**Question-1 What are the object oriented concepts? what is difference between object –based , object-oriented and fully object oriented language?**

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. Object Oriented programming (OOP) is a programming paradigm that relies on the concept of **classes** and **objects**

**Class :-**

To represent data we need class. Class is a blueprint of object that contains variables and methods. It is logical representation of data. Class is non- primitive/ reference type in java

For example, our Car class may have a method repaint that changes the color attribute of our car. This function is only helpful to objects of type Car, so we declare it within the Car class thus making it a method.

Example blue print of house design

**Objects : -**

Objects represents physical entity

For eg. Suppose we have a class car and there are multiple cars .BMW car represents one object of the car

For eg : Car class may have real attributes like color, brand and model and some method repaint to change the color of the car. So we can create instance / object of the car to access the fields with the help of new operator.

**Characteristics of Object :-**

♣ An object has **identity** (each object is a distinct individual).

♣ An object has **state** (it has various properties, which might change).

♣ An object has **behaviour** (it can do things and can have things done to it).

**Advantages of OOPS**

1. To achieve simplicity

2. To achieve data hiding and data security.

3. To minimize the module dependency so that failure in single part should not stop complete system.

4. To achieve reusability so that we can reduce development time/cost/efforts.

5. To reduce maintenance of the system.

6. To fully utilize hardware resources.

7. To maintain state of object on secondary storage so that failure in system should not impact on data.

Object oriented concepts :

Major Pillar : 1.Abstraction 2.Encapsulation 3.Modularity 4.Hierarchy

Minor Pillar :-1.Typing /polymorphism 2.Concurrency 3. Persistence

**Object Oriented Languages**

• Object Oriented Languages supports all the features of Oops including inheritance and polymorphism.

• They support built-in objects.

• C#, Java, VB. Net are the examples of object oriented languages.

**Object Based Languages**

• Object based languages supports the usage of object and encapsulation.

• They does not support inheritance or, polymorphism or, both.

• Object based languages does not supports built-in objects.

• JavaScript, VB are the examples of object bases languages.

**Disadvantages of OOP**

* Size: Object Oriented Programs are much larger than other programs.
* Effort: Object Oriented Programs require a lot of work to create.
* Speed: Object Oriented Programs are slower than other programs, because of their size

**Abstraction :-**

**Getting only essential things from object and hiding internal details.**

**Using abstraction we can achieve simplicity.**

for example, when we ride a bike, we only know about how to ride bikes but can not know about how it work? And also we do not know the internal functionality of a bike.

Creating instance and calling method on it is abstraction.

We can achieve abstraction in java by 1.Abstract class 2. Interfaces

**Encapsulation**

Binding of data and code together is called encapsulation…….Data security….Helps to achieve loose coupling

In java by defining class we achieved encapsulation.

**Que :- How we can achieve encapsulation?**

Encapsulation in Java can be achieved by:

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

**Realtime Example :**  
When you log into your email accounts such as Gmail, Yahoo Mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it.

When you enter the password for logging, they are retrieved in an encrypted form and verified, and then you are given access to your account.

You do not have control over it that how the password has been verified. Thus, it keeps our account safe from being misused

public class Account

{

private double balance;

public double getbalance()

 {

   return balance;

 }

}

**Modularity:**

It is the process of developing complex system using small parts. • Using modularity, we can reduce module dependency. • We can implement modularity by creating library files

**Real Time Example :**

While developing a car, car manufacture don’t produce all the parts at one place. There are some vendors who provides parts to the manufacturer. Eg bolt, tyre, sheets, engine etc.

**Hierarchy :-**

Level / order / ranking of abstraction is called hierarchy. • Main purpose of hierarchy is to achieve reusability.

• Advantages of code reusability

1. We can reduce development time.

2. We can reduce development cost.

3. We can reduce developers effort.

• Types of hierarchy: 1. Has-a / Part-of => Association 2. Is-a / Kind-of => Inheritance / Generalization 3. Use-a => Dependency 4. Creates-a => Instantiation

**Polymorphism/ Typing:-**

An ability of object to take multiple forms is called polymorphism. • Using polymorphism, we can reduce maintenance of the system.

**Real-Life Example** of Polymorphism in Java Before going towards classical definition, let’s understand the concept of polymorphism with the help of a real-life example. Consider a Smartphone. One can use a Smartphone for calling. The same device can be used to listen to music or to watch videos. So, with the help of one Smartphone, we can do several different kinds of tasks with the help of various applications. Thus it can be considered a real-life example of polymorphism

• **Types of polymorphism**:

o **Compile time polymorphism** Ø It is also calling static polymorphism. Ø We can achieve it using: 1. Method Overloading

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in signature, it is known as **Method Overloading**.. We can improve readability of the code

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }
5. **class** TestOverloading1{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

### Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

### Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

1. **class** TestOverloading4{
2. **public** **static** **void** main(String[] args){System.out.println("main with String[]");}
3. **public** **static** **void** main(String args){System.out.println("main with String");}
4. **public** **static** **void** main(){System.out.println("main without args");}
5. }

Output :- main with string

o **Run time polymorphism** Ø It is also calling dynamic polymorphism. Ø We can achieve it using: 1. Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

class Vehicle{

//defining a method

void run(){System.out.println("Vehicle is running");}

}

//Creating a child class

class Bike2 extends Vehicle{

//defining the same method as in the parent class

void run(){System.out.println("Bike is running safely");}

public static void main(String args[]){

Bike2 obj = new Bike2();//creating object

obj.run();//calling method

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance

#### 1. Overriding Access-Modifiers

We can change the access modifier for an overriding method. In the derived class, while overriding a method, we can provide less restriction, but not more, restrictive access than the access of the overridden method of the superclass.

For example, the method declared as public in the super-class cannot be made private or protected while overriding it in the subclass.

Similarly, the protected method can be made public but not private in the subclass. If we provide lesser access in the subclass than that in the superclass, then we will get a compile-time error.

### When to Apply Method Overriding in Java

Method Overriding is when a class has several derived classes and the derived classes need to use the methods of their parent class with the same signature (number, type, and order of parameter), but with the different implementation.

They can override the same method and add specific functionality without even disturbing the code of the parent class.

### Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

you cannot override the static method in Java because the method overriding is based upon dynamic binding at runtime and static methods are bonded using static binding at compile time. This means static methods are resolved even before objects are created, that's why it's not possible to override static methods in Java. Though you can declare a method with the same name and method signature in the subclass which does look like you can override static methods in Java but in reality that is method hiding.

Read more: https://www.java67.com/2012/08/can-we-override-static-method-in-java.html#ixzz77gIRo3ya

### Can we override java main method?

No, because the main is a static method.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4) | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

**Concurrency :-**

It is the process of executing multiple task simultaneously. • Main purpose of concurrency is to utilise CPU efficiently. • In Java, we can achieve concurrency using thread.

**Persistence :-**

It is process of maintaining state of object on secondary storage. • In Java, we can achieve Persistence using file and database

**Association :-**

If has-a relationship is exist between the types then we should use association. If object/instance is a part/component of another instance then it is called as association. • To implement association, we should declare instance of a class as a field inside another class.

Eg. Engine is a part of car

# Aggregation in Java :- Loose Coupling –

Loose coupling – obj are not fully dependent on the each other

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

1. **class** Employee{
2. **int** id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }

**Composition in Java :- Tight Coupling**

A composition in Java **between two objects associated with each other exists when there is a strong relationship between one class and another**

### Benefits of using Composition:

* Composition allows us to reuse the code.
* In Java, we can use multiple Inheritance by using the composition concept.
* The Composition provides better test-ability of a class.
* Composition allows us to easily replace the composed class implementation with a better and improved version.
* Composition allows us to dynamically change our program's behavior by changing the member objects at run time

There are two special forms of Association in Java. They are:  
**1.** Composition  
**2.** Aggregation

**Aggregation: Loose Coupling**

Aggregation in Java is a special kind of association. It represents the Has-A relationship between classes. Java Aggregation allows only one-to-one relationships.

If an object is destroyed, it will not affect the other object, i.e., both objects can work independently.

Let’s take an example. There is an Employee in a company who belongs to a particular Department. If the **Employee** object gets destroyed still the **Department** can work independently.

The end of the Employee object will not affect or destroy the Department object.

