns a **fixed-size** **List** as of size of given array. The resulting List is not a general-purpose List implementation, because it doesn't implement the (optional) **add** and **remove** operations
* Attempting to add/Delete will throw an UnsupportedOperationException exception.
* Element Type of List is of same as type of array element type.
* It returns an List containing all of the elements in this
* arraylist in the same order as array

List<String> ls=Arrays.*asList*("one","Two","Three","ThreeFour");

## Void Arrays.sort()

int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

Arrays.sort(arr); --Entire range in ascending order.

Arrays.sort(arr,fromIndex,toIndex) -- sort in given range

## String Arrays.toString()

int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

Arrays.toString(a)

Returns string representation consists of a list of the array’s elements, enclosed in square brackets (“[]”). Adjacent elements are separated by the characters “, ”

Returns “null” if a is null

## binarySearch()

int **binarySearch**(arrObj,fromIndex,toIndex,searchValue) -- return **index** of the search key in given range; otherwise -1

int **binarySearch**(arrayObject, SearchValue) -- return **index** of the search key in entire range; otherwise -1

int[] arr1 = { 1, 3, 5, 7, 2, 4 };

System.out.println(Arrays.binarySearch(arr1, 3));

## Removing data

int[] test = new int[] { 101, 102, 103, 104, 105};

//let's remove or delete an element from Array using Apache Commons ArrayUtils

test = ArrayUtils.remove(test, 2); //removing element at index 2

# [System.setProperty and System.getProperty](https://stackoverflow.com/questions/21204334/system-setproperty-and-system-getproperty)

System class has a static member variable named props which is of type Properties. Adding to that, Properties is a subtype of Hashtable class. All the property values are stored as Key and Value. So, datastore is Hashtable.Answering the other question, You can very well use System.getProperty(propertyKey) method throughout your application since it is a public static method. You haven't understood how java programs work. **When you run a Java program, you are actually starting a JVM instance. That instance will have its own System properties**. That is where you have to put your property. When you run the other program, that will have its own System properties. So, you cannot expect a property which you set in one JVM instance to be accessible from another JVM instance! You can access the System.getProperty(propertyKey) in all classes running in the same JVM instance.

setProperty(String key, String value)

// this will list the current system properties

Properties p =System.getProperties();

p.list(System.out);

System.*setProperty*("webdriver.chrome.driver", "D:\\chromedriver.exe");

If you set above property then it will contain in system properties list

# Annotation

Annotations is a new feature from Java 5.Annotations are a kind of comment or meta data you can insert in your Java code. These annotations can then be processed at compile time by pre-compiler tools, or at runtime via Java Reflection.

Java **Annotation** is a tag that represents the *metadata* i.e. attached with **class**, **interface**, **methods** or **fields** to indicate some additional information which can be used by java compiler and JVM.

**Java Custom annotations** or Java User-defined annotations are easy to create and use. The ***@interface*** element is used to declare an annotation. For example:

## Built-In Java Annotation used in Java Code

* **@Override--** assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
* **@SuppressWarnings--** used to suppress warnings issued by the compiler.
* **@Deprecated--** marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions.

## Built-In Java Annotations used in other annotations

* **@Target**
* **@Retention**
* **@Inherited**
* **@Documented**

## Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

## Syntax for creating annotation

Annotations are defined like interfaces.

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

public @interface MyAnnotation {

public String name();

public String value();

}

The two directives in the annotation definition, @Retention(RetentionPolicy.RUNTIME) and @Target(ElementType.TYPE), specifies how the annotation is to be used.

@Retention(RetentionPolicy.RUNTIME) means that the annotation can be accessed via reflection at runtime. If you do not set this directive, the annotation will not be preserved at runtime, and thus not available via reflection.

@Target(ElementType.TYPE) means that the annotation can only be used on top of types (classes and interfaces typically). You can also specify METHOD or FIELD, or you can leave the target out alltogether so the annotation can be used for both classes, methods and fields.

### Method Annotations

public class TheClass {

@MyAnnotation(name="someName", value = "Hello World")

public void doSomething(){}

}

### Parameter Annotations

It is possible to add annotations to method parameter declarations too. Here is how that looks:

public class TheClass {

public static void doSomethingElse(@MyAnnotation(name="aName", value="aValue") String parameter){

}

}

### Field Annotations

public class TheClass {

@MyAnnotation(name="someName", value = "Hello World")

public String myField = null;

}

## Types of Annotation

<https://www.javatpoint.com/java-annotation>

There are three types of annotations.

### Marker Annotation

An annotation that has no method, is called marker annotation. For example:

**Public @interface** MyAnnotation{}

### Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

**Public @interface** MyAnnotation{

**int** value();

}

**--Or default value as below----------**

**Public @interface** MyAnnotation{

**int** value() default 0;

}

**Apply** @MyAnnotation(value=10)

### Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

**Public @interface** MyAnnotation{

**int** value1();

String value2();

String value3();

}

**--Or default value as below----------**

**Public @interface** MyAnnotation

{

**int** value1() **default** 1;

String value2() **default** "";

String value3() **default** "xyz";

}

**Apply** @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")

## How to access annotation value

**Step1**: create annotation

**import** java.lang.annotation.Target;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***METHOD***)

**@interface** MyAnnotation{

**int** value();

}

**Step2**: Use annotation in method and other method will use that annotation value

**class** Hello {

@MyAnnotation(value = 10)

@Test

**public** **void** sayHello() {

System.***out***.println("hello annotation");

}

//-------------------------------------

@Test

**public** **void** abc() {

Hello h = **new** Hello();

Method m = **null**;

**try** {

m = h.getClass().getMethod("sayHello");

} **catch** (NoSuchMethodException | SecurityException e) {

e.printStackTrace();

}

MyAnnotation manno = m.getAnnotation(MyAnnotation.**class**);

System.***out***.println("value is: " + manno.value());

}

}

# Object Class

Object is a Superclass for all java class. Every class is a descendant, direct or indirect, of the Object class.

## Clone() Method

This is available by default in all class but not useful unless override this method using Cloneable interface.

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this 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.
* Every class that override clone() should call super.clone() to obtain the cloned object reference.

### 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.

### How to implement clone() method

**class** clsClone **implements** Cloneable {

**int**x = 10;

String y = "Khalid";

**public** Object clone() **throws** CloneNotSupportedException

{

**returnsuper**.clone();

}

}

Main—

clsClone obj = new clsClone();

clsClone cloneobj=(clsClone) obj.clone();

### ****Using Assignment Operator to create copy of reference variable****

**if we use assignment operator then it will create a copy of reference variable and not the object.**

**Test ob1 = new Test();**

**// Creating a new reference variable ob2**

**// pointing to same address as ob1**

**Test ob2 = ob1;**

**// Any change made in ob2 will be reflected**

**// in ob1**

**ob2.x = 100;**

## ****hashCode()****

It returns the hashcode value as an Integer. Hashcode value is mostly used in hashing based collections like HashMap, HashSet, HashTable….etc.

This is a **native method**, which means it will be executed in another language like C, and will return some code regarding the object's memory address.

Hashcode of String

StringStr=newString("Welcome to Tutorialspoint.com");

System.out.println("Hashcode for Str :"+Str.hashCode());

Integer intObj = **new** Integer(10);

System.***out***.println(intObj.hashCode());

Character chObj = **new** Character(‘A’);

System.***out***.println(chObj.hashCode());

Hashcode of integer values is same as value.

Hashcode of Character values is same as ASCII value.

## ****equals(Object)****

**it compares two object based on hashcode.**

**The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.**

## toString()

if this is not overridden then defautlt is packageName.className@haxanumber

## finalize()

finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you.

## getClass()

You cannot override getClass.

The getClass() method returns a **Class object**,which has methods you can use to get information about the class, such as its name (getSimpleName())

The Class class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum())

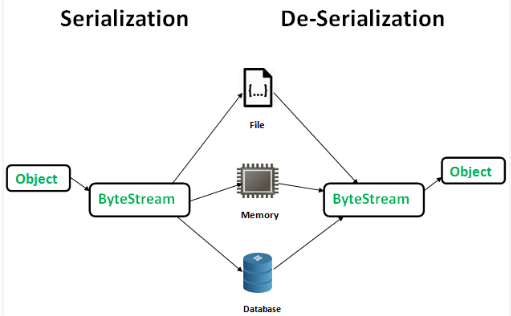
# Java - Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

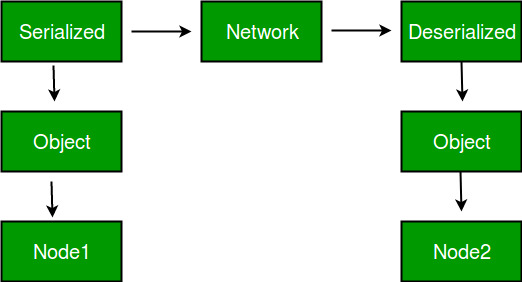
So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

# Serialization and Deserialization in Java

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.



**Advantages of Serialization**  
1. To save/persist state of an object.  
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable**interface.  
Serializable is a **marker interface** (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

**Points to remember**  
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.

|  |  |
| --- | --- |
| ObjectOutputStream | **writeObject()** |
| ObjectInputStream | **readObject()** |

### Calling writeObject()

**FileOutputStream** fos =**newFileOutputStream**("temp.out");

**ObjectOutputStream** oos =**newObjectOutputStream**(fos);

**TestSerial** ts =**newTestSerial**();

oos.writeObject(ts);

oos.flush();

oos.close();

### Calling readObject()

**FileInputStream** fis =**new FileInputStream**("temp.out");

**ObjectInputStream** oin =**new ObjectInputStream**(fis);

**TestSerial** ts =(**TestSerial**) oin.readObject();

**System**.**out**.println("version="+ts.version);

**Difference between Collection and Collections**

**Collection is a root level interface** of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. List, Set and Queue are main sub interfaces of this interface.

**Collections is an utility class** in java.util package. It consists of only **static methods** which are used to operate on objects of type Collection.

# Collections utility class

**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

|  |  |
| --- | --- |
| Collections.max(CollectionObj) | This method returns maximum element in the specified collection. |
| Collections.min(CollectionObj) | This method returns minimum element in the given collection. |
| Collections.sort(ListObj) | This method sorts the specified collection. |
| Collections.reverse(ListObj) | This method reverses the order of elements in the specified collection. |
| Collections.copy(desListObj, srcListObj) | This method copies all elements from one collection to another collection. |
| Collections. singletonList(Object) | return an immutable list containing only the specified object. |

### How To Synchronize ArrayList, HashSet And HashMap In Java?

Collections.synchronizedList(arraylist-Obj);

Collections.synchronizedSet(Hashset-Obj);

Collections.synchronizedMap(Hashmap-Obj)

synchronizedCollection(Collection<T> c)

synchronizedSortedMap(SortedMap<K,V> m)

synchronizedSortedSet(SortedSet<T> s)

# Properties utility class

* **Properties** is a subclass of **Hashtable**.
* It is used to maintain list of value in which the key is a string and the value is also a string.
* One useful capability of the Properties class is that you can specify a default property that will be returned if no value is associated with a certain key.
* Multiple thread can share a single properties object without the need of external synchronisation.

## Method

**String getProperty(String key):** Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.

**String getProperty(String key, String defaultProperty):** Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns the default value argument if the property is not found.

**Object setProperty(String key, String value)**: Associate value with key. Returns the previous value associated with key, or returns null if no such association exist.

**void load(InputStream streamIn):** This method reads a property list (key and element pairs) from the input byte stream.

**void list(PrintStream streamOut):** Sends the property list to the output stream linked to streamOut.

**Set stringPropertyNames():** Returns a set of keys in this property list where the key and its corresponding value are strings, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list. Properties whose key or value is not of type String are omitted.

The returned set is not backed by the Properties object. Changes to this Properties are not reflected in the set, or vice versa.

**Config.properties File**

key1=khalid Anwar

key2=Aurangabad

**Java Class File for reading properties**

Properties prop=**new** Properties();

FileInputStream fis=**new** FileInputStream(System.*getProperty*("user.dir")+"\\src\\config.properties");

**if**(fis !=**null**)

prop.load(fis);

**else**

System.***out***.println("Propery file is not available");

// Reading Property File

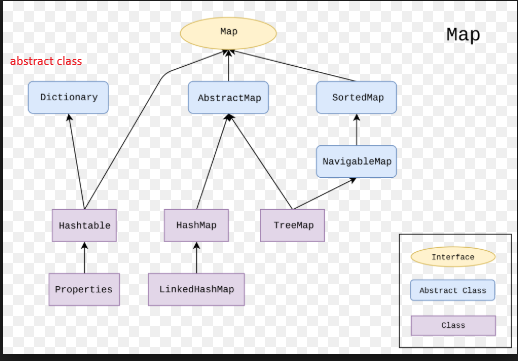
System.***out***.println(prop.getProperty("key1"));

**Java Class File for writing properties**

**FileOutputSteam fos=new FileOutputSteam(**System.*getProperty*("user.dir")+[\\src\\config.properties](file:///\\src\\config.properties));

prop.setProperty(key,Value);

prop.**store**(fos);



# ResourceBundle

If config.properties file is available under src/test/resources then below code read this file and get key value.

**import** java.util.ResourceBundle;

ResourceBundle rb=ResourceBundle.*getBundle*("config");

System.***out***.println(rb.getString("key1"));

# Collection

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

|  |  |  |
| --- | --- | --- |
| single Operation | Bulk Operation | Comment |
| int **size**() |  | No of element in collection |
| boolean **isEmpty**() |  | true if collection is empty |
| void **clear**() |  | remove all elements |
| boolean **add**("value") | boolean **addAll**(CollectionObj) | C1.addAll(C2)--C1 union C2 |
| boolean **remove**("value") | boolean **removeAll**(CollectionObj) | C1.removeAll(C2)--C1 minus C2 |
| boolean **contains**("value") | boolean **containsAll**(CollectionObj) | C1.containsAll(C2)--C1 SuperSet of C2 |
|  | boolean **retainAll**(CollectionObj) | C1.retainAll(C2)--C1 intersection C2 |
| Iterator<E> **iterator**() | **Iterator** is interface and **iterator** is method | return Iterator object. So applicable to  boolean **hasnext**() Element **next**() void **remove**() |
|  | Object[] toArray() | converts to fixed size Object array |
|  | Integer[] toArray(new Integer[]) |  |

## Collection Iteration

### While Loop

 Collection<String> collection = new ArrayList<String>();

        collection.add("zero");

        collection.add("one");

        collection.add("two");

        Iterator<String> iterator = collection.iterator();

        // while loop

        while (iterator.hasNext()) {

        System.out.println("value= " + iterator.next());

        }

### For loop

Collection<String> collection = new ArrayList<String>();

        collection.add("zero");

        collection.add("one");

        collection.add("two");

        // for loop

        for (Iterator<String> iterator = collection.iterator(); iterator.hasNext();) {

        System.out.println("value= " + iterator.next());

        }

### For-Each

Collection<String> collection = new ArrayList<String>();

        collection.add("zero");

        collection.add("one");

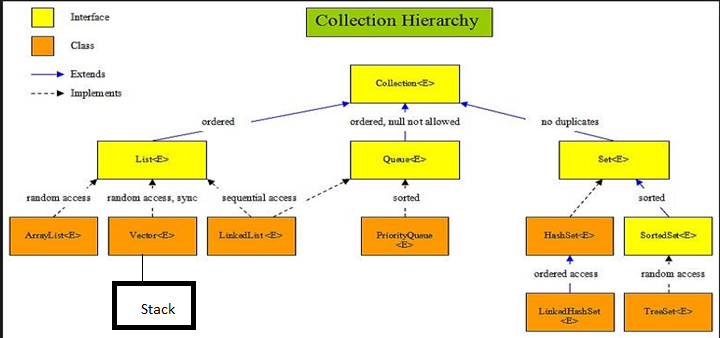
        collection.add("two");

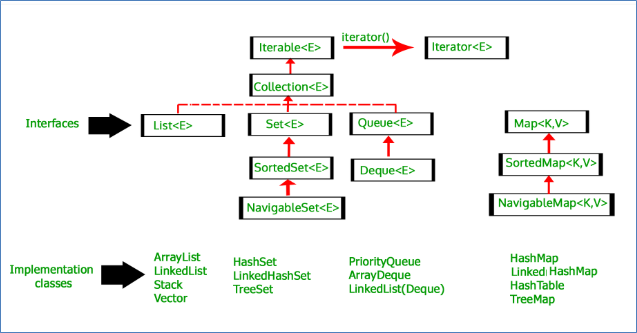
        // for-each loop

        for (String s : collection) {

        System.out.println("value= " + s);

        }





**1) List**  —> It handles sequential list of objects. **ArrayList**, **Vector** and **LinkedList** classes implement this interface.

**2) Queue**  —> It handles the special group of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.

**3) Set**  —> It handles the group of objects which must contain only unique elements. This interface is implemented by **HashSet** and **LinkedHashSet** classes and extended by **SortedSet** interface which in turn, is implemented by **TreeSet**.

**4) Map**  —> This is the one **interface** in Collection Framework which is not inherited from *Collection* interface. It handles the group of objects as Key/Value pairs. It is implemented by **HashMap** and **HashTable** classes and extended by **SortedMap** interface which in turn is implemented by **TreeMap**.

# Iterators interface

Iterators are used in [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) in Java to retrieve elements one by one.It is a universal iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of **remove-ability** of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like **Set**, **List**, **Queue**, **Deque** and also in all implemented classes of **Map** interface. Iterator is the **only** cursor available for entire collection framework.

Iterator interface defines **three** methods:

// Returns true if the iteration has more elements

**public boolean hasNext();**

// Returns the next element in the iteration

// It throws **NoSuchElementException** if no more

// element present

**public Object next();**

// Remove the next element in the iteration

// This method can be called only once per call

// to next()

**public void remove();**

### ****ListIterator Interface** (only applicable in list)**

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration.

ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are **six** more methods.

### Difference between iterator and Listiterator

Iterator ite = Set.iterator();

Iterator ite = List.iterator();

Iterator ite = Collection.iterator();

ListIterator listite = List.**listIterator**();

1. We can use Iterator to traverse **list**,**Set**,**queue** or **Map**. But ListIterator can only be used to traverse a List, it can't traverse a Set.
2. Using ListIterator, you can obtain index of next and previous elements. But, it is not possible with Iterator interface.
3. Using ListIterator, you can perform modifications(insert, replace, remove) on the list. But, using Iterator you can only remove the elements from the collection.
4. Using ListIterator, you can iterate a list from the specified index. It is not possible with Iterator.

**ListIterator Method**

// Forward direction

// Returns true if the iteration has more elements

**public boolean hasNext();**

// same as next() method of Iterator

**public Object next();**

// Returns the next element index

// or list size if the list iterator

// is at the end of the list

**public int nextIndex();**

// Backward direction

// Returns true if the iteration has more elements

// while traversing backward

**public boolean hasPrevious();**

// Returns the previous element in the iteration

// and can throws **NoSuchElementException**

// if no more element present

**public Object previous();**

// Returns the previous element index

// or -1 if the list iterator is at the

// beginning of the list

**public int previousIndex();**

// Other Methods

// same as remove() method of Iterator

**public void remove();**

// Replaces the last element returned by

// next() or previous() with the specified element

**public void set(Object obj);**

// Inserts the specified element into the list at

// position before the element that would be returned

// by next(),

**public void add(Object obj);**

# List

It may contain duplicate record.In addition to the operations inherited from Collection, the List interface includes operations for the following:

|  |  |  |
| --- | --- | --- |
| single Operation | collection views | Comment |
| **add**(index,obj)  add(obj) | Positional access | add value at index.if value is already available then shift existing value at +1 index |
| addAll(index,Collection)  **addAll**(collection) | add collection at index |
| **remove**(index)  remove(“value”) | delete value at index. Overridden method |
| **get**(index) | returns value at index |
| **set**(index,obj) | it will replace existing obj if any |
| **indexOf**(object) | Search | index of first occurrence |
| **lastindexof**(object) | index of last occurance |
| **listIterator**(index);  iterator() | Iteration | ListIterator is another interface which extends Iterator interface. |
| Sublist(originIndex,endIndex) | Range-view | it returns value till endindex-1 |

## ArrayList

* ArrayList internally uses an array to store the elements. Just like arrays, It allows you to retrieve the elements by their index.
* Java ArrayList **allows duplicate** and **null** values.
* Java ArrayList is an ordered collection. It maintains the insertion order of the elements.
* You cannot create an ArrayList of primitive types like int, char etc. You need to use boxed types like Integer, Character, Boolean,String,ArrayList of ArrayLists,user Definedobject etc.
* Java ArrayList is not synchronized. If multiple threads try to modify an ArrayList at the same time, then the final outcome will be non-deterministic. You must explicitly synchronize access to an ArrayList if multiple threads are gonna modify it.
* Equals() : Two List objects are equal if they contain the same elements in the same order.

#### Swaping in arraylist

swap(List<E> a, int i, int j) {

E tmp = a.get(i);

a.set(i, a.get(j));

a.set(j, tmp);

}

#### Looping arraylist

1. Java 8 forEach and lambda expression.
2. iterator().
3. iterator() and Java 8 forEachRemaining() method.
4. listIterator().
5. Simple for-each loop.
6. for loop with index.

#### List Algorithms

**sort** — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)-- Collections.**sort**(ListObj) -- **Ascending** order

**shuffle** — randomly permutes the elements in a List.

**reverse** — reverses the order of the elements in a List.-Collections.**reverse**(ListObj) -**Descending** order

**rotate** — rotates all the elements in a List by a specified distance.

**swap** — swaps the elements at specified positions in a List.-Collections.**swap**(ListObj, index1, index2)

replaceAll — replaces all occurrences of one specified value with another.

fill — overwrites every element in a List with the specified value.

**copy** — copies the source List into the destination List.Collections.**copy**(desListObj, srcListObj)

binarySearch — searches for an element in an ordered List using the binary search algorithm.

**indexOfSubList** — returns the index of the first sublist of one List that is equal to another.-Collections.indexOfSubList(List<?> source, List<?> target)

**lastIndexOfSubList** — returns the index of the last sublist of one List that is equal to another.

Min--Collections.***min***(CollectionObj)

Max--Collections.***max***(CollectionObj)

#### What is fixed size arrayList

FixedSizeList which does not support the **add**, **remove** and **clear** methods(but the set method is allowed because it does not modify the List's size).

#### Create fixed Size ArrayList

List<MyType> fixed = Arrays.asList(new MyType[100]);

#### Size and capacity

The size() method returns an integer equal to a number of elements present in the array list. It's different than the length of the array which is backing the ArrayList, that is called the capacity of ArrayList.When you create an object of ArrayList in Java without specifying a capacity, it is created with a default capacity which is 10. Since ArrayList is a growable array, it automatically resizes when the size (number of elements in array list) grows beyond a threshold.Also, when an ArrayList is first created it is called empty ArrayList and size() will return zero. If you add elements then size grows one by one. You can also remove all elements from ArrayList by using a clear().

The capacity is not publically exposed, as elements are added to an ArrayList, its capacity grows automatically. 

## LinkedList

LinkedList is a doubly-linked list implementation of the List and Deque interfaces. LinkedList allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, LinkedList can be searched forward and backward but the time it takes to traverse the list is directly proportional to the size of the list.

**As it implements deque so below method are extra**

void **addFirst**(Object item) --It adds the item (or element) at the first position in the list.

void **addLast**(Object item): It inserts the specified item at the end of the list.

Object **getFirst**(): It fetches the first item from the list.

Object **getLast**(): It fetches the last item from the list.

Object **removeFirst**(): It removes the first item from the list.

Object **removeLast**(): It removes the last item of the list.

### Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# SET

A Set is a Collection that cannot contain **duplicate** elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

### Properties Of Set

* Set contains only **unique** elements. It does not allow duplicates.
* Set can contain **only one null element**.
* Random access of elements is not possible.
* Set interface contains only methods inherited from Collection interface. It does not have it’s own methods. But, applies restriction on methods so that duplicate elements are always avoided.
* **HashSet** internally uses a [**HashMap**](https://www.callicoder.com/java-hashmap/) to store its elements.
* LinkedHashSet internally uses a [**HashMap**](https://www.callicoder.com/java-hashmap/) and **LinkedList** to store its elements and insertion order
* **TreeSet** is sorted set
* **Equals():**  two set is equal if elements are same

**Example:**

Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

Collection<Type> noDups = new HashSet<Type>(c);

Following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

Collection<Type> noDups = new LinkedHashSet<Type>(c);

## LinkedHashSet

It combines the functionality of a [HashSet](https://www.geeksforgeeks.org/hashset-in-java/) with a **LinkedList** to maintain the insertion order of elements. LinkedHashSet lets us iterate through the elements in the order in which they were inserted.

## TreeSet

The objects of the TreeSet class are stored in **ascending order**. Java TreeSet class access and retrieval times are quiet fast.

## HashSet

When iterating through a HashSet the order is unpredictable

## HashSet uses [HashMap](https://www.callicoder.com/java-hashmap/) to Add and Remove

public class **HashSet** extends AbstractSet implements Set, Cloneable, java.io.Serializable{private transient HashMap<E,Object> map;// Dummy value to associate with an Object in the backing Mapprivate static final Object PRESENT = new Object()public **HashSet**() { map = new HashMap<>();}public boolean add(E e) { return map.put(e, PRESENT)==null;}

public boolean remove(Object o) {

return map.remove(o)==PRESENT;

}/\*\*\* Some code\*/}

**code explanation**

1. when an object of HashSet is created, it will create an object of HashMap.
2. When an element is passed in add(Element e), it is added as a key in the HashMap.
3. Now, a value needs to be associated to the key in HashMap. Java uses a Dummy value (new Object) which is called PRESENT.
4. If map.put(key,value) returns null, then map.put(e, PRESENT)==null will return true and element is added to the HashSet
5. If map.put(key,value) returns the old value of the key, then map.put(e, PRESENT)==nullwill return false and element wont be added to the HashSet.
6. remove() method also works in the same way.

## Differences Between HashSet, LinkedHashSet and TreeSet In Java

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

# SortedSetextends Set

SortedSet is an interface in [collection framework](https://www.geeksforgeeks.org/collections-in-java-2/). This interface extends [Set](http://quiz.geeksforgeeks.org/set-in-java/) and provides a total ordering of its elements.

### Methods of Sorted Set interface:

**comparator()** : Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.

**first()** : Returns the first (lowest) element currently in this set.

**headSet(E toElement)** : Returns a view of the portion of this set whose elements are strictly less than toElement.

**last()** : Returns the last (highest) element currently in this set.

**subSet(E fromElement, E toElement) :** Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElementexclusive.

**tailSet(E fromElement)** : Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

# NavigableSet extends SortedSet

NavigableSet represents a navigable set in Java Collection Framework. The NavigableSet interface inherits from the SortedSet interface. It behaves like a SortedSet with the exception that we have navigation methods available in addition to the sorting mechanisms of the SortedSet.

### Properties Of NavigableSet

NavaigableSet can’t have null elements.

NavigableSet doesn’t support duplicate elements.

NavigableSet can be traversed and accessed in either ascending or descending order.

Methods subSet(), headSet() and tailSet() differ from SortedSet interface in taking additional arguments describing whether upper bound and lower bound are inclusive or exclusive.

### Methods of NavigableSet interface

Lower(E e) : Returns the greatest element in this set which is less than the given element or NULL if there is no such element.

Floor(E e ) : Returns the greatest element in this set which is less than or equal to given element or NULL if there is no such element.

Ceiling(E e) : Returns the least element in this set which is greater than or equal to given element or NULL if there is no such element.

Higher(E e) : Returns the least element in this set which is greater than the given element or NULL if there is no such element.

pollFirst() : Retrieve and remove the first least element. Or return null if there is no such element.

pollLast() : Retrieve and remove the last highest element. Or return null if there is no such element.

descendingset(); By Default, treeset is in ascending order. it changes order

tailSet(E,Boolean) ;it means including object and higher element if boolean is true;if false then element will be excluded

tailSet(E,Boolean) ;it means including object and lower element if boolean is true;if false then element will be excluded

[**https://stackoverflow.com/questions/186118/eclipse-fonts-and-background-color**](https://stackoverflow.com/questions/186118/eclipse-fonts-and-background-color)

#### Sorting of set

Set<String> tree\_Set = **new** TreeSet<String>(hash\_Set);

#### How to remove duplicate from arrayList

Using stream

List<Integer> newList = list.stream().distinct().collect(Collectors.toList());

Using HashSet

Set<Integer> st=new HashSet<>(list);

Using contains method

List<Integer> nlist=new ArrayList<>();

for (int i = 0; i < list.size(); i++) {

if(!nlist.contains(list.get(i)))

nlist.add(list.get(i));

}

nlist.forEach(p->System.out.print(p+ " "));

## Vector

A Vector is a re-sizable collection. It grows its size to accommodate new elements and shrinks the size when the elements are removed.

Same like arrayList where capacity() method gives size of list and it increases by 100%. By Default capacity=10

It is threadSafe whereas arrayList is not threadSafe.

It increase by 100% whereas Arraylist by 50%

## Stack

Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of last-in-first-out.Since the Stack class extends Vector, it also grows and shrinks its size as needed when new elements are added or removed.

Push(itemToBePushed);

Pop(); top item will be removed;

peek(): top item will be returned but not removed

search(itemToBeSearched): it gives position of that element

size() : gives size of stack.

**Iterating stack**

Iterate over a Stack using Java 8 forEach().

Iterate over a Stack using iterator().

Iterate over a Stack using iterator() and Java 8 forEachRemaining() method.

Iterate over a Stack from Top to Bottom using listIterator().

Difference between ArrayList and LinkedList

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

Searching element is faster in arraylist

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

LinkedList element **deletion** is faster compared to ArrayList

LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Inserts Performance: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

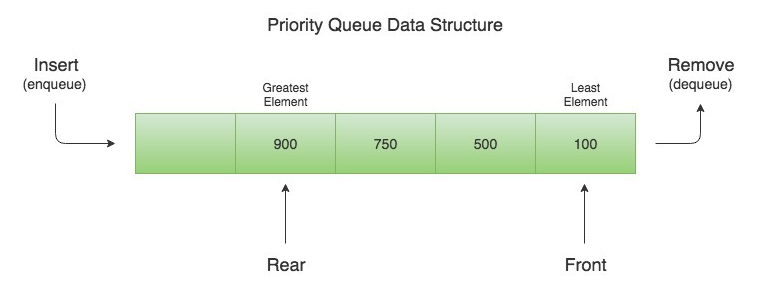
Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

## When to use LinkedList and when to use ArrayList?

LinkedList --if insert or modification is required

ArrayList -- If searching is required

# Queue



The Queue interface is available in java.util package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the **FIFO** or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.It is to be noted that both the implementations are not thread safe. PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.

## Properties Of Queue

* Null elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
* Queue can have duplicate elements.
* Unlike a normal list, queue is not random access. i.e you can’t set or insert or get elements at an arbitrary positions.
* In most of cases, elements are inserted at one end called tail of the queue and elements are removed or retrieved from another end called head of the queue.
* Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations

|  |  |  |
| --- | --- | --- |
| Type of Operation | Throws exception | Returns special value |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

**Implementing class**

**LinkedList**--Not Thread Safe,NULL is allowed.

Queue<Integer> q = **new** LinkedList<>();

q.add(10);

poll() Or remove()

poll() removes element and it returns NULL if the queue is empty whereas remove() throw exception NoSuchElementException

peek()

peek() read element and it returns NULL if the queue is empty

add(9) or offer(9)

offer returns True if the value is successfully inserted into the queue else false.Add() internally just call offer() method and does nothing extra and throw exception.

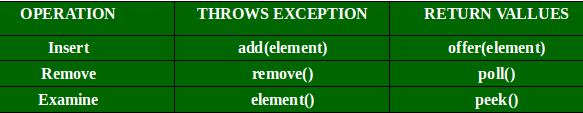
~~It throws two type of exceptions~~

* ~~NullPointerException: If the element to be inserted is NULL for other than LinkedList object.~~
* ~~ClassCastException: If an element to be inserted is of a different type that cannot be compared to the existing elements of the Queue.~~

**PriorityQueue** –extends abstractQueue(extend Queue)

* Not ThreadSafe
* A priority queue in Java is a special type of queue wherein all the elements are ordered as per their natural ordering or based on a custom Comparator supplied at the time of creation.
* So when you remove an element from the priority queue, the least element according to the specified ordering is removed first.
* The PriorityQueue is a special type of queue because it is not a First-In-First-Out (FIFO)
* The PriorityQueue can have duplicate elements but can not have null elements.
* The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.

**ArrayDeque**



# BlockingQueue

is a java Queue that support operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element. Java BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.

## Implementing class

1. ArrayBlockingQueue,
2. LinkedBlockingQueue,
3. PriorityBlockingQueue

* ThreadSafe
* The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as the java.util.PriorityQueue class. You cannot insert null into this queue.

1. SynchronousQueue

Add() internally just call offer() method and does nothing extra. Since, Queue is an interface which extends Collection interface and collection interface has add() method.

So, all the methods declared in collection interface must be reflected in Queue interface too. That’s the only reason for having add() in the Queue besides offer().

//Restricted queue with capacity of 2 elements

BlockingQueue Q\_Offer = new ArrayBlockingQueue(2); //Only 2 elements can

# Circular Array

if you want to keep, say, the last 30 values in a real-time system. The next value gets put into the array, and then the last value in the array gets pushed out. It does this by using a fixed array,

### To Check Whether Queue Is Full Or Not?

# Deque (Double Ended Queue) supports both FIFO or LIFO

The java.util.Deque interface is a subtype of the java.util.Queue interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO). These are faster than Stack and LinkedList.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | | Throws an exception if operation fails. | Returns null or false if operation fails. |
| Insertion | Front End | addFirst() | offerFirst() |
| Rear End | addLast() | offerLast() |
| Retrieval | Front End | getFirst() | peekFirst() |
| Rear End | getLast() | peekLast() |
| Retrieval And Removal | Front End | removeFirst() | pollFirst() |
| Rear End | removeLast() | pollLast() |

**Properties Of Deque**

* Unlike Queue, Deque can have null elements. But, it is recommended not to insert null elements as many methods return null to indicate Deque is empty.
* Deque can have duplicate elements.
* You can’t set or get or insert the elements at an arbitrary position of Deque. i.e Random access is not possible with the Deque.

**Implementing Class of Deque**

1. LinkedList : NULL is allowed.
2. ArrayDeque :class gives way for resizable array and implements the Deque interface.Array deques has no restruction, can have size according to the need.Concurrent access by multiple threads is nor supported. Null elements are not allowed in the array deque. It’s faster than LinkedList and Stack.

Methods of Deque:

1. add(element): Adds an element to the tail.
2. addFirst(element): Adds an element to the head.
3. addLast(element): Adds an element to the tail.
4. offer(element): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
5. offerFirst(element): Adds an element to the head and returns a boolean to explain if the insertion was successful.
6. offerLast(element): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
7. iterator(): Returna an iterator for this deque.
8. descendingIterator(): Returns an iterator that has the reverse order for this deque.
9. push(element): Adds an element to the head.
10. pop(element): Removes an element from the head and returns it.
11. removeFirst(): Removes the element at the head.
12. removeLast(): Removes the element at the tail.
13. poll(): Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
14. pollFirst(): Retrieves and removes the first element of this deque, or returns null if this deque is empty.
15. pollLast(): Retrieves and removes the last element of this deque, or returns null if this deque is empty.
16. peek(): Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
17. peekFirst(): Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.
18. peekLast(): Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.

# Map

<https://www.youtube.com/watch?v=uu20xO9AaU4>

A [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) is an object that maps keys to values. A map cannot contain **duplicate keys**: Each key can map to at most one value. It models the mathematical function abstraction. The Map interface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values).

The Java platform contains three general-purpose Map implementations: [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html), and [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html). Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, as described in [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.

**Example**

// Group employees by department

Map<Department, List<Employee>> byDept

// Compute sum of salaries by department

Map<Department, Integer> totalByDept

// Partition students into passing and failing

Map<Boolean, List<Student>> passingFailing

// Classify Person objects by city

Map<String, List<Person>> peopleByCity

// Cascade Collectors

Map<String, Map<String, List<Person>>> peopleByStateAndCity

1. **HashMap** is implemented as a hash table, and there is no ordering on keys or values.
2. **TreeMap** is implemented based on red-black tree structure, and it is **ordered by the key**.
3. **LinkedHashMap** preserves the insertion order
4. **Hashtable** is synchronized, in contrast to HashMap. It has an overhead for synchronization.

**Sorting Map --** All you have to do is change the implementation type of the Map from HashMap to TreeMap.

Map<String, Integer> m = **new** HashMap<String, Integer>();

Map<String, Integer>sorted=**new** TreeMap<>(m);

Similarly, you want to keep insertion order then simply by changing the implementation type of the map to LinkedHashMap.

Map<String, Integer> LinkedhMap=**new** LinkedHashMap<>(m);

|  |  |  |
| --- | --- | --- |
| single Operation | collection views | Comment |
| int **size**()  HasMap,if key=null is counted as well. | Set entrySet() | Return **Set<**Map.Entry<K,V>> |
| boolean isEmpty() | **keySet**() | return set of Keys |
| void **clear**() | **values**() | gives collection of values |
| V **put**(K key, V value) | void **putAll**(MapObject) | Non-Existing key,value to added and return NULL.  Existing key,values is replaced with new and return old  HashTable,adding key=NULL throw NullPointerException |
| V **remove**(Object key) |  | Non-Existing key, return NULL  Existing key,return deleted value |
| boolean **remove**(Object key, Object value) |  | If Key and value both are matching then return true  If anyone(Key,value)is mismatching then return false |
| V **get**(key) |  | How to get the value associated with a given key ,if key is not available then return NULL |
| Boolean **containsValue**() |  | Search map’s value and return boolean |
| Boolean **containsKey**() |  | Search map’s key and return boolean |
| **replace**(Key,replacingValue) |  |  |

## Fancy Uses of Collection Views: Map Algebra

1. suppose you want to know whether one Map is a submap of another — that is, whether the first Map contains all the key-value mappings in the second.

if (m1.entrySet().containsAll(m2.entrySet())) {

...

}

1. suppose you want to know whether two Map objects contain mappings for all of the same keys.

if (m1.keySet().equals(m2.keySet())) {

...

}

1. Suppose you want to know all the keys common to two Map objects.

Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());

commonKeys.retainAll(m2.keySet());

1. Suppose you want to remove all of the key-value pairs that one Maphas in common with another.

m1.entrySet().removeAll(m2.entrySet());

1. Suppose you want to remove from one Map all of the keys that have mappings in another.

m1.keySet().removeAll(m2.keySet());

## Multimaps

A *multimap* is like a Map but it can map each key to multiple values. It's a fairly simple matter to use a Map whose values are List instances as a multimap.

## HashMap

A Map, as you might know, is a collection of key-value pairs. It maps keys to values.

* A HashMap cannot contain duplicate keys.
* Java HashMap allows null values and the null key.
* HashMap is an unordered collection. It does not guarantee any specific order of the elements.
* Java HashMap is not thread-safe. You must explicitly synchronize concurrent modifications to the HashMap.
* Equals(): Two Map instances are equal if they represent the same key-value mappings.

<https://www.callicoder.com/java-hashmap/>

## TreeMap

The entries in a TreeMap are always sorted based on the natural ordering of the keys, or based on a custom [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) that you can provide at the time of creation of the TreeMap.

* A TreeMap cannot contain duplicate keys.
* TreeMap cannot contain the null key.

## Hashtable

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method based on key and the equals method.

1. It is similar to HashMap, but is synchronised.
2. Hashtable stores key/value pair in hash table.
3. In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
4. Hashtable does not allow null keys or values.

**Constructors:**

* Hashtable(): This is the default constructor. Default size is 11;
* Hashtable(int size): This creates a hash table that has initial size specified by size.
* Hashtable(int size, float fillRatio): This version creates a hash table that has initial size specified by size and fill ratio specified by fillRatio. fill ratio: Basically it determines how full hash table can be before it is resized upward.and its Value lie between 0.0 to 1.0
* Hashtable(Map m): This creates a hash table that is initialised with the elements in m.

## LinkedHashMap

LinkedHashMap is a **HashMap** and **linked list** implementation of the Map interface, with predictable iteration order.

It is same as HashMap with additional feature that it maintains insertion order. To maintain the order of elements, the linked hashmap modifies the Map.Entry class of HashMap by adding pointers to the next and previous entries:

* HashMap doesn’t maintain any order.
* TreeMap sort the entries in ascending order of keys.
* LinkedHashMap maintains the insertion order.

**Constructor:**

LinkedHashMap(int capacity): It is used to initialize a particular LinkedHashMap with a specified capacity.

**Method**

protected **boolean removeEldestEntry(Map.Entry eldest):** The method is used to return true when the map removes its eldest entry from the map.

#### Differences between HashMap and Hashtable in Java:

* Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.
* Hashtable uses Enumerator to iterate through elements.HashMap uses Iterator.
* Hashtable and HashMap does not maintain insertion order. The order is defined by the Hash function.

### Why HashTable doesn’t allow null and HashMap does?

To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can’t implement these methods.

HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.

key == null ? 0 : h = key.hashCode() .HashMap converts key=null as 0 and assign value. If we try to add more than one null then values will be replaced with latest value.

hM.put(null,"oldValue");

hM.put(null,"NewValue");

Here newValue will be replaced against null as key.

# How to iterate Map?

First of all, we cannot iterate a Map directly using iterators, because Map are not Collection.

Since all maps in Java implement Map interface, following techniques will work for any map implementation (HashMap, TreeMap, LinkedHashMap, Hashtable, etc.)

### What is Map.Entry?

**Map.Entry** is an interface. The entrySet() method returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

**Map.Entry** interface has below methods

**Object getKey()--**Returns the key for this map entry.

**Object getValue()-**Returns the value for this map entry.

**Object setValue(Object v)--**Sets the value for this map entry to v. A ClassCastException is thrown if v is not the correct type for the map

### Using Map.entrySet()

**Map.entrySet()** method returns a collection-view(Set<Map.Entry<K, V>>) of the mappings contained in this map.So we can iterate over key-value pair using getKey() and getValue() methods of Map.Entry<K, V>

**Map**<Integer, String> m1 = new **HashMap**<>();

map.put(1, "One");

map.put(2, "Two");

map.put(3, "Three");

**for** (Map.Entry<Integer,String> entry : m1.entrySet())

System.***out***.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());

Using keySet() --**return set of keys**

for (Integer name : m1.keySet())

System.out.println("key: " + name);

Using values() --**return collection of value**

for (String url : m1.values())

System.out.println("value: " + url);

### Using Iterator

**Iterator**<Map.Entry<Integer, String>> itr=mp.entrySet().iterator();  
while(itr.hasNext())  
{  
 Map.Entry<Integer, String> entry=itr.next();  
 System.*out*.println("Key--"+entry.getKey()+" Value--"+entry.getValue());  
}

### Using forEach() lambda

mp.**forEach**((K,V)->  
 {  
 System.*out*.println("Key--"+K+" Value--"+V);  
 }  
 );

## forEach() Method

The forEach() method in Java 8 is a straightforward way to do iteration over collection. It is a member of the Iterable interface

forEach(**(argument)** ->

{

//body

}

);

### List Iteration

List<String> ls=new ArrayList<>(Arrays.*asList*("One","Two","Three"));  
ls.forEach(x-> System.*out*.println(x));

### Set Iteration

Set<String> st=new HashSet<>(Arrays.*asList*("One","Two","Three"));  
st.forEach(x-> System.*out*.println(x));

### Map Iteration

mp.**forEach**((K,V)->  
 {  
 System.*out*.println("Key--"+K+" Value--"+V);  
 }  
 );

### Map entrySet() Iteration

mp.entrySet().forEach( entry ->  
{  
 System.*out*.println("Key--"+entry.getKey()+" Value--"+entry.getValue());  
});

## Count the occurrence of each character in a string

HashMap<Character, Integer> charCountMap = new HashMap<Character, Integer>();

char[] strArray = inputString.toCharArray();

for (char c : strArray) {

if (charCountMap.containsKey(c)) {

// If char is present in charCountMap,

// incrementing it's count by 1

charCountMap.put(c, charCountMap.get(c) + 1);

}

else {

// If char is not present in charCountMap,

// putting this char to charCountMap with 1 as it's value

charCountMap.put(c, 1);

}

}

charCountMap.forEach((x,y)->System.out.println(x+" "+y));

# Stream API

Java provides a new additional package in Java 8 called java.util.stream. It allows functional-style operations on the elements. You can use stream by importing java.util.stream package. To perform **filter**/**map**/**reduce** like operations with the collection of objects. A stream is a sequence of objects that supports various methods.

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source.For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.
* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Only applicable to **collection(list,set)** and **Arrays** utility class
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

## Stream().collect()

It is a method in Java gathers elements from a stream into collections like List or Set and Map

## Stream().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.

List<String> names = Arrays.asList("Reflection", "Collection", "Stream");

List<String> result = names.stream().filter(s -> s.startsWith("S")).collect(Collectors.toList());

System.out.println(result);

**Example**: filter() with multiple conditions

List<String> ls=Arrays.*asList*("one","Two","Three","ThreeFour");  
List<String> ls1=ls.stream().filter(str -> str.length() > 3 && str.length() < 10).collect(Collectors.*toList*());  
ls1.forEach(x-> System.*out*.println(x));

**Map()**

Every list value will be concatenated by “-appened”

List<String> ls1=ls.stream().map(x-> x+"-appened").collect(Collectors.*toList*());  
ls1.forEach(x-> System.*out*.println(x));

## Collectors

Collectors is a class which has method like toList(),toSet(),toMap()

## import java.util.stream.Collectors;

## import java.util.stream.Collectors.toList;

## import java.util.stream.Collectors.toMap;

## import java.util.stream.Collectors.toSet;

The map method is used to map the items in the collection to other objects. Below is mapping of **number** to **square of number**.

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.

### Collectors.toList()

List<Integer> number = Arrays.asList(2, 3, 4, 5);

List<Integer> square = (List) number.stream().map(p -> p \* p).collect(Collectors.toList());

System.out.println(square);

### Collector.toSet()

//converting list into set

List<String> name = Arrays.asList("Reflection", "Collection", "Stream");

Set<String> results = name.stream().sorted().collect(Collectors.toSet());

results.forEach(p -> System.out.print(p + " "));

### Collector.toMap(K,T)

productsList.add(new Product(1,"HP Laptop",25000));

productsList.add(new Product(2,"Dell Laptop",30000));

Map<Integer,String> productPriceMap =productsList.stream().collect(Collectors.toMap(p->p.id, p->p.name));

## 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));

## stream().sorted().collect()

The sorted method is used to sort the stream.

List<String> name = Arrays.asList("Reflection", "Collection", "Stream");

Set<String> results = name.stream().sorted().collect(Collectors.toSet());

results.forEach(p -> System.out.print(p + " "));

**Reduce**()

**T reduce(T identity, BinaryOperator<T> accumulator);**

– identity is initial value of type T which will be used as the first value in the reduction operation.

- accumulator is an instance of a BinaryOperator Function(Functional Interface) of type T.

Reducing in the context of Java 8 Streams refers to the process of combining all elements in the stream repeatedly to produce a single value which is returned as the result of the reduction operation.such as summation of all elements,finding the maximum element from among all the elements.

List<Integer> number3 = Arrays.*asList*(2, 3, 4, 5);

**int** even = number3.stream().filter(x -> x % 2 == 0).reduce(0, (a, b) -> a + b); //after filter(2,4) then summation

System.***out***.println(even);

### Stream().Count()

It gives count of stream collection.

List<Integer> number4 = Arrays.asList(2, 3, 4, 5);

System.out.println(number4.stream().filter(x -> x > 3).count());

### Stream().Sum()

Only applicable to Arrays.stream() and below scenario.

int[] arr = { 1, 3, 5, 7, 2, 4 };

System.out.println(Arrays.stream(arr).sum());

System.out.println(Arrays.stream(arr).count());

<https://www.javatpoint.com/java-8-stream>

<https://www.geeksforgeeks.org/stream-in-java/>

## Arrays with Stream

# What is Comparator ?

We often need to compare two values in our Java programs. Comparing primitive values like int, char, float is very easy and can be done with comparison operators like <, >, == etc.

But comparing objects is a little different. For example, how would you compare two Employees? how would you compare two Students? For this purpose, Java provides two interfaces called

## Comparable --comparision

public interface Comparable<T> {

public int compareTo(T o);

}

The **compareTo** method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

* negative, if this object is less than the supplied object.
* zero, if this object is equal to the supplied object.
* positive, if this object is greater than the supplied object.

Comparable is an public interfaces which is used to impose an natural ordering.Lists (and arrays) of objects that implement this interface can be sorted automatically by **Collections.sort** and **Arrays.sort**. To allow precisely control over the sort order , Comparators can be passed to a sort method (e.g Collections.sort()). Certain type of data structures such as TreeSet or TreeMap can also be sorted using Comparator.

If collection of object or array of object is available then you must use either Comparable or comparator to sort using

**Collections.sort** and **Arrays.sort**.if it is primitive type then not required to use comparator or comparable

## Comparator --sorting

public interface Comparator<T> {

int compare(T obj1, T obj2);

}

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) — an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method.

* returns negative value, if and only if obj1 has to come before obj2.
* returns positive value, if and only if obj1 has to come after obj2.
* returns zero, if and only if obj1 and obj2 are equal.

If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

## Difference Between Comparable and Comparator

If you see then logical difference between these two is Comparator in Java compare two objects provided to it , while Comparable interface compares "this" reference with the object specified. So only one object is provided which is then compared to "this" reference.

Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.

If any class implement Comparable interface in Java then collection of that object either list or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and object will be sorted based on there natural order defined by CompareTo method.

## When to use Comparable and Comparator

an object should implement Comparable if that is the clear **default** (natural)way to sort the class, and anyone would need to sort the class would generally want to do it that way.

Use Comparator if you want to define an external controllable ordering behaviour, this can **override the default ordering behaviour.**

## Lambda Expression use

**Comparable** and **Comparator** are functional interface (only one abstract method) then we can use Lambda Expression at its place.