Example:

**class** Employee {

int id;

String name;

String dept;

Employee(int id, String name, String dept) {

**this**.id = id;

**this**.name = name;

**this**.dept = dept;

System.out.println("\nEmployee name is " + name);

System.out.println("Employee Id is " + id);

System.out.println("Employee belongs to the " + dept + " Department");

}

}

**class** Department {

String deptName;

int noOfemployees;

Department(String name, int numberOfemployees) {

**this**.deptName = name;

**this**.noOfemployees = numberOfemployees;

}

}

public **class** AggregationDemo {

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

Employee e1 = **new** Employee(101, "Rishi", "Engineering");

Employee e2 = **new** Employee(167, "Rohan", "Management");

Employee e3 = **new** Employee(125, "Sneha", "Accounts");

}

}

Composition:

The composition is another form of aggregation which is considered as the restricted form of Association.

In this type of association, the entities are completely dependent on each other, unlike the aggregation. Composition allows for one-to-many relationships between objects.

It represents a part-of relationship between two objects. One entity cannot exist without the other. Composition in Java represents a one-to-many relationship.

Suppose, there is a **House** and inside the house, there are many **rooms**. We consider the relationship between the house and the rooms.

Example

**class** Room {

public String roomName;

public int roomNo;

Room(String name, int number) {

**this**.roomName = name;

**this**.roomNo = number;

}

}

**class** House {

private final List < Room > rooms;

House(List < Room > rooms) {

**this**.rooms = rooms;

}

public List < Room > getTotalRoomsInHouse() {

**return** rooms;

}

}

public **class** CompositionDemo {

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

Room room1 = **new** Room("Dining Room", 2);

Room room2 = **new** Room("Bed Room", 5);

Room room3 = **new** Room("Living Room", 3);

List < Room > books = **new** ArrayList < Room > ();

books.add(room1);

books.add(room2);

books.add(room3);

House house = **new** House(books);

List < Room > rooms = house.getTotalRoomsInHouse();

**for** (Room room: rooms) {

System.out.println("The Room Number of " + room.roomName + " is: " + room.roomNo);

}

}

}

**Inheritance :-**

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship. **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

### Why use inheritance in java

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java)
* For Code Reusability.

**We use extends keyword to implement inheritance.**

**Example of Single Inheritance**

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

**Example of Multilevel Inheritance**

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Example of Hierarchical Inheritance:-

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

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

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

**Because it creates ambiguity.**

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. **public** **static** **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

**Output :- Compile Time Error**

**When to use Inheritance?**

**Ans:-**

1. **If implementation of existing class is partially complete then we can extends it**
2. **To extends the meaning of existing class**
3. **To achieve code reusability**

**What is need of Setter and getter Function in a class?**

For each instance variable, a getter method returns its value while a setter method sets or updates its value. Given this, getters and setters are also known as accessors and mutators, respectively.

1. Getters **and setter are used to protect your data.**
2. Getters and setters allow control over the values. You may validate the given value in the setter before actually setting the value.
3. Encapsulation of behavior associated with getting or setting the property. Inside encapsulation we define fields as a private and this private fields are accessible only by means of getter and setter
4. Providing a debugging interception point for when a property changes at runtime - debugging when and where a property changed to a particular value can be quite difficult without this in some languages.

**Question-11why constructor is considered as special member of class ?**

Constructors is a special member function of class and it is

1. used to initialize the objects of its class.
2. its name is the same as the class name.
3. These constructors get invoked whenever an object of its associated class is created.
4. It is named as "constructor" because it constructs the value of data member of a class.

**Question-12. what is object slicing and explain obj slicing in context of upcasting?**

**Object slicing :**Object slicing is used to describe the situation when you assign an object of a derived class to an instance of a base class.This causes a loss of methods and member variables for the derived class object. This is termed as information being sliced away.when we assign derived class object to a base class reference, doing this all methods and properties of derived class sliced away i.e you can not access these methods and properties.

**Upcasting:**

Converting ref of subclass into ref of super class is called upcasting

Eg. Person p = new Employee();

**In case of upcasting, using super class reference variable, we can not access: 1. fields of sub class 2. Non overridden methods of sub class**

**Hence object is sliced away i.e object slicing**

Using upcasting we can minimize instance dependency in the code. It helps to reduce maintenance of code which represents polymorphism …We can also achieve loose coupling

We use it when we need to develop a code that deals with only the parent class

**Question-13. what is down casting and when it is required?**

Process of converting reference of super class into reference of sub class is called down casting. **If we want to access fields and non overridden methods of sub class then we should do down casting**. Only in case of upcasting, we should do down casting. Otherwise JVM will throw ClassCastException.

• In case of upcasting, using super class reference variable, we can access overridden method of sub class. It is also called as dynamic method dispatch. Example : , Person p = new Employee( ); Employee emp = (Employee)p;

|  |  |  |
| --- | --- | --- |
| **S.No** | **Upcasting** | **Downcasting** |
| 1. | A child object is typecasted to a parent object. | The reference of the parent class object is passed to the child class. |
| 2. | We can perform Upcasting implicitly or explicitly. | Implicitly Downcasting is not possible. |
| 3. | In the child class, we can access the methods and variables of the parent class. | The methods and variables of both the classes(parent and child) can be accessed. |
| 4. | We can access some specified methods of the child class. | All the methods and variables of both classes can be accessed by performing downcasting. |
| 5. | Parent p = new Parent() | Parent p = new Child() Child c = (Chi |

**Que. What are design patterns?**Design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

1. Structural Design Pattern
2. Functional Pattern
3. Behavioural Pattern

suppose we want to deal with how to create object we should use **strucural** design pattern

if we want to achive composition that if we want to include ones object in other then we should use **functional** pattern and if we want to establish relationship between two object then we should use behavioural pattern.

**Singletone pattern :**

**Singleton Pattern** says that just"define a class that has only one instance and provides a global point of access to it". In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

**Advantage of Singleton design pattern**

o Saves memory because object is not created at each request. Only single instance is reused again and again.

**Usage of Singleton design pattern**

o Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc

class singletone {private singletone(){}static s;static void instance(){if(s==null){singletone s = new singletone();}}}

1. Make constructor private

2. Hide the instance Creation

**Shadowing in Java**

If name of super class member (Instance variable) and sub class member(local variable) are same then preference will be given to the sub class member.

The following are the differences between overriding and shadowing −

* Shadowing redefines the complete method, whereas overriding redefines only the implementation of the method.
* In Overriding, you can access the base class using the child class’ object overridden method. Shadowing has cannot access the child class methods.
* Shadowing is also known as method hiding. The method of the parent class is available to the child class without using the override keyword in shadowing. The child class has its own version of the same function.
* Under overriding, you can define a behavior that is specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

**Association Vs Inheritance**

Another type of relationship between classes is the has-a relationship or association relationship. Use this when the object of one class contains a reference to one or more of another class. We use association when we don’t know exact relationship between two objects.

Only use inheritance when the child class is really a type of the parent class, otherwise use association.We use when we know the exact relationship and we want further extends it

**What do you know about super keyword in java?**

## Usage of Java super Keyword

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.

**Q Which methods are not allowed to override in subclass?**

1. **Final Method** : A final method means that it cannot be re-implemented by a subclass, thus it cannot be overridden.
2. **Static Methods** :- we cannot override static methods because method overriding is based on dynamic binding at runtime and the static methods are bonded using static binding at compile time. So, we cannot override static methods.

The calling of method depends upon the type of object that calls the static method. It means:

* If we call a static method by using the parent class object, the original static method will be called from the parent class.
* If we call a static method by using the child class object, the static method of the child class will be called.

1. **Main method**:-because main method is static

Q ***Why main method is static?***

Java **main()**method is always static, so that compiler can call it without the creation of an object or before the creation of an object of the class.

* In any Java program, the **main()** method is the starting point from where compiler starts program execution. So, the compiler needs to call the main() method.
* If the**main()** is allowed to be non-static, then while calling the **main()** method JVM has to instantiate its class.
* While instantiating it has to call the constructor of that class, There will be ambiguity if the constructor of that class takes an argument.
* Static method of a class can be called by using the class name only without creating an object of a class.
* The **main()**method in Java must be declared **public**, **static**and **void**. If any of these are missing, the Java program will compile but a runtime error will be thrown.

**Q What is dynamic method dispatch?**

**Ans:-** In case of upcasting using super class ref var we can access overridden methods of subclass. It is called dynamic method dispatch.