**Sorting**

Integer[] arr= {1,22,10,9,33,5};

Arrays.sort(arr,(o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

ArrayList<Integer> al = new ArrayList<Integer>();

Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

TreeSet<Integer> h =new TreeSet<Integer>((o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

TreeMap<Integer, String> m =new TreeMap<Integer, String>((o1, o2) -> (o1 > o2) ?-1 : (o1 < o2) ? 1 : 0);

# Collections Interview Question

state dropdown is dependent country dropdown . need to validate state dropdown

1. if all are available
2. any state is missing
3. any state is duplicate

# Java Input Output Method

Package java.io

Stream—Sequence of data in **byte**(8 bit)/**Character**(16 bit) format --int(32bit)

**Character Streams**-16bit Stream //only file

**Byte Streams**-8bit data // used to read all types of data. File,image,audio,video

## File Handling Class

1. File
2. FileInputStream --Byte Stream
3. FileOutputStream --Byte Stream
4. BufferedInputStream --Buffered Byte Stream
5. BufferedOutputStream --Buffered Byte Stream
6. FileReader --Character Stream
7. FileWriter --Character Stream
8. BufferedReader --Buffered Character Stream
9. BufferedWriter --Buffered Character Stream

**Data Streams**-Stream of primitive data type values (boolean, char, byte,int,String) **DataInput**(DataInputStream)**DataOutput** (DataOutput)- It takes buffered stream(Byte or character) as constructor.**readDouble**(),**readInt**()

**Object Streams-** Just as data streams support I/O of primitive data types, object streams support I/O of objects

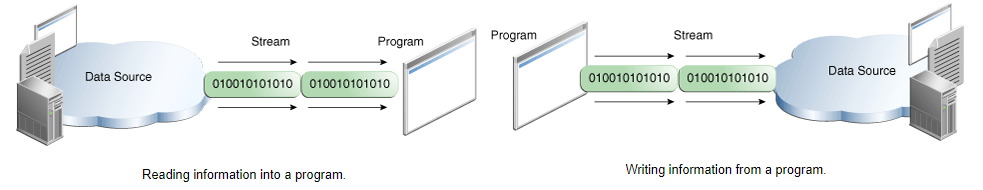
**ObjectInput**(ObjectInputStream),**ObjectOutput**(ObjectOutputStream)-**readObject**(),**writeObject**()

**Standard Stream**: Standard Input->**System.in**; Standard Output->**System.out**

## I/O Streams Properties

Read bit Stream - A program uses an input stream to read data from a source, one item at a time.

Write bit Stream - A program uses an output stream to write data to a destination, one item at time.



Source—File/Keyboard/system.in Target—File/system.out

## Byte Streams Properties

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.

### read() method

This method read 1 byte stream at a time and next 1 byte in next call and return ASCII value (integer). Need to covert that ASCII value by (char)i… It return -1 when reaches end of file.

### Reading file

File f1 = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\abc.txt");  
FileInputStream is = new FileInputStream(f1);*//start-- FileInputStream-----------*int i = is.read();  
while (i != -1) {  
 System.*out*.print((char) i);  
 i = is.read();  
}  
is.close();*//end-------------*FileInputStream is1 = new FileInputStream(f1);*//start-- BufferedInputStream-----------*BufferedInputStream bis = new BufferedInputStream(is1);  
int ii = bis.read();  
while (ii != -1) {  
 System.*out*.print((char) ii);  
 ii = bis.read();  
}  
is1.close();  
bis.close();*//end-------------*FileReader fr = new FileReader(f1);*//start-- File Reader-----------*int j = fr.read();  
while (j != -1) {  
 System.*out*.print((char) j);  
 j = fr.read();  
}  
fr.close();*//end-------------*FileReader fr1 = new FileReader(f1);*//start-- BufferedFile Reader-----------*BufferedReader br = new BufferedReader(fr1);  
int jj = br.read();  
while (jj != -1) {  
 System.*out*.print((char) jj);  
 jj = br.read();  
}

//--------------Using readline() method

String line=br.readLine();  
while(line!=null) {  
 System.*out*.println(line);  
 line = br.readLine();  
}

//-------------  
fr1.close();  
br.close();*//end-------------*

## Reading file and writing into other file

It will override previous content

// FileOutputStream fos = new FileOutputStream(outFile,**true**); --it appends content

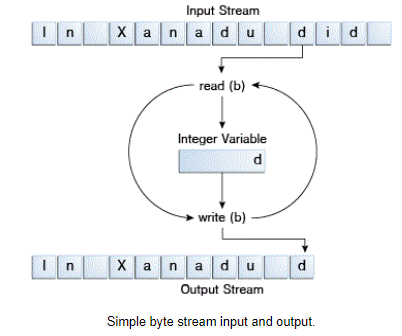
File inFile = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\abc.txt");  
File outFile = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\out.txt");  
*//----using fileOutputStream*FileInputStream fis = new FileInputStream(inFile);  
FileOutputStream fos = new FileOutputStream(outFile);  
int i = fis.read();  
while (i != -1) {  
 fos.write(i);  
 i = fis.read();  
}  
fis.close();  
fos.close();*//----end  
//----using BufferedOutputSteam*FileInputStream fis1 = new FileInputStream(inFile);  
FileOutputStream fos1 = new FileOutputStream(outFile);  
BufferedOutputStream bos = new BufferedOutputStream(fos1);  
int ii = fis1.read();  
while (ii != -1) {  
 bos.write(ii);  
 ii = fis1.read();  
}  
fis1.close();  
bos.close();  
fos1.close();*//----end  
//----using FileWriter*FileInputStream fis2 = new FileInputStream(inFile);  
FileWriter fw = new FileWriter(outFile);  
int j = fis2.read();  
while (j != -1) {  
 fw.write(j);  
 j = fis2.read();  
}  
fis2.close();  
fw.close();*//----end  
//----using BufferedFileWriter*FileInputStream fis3 = new FileInputStream(inFile);  
FileWriter fw1 = new FileWriter(outFile);  
BufferedWriter bw=new BufferedWriter(fw1);  
int jj = fis3.read();  
while (jj != -1) {  
 bw.write(jj);  
 jj = fis3.read();  
}  
fis3.close();  
bw.close();  
fw1.close(); *//----end*

### Always Close Streams

Closing a stream when it's no longer needed is very important.This practice helps avoid serious RAM memory leaks.

Whenever we open an IO steam, it always takes up a bit of system resources. **The resources aren’t released until the IO stream**close()**method is called.**

We often wrap a [**BufferedOutputStream**](https://www.baeldung.com/java-outputstream#outputstream-buffering) around an [**OutputStream**](https://www.baeldung.com/java-outputstream) to provide buffered capability to reduce the overhead of each write operation.  it’s possible that the last chunk of data isn’t yet written to the target, leading to data corruption. **Calling the**close()**method invokes**flush()**to write the remaining data in the buffer.**



## Character Stream Properties

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter.

public class CopyCharacters {

public static void main(String[] args) throws IOException {

FileReader inputStream = null;

FileWriter outputStream = null;

try {

inputStream = new FileReader("xanadu.txt");

outputStream = new FileWriter("characteroutput.txt");

int c;

while ((c = inputStream.read()) != -1) {

outputStream.write(c);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

### Difference in reading/writing between byte and character stream

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.InputStream is used to read binary data(**images**,**video**,**audio**,jar files), while Reader is used to read text data.

**What is difference between binary and text data?**

well everything you read is essentially bytes, but to convert a byte to text, you need a character encoding scheme. Reader classes uses character encoding to decode bytes and return characters to caller.

### Character Streams-Use Byte Streams[InputStreamReader,OutputStreamWriter]

Character streams are often "wrappers" for byte streams. Character stream uses the byte stream.FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: **InputStreamReader** and **OutputStreamWriter**.Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

## Buffered Stream Properties

Most of the examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS.This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer. The native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full. A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class.

There are four buffered stream classes

* BufferedInputStream
* BufferedOutputStream
* BufferedReader
* BufferedWriter

#### Ex1

It internally uses buffer mechanism to make the performance fast.

**FileInputStream** fin = new **FileInputStream**("file1.txt");

**BufferedInputStream** bin = new **BufferedInputStream**(fin);

int c=0;

while ((c=bins.read())!=-1)

{ System.out.print((char)c);}

#### Ex2

**FileOutputStream** fout=new **FileOutputStream**("D:\\testout.txt");

**BufferedOutputStream** bout=new **BufferedOutputStream**(fout);

String s="Welcome to javaTpoint.";

byte b[]=s.getBytes();

bout.write(b);

bout.flush();

bout.close();

fout.close();

### Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.Some buffered output classes support autoflush e.g PrintWriter.To flush a stream manually, invoke its flush method.

## Sanner

Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

while (s.hasNext()) {

if (s.hasNextDouble()) {

sum += s.nextDouble();

} else {

s.next();

}

}

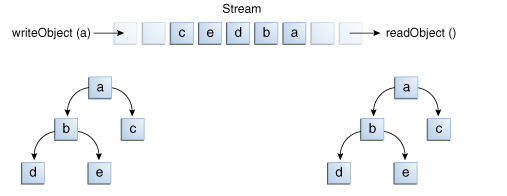
**Console Class**

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

# Object Stream properties

This object a contains references to objects b and c, while b contains references to d and e. Invoking writeobject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also. When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

Object ob = new Object();

out.writeObject(ob);

out.writeObject(ob);

**Standard Stream Properties**

Standard Streams are a feature wherein they read input from the keyboard and write output to the display.

The Java platform supports three Standard Streams:

**Standard Input**, accessed through System.in-returns InputStream object

**Standard Output**, accessed through System.out;returns PrintStream object

**Standard Error**, accessed through System.err.returns PrintStream object

These objects are defined automatically and do not need to be opened. These 3 streams are initialized by the Java runtime when a JVM starts up. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as PrintStream objects.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

InputStreamReader cin = new InputStreamReader(System.in);

**PrintStream**

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

System.out.println(“Hello”); println is a method of class **printStream**

By default, outputStream is console. if you want file as outputStream then code as below

FileOutputStream fout=new FileOutputStream("kk.txt");

PrintStream pout=new PrintStream(fout); //OutputStream as constructor input or new File(“a.txt”) as constructor

pout.println("Hello Java"); // This will write in file kk.txt

**Exchanging system Stream**

Even if the 3 System streams are static members of the java.lang.System class and are pre-instantiated at JVM startup, you can change what streams to use for each of them

To set

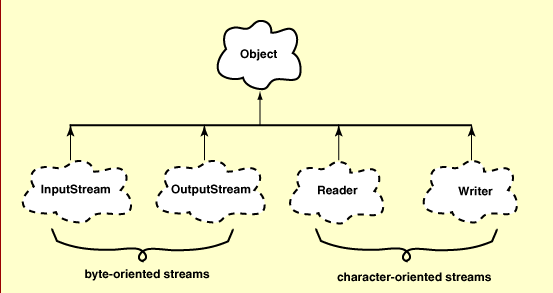
a new System stream, use one of the methods System.setIn(), System.setOut() or System.setErr().

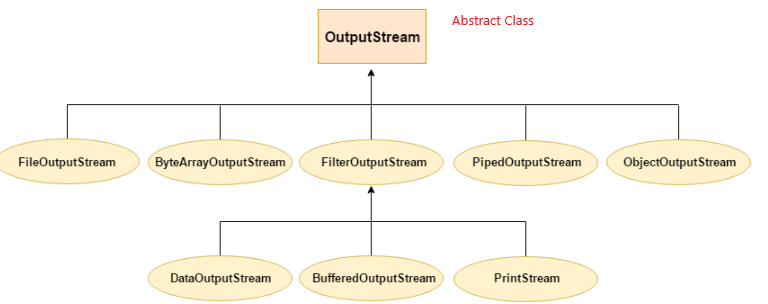
OutputStream output = new FileOutputStream("c:\\data\\system.out.txt");

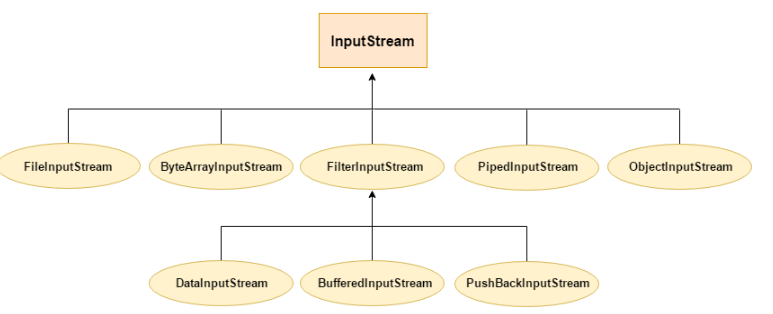
PrintStream printOut = new PrintStream(output);

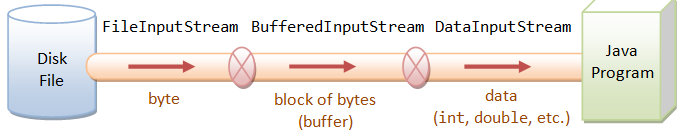
System.setOut(printOut);

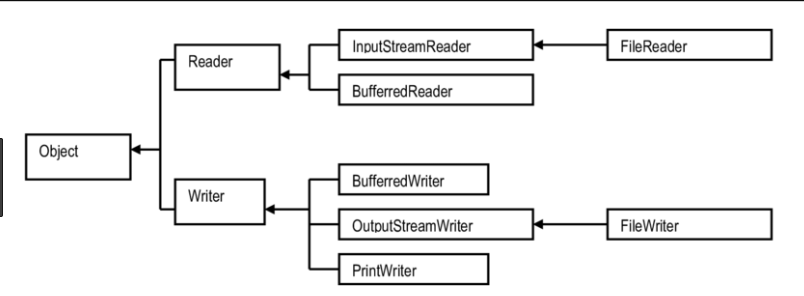
Now all data written to System.out should be redirected into the file "c:\\data\\system.out.txt"











Object🡨InputStream🡨FileInputStream

Object🡨OutputStream🡨FileOutputStream

# ObjectInputStream and ObjectOutputStream --Object to binary file

The process of writing an object to an output stream is called **Serialization**.

The process of reconstructing an object from an input stream is called **deserialization**

he Java ObjectOutputStream class (java.io.ObjectOutputStream) enables you to write Java objects to an OutputStream.

The ObjectOutputStream is used to write the Java objects, and the ObjectInputStream is used to read the objects again.

ObjectOutputStream objectOutput = new ObjectOutputStream(new FileOutputStream("binary.txt"));

List<Student> listStudent = new ArrayList<>();

listStudent.add(new Student("Alice", "02-15-1993", false, 23, 80.5f));

listStudent.add(new Student("Brian", "10-03-1994", true, 22, 95.0f));

for (Student student : listStudent) {

objectOutput.writeObject(student);

}

ObjectInputStream objectInput = new ObjectInputStream(new FileInputStream("binary.txt"));

Student student = (Student) objectInput.readObject();

System.out.print(student.getName() + "\t");

System.out.print(student.getBirthday() + "\t");

System.out.print(student.getGender() + "\t");

System.out.print(student.getAge() + "\t");

System.out.println(student.getGrade());

## OutputStreamWriter extends writer

The Java.io.OutputStreamWriter class is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using a specified charset.

## Byte versus Character Stream class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | Stream of Byte | Remarks  (Byte Stream) | Stream of Character | Remarks(Character Stream) |
| Read | InputStream | Abstract Class | Reader | Abstract Class |
| FileInputStream | (Input) File🡪Byte Stream(output)  **Input**=String e.g fileName  **Return** InputStream object | FileReader | (input)File🡪Character Stream(output)  **Input**=String e.g fileName  **Return** Reader object |
|  | **InputStreamReader**  (input)Byte Stream🡪Charatcer Stream(output) |  | 1. **Input** =InputStream, read file using FileInputStream 2. InputStreamReader read console input as system.in   **Return** Reader object |
| BufferedInputStream | (input)Byte Stream🡪Buffered Stream(output)  **Input** =InputStream, read file using FileInputStream  Read() | BufferedReader | (input)Charatcer Stream🡪Buffered Charatcer Stream(output)  Argument as FileReader Or InputStreamReader  Method: read() Or readLine() |
|  | ObjectInputStream | Byte Stream🡪 Object Stream |  |  |
| Write | OutputStream | Abstract Class | Writer | Abstract Class |
| FileOutputStream | File🡪Byte Stream | FileWriter printWriter | File🡪String writer |
| **OutputStreamWriter** | Byte Stream🡪String Writer |  |  |
| BufferedOutputStream | Byte Stream🡪Buffered Byte Stream  Write() | BufferedWriter | String writer 🡪 Buffered String Writer  Argument as Filewriter Or OutputStreamwriter |
|  | ObjectOutputStream | Byte Stream🡪 Object Stream |  |  |

# Reader Class

Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

#### Subclasses of Reader

* BufferedReader
* InputStreamReader
* FileReader
* StringReader

#### read()

The read() method of [Reader](https://www.geeksforgeeks.org/java-io-reader-class-java/) Class in Java is used to read a **single character** from the stream.

This method is declared as abstract method. It means that the subclasses of Reader abstract class should override this method if desired operation needs to be changed while reading a character.

**Return Value**: This method returns an integer value which is the integer value read from the stream. It can range from 0 to 65535. Else it returns **-1** if no character has been read.

## FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like [FileInputStream](https://www.javatpoint.com/java-fileinputstream-class) class.

It is character-oriented class which reads data character by character and used for [file](https://www.javatpoint.com/java-file-class) handling in [java](https://www.javatpoint.com/java-tutorial).

 FileReader fr=**new** FileReader("D:\\testout.txt");

**int** i;

**while**((i=fr.read())!=-1)

          System.out.print((**char**)i);

          fr.close();

**example-2**

File f = new File("E:\\Mars\_workspace\\MindQ\\src\\FileHandling\\kk4.txt");

FileReader fr = new FileReader(f);

char[] ch = new char[(int) f.length()];

fr.read(ch);

for (char ch1 : ch)

System.out.print(ch1);

## BufferedReader

Main advantage is this reads file line by line instead of character by character. This can not communicate to file directly but through reader object.Important method is **readline().**

* Very important to close the file (or at least call flush())
* closing bufferedReader will automatically close reader

Input Type: FileReading Type: Single Character

 Java BufferedReader class is used to read the text from a character-based input stream. It makes the performance fast.

 FileReader fr=**new** FileReader("D:\\testout.txt");

 BufferedReader br=**new** BufferedReader(fr);

**int** i;

**while**((i=br.read())!=-1){

          System.out.print((**char**)i);

          }

          br.close();

          fr.close();

 Input Type: FileReading Type: Single Line

It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits [Reader](https://www.javatpoint.com/java-reader-class) [class](https://www.javatpoint.com/object-and-class-in-java).

FileReader fr2=**new** FileReader("kk.txt");

BufferedReader br1=**new** BufferedReader(fr2);

String str=**null**;

**while** ((str=br1.readLine())!=**null**)

{

System.***out***.print(str);

}

# InputStreamReader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters

**Ex1**

**InputStreamReader** r=**new** **InputStreamReader**(System.***in***);

**BufferedReader** br=**new** **BufferedReader**(r);

System.***out***.println("Enter your name");

String name=br.readLine();

System.***out***.println("Welcome "+name);

**Ex2**

InputStream in =**new** FileInputStream("input.txt");

**Reader** rdr=**new** **InputStreamReader**(in);

**BufferedReader** br=**new** **BufferedReader**(rdr);

String line=br.**readLine**();

**while**(line!=**null**)

{

System.***out***.println(line);

line=br.readLine();

}

**Ex3**

InputStream in =**new** FileInputStream("input.txt");

InputStreamReader inr=**new** InputStreamReader(in);

**char**[] array = **new** **char**[10]; //it reads max 10 characters only

inr.read(array);

System.***out***.println(array);

# OutputStreamWriter

It is intended to wrap an OutputStream, thereby turning the byte based output stream into a character based Writer. It is useful when you want to write byte to character based in a file.

**For File**

File outFile = new File("C:\\kk\\IdeaWorkspace\\file\\out.txt");  
FileOutputStream fos=new FileOutputStream(outFile,true);  
OutputStreamWriter osw=new OutputStreamWriter(fos);  
osw.write("\n");  
osw.write("This is Third line\n");  
osw.close();

**For Console**

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(System.***in***));

String strLine = in.readLine();

BufferedWriter out = **new** BufferedWriter(**new** OutputStreamWriter(System.***out***));

out.write(strLine);

out.flush();

in.close();

out.close();

# Writer Class

It is an abstract class for writing to character streams. The methods that a subclass must implement are write(char[], int, int), flush(), and close().

Writer w = new FileWriter("output.txt");

String content = "I love my country";

w.write(content);

w.close();

## FileWriter

The Java FileWriter class (java.io.FileWriter) makes it possible to write characters to a file. In that respect it works much like the FileOutputStream except that a FileOutputStream is byte based, whereas a FileWriter is character based.

When you create a Java FileWriter you can decide if you want to overwrite any existing file with the same name, or if you want to append to any existing file. You decide that by choosing what FileWriterconstructor you use.

* Very important to close the file (or at least call flush()),
* otherwise the file isn't actually written

Writer fileWriter = new FileWriter("("output.txt"", true); //appends to file

Writer fileWriter = new FileWriter("("output.txt"", false); //overwrites file

Writer w = new FileWriter("output.txt");

String content = "I love my country";

w.write(content);

w.close();

**Drawback**

* Writing newline is not possible.
* Write data character to character

### BufferedWriter

Practically the same as FileWriter, but the writing is done to a buffer before actually writing to the file. More efficient if you have lots of writing that is smaller than the buffer size.

**newLine()** is extra method which adds line in between in the file.

* Very important to close the file (or at least call flush()),
* otherwise the file isn't actually written
* closing bufferedWriter will automatically close writer

FileWriter writer = new FileWriter("D:\\testout.txt");

BufferedWriter buffer = new BufferedWriter(writer);

buffer.write("Welcome to javaTpoint.");

**buffer.newline();**

bWriter.write(65); //int will converted into corresponding ASCII value. Here A is written

buffer.close();

### PrintWriter

Provides additional methods for formatting. It enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values. The PrintWriter class has all the same methods as the PrintStream except for the methods to write raw bytes. Being a Writer subclass the PrintWriter is intended to write text.

When you are finished writing characters to the Java PrintWriter you should remember to close it. Closing a PrintWriter will also close the Writer instance to which the PrintWriter is writing.

PrintWriter writer = new PrintWriter("filename.txt");

writer.println("The lazy orange for went to sleep");

writer.print(123);

writer.prinf(format,argument);

writer.close();

## How to delete File content

**Example1**

PrintWriter pw = new PrintWriter("filename.txt");

pw.print("");// this is optional

pw.close();//just closing printwriter will delete file content

**Example2**

new FileWriter(FILE\_PATH, false).close(); //using FileWriter

new FileOutputStream(FILE\_PATH).close(); //Using FileOutputStream

**what is difference between write and print method of printwriter class?**

Pw.write(100) will write corresponding ASCII character “d” in file whereas pw.print(100) will write int value 100 in file.

# byte []

A byte array is an array of bytes.You could use a byte array to store a collection of binary data, for example, the contents of a file.

## String to byte array

String str = "PANKAJ";

byte[] byteArr = str.getBytes();

## byte array to String

String str=new String(byteArr)

Practicle Example

File f = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\abc.JPG");  
System.*out*.println(f.length());  
byte[] imageData=new byte[(int)f.length()];  
FileInputStream fis = new FileInputStream(f);  
fis.read(imageData); *//read image into byte array*fis.close();

## Best Practices

* **Memory Efficiency**: Use byte when you need to save memory, especially in large arrays.
* **Range Checking**: Always ensure that the values assigned to a byte variable are within the range of -128 to 127 to avoid unexpected behavior.

## Byte Overflow

byte a = 127;  
a++;  
System.*out*.println("Value of a after overflow: " + a);

// output: -128 which is wrong

# Stream

A stream is a sequence of data. In Java, astream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow

## OutputStream --abstract class

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket

write(byte[]) --is used to write an array of byte to the current output stream.

write(int) – if write(65) means ASCII value of A=65, A is written in file

## InputStream --abstract class

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

 int read() --reads the next byte of data from the input stream. It returns -1 at the end of the file.

### FileInputStream&FileOutputStream

#### FileInputStream(File file)

This creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

**public** **class** FileInputStream **extends** InputStream

 FileInputStream is=**new** FileInputStream ("D:\\testout.txt");

**int** i;

**while**((i=is.read())!=-1)

          System.out.print((**char**)i);

          is.close();

#### FileOutputStream(File file)

This creates a file output stream to write to the file represented by the specified File object.

[void write(byte[] b)](https://www.tutorialspoint.com/java/io/fileoutputstream_write_byte.htm)

This method writes b.length bytes from the specified byte array to this file output stream.