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

# Que. Java instanceof Operator

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false

1. **class** Animal{}
2. **class** Dog1 **extends** Animal{//Dog inherits Animal
4. **public** **static** **void** main(String args[]){
5. Dog1 d=**new** Dog1();
6. System.out.println(d **instanceof** Animal);//true
7. }
8. }

# Que .Difference between == and .equals() method in Java

In general, both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two: 

1. The main difference between the .equals() method and == operator is that one is a method of java.lang.object class and the other is the operator.
2. We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison/value comparison**. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.
3. If a class does not [override the equals method](https://www.geeksforgeeks.org/overriding-equals-method-in-java/), then by default it uses the equals(Object o) method of the closest parent class that has overridden this method. See [this](https://www.geeksforgeeks.org/override-equalsobject-hashcode-method/)for detail
4. Coding Example:

|  |
| --- |
| public class Test {      public static void main(String[] args)      {          String s1 = "HELLO";          String s2 = "HELLO";          String s3 =  new String("HELLO");          System.out.println(s1 == s2); // true          System.out.println(s1 == s3); // false          System.out.println(s1.equals(s2)); // true          System.out.println(s1.equals(s3)); // true      }  } |

### Que. Abstract class in Java

A class which is declared with abstract keyword is known as an **abstract class**. To achieve the feature of abstraction we should write abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.
* It can contains fields

**Why we can not instantiates a abstract class?**

An abstract classs can NOT be instantiated by using new operator. Becuase an abstract may have abstract methods i.e. methods without any body (or implementation). **Because an object can NOT have an abstract methods and JVM can NOT allocate memory of the abstract methods**

Think of repairing a car. Someone has removed the brake pads and is going to replace them in the next day. Now, to prevent someone accidentally driving this car(which has no brakes installed), the mechanic installs a lock on the steering wheel. It's a fail-safe measure.

### Final Variable in Java

Once we declare a variable with the final keyword, we can’t change its value again. If we attempt to change the value of the final variable, then we will get a compilation error.

### Final Method in Java

The Method with Final Keyword cannot be overridden in the subclasses. The purpose of the Final Method is to declare methods of how’s definition can not be changed by a child or subclass that extends it. To prevent the unwanted method definitions, we declare methods as final.

### Final Class in Java

We can also declare a class with a final keyword in Java. When we declare a class as final, then we restrict other classes to inherit or extend it.

In short, Java final class can’t be extended by other classes in the inheritance. If another class attempts to extend the final class, then there will be a compilation error.

public **class** Vidvan **extends** Tech {

**void** test() {

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

}

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

Vidvan obj = **new** Vidvan();

obj.test();

}

}

final **class** Tech {

//code inside class

}

**Que Difference between error and exception**

In java, both Errors and Exceptions are the subclasses of java.lang.Throwable class. [Error](https://www.geeksforgeeks.org/types-of-errors-in-java-with-examples/) refers to an illegal operation performed by the user which results in the abnormal working of the program.

Whereas [exceptions in java](https://www.geeksforgeeks.org/exceptions-in-java/) refer to an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

| **Errors** | **Exceptions** |
| --- | --- |
| Recovering from Error is not possible. | We can recover from exceptions by either using try-catch block or throwing exceptions back to the Errors The program occur caller. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors are mostly caused by the environment in which program is running. | Program itself is responsible for causing exceptions. |
| Errors can occur at compile time as well as run time. Compile Time: eg Syntax Error  Run Time: Logical Error. | All exceptions occurs at runtime but checked exceptions are known to the compiler while unchecked isnot. |
| They are defined in java.lang.Error package. | They are defined in java.lang.Exception package |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException  Unchecked Exceptions : ArrayIndexOutOfBoundException, NullPointerException, ArithmeticException. |

**Que Difference between authentication and validation**

## What are authentication and authorization?

In simple terms, authentication is the process of verifying who a user is, while authorization is the process of verifying what they have access to.

Comparing these processes to a real-world example, when you go through security in an airport, you show your ID to authenticate your identity. Then, when you arrive at the gate, you present your boarding pass to the flight attendant, so they can authorize you to board your flight and allow access to the plane.

|  |  |  |
| --- | --- | --- |
| **S.No** | **Checked Exception** | **Unchecked Exception** |
| 1. | These exceptions are checked at compile time. These exceptions are handled at compile time too. | These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time. |
| 2. | These exceptions are direct subclasses of exception but not extended from RuntimeException class. | They are the direct subclasses of the RuntimeException class. |
| 3. | The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own. | The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic. |
| 4. | These exceptions mostly occur when the probability of failure is too high. | These exceptions occur mostly due to programming mistakes. |
| 5. | Common checked exceptions include IOException, DataAccessException, InterruptedException, etc. | Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc. |
| 6. | These exceptions are propagated using the throws keyword. | These are automatically propagated. |
| 7. | It is required to provide the try-catch and try-finally block to handle the checked exception. | In the case of unchecked exception it is not mandatory |

# ClassCast Exception in Java

ClassCast Exception in [Java](https://www.javatpoint.com/java-tutorial)

is one of the unchecked exceptions that occur when we try to convert one class type object into another class type.

ClassCast Exception is thrown when we try to cast an object of the parent class to the child class object. However, it can also be thrown when we try to convert the objects of two individual classes that don't have any relationship between them.

# Difference between throw and throws in Java

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception Using throw keyword, we **can only propagate unchecked exception** i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |  |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | **throw is used within the method**. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to **throw only one exception at a time i.e. we** cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException |

**With the help of throw we can generate new exception while with the help of throws keyword we can delegate exception from one method to other.**

**Que. What do you know about finally block?**

**Java finally block** is a block used to execute important code such as closing the connection, etc. It is used to release local resources i.e. to cleanup the code

Java finally block is always executed by JVM whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

Finally block must appear after all try catch block

**Finally block do not call by JVM if we call system.exiy(0) inside try and catch block.**

**Que What do you know about try with resources?**

**The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource**.

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

static String readFirstLineFromFile(String path) throws IOException {

**try (BufferedReader br =**

**new BufferedReader(new FileReader(path)))** {

return br.readLine();

}

}

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class BufferedReader, in Java SE 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine throwing an IOException).

# Que Difference between final, finally and finalize

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **final** | **finally** | **finalize** |
| 1. | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| 2. | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| 3. | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| 4. | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is dest |

# Que Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

**Que Java Exception Chaining**

**Process of handling one exception by throwing another exception**

**For instance, consider a method which throws an ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw the ArithmeticException to the caller. The caller would not know about the actual cause of an Exception. Chained Exception is used in such situations.

***Que. Suppressed Exception in Java***

A suppressed exception is an exception that is thrown but somehow ignored. A common scenario for this in Java is when the *finally* block throws an exception. Any exception originally thrown in the *try* block is then suppressed.

Starting with Java 7, we can now use two methods on the [*Throwable*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html) class to handle our suppressed exceptions: *addSuppressed* and *getSuppressed*.

## Q What is User Defined Exception in Java?

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 exception class of java.lang.exception

* 1. Because the existing exceptions may not be suitable for your situation. By creating your own exception you can make it easier to understand why the exception is thrown and it will stand out better than if you were using an existing exception.
  2. You want to pass along extra information such as error codes. For example, if you are a database vendor, you can add extra methods to include your internal error codes.
  3. 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.
* **public** **class** WrongFileNameException **extends** Exception {
* **public** WrongFileNameException(String errorMessage) {
* **super**(errorMessage);
* }
* }

**Que How can we pass argument to the method by reference?**

Java is always a *pass by value*; but, there are a few ways to achieve *pass by reference*:

1. Making a public member variable in a class
2. Return a value and update it
3. Create a single element array

in java there is no pass by reference .but we can simulate it by using array. we can make array and pass it to method as refernce

eg q

class program{

void swap(int []arr){

int temp=arr[0];

int arr[0]=arr[1];

arr[1]=temp;

}

public static void main(String args[]){

int a=10;

int b=20;

int []arr={a,b};

Program p = new program();

p.swap(arr){

a=arr[0];

b= arr1[1];

System.out.printlin(a+" "+b);

}

}

}

**Que Difference between Abstract class and interface.**

|  |  |
| --- | --- |
| **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** 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. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

**If state is involved used abstract class and if not use interface**

**If is – a relationship use--🡪 abstract and if not use interface**

**With the help of abs class we can group releted items together**

**With the help of interface we can declare non related items together**

**Functional Interface:- Interface which includes single abstract methods is called functional interface/ Single Abstract Method**

**Que Interface in Java**

Interface is a standard or specification. If we want to define specification for the subclasses then we should define interface. A **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve*[*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body.We can write method body by defining it default method. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**What is inside interface?**

* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have **default and static methods** in an interface.
* Since Java 9, we can have **private methods** in an interface.
* interfaces can have abstract methods and variables
* **interface** printable{
* **void** print();
* }
* **class** A6 **implements** printable{
* **public** **void** print(){System.out.println("Hello");}
* **public** **static** **void** main(String args[]){
* A6 obj = **new** A6();
* obj.print();
* }
* }

**Que Main Mehhod is Allowed in interface ?**

Since Java 8, static methods are allowed in interfaces.

main() is a static method.