**public** **class** FileOutputStream **extends** OutputStream

**Example**:

FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

String s="Welcome to javaTpoint.";

byte b[]=s.getBytes();//converting string into byte array

fout.write(b);

fout.close();

# File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

## File Operation

Directory creation

1. File Creation
2. File searching
3. File Modification
4. File deletion
5. File information

## File class’s Method

|  |  |  |
| --- | --- | --- |
| File | Directory | Description |
| **createNewFile**() | **mkdir**() | It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. |
| **isFile**() | **isDirectory**() | It tests whether the file denoted by this abstract pathname is a normal file. |
| **getName**() |  | Return file/Directory name |
| **getAbsoluteFile()** |  | Return complete file/Directory path |
|  | String[] **list**() | Only applicable to directory, List of file name |
|  | File[] **listFiles**() | Only applicable to directory, List of file name--f.getName() |
| **exist**() |  | File/Directory availability check |
| **getParent**() |  | For File “**C:\kk\file\abc.txt**” return “**C:\kk\file**”  For Directory “**C:\kk\file**” return “**C:\kk**” |
| **length**() |  | Size of file/Directory in **number of bytes** |
| **delete**() |  | Delete Files/directory |

File f=new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file");  
if(f.mkdir()) //directory “file” will be created if not exist  
 System.*out*.println("Directory created");  
else System.*out*.println("directory not created");

File f1 = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\abc.txt");  
if (f1.createNewFile()) //abc “file” will be created if not exist  
 System.*out*.println("File created");  
else System.*out*.println("File already exist");

--

File f = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file");  
File f1 = new File("C:\\kk\\IdeaWorkspace\\CuCumberTestngParallelExecution\\file\\abc.txt");  
if (f.isDirectory()) {  
 System.*out*.println(f.getName());  
 System.*out*.println(f.getParent());  
 System.*out*.println(f.getAbsoluteFile());  
 System.*out*.println(f.length());

for (**String** ff : f.list())  
 System.*out*.println(ff);

for (**File** ff : f.listFiles())  
 System.*out*.println(ff.getName());

}  
if (f1.isFile()) {  
 System.*out*.println(f1.getName());  
 System.*out*.println(f1.getParent());  
 System.*out*.println(f1.getAbsoluteFile());  
 System.*out*.println(f1.length());  
}

# FileUtils utility class

Package Name: org.apache.commons.io.FileUtils

File screenshotFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);

**FileUtils**.copyFile(screenshotFile, new File("filename\_with\_path"));

File myfile = new File("src/main/resources/myfile.txt");

FileUtils.touch(myfile);

File is created with empty content

FileUtils.deleteQuietly(myfile);

File is deleted

## Copy File

File myfile1 = new File("src/main/resources/myfile.txt");

File myfile2 = new File("src/main/resources/myfile2.txt");

FileUtils.copyFile(myfile1, myfile2);

File docs = new File("src/main/resources/docs");

FileUtils.copyFileToDirectory(myfile2, docs); //copy file into other directory

## Reading File

File f = new File("C:\\kk\\IdeaWorkspace\\file\\abc.txt");  
List<String> line = **FileUtils**.***readLines***(f);  
for (String s : line)  
 System.*out*.println(s);

## Writing File

File f = new File("C:\\kk\\IdeaWorkspace\\file\\abc.txt");  
List<String> ls=new ArrayList<>();  
ls.add("This is 5th line");  
ls.add("This is 6th line");  
FileUtils.writeLines(f,ls);

## Listing files in a directory

 To find files that match either txt or html extensions.

File f = new File("C:\\kk\\IdeaWorkspace\\file ");

Collection<File> ls = FileUtils.*listFiles*(f, new String[]{"txt", "html"}, true);  
for (File ff : ls)  
 System.*out*.println(ff.getName());

# Java Architecture for XML Binding (JAXB)

is a Java standard that defines how Java objects are converted from and to XML. It uses a standard set of mappings.JAXB defines an API for reading and writing Java objects to and from XML documents.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @XmlRootElement(namespace = "namespace") | Define the root element for an XML tree |
| @XmlType(propOrder = { "field2", "field1",.. }) | Allows to define the order in which the fields are written in the XML file |
| @XmlElement(name = "neuName") | Define the XML element which will be used. Only need to be used if the neuNeu is different than the JavaBeans Name |

**HttpURLConnection**

HttpURLConnection class is an abstract class directly extending from [**URLConnection** class](https://www.geeksforgeeks.org/reading-url-using-urlconnection-class/). It includes all the functionality of its parent class with additional HTTP specific features. HttpsURLConnection is another class which is used for the more secured HTTPS protocol.

**getResponseCode() :**Used to retrieve the response status from server.

**setRequestMethod(“POST”) :**Used to set the request method. Default is GET.

**getRequestMethod() :**Returns the request method

**getResponseMessage() :**Retrieves the response message

**setRequestProperty("**Accept", "application/json")

**getHeaderField() :**Returns the nth header field, or null if it does not exist.

# Parallel Execution

While testing a web application it is very important to test the application on different browsers. We can achieve this by using Selenium Web driver and TestNG. If there are more number of scripts to be executed and executing them on each and every browsers sequentially is time consuming. This can be avoided using a concept in Selenium called **Parallel Execution**.

TestNG provides multiple ways to execute tests in separate threads. In testng.xml, if we set **'parallel'** attribute on the tag to **“tests”**, testNG will run all the ‘@Test’ methods in tag in the same thread, but each tag will be in a separate thread.

If we want to run methods/classes in separate threads, we need to set 'parallel' attribute on the tag to **'methods'** / **'classes'**

parallel=“tests” : All test will be executed parallelly

parallel=“methods” : All methods within tests will be executed parallelly but test will be executed one after another

parallel=“classes” :All classes within tests will be executed parallelly but test will be executed one after another

parallel=“none” : no parallel execution

# Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Using maven we can build and manage any Java based project.

## Objective

The primary goal of Maven is to provide developer with the following −

* A comprehensive model for projects, which is reusable, maintainable, and easier to comprehend.
* Plugins or tools that interact with this declarative model.

Maven project structure and contents are declared in an xml file, pom.xml, referred as Project Object Model (POM), which is the fundamental unit of the entire Maven system.

There are two software available in Maven:

* **Maven Eclipse Plug-in**
* **Maven command line plug-in**

### Install command line Maven

It is used to execute the Selenium test script in the command prompt without an Eclipse, and this software should be installed explicitly.

1. Download Apache Maven version 3 and save it on your local machine. <https://maven.apache.org/download.cgi>

🡪”Binary zip archive” link

2. Extract the file you just downloaded using your preferred archive extraction tool.

3. Set path in system environment variable to add the bin directory of the created directory apache-maven-3.5.0 to the PATH environment variable

### Install Maven Eclipse plugin

1. Open Eclipse IDE

2. Click Help -> Install New Software...

3. Click Add button at top right corner

4. At pop up: fill up Name as "M2Eclipse" and Location as <http://download.eclipse.org/technology/m2e/releases/>

5. Now click OK

After that installation would be started.

### How to check maven local directory

1. Open system environment variable for your pc
2. Open PATH variable
3. Check for keyword “apache-maven”
4. **OR** mvn –version 🡪 Maven Home gives maven path in your local system

#### Does Maven need to be installed when we have m2e?

No and no. m2e has an embedded Maven, but you don't have to use it.m2e comes with its own copy of the relevant libraries.Unless you have a specific requirement for a specific Maven version, you don't need a separate installation.

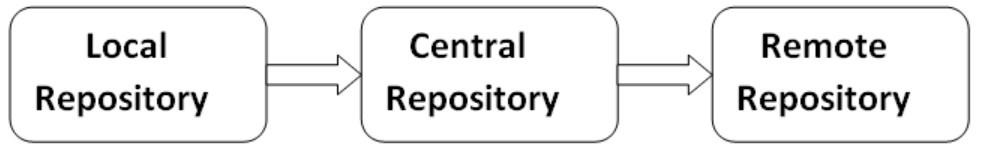
## Maven repository

A maven repository is a directory of packaged JAR file with pom.xml file. Maven searches for dependencies in the repositories. There are 3 types of maven repository:

1. Local Repository
2. Central Repository
3. Remote Repository

Maven searches for the dependencies in the following order:

**Local repository** then **Central repository** then **Remote repository**.



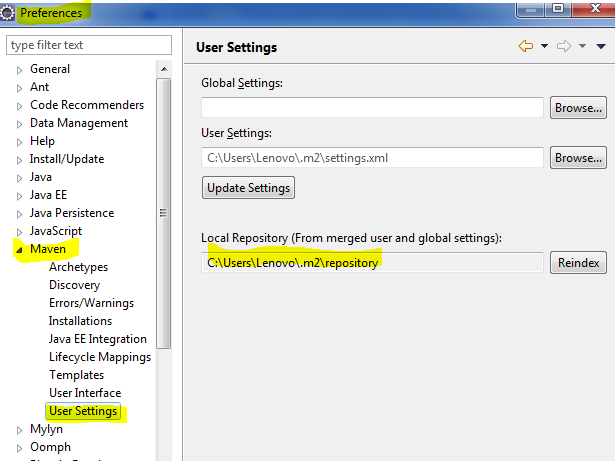
### Maven Central Repository

Maven central repository is located on the web. It has been created by the apache maven community itself.

The path of central repository is: <http://repo1.maven.org/maven2/>.

You can search any repository from Maven official website <https://mvnrepository.com>

### local maven repository

****

### What is Setting.xml File?

🡪Locate your Maven settings.xml file from Eclipse -> Preferences -> Maven -> User Settings

**Default locations: For Windows:**C:\Users\{username}\.m2\

🡪Add the following XML block to your settings.xml file

<proxies>

  <!-- Proxy for HTTP -->

  <proxy>

   <id>optional</id>

   <active>true</active>

   <protocol>http</protocol>

   <username>proxyuser</username>

   <password>proxypass</password>

   <host>proxy.host.net</host>

   <port>80</port>

   <nonProxyHosts>local.net</nonProxyHosts>

  </proxy>

</proxies>

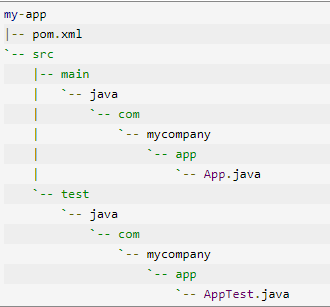
🡪 Modify the host, port, username and password information in the above XML block to match your proxy configuration.

🡪 Go back to Eclipse -> Preferences -> Maven -> User Settings and click "**Update Settings**" to reload the proxy configuration changes.

### What is Archetype in Maven?

#### Folder Structure

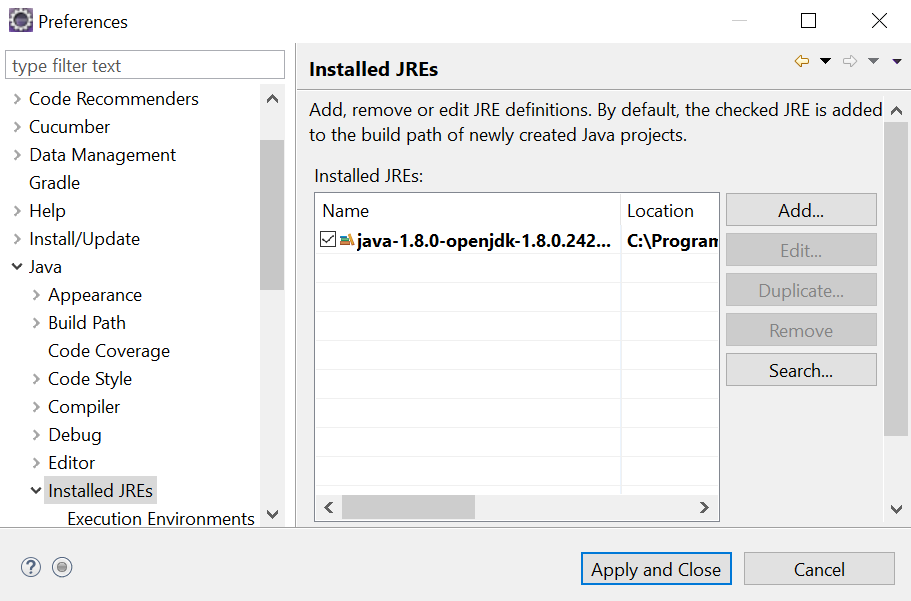
|  |  |
| --- | --- |
| **Item** | **Default** |
| source code | ${basedir}/src/main/java |
| Resources | ${basedir}/src/main/resources |
| Tests | ${basedir}/src/test |
| Complied byte code | ${basedir}/target |
| distributable JAR | ${basedir}/target/classes |

****

# Maven Project using JRE/JDK

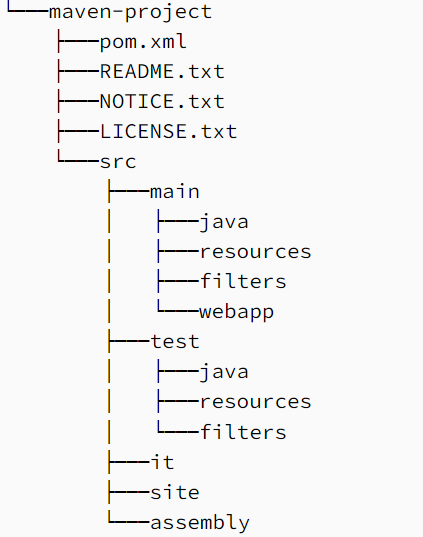
Path>> Windows->Preferences->Java->Installed JREs.

It should point to JDK instead of JRE as it is required to build project.Point to JDK folder before **bin** folder.



# What is POM.xml(Project Object Model) File?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/test/java; and so on. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal.



[**https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html**](https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)

**Note:** Inside src/test/java , All test class must end with “Test” keyword otherwise that will not be considered as test class. E.g seleniumTest.java, MyappTest.java

## Project Identifiers

The minimum requirement for a POM are the following:

* <project> as root element
* <modelVersion> should be set to 4.0.0
* <groupId> Package highrarchy
* <artifactId> Project Name
* version - the version of the artifact under the specified group

Here's an example:

1. <project>
2. <modelVersion>4.0.0</modelVersion>
3. <groupId>com.mycompany.app</groupId>
4. <artifactId>my-app</artifactId>
5. <version>1</version>
6. </project>

A POM requires that its groupId, artifactId, and version be configured. These three values form the project's fully qualified artifact name. This is in the form of <groupId>:<artifactId>:<version>. As for the example above, its fully qualified artifact name is "com.mycompany.app:my-app:1".

* Model version is for maven model .This element indicates what version of the object model this POM is using.
* Every Maven project has a **packaging type**. If it is not specified in the POM, then the **default value "jar"** would be used.
* Furthermore, you can see that in the minimal POM the **repositories** were not specified. If you build your project using the minimal POM, it would inherit the repositoriesconfiguration in the Super POM. Therefore when Maven sees the dependencies in the minimal POM, it would know that these dependencies will be downloaded from http://repo.maven.apache.org/maven2 which was specified in the Super POM.
* **packaging** This element indicates the package type to be used by this artifact (e.g. JAR, WAR, EAR, etc.). This not only means if the artifact produced is JAR, WAR, or EAR but can also indicate a specific lifecycle to use as part of the build process. (The lifecycle is a topic we will deal with further on in the guide. For now, just keep in mind that the indicated packaging of a project can play a part in customizing the build lifecycle.) The default value for the packaging element is JAR so you do not have to specify this for most projects.
* **name** This element indicates the display name used for the project. This is often used in Maven's generated documentation.
* **url** This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
* **description** This element provides a basic description of your project. This is often used in Maven's generated documentation.

## Super POM

The Super POM is Maven’s default POM. All POMs inherit from a parent or default (despite explicitly defined or not). This base POM is known as the Super POM, and contains values inherited by default.

It will include your current project POM + super POM

**mvn** help:effective-pom

Create a pom.xml in any directory on your computer.Use the content of above mentioned example pom.

In example below, We've created a pom.xml in C:\MVN\project folder.

Now open command console, go the folder containing pom.xml and execute the following mvn command.

C:\MVN\project>mvn help:effective-pom

In above pom.xml, you can see the default project source folders structure, output directory, properties, dependencies, repositories, build(plugins), reporting and profiles.

## Project Inheritance

Elements in the POM that are merged are the following:

1. dependencies
2. developers and contributors
3. plugin lists (including reports)
4. plugin executions with matching ids
5. plugin configuration
6. resources

The Super POM is one example of project inheritance, however you can also introduce your own parent POMs by specifying the parent element in the POM, as demonstrated in the following examples.

**Example:**

As an example, let us reuse our previous artifact, com.mycompany.app:my-app:1. And let us introduce another artifact, com.mycompany.app:my-module:1.

1. <project>
2. <modelVersion>4.0.0</modelVersion>
3. <groupId>com.mycompany.app</groupId>
4. <artifactId>my-module</artifactId>
5. <version>1</version>
6. </project>

And let us specify their directory structure as the following:

1. .
2. |--my-module
3. |`-- pom.xml
4. `-- pom.xml

**Note:** my-module/pom.xml is the POM of com.mycompany.app:my-module:1 while pom.xml is the POM of com.mycompany.app:my-app:1

Now, if we were to turn com.mycompany.app:my-app:1 into a parent artifact of com.mycompany.app:my-module:1,we will have to modify com.mycompany.app:my-module:1's POM to the following configuration:

**com.mycompany.app:my-module:1's POM**

1. <project>
2. <parent>
3. <groupId>com.mycompany.app</groupId>
4. <artifactId>my-app</artifactId>
5. <version>1</version>
6. </parent>
7. <modelVersion>4.0.0</modelVersion>
8. <groupId>com.mycompany.app</groupId>
9. <artifactId>my-module</artifactId>
10. <version>1</version>
11. </project>

Notice that we now have an added section, the parent section. This section allows us to specify which artifact is the parent of our POM. And we do so by specifying the fully qualified artifact name of the parent POM. With this setup, our module can now inherit some of the properties of our parent POM.

Alternatively, if we want the groupId and / or the version of your modules to be the same as their parents, you can remove the groupId and / or the version identity of your module in its POM.

1. <project>
2. <parent>
3. <groupId>com.mycompany.app</groupId>
4. <artifactId>my-app</artifactId>
5. <version>1</version>
6. </parent>
7. <modelVersion>4.0.0</modelVersion>
8. <artifactId>my-module</artifactId>
9. </project>

This allows the module to inherit the groupId and / or the version of its parent POM.

Project Aggregation?

References: <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

<https://maven.apache.org/ref/3.5.4/maven-model/maven.html> --All POM child tag and their details

<https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html> --mvn folder structure

<https://maven.apache.org/guides/getting-started/>

## Maven Properties

You need to change all of those to be 3.13, then it'll work.Accessible using **${X}** where X being property

I'd suggest something like:

<properties>

<poi.version>3.13</poi.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>${poi.version}</version>

</dependency>

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-scratchpad</artifactId>

<version>${poi.version}</version>

</dependency>

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-ooxml</artifactId>

<version>${poi.version}</version>

</dependency>

</dependencies>

<!-- etc as needed -->

How to fetch environment variable??

How to fetch java system property??

<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference>

## ****Build****

The build section is also a very important section of the Maven POM. It provides information about the default Maven goal, the directory for the compiled project, and the final name of the application. The default build section looks like this:

<**build**>

<**defaultGoal**>install</**defaultGoal**>

<**directory**>${basedir}/target</**directory**>

<**finalName**>${artifactId}-${version}</**finalName**>

<**filters**> <**filter**>filters/filter1.properties</**filter**>

</**filters**>

//...

</**build**>

## Dependency Exclusions

Exclusions are set on a specific dependency in your POM, and are targeted at a specific groupId and artifactId. When you build your project, that artifact will not be added to your project's classpath by way of the dependency in which the exclusion was declared.

## Dependency Scope

The <scope> element can take 6 values: compile, provided, runtime, test, system and import.

**Test:** scope allows to use dependencies only for the test phase.

**Compile**:This is the default scope when no other scope is provided.

# Maven Phases and its Goal(Life Cycle)

## Life Cycle vs Phase

A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed.

## Plugin vs Goal

Goal is like an action in Plugin. So if plugin is a class, goal is a method.

mvn clean:clean

This means "call the clean goal, in the clean plugin"

## Relation between Phase & Goal

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases. A plugin goal represents a specific task (finer than a build phase).  It may be bound to zero or more build phases.

Example-

The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The **clean** and **package** arguments are build phases while the **dependency:copy-dependencies** is a goal.

mvn clean dependency:copy-dependencies package

If this were to be executed, the clean phase will be executed first (meaning it will run all preceding phases of the clean lifecycle, plus the clean phase itself), and then the dependency:copy-dependencies goal, before finally executing the package phase (and all its preceding build phases of the default lifecycle).

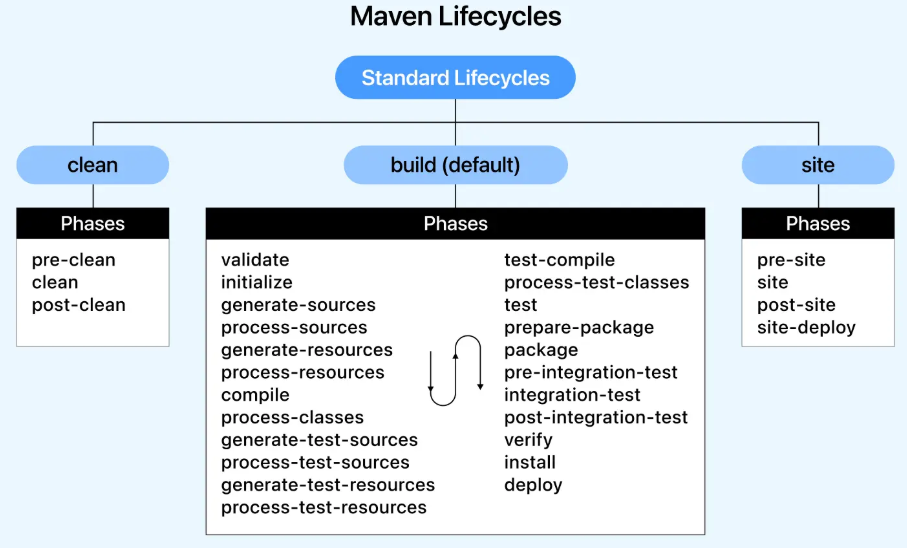
Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals.

## Built-in Lifecycles

A build lifecycle is an organized sequence of phases that exist to give order to a set of goals.

There are three built-in build lifecycles: default, clean and site. The default build life cycle consists of 23 phases as it's the main build lifecycle.On the other hand, clean life cycle consists of 3 phases, while the site lifecycle is made up of 4 phases.

1. **Default** -- default lifecycle handles your project deployment.sometimes called build lifecycle.Consist of **23** phse
2. **Clean** -- Clean lifecycle handles project cleaning.Consist of **3** phases
3. **Site** -- site lifecycle handles the creation of your project's site documentation.Consist of **4** phases.



### Default lifecycle-Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.Out of 23 , below are most popular phases

validate –

validate the project is correct and all necessary information is available.POM.xml check,Configuration check, correct folder structure

compile –

compile the only source code**{src/main/java}** of the project into class file.In Target Foler, **Classes** folder is created and class files will be placed. **Generated-source** and **maven-Status** folder will also be created.

test –

How do I compile my test sources and run my unit tests?

test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.

**In Target Folder**, surefire-reports and test-classes folder are created.

If you simply want to compile both test sources and main source code (but not execute the tests) -- mvn **test-compile**

This will create all folder in target folder e,g source and target classes

package –

How do I create a JAR and install it in my local repository?

take the compiled code and package it in its distributable format, such as a JAR,WAR and EAR.

In Target Folder, maven-archiver folder is created and newly created **jar** file is placed in target folder.

verify

run any checks on results of integration tests to ensure quality criteria are met

install

install the package into the local repository(.m2 folder), for use as a dependency in other projects locally.Example if prject is dependent on other project which is on server or other machine. It pulls all related jar and put in local directory to run project smoothly.