Hence, main() is allowed in interfaces.

We don't **need** this, since it wasn't allowed before, and yet we survived. But since static methods, by definition, are not bound to an instance of a class, but to the class itself, it makes sense to allow them in interfaces. It allows defining utility methods related to an interface (like the ones found in Collections, for example), in the interface itself, rather than a separate class).

**Que What is difference between default method and abstract method?**

Default methods are to **add external functionality** to existing classes without changing their state.

And abstract classes are a normal type of inheritance, they are **normal classes** which are intended to be extended.

# Que. Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

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

Natural Ordering—as per owner Custom ordering---acc to client

The implementation logic is built into Implemented outside the class

The class as top level class, inner or lambda

**4.What is Marker Interface In java ?**

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

Marker interface is used as a tag that inform the Java compiler by a message so that it can add some special behaviour to the class implementing it. Java marker interface are useful if we have information about the class and that information never changes, in such cases, we use marker interface to represent the same. Implementing an empty interface tells the compiler to do some operations.

It is used to logically divide the code and a good way to categorize code. It is more useful for developing API and in frameworks like Spring.

**Difference between Annotation Vs Marker Interface :**Everything that can be done with a marker or tag interfaces in earlier versions of Java can now be done with annotations at runtime using reflection. One of the common problems with the marker or tag interfaces like Serializable, Cloneable, etc. is that when a class implements them, all of its subclasses inherit them as well whether you want them to or not. You cannot force your subclasses to un-implement an interface. Annotations can have parameters of various kinds, and they're much more flexible than the marker interfaces. This makes tag or marker interfaces obsolete, except for situations in which empty or tag interfaces can be checked at compile-time using the type-system in the compiler whereas annotation check has to be done at runtime

**Que What are the advantages of Collection Framework?**

Library of reusable data structure classes that is used to develop java application is called collection framework. • Main purpose of collection framework is to manage data in RAM efficiently

## Benefits of the Java Collections Framework

* 1. **Reduces programming effort**
  2. **Increases program speed and quality:**
  3. **Allows interoperability among unrelated APIs:**
  4. **Reduces effort to learn and to use new APIs**
* Reduced effort for code maintenance by using collection classes shipped with JDK.
* Reusability and Interoperability

**Collection:** Collection is a [interface](https://www.geeksforgeeks.org/interfaces-in-java/) present in java.util.package. It is used to represent a group of individual objects as a single unit. It is similar to the container in the [C++](https://www.geeksforgeeks.org/c-plus-plus/) language. The collection is considered as the root interface of the collection framework. It provides several classes and interfaces to represent a group of individual objects as a single unit.

The [List](https://www.geeksforgeeks.org/list-interface-java-examples/), [Set](https://www.geeksforgeeks.org/set-in-java/), and [Queue](https://www.geeksforgeeks.org/queue-interface-java/) are the main sub-interfaces of the collection interface. The map interface is also part of the java collection framework, but it doesn’t inherit the collection of the interface. The**add()**,**remove()**, **clear()**, **size()**, and **contains()** are the important methods of the Collection interface.

**Declaration:**

public interface Collection<E> extends Iterable<E>

**Type Parameters:** E - the type of elements returned by this iterator

**Collections:** Collections is a utility class present in java.util.package. It defines several utility methods like sorting and searching which is used to operate on collection. It has all static methods. These methods provide much-needed convenience to developers, allowing them to effectively work with [Collection Framework](https://www.geeksforgeeks.org/collections-in-java-2/). For example, It has a method *sort()* to sort the collection elements according to default sorting order, and it has a method *min()*, and *max()* to find the minimum and maximum value respectively in the collection elements.

Que Difference between collection and collections

| **Collection** | **Collections** |
| --- | --- |
| It is an interface. | It is a utility class. |
| It is used to represent a group of individual objects as a single unit. | It defines several utility methods that a are used to operate on collection. |
| The Collection is an interface that contains a static method since java8. The Interface can also contain abstract and default methods. | It contains only static methods. |

#### What is the benefit of Generics in Collections Framework?

#### In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

#### Type Safety:

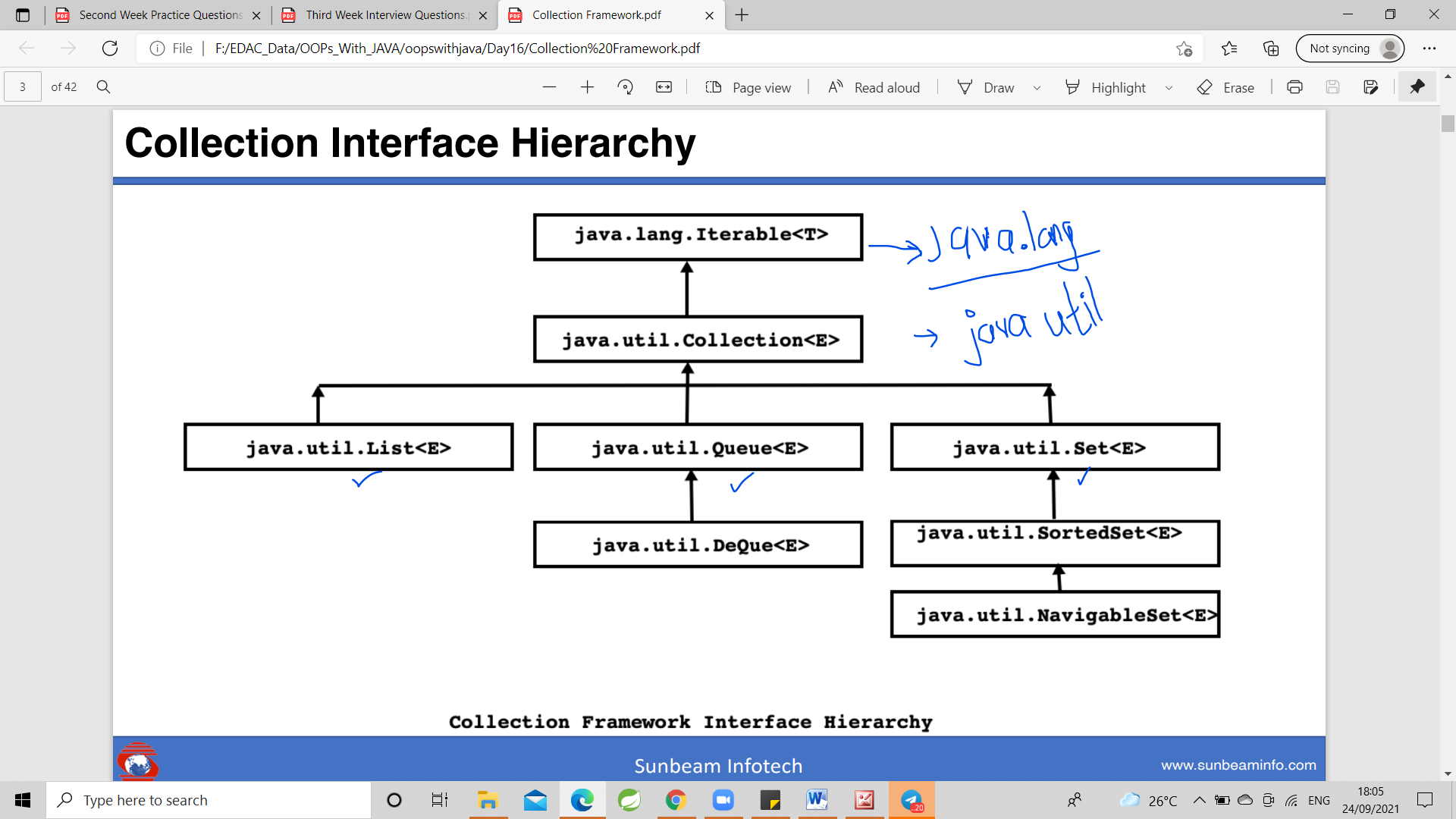
#### 1.Generic gives stronger type checking at compile time

#### 2.No need of explicit type casting

#### 3. Help us to implement generic data structure and algorithm

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error.