* 1. It downloads all dependency jar into .m2 folder
  2. Other project jar if any into .m2 folder
  3. Put cuurent project jar into .m2 folder
  4. Convert POM.xml into filename.pom and put into .m2 folder

Installing E:\Mars\_workspace\MavenProject\target\MavenProject-0.0.1-SNAPSHOT.jar to C:\Users\Lenovo\.m2\repository\mavenTest\MavenProject\0.0.1-SNAPSHOT\MavenProject-0.0.1-SNAPSHOT.jar

Installing E:\Mars\_workspace\MavenProject\pom.xml to C:\Users\Lenovo\.m2\repository\mavenTest\MavenProject\0.0.1-SNAPSHOT\MavenProject-0.0.1-SNAPSHOT.pom

deploy –

done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

deploy (default-deploy) on project MavenProject: Deployment failed: repository element was not specified in the POM inside distributionManagement element or in -DaltDeploymentRepository=id::layout::url parameter -> [Help 1]

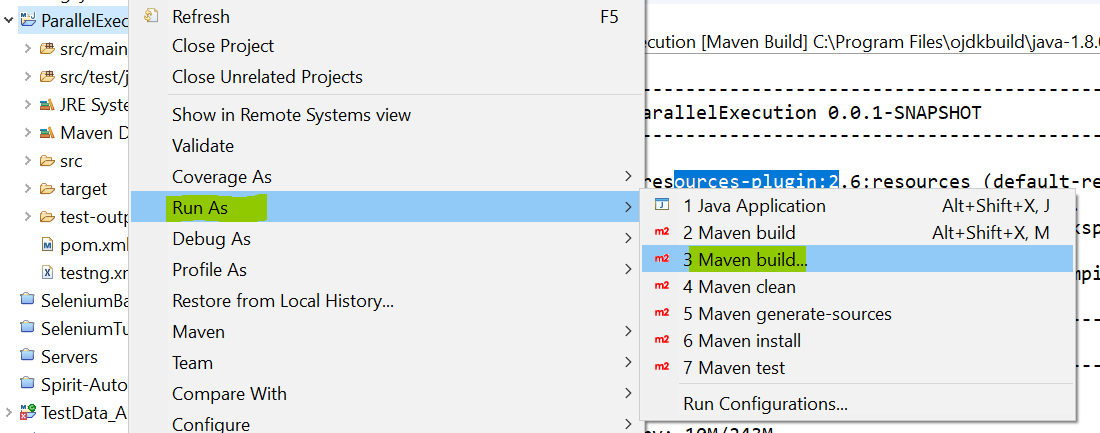
<https://www.youtube.com/watch?v=IYRYbPR5Gek>

|  |  |
| --- | --- |
| Default Lifecycle(23 phases) | |
| **Phase** | **Description** |
| **Validate** | validate the project is correct and all necessary information is available. |
| initialize | initialize build state, e.g. set properties or create directories. |
| generate-sources | generate any source code for inclusion in compilation. |
| process-sources | process the source code, for example to filter any values. |
| generate-resources | generate resources for inclusion in the package. |
| process-resources | copy and process the resources into the destination directory, ready for packaging. |
| **Compile** | compile the source code of the project. |
| process-classes | post-process the generated files from compilation, for example to do bytecode enhancement on Java classes. |
| generate-test-sources | generate any test source code for inclusion in compilation. |
| process-test-sources | process the test source code, for example to filter any values. |
| generate-test-resources | create resources for testing. |
| process-test-resources | copy and process the resources into the test destination directory. |
| **test-compile** | compile the test source code into the test destination directory |
| process-test-classes | post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. |
| **Test** | run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed. |
| prepare-package | perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. |
| **Package** | take the compiled code and package it in its distributable format, such as a JAR,WAR |
| pre-integration-test | perform actions required before integration tests are executed. This may involve things such as setting up the required environment. |
| integration-test | process and deploy the package if necessary into an environment where integration tests can be run. |
| post-integration-test | perform actions required after integration tests have been executed. This may including cleaning up the environment. |
| **Verify** | run any checks to **verify the package** is valid and meets quality criteria. |
| **Install** | install the package(jar,war) into the local repository, for use as a dependency in other projects locally. |
| **Deploy** | copies the final package to the remote repository for sharing with other developers and projects. |

#### Run Maven command from Eclipse

Option1: Already saved previous goal will be executed.

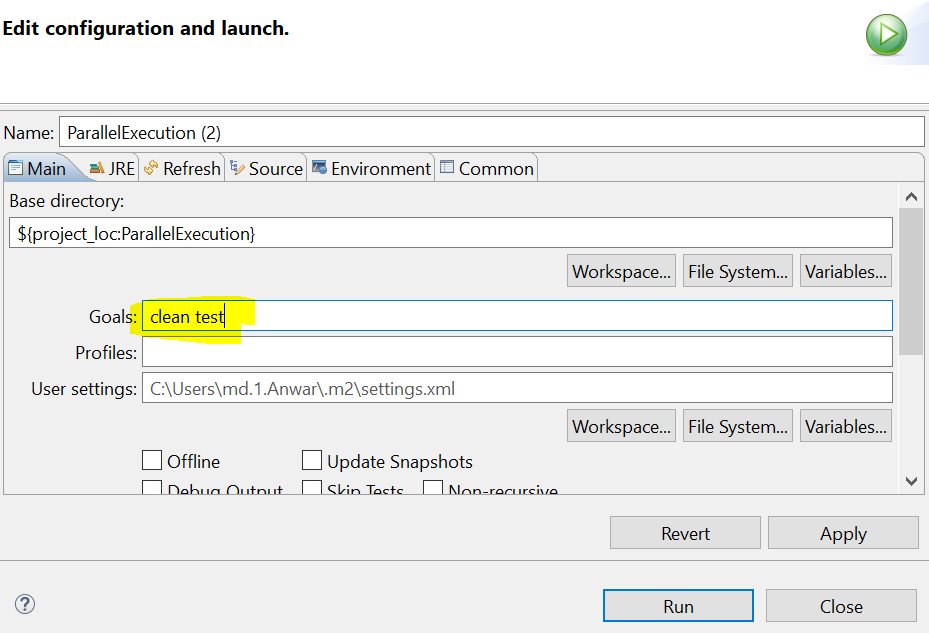
Option2: if you want to run maven command manually, always choose second option “**Maven build**” as below and enter your command. First option “**Maven Build**” will get executed directly with previous saved goal option.



Option3: mvn clean

Option5: mvn install

Option6: mvn test



### Clean lifecycle-Phases

|  |  |
| --- | --- |
| Clean Lifecycle | |
| **Phase** | **Description** |
| pre-clean | execute processes needed prior to the actual project cleaning |
| **Clean** | remove all files generated by the previous build |
| post-clean | execute processes needed to finalize the project cleaning |

### Site lifecycle-Phases

|  |  |
| --- | --- |
| Site Lifecycle | |
| **Phase** | **Description** |
| pre-site | execute processes needed prior to the actual project site generation |
| **Site** | generate the project's site documentation |
| post-site | execute processes needed to finalize the site generation, and to prepare for site deployment |
| site-deploy | deploy the generated site documentation to the specified web server |

### Default lifecycle Binding --Phase vs plugin:goal

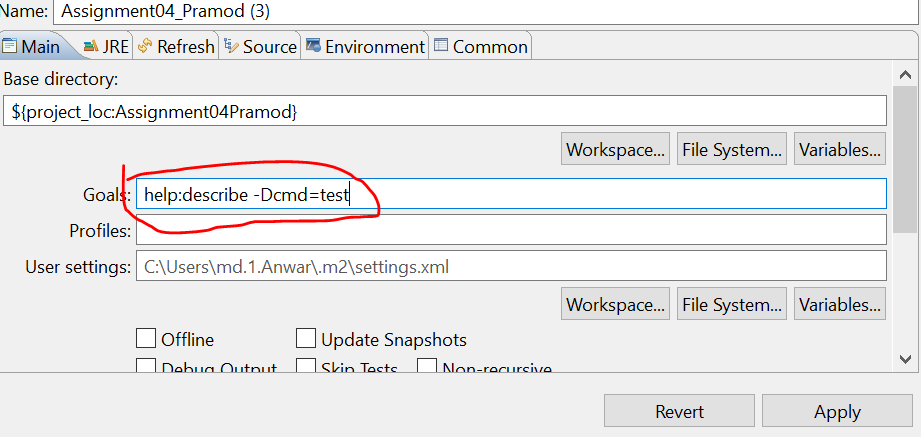
The first, and most common way, is to set the packaging for your project via the equally named **POM element** <packaging>. Some of the valid packaging values are jar, war, ear and pom. If no packaging value has been specified, it will default to jar.

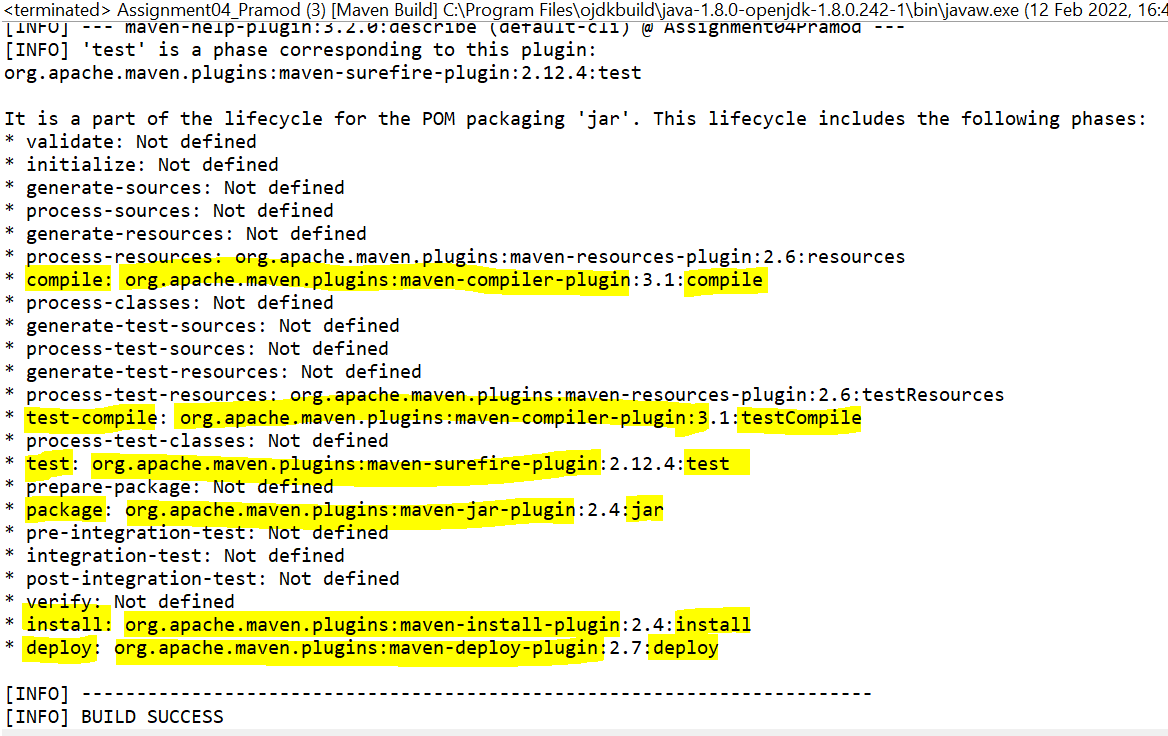
Each packaging contains a list of goals to bind to a particular phase. For example, the jar packaging will bind the following goals to build phases of the default lifecycle. you may also need to include a particular plugin in the <build> section of your POM

|  |  |
| --- | --- |
| **Phase** | **plugin:goal** |
| process-resources | resources:resources |
| **Compile** | **compiler**:**compile** |
| process-test-resources | resources:testResources |
| test-compile | **compiler**:**testCompile** |
| **Test** | **surefire**:**test** |
| **Package** | **jar**:**jar** |
| **Install** | **install**:**install** |
| **Deploy** | **deploy**:**deploy** |

We can list all goals bound to a specific phase and their plugins using the command:

mvn **help:describe** -Dcmd=**PHASENAME**





### Clean Lifecycle Binding

|  |  |
| --- | --- |
| Clean Lifecycle Bindings | |
| **Phase** | **plugin:goal** |
| Clean | clean:clean |

### Site Lifecycle Binding

|  |  |
| --- | --- |
| Site Lifecycle Bindings | |
| **Phase** | **plugin:goal** |
| Site | site:site |
| site-deploy | site:deploy |

### Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin.

## Maven Phase

These lifecycle phases are executed sequentially to complete the default lifecycle.

Example **mvn <PHASE>**

**mvn install** --> **validate** **>> compile** **>> test-compile >> test** **>> package** **>> verify** **>> install**

**mvn clean deploy**

For project with one or more subprojects, Maven traverses into every subproject and executes clean phase, then executes deploy phase in the default build lifecycle – that will execute all phases before the deploy phase as well, which is the entire default lifecycle

## Maven Goal

Each phase is a sequence of goals, and each goal is responsible for a specific task.

When we run a phase – all goals bound to this phase are executed in order.

Here are some of the phases and default goals bound to them:

* compiler:compile – the compile goal from the compiler plugin is bound to the compile phase
* compiler:testCompile is bound to the test-compile phase
* surefire:test is bound to test phase
* install:install is bound to install phase
* jar:jar and war:war is bound to package phase

We can list all goals bound to a specific phase and their plugins using the command:

mvn help:describe -Dcmd=<PHASENAME>

For example, to list all goals bound to the compile phase, we can run:

mvn help:describe -Dcmd=compile

### Maven Plugin

Each plugins is a sequence of goals.

A Maven plugin is a group of goals. However, these goals aren't necessarily all bound to the same phase.

We can use the following command to list all goals in a specific plugin:

mvn <PLUGIN>:help --here help is one of goal for particular plugin

mvn compiler:help --here compiler plugin has help goal

mvn surefire:help --here surefire plugin has help goal

**To run a specific goal, without executing its entire phase (and the preceding phases)** we can use the command:

**mvn <PLUGIN>:<GOAL>**

like mvn surefire:test will execute all test classes

only compile code mvn compiler:compile

**Note**

that if we tried to build a Maven project without specifying a phase or a goal, that will cause the error:

**mvn <Phase>** Or **mvn <plugin>:<goal>** or **mvn <Clean Phasee> <build Phase>**

e.g mvn clean dependency:copy-dependencies package

phase-------plugin-----------------phase

Here the clean phase will be executed first, followed by the dependency:copy-dependencies goal, and finally package phase will be executed.

# Maven Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

As you will see in the later sections, plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough information - you must also specify the goals you want to run as part of your build.

The goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. Note that you can use the <executions> element to gain more control over the order of particular goals.

## Types of Plugins

1. **Build Plugins**: These plugins will execute during build phase.under **<build>** tag in POM.xml
2. **Report Plugins**: These plugins are executed during(Report or javadocs generation.Under **<reporting>** tag in POM

## Build Plugins

### Maven surefire plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version>

</plugin>

</plugins>

</build>

<!-- NOTE: We don't need a groupId specification because the group is

org.apache.maven.plugins ...which is assumed by default.

-->

### Maven clean plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-clean-plugin</artifactId>

<version>3.2.0</version>

</plugin>

</plugins>

</build>

### Maven resources plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<version>3.3.1</version>

</plugin>

</plugins>

</build>

### Maven jar plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.4.1</version>

</plugin>

</plugins>

</build>

### Maven compiler plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.13.0</version>

</plugin>

</plugins>

</build>

### Maven install plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-install-plugin</artifactId>

<version>3.1.2</version>

</plugin>

</plugins>

</build>

### Maven deploy plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-deploy-plugin</artifactId>

<version>3.1.2</version>

</plugin>

</plugins>

</build>

### Maven site plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-site-plugin</artifactId>

<version>3.12.1</version>

</plugin>

</plugins>

</build>

## Maven Resources Plugin

This plugin is responsible for copying of java project resources to the output directory. There are two different kind of resources in maven projects.

1. Main resources

2. Test resources

It has four goals.

*resources:help*

*resources:resources*

  Copy resources for the main source code to the main output directory.

  Always uses the project.build.resources element to specify the resources to copy.

*resources:testResources*

  Copy resources for the test source code to the test output directory.

  Always uses the project.build.testResources element to specify the resources to copy.

*resources:copy-resources*

  Copy resources(image,file) to target output (target>>class for (main resource) OR target>>test-classed

### Specifying Main Resources Directory

Usually, all our maven project resources are defined under src/main/resources directory. But what if we choose to have some of our resources defined elsewhere?For such a case, we’ll need to specify the path to such resources in our POM.

#### Additional directory Other than src/main/resources

Below is for main resources and hence dir1 and dir2 will be copied to target/classes folder.Moreover java file from src/main/java will automatically be copied to target/classes folder.**This is only for resources file.**

<build>

<resources>

<resource>

<directory>src/otherResources/Dir1</directory>

</resource>

<resource>

<directory>src/otherResources/Dir2</directory>

</resource>

<resource> **<!—you have to mention this as well -->**

<directory>src/main/resources</directory>

</resource>

...

</resources>

...

</build>

Similarly, we can define the additional testResources. Dir1 resources will be copied to target/test-classes folder

<build>

<testResources>

<testResource>

<directory>src/otherTestResources/Dir1</directory>

</testResource>

< testResource> **<!—you have to mention this as well -->**

<directory>src/test/resources</directory>

</ testResource>

<testResources>

...

</build>

<https://www.baeldung.com/maven-resources-plugin>

one of the core plugins of the Maven build tool. **The**maven-resources plugin **copies files from input resource directories to an output directory.**

<**plugin**>

<**artifactId**>maven-resources-plugin</**artifactId**>

<**version**>3.0.2</**version**>

<**configuration**>

...

</**configuration**>

</**plugin**>

Maven filter?? <https://www.codetab.org/tutorial/apache-maven/plugins/maven-resources-plugin/>

## Filters

The Maven Resources Plugin also has features to apply filters to the resource files. Filters replaces the variables denoted by **${…}** in resource files with the actual value.

**Step1**: to place the filter files, create a new directory src/test/kk and to it, add a properties file abc.properties.

Abc.properties file content as below

db.url=jdbc:hsqldb:mem:mydb

**Step2**: Create test-config.properties inside src/test/config with content as below

Key1=${db.url}

**Step3:**

Add below in pom.xml

**<build>**

**<filters>**

**<filter>**src/test/kk/abc.properties**</filter>**

**</filters>**

**<resources>**

**<resource>**

**<directory>**src/test/config**</directory>**

**<filtering>**true**</filtering>**

**</resource>**

**</resources>**

**</build>**

**Step4:** after mvn package

Test-config.properties file will be copied to output folder target/test-classed/test-config.properties with content as below

Key1=jdbc:hsqldb:mem:mydb

## Maven Compiler Plugin

### set the Java version in Maven

**By default, the**compiler**plugin compiles source code compatible with Java 5, and the generated classes also work with Java 5 regardless of the JDK in use.** This plugin compiles your java code from the standard location e.g *src/main/java* and *src/test/java*.

We can customize default behavior of maven compiler.The Maven compiler accepts this command with –target and –source versions. If we want to use the Java 8 language features the –source should be set to 1.8.

Also, for the compiled classes to be compatible with JVM 1.8, the –target value should be 1.8.

Maven compiler version is

The default value for both of them is 1.6 version.

<properties>

<maven.compiler.target>1.8</maven.compiler.target>

<maven.compiler.source>1.8</maven.compiler.source>

</properties>

**Alternatively**, we can configure the compiler plugin directly:

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

For Java 9 and above, we can just use the version number directly

<**configuration**>

<**source**>9</**source**>

<**target**>9</**target**>

</**configuration**>

## Maven Surefire Plugin

The Maven Surefire plugin is the **default plugin** for running unit test.This plugin generates XML reports in the directory target/surefire-reports. You can also customize the behaviour of this plugin be specifying the **configuration** in POM.xml

This plugin has only one goal, test. This goal is bound to the test phase of the default build lifecycle, and the command *mvn test* will execute it.

* The Surefire plugin is compatible with the JUnit and TestNG test frameworks. Surefire's behavior remains consistent regardless of the framework used.
* By default, this plugin generates XML reports in the directory *target/surefire-reports*.
* This plugin has only one goal, *test*. This goal is bound to the *test* phase of the default build lifecycle

<https://maven.apache.org/surefire/maven-surefire-plugin/> 🡪 Usage link

<https://maven.apache.org/surefire/maven-surefire-plugin/examples/testng.html> useful link surefire

<https://www.softwaretestinghelp.com/integration-of-maven-with-testng-using-surefire-plugin/> --Latest content

### Use of Surefile plugin

**mvn test**

above command will use surefire plugin to execute only test class present in **src/test/java** with xml-1 in above**.**Above command will only cosider java file which ends with “Test” keyword in file name.e.g myTest.java

By default, surefire automatically includes all test classes whose name starts with Test, or ends with Test, Tests or TestCase.

Case1: There is no configuration in surefire plugin

### Running testing xml Using Suite XML Files

If **Configurations** tag is added into surefire plugin then mvn test will execute **testng.xml** file instead of test class present in src/test/java folder.

<build>

<pluginManagement>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.20</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</pluginManagement>

</build>

### Skipping unit test

To skip running the tests for a particular project, set the skipTests property to true.

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

<configuration>

<skipTests>true</skipTests>

</configuration>

</plugin>

### Showing Only Failed Tests

By default, the Surefire Report Plugin shows the full test result (i.e., successes as well as failures) in the generated HTML. To be able to show the failures only, the property showSuccess should be set to false.

<configuration>

<showSuccess>false</showSuccess>

</configuration>

#### Running a Single Test

mvn -Dtest=Mytestclassname test

mvn -Dtest=Mytestclassname,test2 test -- executing multiple tests

<https://maven.apache.org/surefire/maven-surefire-plugin/examples/single-test.html>

#### Running testing.xml file with mvn parametrization

**Step1**: You need to create variable name e.g ${filename} in place of actual file name in pom.xml file as below

<build>

<pluginManagement>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.20</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>${fileName}</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</pluginManagement>

</build>

**Step2**: run mvn command as below

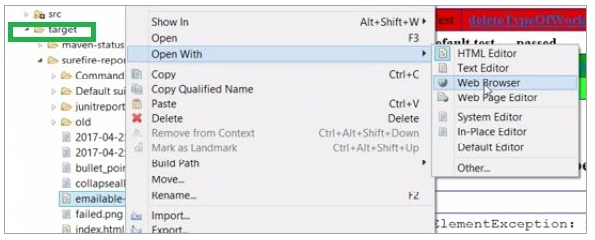
Mvn clean test –DfileName=testng1.xml

#### Surefire report

Refresh the entire project and the test suite Report can be seen in the target folder of the Project Explorer window.

Path: *target/surefire-report/emailable-report.html*

*target/surefire-report/index.html*



mvn test ----will execute unit test and generate surefire report. It can also be done executing below command.

mvn surefire-report:report

**Skipping the Tests**

Sometimes you might want Maven to generate a unit test report without running all the unit tests again. You might just want to use the results from the last run.

mvn surefire-report:report-only

## Maven-failsafe plugin

The failsafe plugin is used for integration tests of a project. It has two goals:

* integration-test – run integration tests; this goal is bound to the integration-test phase by default
* verify – verify that the integration tests passed; this goal is bound to the verify phase by default

<plugin>

<artifactId>maven-failsafe-plugin</artifactId>

<version>2.21.0</version>

<executions>

<execution>

<goals>

<goal>integration-test</goal>

<goal>verify</goal>

</goals>

<configuration>

...

</configuration>

</execution>

</executions>

</plugin>

This plugin runs methods in test classes just like the surefire plugin. We can configure both plugins in similar ways. However, there're some crucial differences between them.

<https://www.baeldung.com/maven-surefire-vs-failsafe>

## Maven-jar-plugin

This plugin creates a java archive(JAR) file from the compiled project classes and resources. You almost always use this plugin to create JAR file.

## Maven-war-plugin

This plugin create a web archive(WAR) file from the compiled project classes,resources and web.xml. You use this plugin while building java web application as they are deployed into servlet containers like tomcat.

**maven-clean-plugin**

The Clean Plugin is used when you want to remove files generated at build-time in a project's directory.

*mvn clean:clean*

## Maven common command

|  |  |
| --- | --- |
| **Maven Command** | **Description** |
| mvn –version | Prints out the version of Maven you are running. |
| mvn clean | Clears the target directory into which Maven normally builds your project. |
| mvn package | Builds the project and packages the resulting JAR file into the target directory. |
| mvn clean package | Clears the target directory and Builds the project and packages the resulting JAR file into the target directory. |
| mvn install | Builds the project described by your Maven POM file and installs the resulting artifact (JAR) into your local Maven repository |
| mvn clean install | Clears the target directory and builds the project described by your Maven POM file and installs the resulting artifact (JAR) into your local Maven repository |
| mvn dependency:copy-dependencies | Copies dependencies from remote Maven repositories to your local Maven repository. |
| mvn clean dependency:copy-dependencies | Cleans project and copies dependencies from remote Maven repositories to your local Maven repository. |
| mvn dependency:tree | Prints out the dependency tree for your project - based on the dependencies configured in the pom.xml file. |
| mvn dependency:build-classpath | Prints out the classpath needed to run your project (application) based on the dependencies configured in the pom.xml file. |

# How to Create an Executable JAR with Maven

we'll focus on packaging a Maven project into an executable Jar file.

When creating a jar file, we usually want to run it easily, without using the IDE. To that end, we'll discuss the configuration and pros/cons of using each of these approaches for creating the executable.

## ****Apache Maven Assembly Plugin****

The Apache Maven Assembly Plugin allows users to aggregate the project output along with its dependencies, modules, site documentation, and other files into a single, runnable package.

The main goal in the assembly plugin is the [**single**](https://maven.apache.org/plugins/maven-assembly-plugin/single-mojo.html) goal, which is used to create all assemblies (all other goals are deprecated and will be removed in a future release).

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<version>2.6</version>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

<configuration>

<archive>

<manifest>

<mainClass>com.testRunner.testRunner</mainClass>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

</plugins>

## </build>

## User defined properties in Maven

<properties>

<spring.version>3.1.2.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

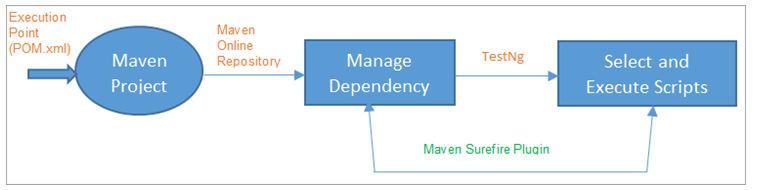
</dependency>

</dependencies>

If you want to upgrade Spring to 3.1.5, just change the “spring.version” to 3.1.5, and all the dependencies will be affected.