This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don’t need to use casting and **instanceof** operator. It also adds up to runtime benefit because the bytecode instructions that do type checking are not generated.



**Que Ways to create object :-**

Using new keyword

Using new instance method of constructor class

Using new instance method of class

Using clone method

By using deserializable

**5. Process of sorting list collection in java ?**

We can use the following methods to sort the list:

* Using stream.sorted() method
* Using Comparator.reverseOrder() method
* Using Comparator.naturalOrder() method
* Using Collections.reverseOrder() method
* Using Collections.sort() method : String class implements comparable interface

sorted() method

Stream interface provides a sorted() method to sort a list. It is defined in Stream interface which is present in **java.util package**. It returns a stream sorted according to the natural order. If the elements are not comparable, it throws java.lang.ClassCastException.

**6. how to copy one collection to other ?**

The **copy()** method of **java.util.Collections** class is used to copy all of the elements from one list into another.

After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

This method runs in linear time.

**Syntax:**

public static void copy(List dest, List src)

**7. Which are synchronized collection?**

Stack,vector,Hashtable and property

The synchronizedCollection**()** method of java. util. Collections class is used to return a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that all access to the backing collection is accomplished through the returned collection.

**Que Java Iterable**

The Java Iterable interface of java.lang package represents a collection of objects which is iterable - meaning which can be iterated. This means, that a class that implements the Java Iterable interface can have its elements iterated

3 Methods: iterator, splititerator, forEach

**8. diff between iterator and enumeration?**

[**Iterator**](https://www.geeksforgeeks.org/iterators-in-java/)**: It is an interface of java.util.** 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 an improved version of Enumeration with the additional functionality of remove-ability of an 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.

[**Enumeration**](https://www.geeksforgeeks.org/enum-in-java/)**:**Enumeration (or enum) is a user-defined data type. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain. In Java (from 1.5), enums are represented using the enum data type. Java enums are more powerful than C/C++ enums. In Java, we can also add variables, methods and constructors to it. The main objective of the enum is to define our own data types(Enumerated Data Types).

**Que Difference between Iterator and Enumeration:**

The functionality of Enumeration and the Iterator are same. Using Enumeration you can only traverse and fetch the objects, where as using Iterator we can also add and remove the objects. So Iterator can be useful if you want   
to manipulate the list and Enumeration is for read-only access. 

| Iterator | Enumeration |
| --- | --- |
| Iterator is a universal cursor as it is applicable for all the collection classes. | Enumeration is not a universal cursor as it applies only to legacy classes.We cannot traverse arraylist |
| Iterator has the remove() method. | Enumeration does not have the remove() method. |
| Iterator can do modifications (e.g using remove() method it removes the element from the Collection during traversal). | Enumeration interface acts as a read only interface, one can not do any modifications to Collection while traversing the elements of the Collection. |
| Iterator is not a legacy interface. Iterator can be used for the traversal of HashMap, LinkedList, ArrayList, HashSet, TreeMap, TreeSet . | Enumeration is a legacy interface which is used for traversing vector, hastable , dictionary  Method :- hasMoreElement(), nextElement() |

Method :- hasNext(), next(), remove

forEachRemaining()

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator<String> it = cars.iterator();

// Print the first item

System.out.println(it.next());

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

/ Java program to demonstrate

// enumeration() method

// for String value

import java.util.\*;

public class GFG1 {

    public static void main(String[] argv) throws Exception

    {

        try {

            // creating object of List<String>

            List<String> arrlist = new ArrayList<String>();

            // Adding element to srclst

            arrlist.add("Ram");

            arrlist.add("Gopal");

            arrlist.add("Verma");

            // Print the list

            System.out.println("List: " + arrlist);

            // creating object of type Enumeration<String>

            Enumeration<String> e = Collections.enumeration(arrlist);

            // Print the Enumeration

            System.out.println("\nEnumeration over list: ");

            // print the enumeration

            while (e.hasMoreElements())

                System.out.println("Value is: " + e.nextElement());

        }

        catch (IllegalArgumentException e) {

            System.out.println("Exception thrown : " + e);

        }

        catch (NoSuchElementException e) {

            System.out.println("Exception thrown : " + e);

        }

    }

}

# Legacy Class in Java

In the past decade, the **Collection** framework didn't include in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. After adding the [Collection framework](https://www.javatpoint.com/collections-in-java) in **JSE 1.2**, for supporting the collections framework, these classes were re-engineered. So, classes and interfaces that formed the collections framework in the older version of [Java](https://www.javatpoint.com/java-tutorial) are known as **Legacy classes**. For supporting generic in JDK5, these classes were re-engineered.

All the legacy classes are synchronized. The **java.util** package defines the following **legacy** classes:

1. HashTable
2. Stack
3. Dictionary
4. Properties
5. Vector

**Q. Diff between iterator and splititerator**

# Difference Between Iterator and Spliterator in Java

The Java **Iterator** interface represents an object capable of iterating through a collection of Java objects, one object at a time. The Iterator interface is one of the oldest mechanisms in Java for iterating collections of objects

***Moreover, an iterator differs from the enumerations in two ways:***

**1.** Iterator permits the caller to remove the given elements from the specified collection during the iteration of the elements.

**2.**Method names have been enhanced.

Like **Iterator** and **ListIterator**, [**Spliterator**](https://www.geeksforgeeks.org/java-util-interface-spliterator-java8/)is a Java Iterator, which is used to iterate elements one-by-one from a List implemented object.

the main functionalities of Spliterator are:

* Splitting the source data
* Processing the source data

The Interface Spliterator is included in JDK 8 for taking the advantages of parallelism in addition to sequential traversal. It is designed as a parallel analogue of an iterator

| ***Iterator*** | ***Spliterator*** |
| --- | --- |
| Introduced in Java 1.2 | Introduced in Java 1.8 |
| Iterator only iterates elements individually | Spliterator traverse elements individually as well as in bulk |
| It is an iterator for whole collection API | It is an iterator for both Collection and [Stream](https://www.geeksforgeeks.org/stream-in-java/#:~:text=A%20stream%20is%20a%20sequence,Arrays%20or%20I%2FO%20channels.) API, except [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) implementation classes |
| It uses external iteration | It uses internal iteration. |
| It is a Universal iterator | It is Not a Universal iterator |
| It does not support parallel programming | It supports parallel programming by splitting the given element set so that each set can be processed individually. |

**Parallel Programming :**

Many elements processed at a time

**ListIterator :-**

• It is sub interface of Iterator interface

• It is used to traverse only List Collection in bidirectional.

• During traversing, we can add, set as well as remove element from collection.

• It is introduced in jdk 1.2

• Methods of ListIterator 1. boolean hasNext() 2. E next() 3. boolean hasPrevious() 4. E previous() 5. void add(E e) 6. void set(E e) 7. void remove(

**Que Difference between Array and ArrayList**

Array is a fixed length data structure whereas ArrayList is a variable length Collection class. We cannot change length of array once created in Java but ArrayList can be changed.