# Maven With TestNG Integration?

* Whenever we are executing test scripts or suites using the Maven project, our dependencies are managed in the POM.xml file. However, a specific test suite cannot be selected to execute from a list of available suites.
* In TestNG, we cannot manage our dependencies but we can select and execute particular test scripts or suites.
* Given that Maven and TestNG have different capabilities, we are integrating both using the Maven Surefire plugin



* Here, execution starts from the Maven project using POM.xml. Initially, it connects to the Maven Online Repository and downloads the latest version of the dependencies.
* As TestNG has the capability to select and execute particular test scripts or suites, we are integrating this with Maven using the Maven Surefire plugin.

## Maven Profile

A Build profile is a set of configuration values, which can be used to set or override default values of Maven build. Using a build profile, you can customize build for different environments such as Production v/s Development environments.

There are 3 build profile types.

<https://www.youtube.com/watch?v=594jY1QDDkc>

### Per Project

defined in the pom.xml file

### Per User

defined in the Maven settings xml file (%USER\_HOME%/.m2/settings.xml)

### Global

Defined in Maven global settings xml file (%M2\_HOME%/conf/settings.xml)

<profiles>

<profile>

<id>regression</id>

<build>

<pluginManagement>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.20 </version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</pluginManagement>

</build>

</profile>

</profiles>

## How to execute mvn test with profile

mvn test –P<profileId>

if you use mvn test then it will not use profile configuration and default configuration will be considered.

## How do I compile my application sources?

mvn compile

The first time you execute this (or any other) command, Maven will need to download all the plugins and related dependencies it needs to fulfill the command. From a clean installation of Maven, this can take quite a while (in the output above, it took almost 4 minutes). If you execute the command again, Maven will now have what it needs, so it won't need to download anything new and will be able to execute the command much more quickly.

As you can see from the output, the compiled classes were placed in ${basedir}/target/classes, which is another standard convention employed by Maven.

## How do I compile my test sources and run my unit tests?

mvn test

Some things to notice about the output:

* Maven downloads more dependencies this time. These are the dependencies and plugins necessary for executing the tests (it already has the dependencies it needs for compiling and won't download them again).
* Before compiling and executing the tests Maven compiles the main code (all these classes are up to date because we haven't changed anything since we compiled last).

If you simply want to compile your test sources (but not execute the tests), you can execute the following:

*mvn test-compile*

## How do I create a JAR and install it in my local repository?

*mvn package*

If you take a look at the POM for your project you will notice the packaging element is set to jar. This is how Maven knows to produce a JAR file from the above command (we'll talk more about this later). You can now take a look in the ${basedir}/target directory and you will see the generated JAR file.

Now you'll want to install the artifact you've generated (the JAR file) in your local repository (${user.home}/.m2/repository is the default location). mvn install

*mvn clean*

This will remove the target directory with all the build data before starting so that it is fresh.

## How do I add resources to my JAR?

Another common use case that can be satisfied which requires no changes to the POM that we have above is packaging resources in the JAR file. For this common task, Maven again relies on the [Standard Directory Layout](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html), which means by using standard Maven conventions you can package resources within JARs simply by placing those resources in a standard directory structure.

${basedir}/src/main/resources

${basedir}/src/test/resources

## Difference between Eclipse Build Project and Maven Compile command

The short answer is no, maven build and eclipse build are not the same. Basically, eclipse has its own way of building things, which has little to do with maven.

To get an as close as possible approximation so that what Eclipse does, resembles most to what maven does on the commandline you should install m2e (maven eclipse tooling).

M2E tries to make your Eclipse IDE's behavior 'emulate' as closely as possible to maven's commandline behavior. It does this by configuring your eclipse project. For example, setting source folders, classpath etc. based on the maven poms. This works pretty well if your poms don't do 'fancy' things (i.e. use some not so common maven plugins).

# Selenium-Maven

<https://www.javatpoint.com/selenium-maven#:~:text=Maven%20is%20the%20latest%20build%20testing%20tool.&text=Maven%20is%20a%20project%20build,files%20into%20the%20same%20framework>.

* Create Maven Project
* To create a package in the MavenProject, we will right-click on the **src/test/java → New → Package**
* After that, we will provide the name of the package as "**testpackage**", and click on the **Finish** button
* Right-click on the newly created package, and select **New** then select **Class MavenTest1.java**
* After adding the dependencies, our pom.xml file look like this:

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>3.7.0</version>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.8.8</version>

<scope>test</scope>

</dependency>

## Execute the code

We can run above code in multiple ways:

1. **Run through TestNG**
2. **Run through Maven**
3. **Run through Command Line**

### Run through TestNG

If we run the above code with the TestNG, we will follow the below process:

Right-click on the **java** file, and select **Run As → 1 TestNG Test**

### Run Through Maven

* First, we need to convert the **MavenTest1.java** file into the **TestNG** File, for this follow the below process:
* Right-click on the **java** file and select **TestNG** and then select **Convert to TestNG**options in the given list.
* Once we select the **Convert to TestNG** options, it will open window to create testing.xml, and after that, we will click on the **Finish**

After that, we will run the **testng.xml** file, so for this, we need to add the Maven Plugins in the **pom.xml** files.

So, we will add the three different plugins, which are as follows:

* Maven compiler plugin
* Maven surefire plugin
* Maven source plugin

The **Maven compiler plugin** is used to compile the source code of a Maven project.

The **Maven surefire plugin** is used when we have to run the unit tests of the application.

The **Maven source plugin** is used to build the jars files that were having the **.java** source files.

After adding all the plugins, our pom.xml look like this:

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.19.1</version>

<configuration>

<suiteXmlFiles> <suiteXmlFile>C:\kk\JavaWorkspace\Selenium\testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-source-plugin</artifactId>

<executions>

<execution>

<id>attach-sources</id>

<goals>

<goal>jar</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

For executing the above code, right-click on the **MavenProject → Run As → Maven Test**

### Run through Command Line

If we are using some remote machine with the help of Maven commands, then we need to go to the command prompt. For this, we will go to that particular directory of the Maven project. Open command prompt and

cd eclipse-workspace\MavenProject

|  |  |
| --- | --- |
| **Maven commands** | **Description** |
| **mvn clean install** | This command is used to generate, compile, and execute the jars files. |
| **mvn test** | We will use this command when we have to execute the tests against the compiled source code with the help of an appropriate unit testing framework. |
| **mvn compile** | It is used to compile the source code of the Maven project. |
| **mvn package** | It will pack the executed code in a different for |

### Tomcat maven plugin

configure [tomcat](https://howtodoinjava.com/tomcat/tomcats-architecture-and-server-xml-configuration-tutorial/) plugin to pom.xml and use it deploy the web application without any tomcat installation in machine. It is very useful when you want to test your application in developer’s machines where actual tomcat installation is not available due to any reason.

#### Run application with tomcat plugin

To run the application with tomcat maven plugin, use maven goal as –

mvn tomcat7:run

<build>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<port>9000</port> //Configure port number

<path>/spring5-webmvc-demo</path> //Configure application root URL

</configuration>

</plugin>

</plugins>

</build>

# Maven Interview Question

## List out what are the aspects does Maven Manages?

Maven handles following activities of a developer

* Build
* Documentation
* Reporting
* Dependencies
* SCMs
* Releases
* Distribution
* Mailing list

## ****Explain what is Maven artifact?****

Usually an artifact is a JAR file which gets arrayed to a Maven repository. One or more artifacts a maven build produces such as compiled JAR and a sources JAR.

Each artifact includes a group ID, an artifact ID and a version string.

## List out the dependency scope in Maven?

The various dependency scope used in Maven are:

* **Compile**: It is the default scope, and it indicates what dependency is available in the classpath of the project
* **Provided**: It indicates that the dependency is provided by JDK or web server or container at runtime
* **Runtime**: This tells that the dependency is not needed for compilation but is required during execution
* **Test**: It says dependency is available only for the test compilation and execution phases
* **System**: It indicates you have to provide the system path
* **Import**: This indicates that the identified or specified POM should be replaced with the dependencies in that POM’s section

<dependencies>

<dependency>

<groupId>sample.ProjectA</groupId>

<artifactId>Project-A</artifactId>

<version>1.0</version>

<scope>compile</scope>

</dependency>

</dependencies>

## Optional Dependencies

Optional dependencies are used when it's not possible (for whatever reason) to split a project into sub-modules. The idea is that some of the dependencies are only used for certain features in the project and will not be needed if that feature isn't used. Ideally, such a feature would be split into a sub-module that depends on the core functionality project. This new subproject would have only non-optional dependencies, since you'd need them all if you decided to use the subproject's functionality.

<dependencies>

<!-- declare the dependency to be set as optional -->

<dependency>

<groupId>sample.ProjectA</groupId>

<artifactId>Project-A</artifactId>

<version>1.0</version>

<scope>compile</scope>

<optional>true</optional> <!-- value will be true or false only -->

</dependency>

</dependencies>

## Explain how you can exclude dependency?

Since Maven resolves dependencies transitively, it is possible for unwanted dependencies to be included in your project's classpath. For example, a certain older jar may have security issues or be incompatible with the Java version you're using. To address this, Maven allows you to exclude specific dependencies. Exclusions are set on a specific dependency in your POM, and are targeted at a specific groupId and artifactId. When you build your project, that artifact will not be added to your project's classpath by way of the dependency in which the exclusion was declared.

<dependencies>

<dependency>

<groupId>sample.ProjectA</groupId>

<artifactId>Project-A</artifactId>

<version>1.0</version>

<scope>compile</scope>

<exclusions>

<exclusion> <!-- declare the exclusion here -->

<groupId>sample.ProjectB</groupId>

<artifactId>Project-B</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

## In Maven what are the two setting files called and what are their location?

In Maven, the setting files are called settings.xml, and the two setting files are located at

Maven installation directory: $Maven\_Home/conf/settings.xml

User’s home directory: ${ user.home }/ .m2 / settings.xml

## List out what are the build phases in Maven?

Build phases in Maven are

1. validate
2. compile
3. test
4. package
5. install
6. deploy

## Where do you find the class files when you compile a Maven project?

You will find the class files ${basedir}/target/classes/

## List out what are the Maven’s order of inheritance?

The maven’s order of inheritance is

Parent Pom

Project Pom

Settings

CLI parameters

## For POM what are the minimum required elements?

The minimum required elements for POM are project root, modelVersion, groupID, artifactID and version.

## What types of Maven repository?

Maven repositories are of three types –

**Local**: Maven local repository is a folder location that is present on your machine. It is created when you run any maven command for the first time. Maven local repository is a location where you can find your project’s all dependencies (library jars, plugin jars etc).

Maven local repository by default get created by Maven in ${user.home}/.m2/repository directory. To override the default location, mention another path in Maven settings.xml file available at %MAVEN\_HOME%\conf directory.

<settings xmlns = "http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/SETTINGS/1.0.0

http://maven.apache.org/xsd/settings-1.0.0.xsd">

<localRepository>C:/MyLocalRepository</localRepository>

</settings>

When you run Maven command, Maven will download dependencies to your custom path.

**Central**: It is a repository provided by the Maven community. It contains a huge collection of commonly used libraries. When Maven does not find any dependency in local repository, it starts searching in central repository using the following URL: <http://repo1.maven.org/maven2/>.

Key concepts of Central repository are as follows −

* This repository is managed by Maven community.
* It is not required to be configured.
* It requires internet access to be searched.

**Remote**: Sometimes, Maven is not able to find a mentioned dependency in the central repository as well then it stops the build process and an output error message is displayed on the console. To avoid such a situation, Maven provides the idea of Remote Repository which is nothing but the developer’s own custom repository containing required libraries or other project jars.

<dependencies>

<dependency>

<groupId>com.companyname.common-lib</groupId>

<artifactId>common-lib</artifactId>

<version>1.0.0</version>

</dependency>

<dependencies>

<repositories>

<repository>

<id>companyname.lib1</id>

<url>http://download.companyname.org/maven2/lib1</url>

</repository>

<repository>

<id>companyname.lib2</id>

<url>http://download.companyname.org/maven2/lib2</url>

</repository>

</repositories>

## What are the uses of Maven Plugins?

Maven Plugins are used to −

create jar file.

create war file.

compile code files.

unit testing of code.

create project documentation.

create project reports.

## What is Archetype?

An archetype is a Maven plugin whose task is to create a project structure as per its template.

# What’s the difference between Unit Testing, TDD and BDD?

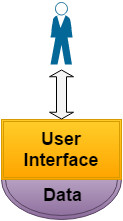
Test Driven development

Behavior-Driven Development

Unit Testing

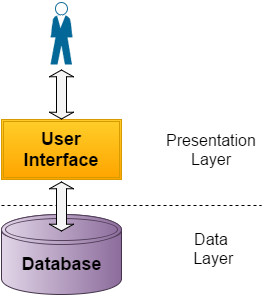
# One,Two,Three and N-Tier architecture

**One-tier architecture,** where all software components are kept in one place.



Such software can be run on a single computer, and is the simplest and most direct option. All components are included in one application. If data needs to be changed, this can only be done using the computer on which the software is installed. One-tier architecture is not sufficient for web applications or cloud solutions.

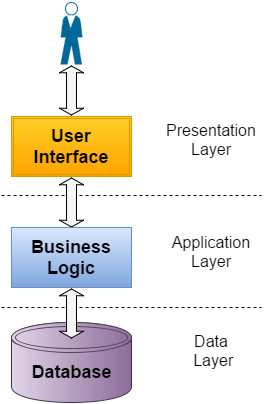
**Two-tier architecture** includes two layers: a presentation layer and a data layer. The former runs on the client side, while the latter stores data on a server.



This approach improves scalability and divides the user interface from the data layer. Most of the processing occurs either on the client side or on a server. In the context of an increasing number of users, the performance of this software architecture may be poor.

**Three-tier architecture** is the most popular option. It provides the following independent layers:

* A presentation layer, which displays information and communicates with other layers.
* An application layer, which contains business logic and controls application functionality performing processing.
* A data layer, which stores and retrieves information. Data in this tier is kept independent of application servers or business logic.



This division allows each layer to be developed, tested, executed and reused individually. The application layer can also be multi-tiered itself. In this case, the general architecture is called **N-tier architecture**.

## Typical Solutions for Multi-Tier Architecture

Here, I will describe some technological solutions we often use. As you know, each tier can be developed using particular technologies. This is why I want to show the best solutions for each tier separately.

### Presentation tier

The presentation tier can be presented using desktop, web or mobile applications. With respect to web applications, we prefer the following technologies: JavaScript with Jquery, and AngularJS.

Mobile applications can be created using Cordova, ReactJS, or Native App. [Choosing the best technology](https://www.linkedin.com/pulse/mobile-development-native-application-vs-cordova-vasiliy-soloshchuk) depends on specific project needs. Cordova allows for quick development of a simple application with few pages. However, if a full-featured mobile application is required, Native App with direct access to Native SDK can be the best choice.

### Integration/API tier

You may need APIs to interact with a business layer or third-party objects. In this case, you can use SOAP or REST techniques. Which one you use depends on the third-party object’s specification and on the particular project requirements.

SOAP is language, platform, and transport independent. It works well in distributed enterprise environments. On the other hand, REST is more efficient, faster, and closer to other web technologies in design philosophy.

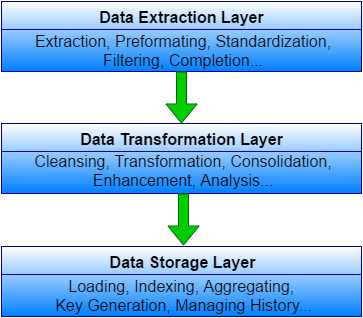
### Application tier

The application tier contains business logic, which can consist of one or more modules. Technologies we recommend for use in this tier are as follows: Java EE 7 and Java SE 8, Spring Framework, and JPA/Hibernate ORM.

Depending on project requirements, the application tier can be separated into several tiers. This step may increase the application scalability and maintainability. Clustering is also possible.

### Data tier

The data tier can consist of additional sub-tiers. For example, a typical ETL (Extract, Transform and Load) process includes extraction, transformation, and storage layers; each of the layers may be executed in a separate tier.



Depending on project requirements, SQL or NoSQL databases can be used. A hybrid SQL+NoSQL approach is also possible. Our recommendations are as follows:

* **SQL** databases: PostgreSQL, MySQL, Oracle. Each of these has its own pros and cons.
* **NoSQL** databases: Choosing a NoSQL database depends on the data model that is most relevant for a particular project. We prefer working with MongoDB, Couchbase (document-oriented), and Neo4j (graph-oriented) databases.

Depending on the project specifics, you may consider using Amazon Web Services. This enables you to save money on infrastructure and maintaining the hardware. In addition, the autoscaling feature increases and decreases capacity according to demands, and in this way reduces costs.

### Business Logic Tier

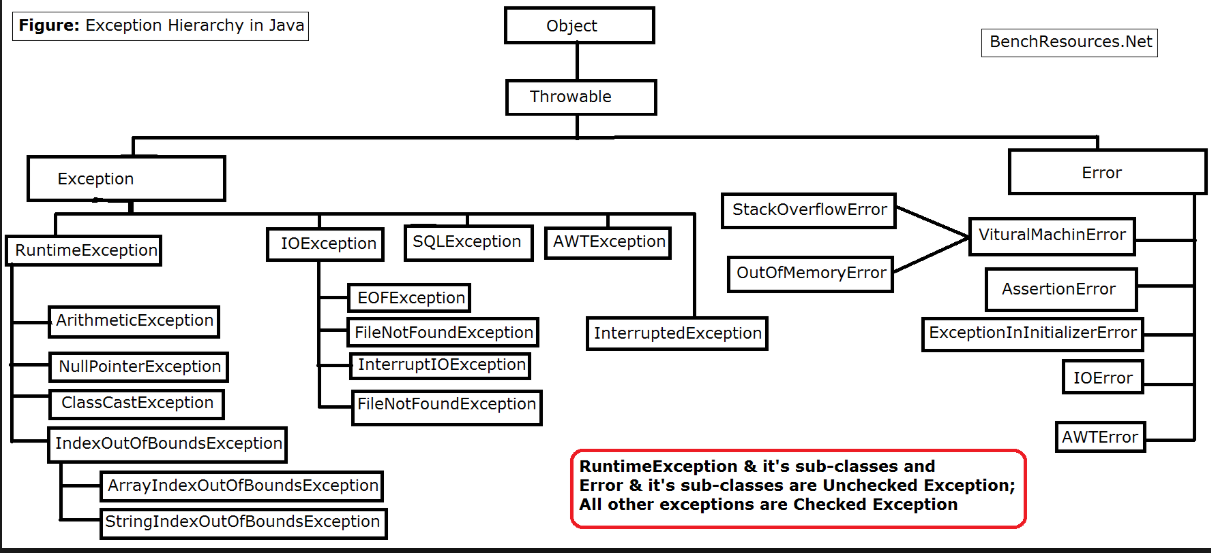
Now I would like to focus on the application layer, which is also called the business logic tier. The best approach with respect to this layer is to use service-oriented architecture (SOA). In this case, the business logic is implemented as a set of services; each service is a separate application.

Since each service is independent, they may each be used for individual goals. The services may be divided into the following groups:

* Algorithms for calculations and predictions;
* Preparing and processing data for the algorithms;
* Processing output and reporting.

The SOA approach enables every specific component in the business logic architecture to be clustered. Clustering provides fault tolerance. Each service can be a cluster of servers, and enabling the autoscaling feature in Amazon Cloud allows the software product to withstand load spikes.

# Exceptions Handling



A **NullPointerException in Java** is a [RuntimeException](https://www.geeksforgeeks.org/java-program-to-handle-runtime-exceptions/). In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when a program attempts to use an object reference that has the **null**value.

There are three category of exception

**Why Exception handling**

when you don’t know the reason of failure, how you would prevent it in future? Never do this !!

1. **Checked exceptions / Compile Time exceptions –**

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a **FileNotFoundException** occurs, and the compiler prompts the programmer to handle the exception.**It can be detected by eclipse**

1. **Unchecked exceptions / RunTime exceptions** –

An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

1. **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

# Exception Method List

public String **getMessage**() : Only exception Message

public String **toString**() : name of the exception class concatenated with the result of getMessage().

public void **printStackTrace**() : result of toString() along with the stack trace to System.err, the error output stream.

String **e.getClass().getSimpleName()** : get only exception name

# Throws/Throw Keywords

## Throws

If exception is not handled for compile Time error then

method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

**Example:** if you don’t want to handle exception for the time being, use Throws key

And later remove Throws keyword and handle exception

Try to understand the difference between **throws** and **throw** keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

**Important points to remember about throws keyword:**

* throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
* throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
* By the help of throws keyword we can provide information to the caller of the method about the exception.

## Throw

Throw keyword in Java is used to explicitly throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions. But this exception i.e, Instance must be of type **Throwable** or a subclass of **Throwable**.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program.

**try** {

**throw new** RemoteException();

} **catch** (RemoteException e) {

System.***out***.println("catching manually exception");//e.printStackTrace();

}

In the above example we have throw an unchecked exception, same way we can throw unchecked and user-defined exception as well.

### Custom exception

In Java, we can create our own exceptions that are derived classes of the **Exception class**. Creating our own Exception is known as custom exception or user-defined exception. Using the custom exception, we can have your own exception and message.

#### Why Custome exception

* To catch and provide specific treatment to a subset of existing Java exceptions.
* Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

class myexception extends Exception {  
 myexception(String str) {  
 super(str);  
 }

**Note**: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

#### How to use custom exception

try {  
 throw new myexception("There is some error");  
} catch (myexception e) {  
 System.*out*.println("Exception details--" + e.getMessage());  
}

o/p-- Exception details--There is some error

**Rethrow Exception**

classThrowExcep

{

    staticvoidfun()

    {

        try

        {

            Throw new NullPointerException("demo");

        }

        catch(NullPointerException e)

        {

            System.out.println("Caught inside fun().");

            Throw e; // rethrowing the exception

        }

    }

    Public static void main(String args[])

    {

        try

        {

            fun();

        }

        catch(NullPointerException e)

        {

            System.out.println("Caught in main.");

        }

    }

}

**Output**

Caught inside fun().

Caught in main.

**How to throw your own exception explicitly**

class MyOwnException extends Exception {

public MyOwnException(String msg){

super(msg);

}

}

--test class method

static void employeeAge(int age) throws MyOwnException{

if(age < 0)

throw new MyOwnException("Age can't be less than zero");

else

System.out.println("Input is valid!!");

}

**How to throw an already defined exception using throw keyword**

class Exception2{

static int sum(int num1, int num2){

if (num1 == 0)

throw new ArithmeticException("First parameter is not valid");

else

System.out.println("Both parameters are correct!!");

return num1+num2;

}

public static void main(String args[]){

int res=sum(0,12);

System.out.println(res);

System.out.println("Continue Next statements");

}

}

Similarly other exceptions, such as NullPointerException, ArrayIndexOutOfBoundsException etc. can be thrown.

**What is the need of having throws keyword when you can handle exception using try-catch?**

Throws keyword is used for handling checked exceptions .Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch.But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

# Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Try .. catch

Try .. catch..catch

Try .. finally

Try..catch..finally

Try..Catch{Try..Catch}

Points to remember for exception handling

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses whenever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

**Best practices while Exception handling**

1. Clean up Resources in a Finally Block or Use a Try-With-Resource Statement
2. Use a Finally Block

-

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

log.error(e);

}

1. **Catch the most specific exception first (child exception first then parent)**

Most IDEs help you with this best practice. They report an unreachable code block when you try to catch the less specific exception first.

The problem is that only the first catch block that matches the exception gets executed. So, if you catch an IllegalArgumentException first, you will never reach the catch block that should handle the more specific NumberFormatException because it’s a subclass of the IllegalArgumentException.

Always catch the most specific exception class first and add the less specific catch blocks to the end of your list.

You can see an example of such a try-catch statement in the following code snippet. The first catch block handles all NumberFormatExceptions and the second one all IllegalArgumentExceptions which are not a NumberFormatException.

public void catchMostSpecificExceptionFirst() {

try {

doSomething("A message");

} catch (NumberFormatException e) {

log.error(e);

} catch (IllegalArgumentException e) {

log.error(e)

}

catch (Exceptione) {

log.error(e)

}

}

1. **Don’t catch Throwable**

Throwable is the superclass of all exceptions and errors. You can use it in a catch clause, but you should never do it!

If you use Throwable in a catch clause, it will not only catch all exceptions; it will also catch all errors. Errors are thrown by the JVM to indicate serious problems that are not intended to be handled by an application.

Throwable is the superclass of all errors and exceptions in Java. Error is the superclass of all errors which are not meant to be caught by applications. Thus, catching Throwable would essentially mean that Errors such as system exceptions (e.g., OutOfMemoryError, StackOverFlowError or InternalError) would also get caught. And, the recommended approach is that application should not try and recover from Errors such as these. Thus, Throwable and Error classes should not be caught. Only Exception and its subclasses should be caught.

Above said, there are reasons why people still go for catching Throwable. The data errors such as encoding issues etc which are not known at programming time can be caught using this technique. However, catching Throwable such as InternelError or OutofMemoryError would not be of any help and should therefore be thrown. Thus, one should avoid writing code consisting of catching Throwable as general practice.

**Avoid printStackTrace method**

1. Difficult to Retrieve Logs for Debugging:The logs written using printStackTrace is written to System.err which is hard to route or filter elsewhere. Instead, using Loggers, it is easy to retrieve logs for debugging purpose.
2. Violation of Coding Best Practices:Generally, as per coding guidelines in production-ready applications, developers need to use Logger methods for logging different level of information. However, when it comes to exception handling, the instances of printStackTrace are commonly found in various places. This is, thus, a violation of coding practice and, thus, should be avoided.

A stack trace should never be visible to end users (for user experience and security purposes)

1. Don’t ignore exceptions

That’s often caused by an ignored exception. The developer was probably pretty sure that it would never be thrown and added a catch block that doesn’t handle or logs it. And when you find this block, you most likely even find one of the famous “This will never happen” comments.

So, please, never ignore an exception. You don’t know how the code will change in the future. Someone might remove the validation that prevented the exceptional event without recognizing that this creates a problem. Or the code that throws the exception gets changed and now throws multiple exceptions of the same class, and the calling code doesn’t prevent all of them.

1. **Wrap the Exception/ User Defined Exception**

Anytime when user feels that he wants to use its own application specific exception for some reasons, he can create a new class extending appropriate super class (mostly its Exception.java) and start using it in appropriate places. These user defined exceptions can be used in two ways:

1. Either directly throw the custom exception when something goes wrong in application

throw new DaoObjectNotFoundException("Couldn't find dao with id " + id);

1. Or wrap the original exception inside custom exception and throw it

catch (NoSuchMethodException e) {

throw new DaoObjectNotFoundException("Couldn't find dao with id " + id, e);

}

// wrong way as below

catch (NoSuchMethodException e) {

throw new MyServiceException("Some information: " + e.getMessage()); //Incorrect way

}

-

Wrapping an exception can provide extra information to the user by adding your own message/ context information, while still preserving the stack trace and message of the original exception. It also allows you to hide the implementation details of your code, which is the most important reason to wrap exceptions.

**Bad Practics**

try {

/\* ... \*/

} catch( Exception e ) {

SomeLogger.info( e.getMessage() ); // The exception is lost. Just that exception message is written; Also, context information is not logged.

}

**Bad Practics**

try {

/\* ... \*/

} catch( Exception e ) {

SomeLogger.info( "some context message" ); // The exception is lost

}

**Bad Practics**

try {

/\* ... \*/

} catch( Exception e ) {

SomeLogger.info( e ); // No context message

}

// **Best Practices**

try {

/\* ... \*/

} catch( Exception e ) {

SomeLogger.info( "some context message", e ); // Context message is there. Also, exception object is present

}

1. Declare the specific checked exceptions that your method can throw

public void foo() throws Exception { //Incorrect way

}

Always avoid doing this as in above code sample. It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

public void foo() throws SpecificException1, SpecificException2 { //Correct way

}

1. **Do not catch the Exception class rather catch specific sub classes**

try {

someMethod();

} catch (Exception e) {

LOGGER.error("method has failed", e);

}

The problem with catching Exception is that if the method you are calling later adds a new checked exception to its method signature, the developer’s intent is that you should handle the specific new exception. If your code just catches Exception (or Throwable), you’ll never know about the change and the fact that your code is now wrong and might break at any point of time in runtime.

1. Either log the exception or throw it but never do the both

catch (NoSuchMethodException e) {

LOGGER.error("Some information", e);

throw e;

}

1. Use finally blocks instead of catch blocks if you are not going to handle exception

try {

someMethod(); //Method 2

} finally {

cleanUp(); //do cleanup here

}

This is also a good practice. If inside your method you are accessing some method 2, and method 2 throw some exception which you do not want to handle in method 1, but still want some cleanup in case exception occur, then do this cleanup in finally block. Do not use catch block.

# User Defined Exception / Custom Exception

User Defined Exception or custom exception is creating your own exception class and throws that exception using ‘throw’ keyword. This can be done by extending the class Exception.

There is no need to override any of the above methods available in the Exception class, in your derived class. But practically, you will require some amount of customizing as per your programming needs.

The keyword “throw” is used to create a new Exception and throw it to the catch block.The keyword “throw” is used to create a new Exception and throw it to the catch block.

* User defined exception class must extend Exception class
* If Throw is used inside Try block then Catch must have throwing exception name
* If throw is not used inside try block and throws (user defined exception name) is used in method signature.

# What is JSON?

* JSON stands for **J**ava**S**cript **O**bject **N**otation
* JSON is a lightweight data-interchange format
* JSON is "self-describing" and easy to understand
* JSON is language independent

## JSON Data - Key and Value

Key: keys must be strings, written with double quotes:

values must be one of the following data types:

* **String**
* **Number**
* **Object** (JSON object)
* **Array of Object**
* **Array**
* **Boolean**
* **Null**

File Name Extension: .json

**Json Array**

{

  "ids" : ["1","2","3"]

}

//or array of object

{

  "ids" : [

        {"id" : 1},

        {"id" : 2},

        {"id" : 3}

   ]

}

# Process Builder

<http://www.xyzws.com/Javafaq/how-to-run-external-programs-by-using-java-processbuilder-class/189>

public class ProcessBuildDemo {

    public static void main(String [] args) throws IOException {  
          
        String[] command = {"CMD", "/C", "dir"};  
        ProcessBuilder probuilder = new ProcessBuilder( command );

        //You can set up your work directory  
        probuilder.directory(new File("c:\\xyzwsdemo"));  
          
        Process process = probuilder.start();  
          
        //Read out dir output  
        InputStream is = process.getInputStream();  
        InputStreamReader isr = new InputStreamReader(is);  
        BufferedReader br = new BufferedReader(isr);  
        String line;  
        System.out.printf("Output of running %s is:\n",  
                Arrays.toString(command));  
        while ((line = br.readLine()) != null) {  
            System.out.println(line);  
        }  
          
        //Wait to get exit value  
        try {  
            int exitValue = process.waitFor();  
            System.out.println("\n\nExit Value is " + exitValue);  
        } catch (InterruptedException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
}

# What is printf

printf is used when you want to format your string. This will clean up any concatenation. For a simple example -> ("Hello, " + username + "! How are you?") could easily be cleaned up a bit by using ("Hello, %s! How are you?", username). It's up to you really when you want to use this.

%s 🡪 it will be replaced with username as string

# 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. Each part of such program is called a thread. So, threads are light-weight processes within a process.  
  
Threads can be created by using two mechanisms :  
1. Extending the Thread class

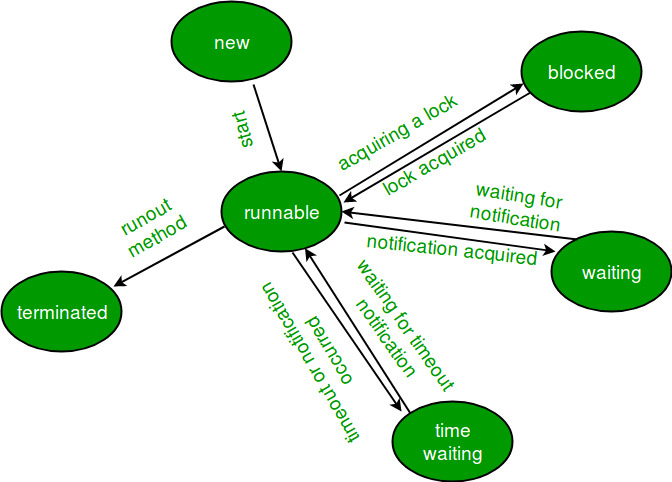
* The class should extend Java Thread class.
* The class should override the run() method.
* The functionality that is expected by the Thread to be executed is written in the run() method.
* void start(): Creates a new thread and makes it runnable.  
  void run(): The new thread begins its life inside this method.

2. Implementing the Runnable Interface

* The class should implement the Runnable interface
* The class should implement the run() method in the Runnable interface
* The functionality that is expected by the Thread to be executed is put in the run() method

## Lifecycle and States of a Thread in Java

A [thread](http://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:



1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.  
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
3. **Blocked/Waiting state:**When a thread is temporarily inactive, then it’s in one of the following states:

Blocked

Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

1. **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.
2. **Terminated State:** A thread terminates because of either of the following reasons:

Because it exists normally. This happens when the code of thread has entirely executed by the program.

Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in terminated state does no longer consumes any cycles of CPU.

### Why don’t we call run() method directly, why call start() method?

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of [multithreading](https://beginnersbook.com/2013/03/multithreading-in-java/).When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

On the other hand if the run() method of these threads are being called directly then the execution of all of them is being handled by the same current thread and no multithreading will take place, hence the output would reflect the sequential execution of threads in the specified order.

## Thread vs Process

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

4) A process has its own address space. A thread uses the process’s address space and share it with the other threads of that process.

6) A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll(). A process can communicate with other process by using inter-process communication.

7) New threads are easily created. However the creation of new processes require duplication of the parent process.

8) Threads have control over the other threads of the same process. A process does not have control over the sibling process, it has control over its child processes only.

# What are ways to Prevent Inheritance in Java Programming?

##### 1: Using final keyword

Using final keyword before a class declaration we can stop a class to be inherited by other classes. For example,

public final class A

{

}

If we try to extend the class A which is final, compiler will flash an error i.e.  
**“The Type B cannot the subclass the final Class A”,**if class B is trying to extend final class A.

##### ****2: By making a class constructor private:****

If we make the class constructor private we’ll not be able to create the object of this class from outside of this class. But, our purpose is to just prevent a class to be inherited and not to stop object creation. Hence, we need a method that can create an object of this class and return it.

We need to put a static method that will create and return an object. Why Static method? Because, from outside of a class, to call a normal method we need an object of the class, but, as constructor is private, we cannot create an object, hence, only solution is to have a static method that can be called using class name.

So, as a solution to stop a class to be extended, we need to make a constructor private and have one static method that will create an object of this class and return it.

class A {

   // Make constructor private to prevent object creation

   //From outside of this class.

    private A() {

    }

//Static method to create and return an object.

//this method will be called from outside by using

//class name only.

    public static A GetInstance() {

        return new A();

    }

}

# [Can we overload the main method in Java?](https://stackoverflow.com/questions/3759315/can-we-overload-the-main-method-in-java)

You *can* overload the main() method, but only public static void main(String[] args) will be used when your class is launched by the JVM. For example:

publicclassTest {

publicstaticvoid main(String[] args) {

System.out.println("main(String[] args)");

}

publicstaticvoid main(String arg1) {

System.out.println("main(String arg1)");

}

publicstaticvoid main(String arg1, String arg2) {

System.out.println("main(String arg1, String arg2)");

}

}

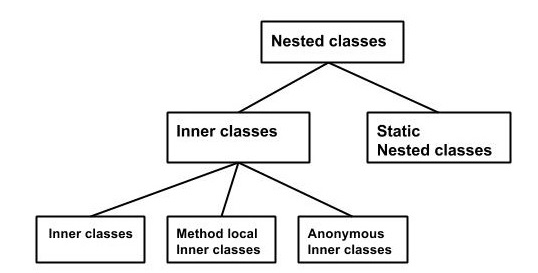
That will *always* print main(String[] args) when you run java Test ... from the command line, even if you specify one or two command-line arguments.

You can call the main() method yourself from code, of course - at which point the normal overloading rules will be applied.

Yes, you can overload main method in Java. But the program doesn't execute the overloaded main method when you run your program, you have to call the overloaded main method from the actual main method.

# Inner Class

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.



Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

* [static keyword](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/) can be applied on classes and interfaces only when they are nested. Top-level classes and interfaces cannot be static.
* **the static inner class cannot access outer class non-static members**"

## Why Use Nested Classes?

* It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
* It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
* It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

## Inner Class-Static

Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected.

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

### Static Inner Class --with non-static method

class TestOuter1{

static int data=30;

static class Inner{

void msg(){System.out.println("data is "+data);}

}

public static void main(String args[]){

TestOuter1.Inner obj=new TestOuter1.Inner();

obj.msg(); }

}

you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

### Static Inner Class --with static method

class TestOuter2{

static int data=30;

static class Inner{

static void msg(){System.out.println("data is "+data);}

}

public static void main(String args[]){

TestOuter2.Inner.msg();//no need to create the instance of static nested class

}

}

<https://www.javatpoint.com/static-nested-class>

### If Static inner class is public/default/protectedinside Outer class

#### Inner class Object- Outside/Inside Outer class

Outer\_class.Inner\_StaticClass obj = **new**Outer\_class.Inner\_StaticClass ();

#### If static inner class is inside interface then implement interface

interface.Inner\_StaticClass obj = **new**interface.Inner\_StaticClass ();

#### If static inner class is inside interface then extend inner class

by interface.Inner\_StaticClass

implClass obj=new implClass();

obj.InnerClassMethod()

## Inner Class -Non Static

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.These inner classes have access to other members of the enclosing class, even if they are declared private.

Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Since inner classes are associated with instance, we can’t have any **static variables** in them.

### If inner class is public/default/protectedinside Outer class

#### Inner class Object- Outside Outer class

Outer\_Class outer = **new** Outer\_Class ();

Inner\_Class obj=outer.**new** Inner\_Class ();

#### Inner class Object inside Outer class

Inner\_Class obj=**new** Inner\_Class ();

This obj can access even private member of inner class.

#### Accessing Members of Outer Class within Inner Class

inner classes can access the members of the outer class using outerClassName.this.variableName/methodName

**member of inner class** can be accessed by this.variableName/MethodName

public class ShadowTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

System.out.println("x = " + x);

System.out.println("this.x = " + this.x);

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

}}

public static void main(String... args) {

ShadowTest st = new ShadowTest();

ShadowTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}}

output of this example:

x = 23

this.x = 1

ShadowTest.this.x = 0

# Inner Class extend?

## Inner class-static

### Outside Outer class

**publicclass** InnerClass\_Static\_Extend **extends** AA.BB{

**publicstaticvoid** main(String[] args) {

InnerClass\_Static\_Extend obj=**new** InnerClass\_Static\_Extend();

obj.bMethod();

}

}

**class** AA

{

**staticclass** BB

{

**int**i=10;

**void** bMethod() {System.***out***.println("this is bMethod");}

}

**void** aMethod() {System.***out***.println("this is aMethod");}

}

## Inner class-non static

If you are extending non-static inner class, then sub class constructor must explicitly call super class constructor using an instance of outer class. Because, you can’t access non-static inner class without the instance of outer class.

**publicclass** InnerClassExtend **extends** A.B{

InnerClassExtend()

{**new** A().**super**(); }

**publicstaticvoid** main(String[] args) {

InnerClassExtend obj=**new** InnerClassExtend();

obj.bMethod();

}

}

class A

{

class B

{

inti=10;

void bMethod() {System.*out*.println("this is bMethod");}

}

void aMethod() {System.*out*.println("this is aMethod");}

}

<https://javaconceptoftheday.com/inheritance-inner-classes-java/>

If inner class is public/default/protected**inside Interface**

If you want to use class that it has to extended as below

public class Test extends InterfaceName. Innerclass

Test obj=new Test();

If you are only implementing interface then (inner class inside interface will be useful)

public class Test implementsInterfaceName

Test obj=new Test();

OR

public class Test extends InterfaceName. InnerclassimplementsInterfaceName

If inner class is private then it can not be access outside of class

Inner\_Class obj=new Inner\_Class(); create object inside outer class method

# Method--local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.Since local inner class is not associated with Object, no access modifier is allowed. A method-local inner class can be instantiated only within the method where the inner class is defined.

Class object can not be created before class definition.

## Properties

* local class can access local variables and parameters of the enclosing block that are final or effectively final.
* A variable or parameter whose value is never changed after it is initialized is **effectively final**.
* Local classes are similar to inner classes because they cannot define or declare any static members.

# Anonymous Inner Class --only override method

An inner class declared without a class name is known as an anonymous inner class for which only a single object is created.An anonymous class is declared and instantiated in a single statement. They are like local classes except that they do not have a name.

## Accessing Local Variables of the Enclosing Scope

Like local classes, anonymous classes can capture variables

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name. See Shadowing for more information.
* An anonymous class cannot have a constructor. Thus, you cannot pass parameters to an anonymous class when you instantiate it.

**Anonymous classes also have the same restrictions as local classes with respect to their members:**

* You cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

**you can declare the following in anonymous classes:**

* Fields
* Extra methods (even if they do not implement any methods of the supertype)
* Instance initializers
* Local classes

## Use of anonymous class

* Override method without extending class
* If you want to call parent method then create another object inside override method.

## Anonymous class properties

* Anonymous inner class always extend a class(abstract or concrete) or implement an interface.
* Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class.
* Anonymous inner classes are accessible only at the point where it is defined.

## anonymous class expression consists of the following.

* The new operator
* The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface HelloWorld.
* Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
* A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

### Syntax:

|  |
| --- |
| //Java program to demonstrate need for Anonymous Inner class  Interface Age  {      intx = 21;      voidgetAge();  }  // Myclass implement the methods of Age Interface  Class **MyClass** implements Age  {      publicvoidgetAge()      {          // printing the age          System.out.print("Age is "+x);      }  }  Class AnonymousDemo  {      publicstaticvoidmain(String[] args)      {          // Myclass is implementation class of Age interface          MyClass obj=newMyClass();            // calling getage() method implemented at Myclass          obj.getAge();      }  } |

**Anonymous inner class version of the above Program**

//Java program to demonstrate Anonymous inner class

Interface Age

{

    intx = 21;

    void getAge();

}

classAnonymousDemo

{

    publicstaticvoidmain(String[] args) {

        // Myclass is hidden inner class of Age interface

        // whose name is not written but an object to it

        // is created.

        Age oj1 = new Age() {

            @Override

            Public void getAge() {

                 // printing  age

                System.out.print("Age is "+x);

            }

        };

        oj1.getAge();

    }

}

Here, an object to Age is not created but an object of Myclass is created and copied in the entire class code as shown above. This is possible only with anonymous inner class. Such a class is called ‘anonymous inner class’, so here we call ‘Myclass’ as anonymous inner class.

In the program, interface Age is created with getAge() method and x=21.  Myclass is written as implementation class of Age interface. As done in Program, there is no need to write a  separate class Myclass. Instead,   directly copy the code of Myclass into this parameter, as shown here:

### Anonymous class for interface inside in outside class

class J {

interface K {

void kMethod();

}

}

class P {

void mImplement() {

J.K jkobj = new J.K() { //anonymous class for interface K

public void kMethod() {

System.out.println("this is anonymous kmethod");

}};

jkobj.kMethod();

}

### Anonymous class for interface inside in outer class

class P {

interface KK {

void kkMethod();

}

void mImplement() {

KK kkobj = new KK() {

public void kkMethod() {

System.out.println("This is kk method");

}};

kkobj.kkMethod();

}

}

## ****Types of anonymous inner class****

Based on declaration and behavior, there are 3 types of anonymous Inner classes:

1. **Anonymous Inner class that extends a class**
2. **Anonymous Inner class that implements a interface**
3. **Anonymous Inner class that defines inside method/constructor argument as below**

### Anonymous class as contructor

class MyThread

{

public static void main(String[] args)

{

//Here we are using Anonymous Inner class

//that define inside argument, here constructor argument

Thread t = new Thread(new Runnable()

{

public void run()

{

System.out.println("Child Thread");

}

});

t.start();

System.out.println("Main Thread");

}

}

# Java Anonymous Class Important Points

1. We can use any constructor while creating anonymous class. Notice the constructor argument is being used too.
2. Anonymous class extends the top-level class and implements the abstract class or interface. So access modifier rules apply as usual. We are able to access protected variable, if we change it to private then we won’t be able to access it.
3. Since we are extending concrete class above, we are not required to override all the methods. However if it would have been an interface or abstract class then we have to provide implementation of all the unimplemented methods.
4. **You cannot declare static initializers or member interfaces in an anonymous class.**
5. An anonymous class can have static members provided that they are constant variables.

# Can we define a class inside the interface?

Yes, If we define a class inside the interface, java compiler creates a **static** nested class. Let's see how can we define a class within the interface:

public class test2 extends M.MN {

public static void main(String[] args) {

test2 obj = new test2();

obj.classprint();

}

}

interface M {

void print();

class MN {

void classprint() {

System.out.println("This is classPrint");

}

}

}

# Inner Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain.

* Nested interface must be public if it is declared inside the interface
* Nested interfaces are declared implicitely public and static.
* it can have any access modifier if declared inside the class.

## Implementation of inner Interface (inside Interface)

**interface** Showable{

**void** show();

**interface** Message{

**void** msg();

  }

}

**class** TestNestedInterface1 **implements** Showable.Message{  //implementing inner interface “message”

**public** **void** msg(){System.out.println("Hello nested interface");}

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

  Showable.Message message=**new** TestNestedInterface1(); //upcasting here

  message.msg();

 }

}

## Implementation of inner Interface (inside Class)

class Test

{

    interface Yes

    {

        void show();

    } }

classTesting implementsTest.Yes

{

    publicvoidshow()

    {

        System.out.println("show method of interface");

    }

}

classA

{

    publicstaticvoidmain(String[] args)

    {

        Test.Yes obj = newTesting();

       obj.show();

    }

}

# Lambda Expression

## Functional interfaces

An interface with **single abstract method** is called functional interface.An interface is still a functional interface even if it contains default and static methods, as long as the interface only contains a single unimplemented (abstract) method.

Functional Interface is also know as SAM Interface.SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

### Properties

1. Define an interface with one and only one abstract method.
2. We cannot define more than one abstract method.
3. Use @FunctionalInterface annotation in interface definition.
4. We can define any number of other methods like Default methods, Static methods.

If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

### Is is possible to define our own Functional Interface? What is @FunctionalInterface?

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

It is not mandatory to define a Functional Interface with @FunctionalInterface annotation.

However, if we use it in Functional Interface definition, Java Compiler forces to use one and only one abstract method inside that interface.

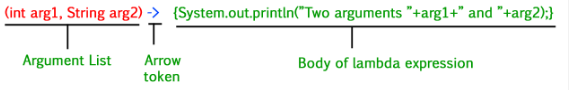
## Lambda properties

Lambda expressions basically express instances of functional interfaces.lambda expressions implement the only abstract function and therefore implement functional interfaces. A Java lambda expression is thus a function which can be created without belonging to any class.

* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.
* This expression is only used for single abstract method interface
* You can omit the data type of the parameters in a lambda expression
* In addition, you can omit the parentheses if there is only one parameter
* you can consider lambda expressions as **anonymous methods—**methods without a name.

### Syntax:

lambda operator -> body



**Zero parameter**

() -> body

**One parameter**

(p) -> body --It is not mandatory to use parentheses for single parameter. We can directly use **p->**

**Multiple parameters**

(p1, p2) -> body

**Body**

* The body of a lambda expression can contain zero, one or more statements.
* When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.
* If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

p -> a+b

**OR**

p -> {

return a+b

}

**Example:**

public class LambdaExpression {

public static void main(String[] args) {

FuncInterface fobj = (int x)->

{ //method implementation

System.out.println(2\*x);

System.out.println("Hello World");

};

// This calls above lambda expression and prints 10.

fobj.abstractFun(5);

fobj.normalFun();

}

}

interface FuncInterface

{

// An abstract function

void abstractFun(int x);

// A non-abstract (or default) function

default void normalFun()

{

System.out.println("Hello");

}

}

## Accessing Local Variables in lambda expression

**Instance,local and static variable**

Like local and anonymous classes, lambda expressions can [capture variables](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class); they have the same access to local variables of the enclosing scope.

public class LambdaScopeTest {

public int x = 0;

static int staticintOuter=10;

class FirstLevel {

public int x = 1;

static final int staticintInner=100;

void methodInFirstLevel(int x) {

Consumer<Integer> myConsumer = (y) ->

{

System.out.println("x = " + x); // Statement A

System.out.println("y = " + y);

System.out.println("this.x = " + this.x);

System.out.println("LambdaScopeTest.this.x = " + LambdaScopeTest.this.x);

println("staticintOuter="+ LambdaExpression2.staticintOuter + " staticintInner="+LambdaExpression2.FirstLevel.staticintInner);

};

myConsumer.accept(x);

}

}

public static void main(String... args) {

LambdaScopeTest st = new LambdaScopeTest();

LambdaScopeTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

This example generates the following output:

x = 23

y = 23

this.x = 1

LambdaScopeTest.this.x = 0

## When to Use lambda Expression

* Use it if you are encapsulating a single unit of behavior that you want to pass to other code.
* Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

## Lambda Expressions vs. Anonymous Interface Implementations

The major difference is, that an anonymous interface implementation can have state (member variables) whereas a lambda expression cannot. Look at this interface.

public interface MyEventConsumer {

public void consume(Object event);

}

//----------------------------------------

MyEventConsumer myEventConsumer = new MyEventConsumer() {

private int eventCount = 0;

public void consume(Object event) {

System.out.println(event.toString() + " consumed " + this.eventCount++ + " times.");

}

};

**Notice** how the anonymous MyEventConsumer implementation now has a field named eventCount.

A lambda expression cannot have such fields. A lambda expression is thus said to be stateless.