We cannot store primitives in ArrayList, it can only store objects. But array can contain both primitives and objects in Java. Since Java 5, primitives are automatically converted in objects which is known as auto-boxing

| **Base** | **Array** | **ArrayList** |
| --- | --- | --- |
| Dimensionality | It can be single-dimensional or multidimensional | It can only be single-dimensional |
| Traversing Elements | For and for each generally is used for iterating over arrays | Here iterator is used to traverse riverArrayList |
| Length | length keyword can give the total size of the array. | size() method is used to compute the size of ArrayList. |
| Size | It is static and of fixed length | It is dynamic and can be increased or decreased in size when required. |
| Speed | It is faster as above we see it of fixed size | It is relatively slower because of its dynamic nature |
| Primitive Datatype Storage | Primitive data types can be stored directly unlikely objects | Primitive data types are not directly added unlikely arrays, they are added indirectly with help of autoboxing and unboxing |
| Generics | They can not be added here hence type unsafe | They can be added here hence makingArrayList type-safe. |
| Adding Elements | Assignment operator only serves the purpose | Here a special method is used known as add() method |

**Q. Diff Between ArrayList andLinkedList**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an 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 a doubly linked list, so no bit shifting is required in memory. |
| 3) An 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. |

**Que. Diff Between List and Set**

The [**List interface**](https://www.geeksforgeeks.org/list-interface-java-examples/) allows storing the ordered collection. It is a child interface of [Collection](https://www.geeksforgeeks.org/collections-in-java-2/). It is an ordered collection of objects in which duplicate values are allowed to store. List preserves the insertion order, it allows positional access and insertion of elements.

**Declaration:**

public abstract interface List extends Collection

The[**set interface**](https://www.geeksforgeeks.org/set-in-java/) in the [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package and extends [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/) is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the maths set. This interface contains the methods inherited from the Collection interface and adds a feature that restricts to insert the duplicate elements.

| List | Set |
| --- | --- |
| 1. The List is an ordered sequence. | 1. The Set is an unordered sequence. |
| 2. List allows duplicate elements | 2. Set doesn’t allow duplicate elements. |
| 3. Elements by their position can be accessed. | 3. Position access to elements is not allowed. |
| 4. Multiple null elements can be stored. | 4. Null element can store only once. |
| 5. List implementations are ArrayList, LinkedList, Vector, Stack | 5. Set implementations are HashSet, LinkedHashSet. |

**Que Diff Between HashSet And TreeSet**

## HashSet

[**HashSet**](https://www.javatpoint.com/java-hashset) is a generic class of the Java collection framework. It extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a [**hash table**](https://www.javatpoint.com/java-hashtable) for storage. The hash table stores the information by using the **hashing** mechanism.

Hashing uses the informational content to determine a unique value which is known as **hash code**. It is used as the index in which data is stored that is associated with the key. The transformation of the key into hash code performed automatically. The benefit of hashing is that it allows the execution time of add, contain, remove, and size operation to remain constant even for large sets. Its time complexity for the operation search, insert, and delete is **O(1)**.

The HashSet class does not provide any additional methods. It uses the methods of its superclasses and interfaces. It is to be noted that it does not guarantee the order of its elements.

## TreeSet

[**TreeSet**](https://www.javatpoint.com/java-treeset) is a class of Java collection framework that extends AbstractSet and implements the Set, NavigableSet, and SortedSet interface. It creates a collection that uses a tree for storage.

TreeSet is a generic class of the Java collection framework. It implements the Set interface. It uses [**TreeMap**](https://www.javatpoint.com/java-treemap) internally to store the TreeSet elements. By default, it sorts the elements in natural order (ascending order). The order of sorting depends on the Comparator that we have parsed. If no Comparator is parsed, it sorts the elements in the natural order.

Its performance is slow in comparison to HashSet because TreeSet sorts the elements after each insertion and deletion operation.

It uses two methods **comaperTo()** or **compare()** to compare the elements. It is to be noted that the implementation of TreeSet is not synchronized. It means that it is not thread-safe. The implementation must be synchronized externally if multiple threads accessing a TreeSet concurrently and a thread try to modify the TreeSet.

It does not allow to store null elements. It throws **NullPointerException** if we try to insert a null element. It requires more memory than TreeSet because it also maintains the comparator to sort the elements.

Its time complexity for the operation search, insert, and delete is **O(log n)** which is much higher than HashSet. It uses a **self-balancing BST** (Red-Black Tree) to implement the TreeSet

|  |  |  |
| --- | --- | --- |
| **Parameters** | **HashSet** | **TreeSet** |
| **Ordering or Sorting** | It does not provide a guarantee to sort the data. | It provides a guarantee to sort the data. The sorting depends on the supplied Comparator. |
| **Null Objects** | In HashSet, **only an element** can be null. | It does not allow null elements. |
| **Comparison** | It uses **hashCode()** or **equals()** method for comparison. | It uses **compare()** or **compareTo()** method for comparison. |
| **Performance** | It is **faster** than TreeSet. | It is **slower** in comparison to HashSet. |
| **Implementation** | Internally it uses **HashMap** to store its elements. | Internally it uses **TreeMap** to store its elements. |
| **Data Structure** | HashSet is backed up by a hash table. | TreeSet is backed up by a Red-black Tree. |
| **Values Stored** | It allows only **heterogeneous** value. | It allows only **homogeneous** value. |

**Que . Diff between Hashtable and HashMap**

[HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/) and [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/) store key and value pairs in a hash table. When using a Hashtable or HashMap, we specify an object that is used as a key and the value that you want to be linked 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.

**Hashmap vs Hashtable**

* HashMap is non-synchronized. It is not thread-safe and can’t be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
* HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value.
* HashMap is generally preferred over HashTable if thread synchronization is not needed

*Geek now****you must be wondering why HashTable doesn’t allow null and HashMap***

*the****answer****is simple s in order 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.*

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

***Q. Hashing ,bucket ,working of hashmap***

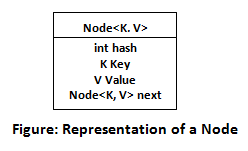
# Working of HashMap in Java

## What is Hashing

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

## What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



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

* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

**Que Difference between String and StringBuffer**

| **Sr. No.** | **Key** | **String** | **StringBuffer** |
| --- | --- | --- | --- |
| 1 | Basic | String is an immutable class of java.lang package and its object can’t be modified after it is created | String buffer is mutable classes java.lang which can be used to do operation on string object (thread-safe) |
| 2 | Methods | Methods are not synchronized (toLowerCase, substring, equals) | All methods are synchronized in this class.(append, delete, reverse) |
| 3 | Performance | It is fast | Multiple thread can’t  access at the same time therefore  it is slow |
| 4. | Memory Area | If a String is created using constructor or method then those strings will be stored in Heap Memory  as well as SringConstantPool | Heap Space |

String class is slower while performing concatenation operation

StringBuffer class is faster while performing concatenation operation.

String consumes more memory while stringbuffer consumes less

String length is fixed while length of string buffer object can be increased when required i.e dynamic in nature

String class overrides equals and hashcode method while stringbuffer dont

**Que Difference between StringBuffer and StringBuilder**

**Both are mutable**

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |
| 3) | StringBuffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34

**Que Difference between fail fast iterator and fail safe iterator**

Iterators in Java are part of the Java Collection framework. They are used to retrieve elements one by one. The Java

Collection supports two types of iterators; Fail Fast and Fail Safe. These iterators are very useful in exception handling.

***Fail fast iterator***

* The Fail fast iterator aborts the operation as soon it exposes failures and stops the entire operation The Fail Fast iterator throws a ConcurrentModificationException if a collection is modified while iterating over it.
* The Fail Fast iterator uses an original collection to traverse over the collection's elements.
* They are memory savers, don't require extra memory.
* The Fail Fast iterators returned by ArrayList, HashMap, Vector classes.

## *Fail Safe Iterator*

## The Fail Safe iterators are just opposite to Fail Fast iterators; unlike them, A fail-safe iterator does not throw any exceptions unless it can handle if the collection is modified during the iteration process. This can be done because they operate on the copy of the collection object instead of the original object. The structural changes performed on the original collection ignored by them and affect the copied collection, not the original collection. So, the original collection will be kept structurally unchanged.

|  |  |  |
| --- | --- | --- |
| **Base of Comparison** | **Fail Fast Iterator** | **Fail Safe Iterator** |
| **Exception** | It throws a ConcurrentModificationException in modifying the object during the iteration process. | It does not throw Exception. |
| **Clone Object** | No clone object is created during the iteration process. | A copy or clone object is created during the iteration process. |
| **Memory utilization** | It requires low memory during the process. | It requires more memory during the process. |
| **Modification** | It does not allow modification during iteration. | It allows modification during the iteration process. |
| **Performance** | It is fast. | It is slightly slower than Fail Fast. |
| **Examples** | HashMap, ArrayList, Vector, HashSet, etc | CopyOnWriteArrayList, ConcurrentHashMap, etc |

**Que Difference between Collection and Map**

**Collection**

Collection is interface declared in java.util package. • It is sub interface of Iterable interface. • It is root interface in collection framework interface hierarchy.