## Lambdas as Objects

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around.

public interface MyComparator {

public boolean compare(int a1, int a2);

}

MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);

* The first code block shows the interface which the lambda expression implements.
* The second code block shows the definition of the lambda expression, how the lambda expression is assigned to variable, and
* finally how the lambda expression is invoked by invoking the interface method it implements.

**Question**

Can we call create class object or call other method inside expression?

## Method References

In Java, we can use references to objects, either by creating new objects.

List list = new ArrayList();

store(new ArrayList());

Or by using existing objects

List list2 = list;

isFull(list2);

But what about a reference to a method?

If we only use a method of an object in another method, we still have to pass the full object as an argument. Wouldn't it be more practical to just pass the method as an argument? For example.

isFull(list.size);

we can do something like this using Method reference. We can use methods as if they were objects, or primitive values.

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

**Properties**

Method references are a special form of the lambda expression. Since your lambda expressions are doing nothing other than invoking existing behavior (methods), you can achieve the same result by referring to it by name.

* **::** is used to refer to a method.
* Method type arguments are inferred by JRE at runtime from the context it is defined. So no of argument and data type order must match between abstract method and reference method.
* :: tells signal to the Java compiler that this is a method reference.The method referenced(without arguments) is what comes after the double colons. Whatever **class or object** that owns the referenced method comes before the double colons.
* First of all, a method reference can't be used for any method. They can only be used to replace a single-method lambda expression.So to use a method reference, you first need a lambda expression with one method. And to use a lambda expression, you first need a functional interface, an interface with just one abstract method.

### Syntax

ContainingClass::MethodName -- for static method

Object::methodName -- for Non-Static or instance method

>>Instead of using AN ANONYMOUS CLASS

>>you can use A LAMBDA EXPRESSION

>> And if this just calls one method, you can use A METHOD REFERENCE

--

### Types of Method References

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

#### Reference to a static method

(args) -> Class.staticMethod(args) **OR** Class::staticMethod

Notice that between a static method and a static method reference, instead of the **.**operator, we use the**::**operator, and that we don't pass arguments to the method reference.

In general, we don't have to pass arguments to method references. However, arguments are treated depending on the type of method reference.

public class MethodReferences {

public static void staticMethod() {

System.out.println("Hello, this is static method.");

}

public static void main(String[] args) {

// Referring static method

Sayable sayableStatic = MethodReferences::staticMethod;

// Calling interface method

sayableStatic.say();

}}

interface Sayable {

void say();

}

#### Reference to an Instance Method

(obj, args) -> obj.instanceMethod(args) **OR** ObjectType::instanceMethod

Where an instance of an object is passed, and one of its methods is executed with some optional(s) parameter(s).

public class MethodReferences {

public void nonstaticMethod(){

System.out.println("Hello, this is non-static method.");

}

public static void main(String[] args) {

MethodReferences obj=new MethodReferences();

Sayable sayableNonStatic =obj::nonstaticMethod;

sayableNonStatic.say();

}}

interface Sayable {

void say();

}

**Example:2**

Existing functional interface **– Consumer**

Consumer<String>c=System.***out***::println;

c.accept("Hello");

#### Reference to a Constructor

(args) -> new ClassName(args) OR ClassName::new

The only thing this lambda expression does is to create a new object and we just reference a constructor of the class with the keyword new.

Java method returns object as below

public List<String> get() {

return new ArrayList<String>();

}

public class MethodReferences {

MethodReferences(String str)

{

System.out.println("This is MethodReferences construtor--" + str);

}

public static void main(String[] args) {

Messageable messageableobj=MethodReferences::**new**;

messageableobj.getMessage("Hello Method references");

}}

interface Messageable{

MethodReferences getMessage(String msg);

}

# What is Optional class

Every Java Programmer is familiar with **NullPointerException**. It can crash your code. And it is very hard to avoid it without using too many null checks.It can help in writing neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run.

# String Class

**Constructor**

String() -- return String object

String(String str) – returns String object with value=str

String(byte[] byt) –returns String by decoding byte

**length():**

method, which returns the number of characters contained in the string object.

String str=””; then it returns 0;

String str=null; then it throw exception “NullPointerException”

**string1.concat(string2):**

This returns a new string that is string1 with string2 added to it at the end.

Another Way: "Hello," + " world" + "!"

**char charAt(int index)**

Returns the character at the specified index. Str.charAt(0)

**boolean equals(Object anObject)**

Compares this string to the specified object.

**boolean equalsIgnoreCase(String anotherString)**

Compares this String to another String, ignoring case considerations.

**byte getBytes()**

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

**int indexOf(String str)**

Returns the index within this string of the first occurrence of the specified substring.

**int indexOf(Stringstr, int fromIndex)**

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

**String replaceAll(String regex, String replacement)**

Replaces each substring of this string that matches the given regular expression with the given replacement.

**String replaceFirst(String regex, String replacement)**

Replaces the first substring of this string that matches the given regular expression with the given replacement.

**String[] split(String regex)**

Splits this string around matches of the given regular expression.

**char[] toCharArray()**

Converts this string to a new character array. For revert back , String str=new String(charArrayObj)

**boolean startsWith(String prefix)**

**boolean endsWith(String prefix)**

**String substring(int beginIndex)**

Returns a new string that is a substring of this string.

**String substring(int beginIndex, int endIndex)**

Returns a new string that is a substring of this string.

**String toLowerCase()**

**String toUpperCase()**

**String trim()**

**valueOf(int i)**

Converts int to String

Traversing each character of string

String str = "This is khalid";

for (int i = 0; i < str.length(); i++)

char c = str.charAt(i);

char[] ch = str.toCharArray();

for(int i=0; i<ch.length;i++)

System.out.print(ch[i]);

What are different ways to create String Object?

String str = new String("abc");

String str1 = "abc";

Write a method that will remove given character from the String?

String str="aaaaabbbbccc";

str.replaceAll("c", "");

How to convert String to char and vice versa?

String is a sequence of characters;so we can’t convert it to a single character.

We can use use charAt method to get the character at given index or we can use toCharArray() method to convert String to character array.

String str = "This is khalid Anwar";

System.out.println(str.charAt(5));

char[] ch = str.toCharArray(); //convert String to array-character

String str1 = new String(ch); //convert array-character to String

Write a java program to find the character count and their number of occurrences in a string?

Map<Character, Integer> hm = new HashMap<>();

String str = "abcdefghiabcdefghiabcdfeghijk";

char[] ch = str.toCharArray();

for (int i = 0; i < ch.length; i++) {

hm.put(ch[i], hm.get(ch[i]) == null ? 1 : hm.get(ch[i]) + 1);

}

System.out.println("a count: " + hm.get('a'));

# Character Class

Java provides a wrapper class Character in java.lang package. An object of type Character contains a single field, whose type is char.

Character ch = new Character('a');

The above statement creates a Character object which contain ‘a’ of type char.There is only one constructor in Character class which expect an argument of char data type.

Note : The Character class is immutable like String class i.e once it’s object is created, it cannot be changed.

**is --Method**

boolean **isLetter**(char ch) -- **Character**.*isLetter*(c)

The method will return true if it is letter([A-Z],[a-z])

boolean **isDigit**(char ch)

This method is used to determine whether the specified char value(ch) is a digit or not.

boolean **isWhitespace**(char ch)

It determines whether the specified char value(ch) is white space. A whitespace includes space, tab, or new line

System.out.println(Character.isWhitespace(' ')); //true

System.out.println(Character.isWhitespace('\n')); //true

System.out.println(Character.isWhitespace('Q'));//false

boolean **isUpperCase**(char ch)

It determines whether the specified char value(ch) is uppercase or not.

boolean **isLowerCase**(char ch)

It determines whether the specified char value(ch) is lowercase or not.

boolean **isLetterOrDigit** (char ch)

It determines whether the specified char value(ch) is integer or alphabet.

**to--Method**

char **toUpperCase**(char ch) : It returns the uppercase of the specified char value(ch)

char **toLowerCase**(char ch) : It returns the lowercase of the specified char value(ch)

**toString**(char ch) : It returns a String class object representing the specified character value(ch) i.e a one-character string

#### What is difference between isLetter and isAlphabet?

Almost same.need to google for difference

Note: \(backslash) can not be alone. It must be followed by special keyword.

|  |  |
| --- | --- |
| **ESCAPE SEQUENCE** | **DESCRIPTION** |
| \t | Insert a tab in the text at this point. |
| \b | Insert a backspace in the text at this point. |
| \n | Insert a newline in the text at this point. |
| \r | Insert a carriage return in the text at this point. |
| \f | Insert a formfeed in the text at this point. |
| \' | Insert a single quote character in the text at this point. |
| \" | Insert a double quote character in the text at this point. |
| \\ | Insert a backslash character in the text at this point. |

# Integer Class

**Integer.parseInt(int i)**

public static int parseInt(String val) throws NumberFormatException

**Integer.valueof(anyTypevalue)**

Return int

**Integer.toString(int i)**

public static String toString(int i)

int a = 1234;

int b = -1234;

String str1 = Integer.toString(a);

String str2 = Integer.toString(b);

System.out.println("String str1 = " + str1);

System.out.println("String str2 = " + str2);

Output:

String str1 = 1234

String str2 = -1234

# Double class

Double.valueof(any\_value) -- return double

# Java Regex

It is widely used to define the constraint on strings such as password and email validation, word searches.Java Regex API provides 1 interface and 3 classes in java.util.regex package.

The java.util.regex package provides following classes and interfaces for regular expressions.

* MatchResult interface
* Matcher class
* Pattern class
* PatternSyntaxException class

Pattern p = Pattern.compile(".s");

Matcher m = p.matcher("as");

boolean b = m.matches();

// OR

System.out.println(Pattern.compile(".s").matcher("as").matches());

To develop regular expressions, ordinary and special characters are used:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \$ | ^ | . | + | ? | [ | ] | \. | \* |

|  |  |
| --- | --- |
| **Character Class** | **Description** |
| [abc] | Find one character from the options between the brackets |
| [^abc] | Find one character NOT between the brackets |
| [a-zA-Z] | a through z or A through Z, inclusive (range) |
| [0-9] | Find one character from the range 0 to 9 |
| [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| [a-z&&[def]] | d, e, or f (intersection) |
| [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |
| \\ | Backslash character |
| \t \n \s \S | Tab New line whitespace non-whiteSpace(Find a whitespace character) |
| \d \D | Find a digit e.g [0-9] [^0-9] |
| \w \W | wordCharacter e.g[a-zA-Z0-9] non-wordCharaacter [^\w] |
| . | Find just one instance of any character |
| ^ | Finds a match as the beginning of a string as in: ^Hello |
| $ | Finds a match at the end of the string as in: World$ |
| n+ | Matches any string that contains at least one *n* |
| n\* | Matches any string that contains zero or more occurrences of *n* |
| n? | Matches any string that contains zero or one occurrences of *n* |

# String Utility Class

## StringBuilder

StringBuilder objects are like **String** objects, except that they can be modified. Java StringBuilder class is mutable sequence of characters. StringBuilder Class can be comparable to String however the StringBuilder class provides more versatility because of its modification features.

|  |  |
| --- | --- |
| CONSTRUCTOR | DESCRIPTION |
| StringBuilder() | Creates an empty string builder with a default capacity of 16 (16 empty elements). |
| StringBuilder(CharSequence cs) | Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence. |
| StringBuilder(int initCapacity) | Creates an empty string builder with the specified initial capacity. |
| StringBuilder(String s) | Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string. |

### Length() and Capacity()

Capacity returns the number of **character spaces** that have been allocated. The returned value is always greater than or equal to the length (usually greater than) and automatically expands whenever necessary to accommodate character additions to the string builder.

#### Capacity Calculation

Default is 16 and increases by next (prevLength)\*2+2=e.g 16, 34, 70,142

StringBuilder sb = new StringBuilder("Hello");

System.out.println(sb.capacity());

System.out.println(sb.length());

#### Length

Returns an integer equal to a number of elements present in the stringBuilder

### Convert StringBuilder to String

StringBuilder sb=new StringBuilder(“Hello World”);

String str=sb.toString();

|  |  |
| --- | --- |
| **Method** | **Description** |
| StringBuilder **append**(boolean b) | Appends the argument to this string builder. The data is converted to a string before the append operation takes place. |
| StringBuilder append(char c) |
| StringBuilder append(char[] str) |
| StringBuilder append(char[] str, int offset, int len) |
| StringBuilder append(double d) |
| StringBuilder append(float f) |
| StringBuilder append(int i) |
| StringBuilder append(long lng) |
| StringBuilder append(Object obj) |
| StringBuilder append(String s) |
| StringBuilder delete(int start, int end) | The first method deletes the subsequence from start to end-1 (inclusive) in the StringBuilder's char sequence. |
| StringBuilder deleteCharAt(int index) |
| StringBuilder **insert**(int offset, boolean b) | Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place. |
| StringBuilder insert(int offset, char c) |
| StringBuilder insert(int offset, char[] str) |
| StringBuilder insert(int index, char[] str, int offset, int len) |
| StringBuilder insert(int offset, double d) |
| StringBuilder insert(int offset, float f) |
| StringBuilder insert(int offset, int i) |
| StringBuilder insert(int offset, long lng) |
| StringBuilder insert(int offset, Object obj) |
| StringBuilder insert(int offset, String s) |
| StringBuilder replace(int start, int end, String s) | Replaces the specified character(s) in this string builder. |
| void setCharAt(int index, char c) |
| StringBuilder **reverse()** | Reverses the sequence of characters in this string builder. |
| String **toString()** | Returns a string that contains the character sequence in the builder. |

**Note:** There is also a StringBuffer class that is *exactly* the same as the StringBuilder class, except that it is thread-safe by virtue of having its methods synchronized.

**StringBuffer is thread safe**

# String Vs StringBuilder vs StringBuffer

## StringReader

Java StringReader class is a character stream with string as a source. It takes an input string and changes it into character stream. It inherits **Reader** class.

public class StringReader extends Reader

String srg = "Hello Java!! \nWelcome to Javatpoint.";

StringReader reader = new StringReader(srg);

int k=0;

while((k=reader.read())!=-1){

System.out.print((char)k);

}

## StringWriter

Java StringWriter class is a character stream that collects output from string buffer, which can be used to construct a string. The StringWriter class inherits the **Writer** class.

public class StringWriter extends Writer

StringWriter sw = new StringWriter();

sw.write(65);

sw.write('a');

// Use of toString() : Value written by write(int char)

System.out.println("Using write(int char) : " + sw.toString());

String str = "Hello Geeks";

sw.write("\n");

sw.write(str);

System.out.println("Using write(int char) : " + sw.toString());

# Generic Type

## Why Generic is used

* Stronger type checks at compile time.
* A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.

**The following code snippet without generics requires casting:**

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

**When re-written to use generics, the code does not require casting:**

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

list.add("hello");

String s = list.get(0); // no cast

## Type Parameter Naming Conventions

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value

## It is applicable to

1. Class
2. Method
3. Interfaces

 For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

**Note:** In Parameter type we can not use primitives like 'int','char' or 'double'.

### Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that appears **before thethe method's return type**.
* **Each type parameter section contains one or more type parameters separated by commas**. A type parameter, also known as a type variable, is an identifier that specifies a generic type name

class Test

{

// A Generic method example

static <T> void genericDisplay (T element)

{

System.out.println(element.getClass().getName() +

" = " + element);

}

// Driver method

public static void main(String[] args)

{

// Calling generic method with Integer argument

genericDisplay(11);

// Calling generic method with String argument

genericDisplay("GeeksForGeeks");

// Calling generic method with double argument

genericDisplay(1.0);

}

}

### Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

#### Single argument

Public class Box<T>{

private T t;

public void add(T t){

this.t = t;

}

public T get(){

return t;

}

Public static void main(String[] args){

Box<Integer> integerBox =newBox<Integer>();

Box<String> stringBox =newBox<String>();

integerBox.add(newInteger(10));

stringBox.add(newString("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond.

Box<Integer> integerBox = new Box<>();

### Multiple argument

// A Simple Java program to show multiple

// type parameters in Java Generics

// We use <> to specify Parameter type

classTest<T, U>

{

    T obj1;  // An object of type T

    U obj2;  // An object of type U

    // constructor

    Test(T obj1, U obj2)

    {

        this.obj1 = obj1;

        this.obj2 = obj2;

    }

    // To print objects of T and U

    publicvoidprint()

    {

        System.out.println(obj1);

        System.out.println(obj2);

    }

}

// Driver class to test above

classMain

{

    publicstaticvoidmain (String[] args)

    {

        Test <String, Integer> obj =

            newTest<String, Integer>("GfG", 15);

        obj.print();

    }

}

## Raw Types

A raw type is the name of a generic class or interface without any type arguments.

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

Box<Integer> intBox = new Box<>();

If the actual type argument is omitted, you create a raw type of Box<T>:

Box rawBox = new Box();

Therefore, you should avoid using raw types

# Enumerations

Enums are lists of constants like unchangeable variables. Have you heard of Final keyword? It’s like that.

An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. E.g days of the week, month of year here you know all possible values at compile time.

In Java, enumeration defines a class type. An Enumeration can have

* Constant --implicitly **public**, **static** and **final**
* Constructors
* methods(only concrete)
* instance variables.
* It is created using **enum** keyword
* Every enum are implicitly **static and final class** (public/default)

**enum** WeekDays

{ ***sun***, ***mon***, ***tues***, ***wed***, ***thurs***, ***fri***, ***sat*** }

**publicstaticvoid** main(String[] args) {

WeekDays wk=**null**;

wk=WeekDays.***mon***;

Variables of Enumeration type can have only enumeration constants as value. We define an enum variable as enum\_variable = enumType.enumConstant;

## Calling Method,variable,Constructor

In order to call Enum’s method, we can not just directly call as usual class’s method “myClass.doSomething()”.

Instead, we must assign a value to the Enum before we can invoke the method.

## Values( ) and ValueOf( ) method

All the enumerations predefined methods values() and valueOf(). **values()** method returns an array of enum-type containing all the enumeration constants in it.

## Accessing Enum

enum outClass

{

AA,BB,CC,DD;

int i=10;

void print()

{

System.out.println("Hello print");

}

}

class Outer

{

enum innerenum

{

IA,IB,IC;

}

}

public class EnumOutsideClass {

enum inClass

{

A,B,C;

}

public static void main(String[] args) {

System.out.println(outClass.AA);

System.out.println(inClass.A);

System.out.println(Outer.innerenum.IA);

}

}

## Enumerations Properties

1. Enumerations are of class type, and have all the capabilities that a Java class has.

enum Color

{

RED, GREEN, BLUE;

}

Every enum internally implemented by using Class.internally creates a **static and final class** that extends the Enum class.

Static final class Color extends Enum

{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}

1. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.

enum Color

{

RED, GREEN, BLUE;

private Color()

{

System.out.println("Constructor called for : " + this.toString());

}

// Only concrete (not abstract) methods allowed

public void colorInfo()

{

System.out.println("Universal Color");

}

}

Main method—

Color c1;

c1 = Color.***RED***;

System.***out***.println(c1);

c1.colorInfo();

1. Enumerations are not instantiated using new keyword.
2. All Enumerations by default inherit java.lang.Enum class.
3. Enum declaration can be done outside a Class or inside a Class but not inside a Method.
4. Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name.
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

## Can we create the instance of enum by new keyword?

No, because it contains private constructors only.

## Can we have abstract method in enum?

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

--

class EnumExample1{

public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {

for (Season s : Season.values())

System.out.println(s);

}}

Equivalent Class internally by Java Compiler

public static final class EnumExample1$Season extends Enum

{

private EnumExample1$Season(String s, int i)

{

super(s, i);

}

public static EnumExample1$Season[] values()

{

return (EnumExample1$Season[])$VALUES.clone();

}

public static EnumExample1$Season valueOf(String s)

{

return (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);

}

public static final EnumExample1$Season WINTER;

public static final EnumExample1$Season SPRING;

public static final EnumExample1$Season SUMMER;

public static final EnumExample1$Season FALL;

private static final EnumExample1$Season $VALUES[];

static

{

WINTER = new EnumExample1$Season("WINTER", 0);

SPRING = new EnumExample1$Season("SPRING", 1);

SUMMER = new EnumExample1$Season("SUMMER", 2);

FALL = new EnumExample1$Season("FALL", 3);

$VALUES = (new EnumExample1$Season[] {

WINTER, SPRING, SUMMER, FALL

});

}

}

# Enum Fields

You can add fields to a Java enum. Thus, each constant enum value gets these fields. The field values must be supplied to the constructor of the enum when defining the constants. Here is an example:

public enum Level {

HIGH (3), //calls constructor with value 3

MEDIUM(2), //calls constructor with value 2

LOW (1) //calls constructor with value 1

; // semicolon needed when fields / methods follow

private final int levelCode;

private Level(int levelCode) {

this.levelCode = levelCode;

}

}

# EnumSet

Java contains a special Java Set implementation called EnumSet which can hold enums more efficiently than the standard Java Set implementations. Here is how you create an instance of an EnumSet :

EnumSet<Level> enumSet = EnumSet.of(Level.HIGH, Level.MEDIUM);

Once created, you can use the EnumSet just like any other Set.

# EnumMap

Java also contains a special Java Map implementation which can use Java enum instances as keys. Here is a Java EnumMap example:

EnumMap<Level, String> enumMap = new EnumMap<Level, String>(Level.class);

enumMap.put(Level.HIGH , "High level");

enumMap.put(Level.MEDIUM, "Medium level");

enumMap.put(Level.LOW , "Low level");

String levelValue = enumMap.get(Level.HIGH);

# What is Java ClassLoader?

Running a java program means telling JVM to load class then start executing its main() method,keep running till all code in main method is finished.

We know that Java Program runs on Java Virtual Machine (JVM). When we compile a Java Class, it transforms it in the form of bytecode that is platform and machine independent compiled program and store it as a .class file. After that when we try to use a Class, Java ClassLoader loads that class into memory.

There are three types of built-in ClassLoader in Java:

**Bootstrap Class Loader**

It loads JDK internal classes, typically loads rt.jar (C:\Program Files\Java\jdk1.8.0\_172\jre\lib\rt.jar) and other core classes for example java.lang.\* package classes

**Extensions Class Loader**

It loads classes from the JDK extensions directory, usually C:\Program Files\Java\jdk1.8.0\_172\jre\lib\ext directory.

**System Class Loader**

It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.

# What is POJO Class?

There are websites which actually helps you to create POJO java classes out of JSON like

<http://www.jsonschema2pojo.org/>

## ****Maven Dependencies****

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.8.6</version>

</dependency>

## Json file

[{

"name":*"Khalid1"*,

"age":30

},

{

"name":*"Khalid2"*,

"age":35

}

]

## Pojo class

**public** **class** POJO {

**public** String name;

**public** Integer age;

## }

## Driver Class

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.util.Arrays;

**import** java.util.List;

**import** com.google.gson.Gson;

**public** **class** jsonExample {

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

Gson gson = **new** Gson();

BufferedReader bufferReader = **null**;

bufferReader = **new** BufferedReader(**new** FileReader("C:\\kk\\JavaWorkspace\\JavaBasic\\src\\jsonPOJO\\k.json"));

POJO[] pj = gson.fromJson(bufferReader, POJO[].**class**);

List<POJO> pjj = Arrays.*asList*(pj);

**for** (**int** i = 0; i < pjj.size(); i++) {

System.***out***.println(pjj.get(i).name);

System.***out***.println(pjj.get(i).age);

}

}

## }

# Apache Tomcat configuration

## The Tomcat Installed Directory

**bin**: for Tomcat’s binaries and startup scripts.

**conf**: global configuration applicable to all the webapps. The default installation provides:

catalina.policy for specifying security policy.

**Two** Properties Files: catalina.properties and logging.properties,

**Four** Configuration XML Files: server.xml (Tomcat main configuration file), web.xml (global web application deployment descriptors), context.xml (global Tomcat-specific configuration options) and tomcat-users.xml (a database of user, password and role for authentication and access control).

**lib**: Keeps the JAR-file that are available to all webapps. The default installation include servlet-api.jar (Servlet), jasper.jar (JSP) and jasper-el.jar (EL). External JARs can be put here such as MySQL JDBC driver (mysql-connector-java-5.1.{xx}-bin.jar) and JSTL (jstl.jar and standard.jar).

**logs**: contains the engine logfile Catalina.{yyyy-mm-dd}.log, host logfile localhost.{yyyy-mm-dd}.log, and other application logfiles such as manger and host-manager. The access log (created by theAccessLogValve) is also kept here.

**webapps**: the default appBase – web applications base directory of the host localhost.

**work**: contains the translated servlet source files and classes of JSP/JSF. Organized in hierarchy of engine name (Catalina), host name (localhost), webapp name, followed by the Java classes package structure.

temp: temporary files.