It consist of List, Queue and Set collections

• Default methods of Collection interface 1. default Stream stream() 2. default Stream parallelStream() 3. default boolean removeIf(Predicate filter)

collection objects are able to store data in the form of individual objects or in the form of single objects

**Map**

It is part of collection framework but it doesn't extend Collection interface. • This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

• HashMap, Hashtable, TreeMap etc are Map collection’s.

• Map collection stores data in key/value pair format.

• In map we can not insert duplicate keys but we can insert duplicate values. • It is introduced in jdk 1.2 • Map.Entry is nested interface of Map.

• Following are abstract methods of Map.Entry interface. 1. K getKey() 2. V getValue() 3. V setValue(V value

maps are able to store the data in the form of key value pairs, where both keys and values are objects

**Que Difference between ArrayList and Vector**

ArrayList and Vector both implements List interface and maintains insertion order.

The **Vector** class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections. It is found in the**java.util package** and implements the [List](https://www.geeksforgeeks.org/list-interface-java-examples/) interface, so we can use all the methods of List interface here.

* Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
* They are very similar to [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) but Vector is synchronized and has some legacy method that the collection framework does not contain.
* It also maintains an insertion order like an ArrayList but it is rarely used in a non-thread environment as it is **synchronized** and due to which it gives a poor performance in adding, searching, delete and update of its elements.
* The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException.**

However, there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
|  |  |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

**Que Synchronization**

<https://www.mygreatlearning.com/blog/synchronization-in-java/>

**Que Serialization and Deserialization**

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. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

**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.

**Que :- SerialVersionUID**  
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.  
It must be static, final and of type long.  
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data

**Que Java Program to count no of words in a file**

1. **import** java.io.BufferedReader;
2. **import** java.io.FileReader;
4. **public** **class** CountWordFile
5. {
6. **public** **static** **void** main(String[] args) **throws** Exception {
7. String line;
8. **int** count = 0;
10. //Opens a file in read mode
11. FileReader file = **new** FileReader("data.txt");
12. BufferedReader br = **new** BufferedReader(file);
14. //Gets each line till end of file is reached
15. **while**((line = br.readLine()) != **null**) {
16. //Splits each line into words
17. String words[] = line.split(" ");
18. //Counts each word
19. count = count + words.length;
20. }
22. System.out.println("Number of words present in given file: " + count);
23. br.close();
24. }
25. }

**Que program to return hashmap values in list in sorted form java**

1. **import** java.util.Collections;
2. **import** java.util.Comparator;
3. **import** java.util.HashMap;
4. **import** java.util.Iterator;
5. **import** java.util.LinkedHashMap;
6. **import** java.util.LinkedList;
7. **import** java.util.List;
8. **import** java.util.Map;
9. **import** java.util.Set;
10. **public** **class** SortHashMapValue
11. {
12. **public** **static** **void** main(String[] args)
13. {
14. //implementing HashMap
15. HashMap<Integer, String> hm = **new** HashMap<Integer, String>();
16. hm.put(6, "Tushar");
17. hm.put(12, "Ashu");
18. hm.put(5, "Zoya");
19. hm.put(78, "Yash");
20. hm.put(10, "Praveen");
21. hm.put(67, "Boby");
22. hm.put(1, "Ritesh");
23. System.out.println("Before Sorting:");
24. Set set = hm.entrySet();
25. Iterator iterator = set.iterator();
26. **while**(iterator.hasNext())
27. {
28. Map.Entry map = (Map.Entry)iterator.next();
29. System.out.println("Roll no:  "+map.getKey()+"     Name:   "+map.getValue());
30. }
31. Map<Integer, String> map = sortValues(hm);
32. System.out.println("\n");
33. System.out.println("After Sorting:");
34. Set set2 = map.entrySet();
35. Iterator iterator2 = set2.iterator();
36. **while**(iterator2.hasNext())
37. {
38. Map.Entry me2 = (Map.Entry)iterator2.next();
39. System.out.println("Roll no:  "+me2.getKey()+"     Name:   "+me2.getValue());
40. }
41. }
42. //method to sort values
43. **private** **static** HashMap sortValues(HashMap map)
44. {
45. List list = **new** LinkedList(map.entrySet());
46. //Custom Comparator
47. Collections.sort(list, **new** Comparator()
48. {
49. **public** **int** compare(Object o1, Object o2)
50. {
51. **return** ((Comparable) ((Map.Entry) (o1)).getValue()).compareTo(((Map.Entry) (o2)).getValue());
52. }
53. });
54. //copying the sorted list in HashMap to preserve the iteration order
55. HashMap sortedHashMap = **new** LinkedHashMap();
56. **for** (Iterator it = list.iterator(); it.hasNext();)
57. {
58. Map.Entry entry = (Map.Entry) it.next();
59. sortedHashMap.put(entry.getKey(), entry.getValue());
60. }
61. **return** sortedHashMap;
62. }
63. }

***1.What do you know about JDK, JVM and JRE?***

Ans**:- JDK**: JDK is a software development environment used for making applets and Java applications. The full form of JDK is Java Development Kit. JDK helps them to code and run Java programs.

We need to install jdk on client machine to run java application .

 It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

**JDK** = Java Development Tools + Java Docs+ rt.jar +JVM

**JRE** : JRE is a piece of a software which is designed to run other software. It contains the class libraries, loader class, and JVM. In simple terms, if you want to run Java program you need JRE

JRE = rt.jar + JVM

* JDK is platform dependent, JRE is also platform dependent, but JVM is platform independent.
* JDK contains tools for developing, debugging, etc. JRE contains class libraries and other supporting files, whereas software development tools are not included in JVM

**JVM** : JVM is an engine that provides a runtime environment to drive the Java Code or applications. It converts Java bytecode into machine language. In other words, an engine which manages execution of Java application. It is virtual machine it don’t have physical existence . we can not download it seperatly .

is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an [**i*nterpreter***](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)**.**

| **JVM** | **JIT** |
| --- | --- |
| JVM stands for Java Virtual Machine. | JIT stands for Just-in-time compilation. |
| JVM was introduced for managing system memory and providing a transportable execution environment for Java-based applications | JIT was invented to improve the performance of JVM after many years of its initial release. |
| JVM consists of many other components like stack area, heap area, etc. | JIT is one of the components of JVM. |
| JVM compiles complete byte code to machine code. | JIT compiles only the reusable byte code to machine code. |
| JVM provides platform independence. | JIT improves the performance of JVM. |

***2.Is JVM is platform dependent of independent? What about JRE?***

Ans: JDK, JRE and JVM are all platform dependent.

***3.What is bytecode?***

Ans: - Java bytecode is the instruction set for the Java Virtual Machine. As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we **achieve platform independence in java.**

## What happens at compile time?

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



## What happens at runtime?

At runtime, the following steps are performed:



**Classloader:** It is the subsystem of JVM that is used to load class files.HTML Tutorial

**Bytecode Verifier:** Checks the code fragments for illegal code that can violate access rights to objects.

 The following are some of the checks carried out:

* Variables are initialized before they are used.
* Method calls match the types of object references.
* Rules for accessing private data and methods are not violated.
* Local variable accesses fall within the runtime stack.
* The run time stack does not overflow.

**Interpreter:** Read bytecode stream then execute the instructions.

***4. Write a Hello World Program without giving semicolon?***

class HelloWorld {

    public static void main(String args[])

    {

        if (System.out.printf("Helllo World") == null) {

        }

    }

}

***5.What is meaning of System.out.Println?***

Ans: object of PrintStream class that is **System.out**.

Where **System** is the class name, it is declared as final.

The out is an instance of the System class and is of type PrintStream. Its access specifiers are public and final. It is an instance of java.io.PrintStream. When we call the member, a PrintStream class object creates internally.

As all instances of [PrintStream class](https://www.geeksforgeeks.org/java-io-printstream-class-java-set-1/) have a public method println(), hence we can invoke the same on out as well.

***6. What is method overloading?***

Ans: If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**. Increases the readability of the [program](https://www.javatpoint.com/java-programs).

***7.Which are the rules of method overloading?***

Ans:- Rule 1 : If we want to give same name to the method and if type of all the parameters are same then number of parameters passed to the method must be different.

Rule 2 : If we want to give same name to the method and if number of parameters are same then type of at least one parameter must be different.

Rule 3 : If we want to give same name to the method and if number of parameters are same then order of type of type of parameters must be different.

Rule 4 : Only on the basis of different return type, we can not give same name to the method.

***8.Can we overload main method?***

Ans: Yes.

You can change the method signature by changing the type of argument, a number of arguments, or order of arguments.

public static void main(String[] args) { System.out.println("Inside main(String[] args) method ...."); }  
  
public static void main(Integer[] args){ System.out.println("Inside main(Integer[] args) method ....");

***9.What do you know about wrapper class?***

Ans: - In java primitive types are not classes. For every primitive types java has defined a class called wrapper class. All wrapper classes are final. Declared in java.lang package.

For eg: Boolean, Byte, Character, Short, Integer, Float, Double, Long

***10.What is boxing and unboxing? Explain with examples.***

Ans: - Boxing : Conersion of Primitive values to non- primitive values

Unboxing : Conversion of Non primitive to primitive values

***Wrapper classes :***

wrapper class are used to convert primitive to object and object to primitive data types .

in java primitive data types are not classes but java has provided a class for every primitive data type which is called as wrapper class .

***Need of Wrapper Classes***

.They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

.The classes in java.util package handles only objects and hence wrapper classes help in this case also.

.Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

.An object is needed to support synchronization in multithreading.

**Autoboxing :**

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.

Example:

// Java program to demonstrate Autoboxing

import java.util.ArrayList;

class Autoboxing

{

public static void main(String[] args)

{

char ch = 'a';

// Autoboxing- primitive to Character object conversion

Character a = ch;

}

***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.

// Java program to demonstrate Unboxing

import java.util.ArrayList;

class Unboxing

{

public static void main(String[] args)

{

Character ch = 'a';

// unboxing - Character object to primitive conversion

char a = ch;

}

Que Difference between import and import static

***11.Explain class and instance by giving real time examples***.

Ans:- To represent data we need class. Class is a blueprint of object that contains variables and methods. It is logical representation of data. Class is non- primitive/ reference type in java

Instance or object represents physical entity. With the help of a new operator we may easily create object of class and memory is created in the HEAP and the object is called an instance of class.

If Engine is a class then components inside engine represents instance

If animal is a class then dog, buffalo is an instance.

***12. What is difference between primitive and non- primitive types?***

|  |  |
| --- | --- |
| **Primitive Type(Value Type)** | **Non Primitive Type(Reference Type)** |
| boolean, byte, char, short, int, float, double, long are primitive/value type | Interface, class, type variable and array are non primitive/reference type |
| Variable of value type contains value. | Variable of reference type contains reference |
| Variable of value type by default contains 0 value. | Variable of reference type by default contain null reference. |
| We can not create variable of value type using new operator. | It is mandatory to use new operator to create instance of reference type. |
| variable of value type get space on Java stack. | Instance of reference type get space on heap section. |
| We can not store null value inside variable of value type. | We can store null value inside variable reference type. |
| In case of copy, value gets copied. | In case of copy, reference gets copied. |

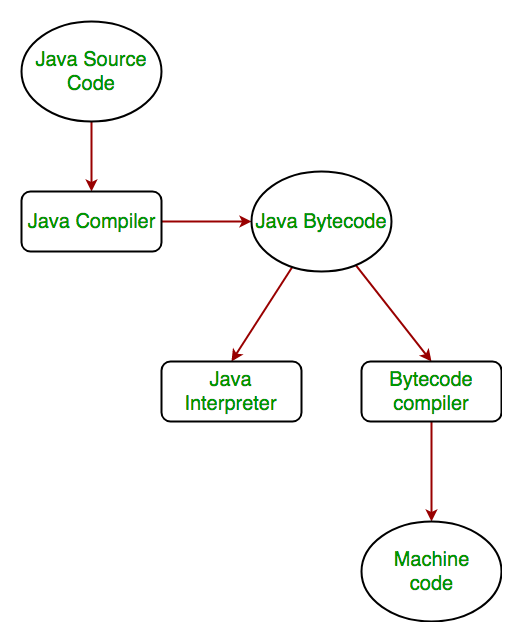
# How is Java platform independent?

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 07 Jan, 2020

The meaning of platform-independent is that the java compiled code(byte code) can run on all operating systems.  
A program is written in a language that is a human-readable language. It may contain words, phrases, etc which the machine does not understand. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines. Therefore, a compiler is a program that translates the source code for another program from a programming language into executable code.  
This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code.**

**Step by step Execution of Java Program:**

* Whenever, a program is written in JAVA, the javac compiles it.
* The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).
* The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
* And finally program runs to give the desired output.

.  
  
In case of C or C++ (language that are not platform independent), the compiler generates an .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS dependent and hence is not compatible with the other OS.

**Java is platform-independent but JVM is platform dependent**

In Java, the main point here is that the JVM depends on the operating system – so if you are running Mac OS X you will have a different JVM than if you are running Windows or some other operating system. This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will be given a list of JVMs corresponding to different operating systems, and you will obviously pick whichever JVM is targeted for the operating system that you are running. So we can conclude that JVM is platform-dependent and it is the reason why Java is able to become “Platform Independent”.  
**Important Points:**

* In the case of Java, **it is the magic of Bytecode that makes it platform independent**.
* This adds to an important feature in the JAVA language termed as **portability**. Every system has its own JVM which gets installed automatically when the jdk software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.
* An important point to be noted is that while **JAVA is platform-independent language, the JVM is platform-dependent.** Different JVM is designed for different OS and byte code is able to run on different OS.

**java** is object oriented programming  language but not pure object oriented because  supports Primitive datatype such as int, byte, long... Etc, to be used these are not object.

**Main features are:**

* Inheritance
* Encepculation
* Polymorphism
* Association
* Aggregation

**14 .What is Constructor Chaining?**

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.   
Constructor chaining can be done in two ways: 

* **Within same class**: It can be done using **this()** keyword for constructors in same class
* **From base class:**by using **super()** keyword to call constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub class constructor’s task is to call super class’s constructor first. This ensures that creation of sub class’s object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.  
**Why do we need constructor chaining ?**   
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

**Rules of constructor chaining :**

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

**Constructor :-**

In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

* 1. We can make constructor private. Public , protected or default . It helps to control object creation
  2. Constructore have no return types
  3. A Java constructor cannot be abstract, static, final, and synchronized
* **We cannot make constructor static because it will call before object creation as static members loads every time when class loads.**
* **Java constructor can not be final**
* One of the important property of java constructor is that it can not be [final](https://www.geeksforgeeks.org/final-keyword-java/). As we know, [constructors are not inherited in java](https://www.geeksforgeeks.org/constructors-not-inherited-java/). Therefore, constructors are not subject to hiding or [overriding](https://www.geeksforgeeks.org/overriding-in-java/#:~:text=In%20any%20object%2Doriented%20programming,super%2Dclasses%20or%20parent%20classes.&text=Method%20overriding%20is%20one%20of,java%20achieve%20Run%20Time%20Polymorphism.). When there is no chance of constructor overriding, there is no chance of modification also. When there is no chance of modification, then no sense of restricting modification there. We know that the final keyword restricts further modification. So a java constructor can not be final because it inherently it cannot be modified. Also, a java constructor is internally final. So again there is no need for final declaration further.
* **Java constructor can not be static**
* One of the important property of java constructor is that it can not be [static](https://www.geeksforgeeks.org/static-keyword-java/). We know static keyword belongs to a class rather than the object of a class. A constructor is called when an object of a class is created, so no use of the static constructor. Another thing is that if we will declare static constructor then we can not access/call the constructor from a subclass. Because we know static is allowed within a class but not by a subclass.

**15. What Do u Know about Static initializer block?**

Static init block is used to initialize static members of the class .

Static block in java is executed **every time when a class loads**..

**Why static method do not get this reference:?**

1. If we call, non static method on instance then method get this reference. 2. Static method is designed to call on class name. 3. Since static method is not designed to call on instance, it doesn't get this reference.

• Normally, you would put code to initialize an instance variable in a constructor. • There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods. • Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

• The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

…………………………………………………………………………………………………………

Instance intilizer block:

• Normally, you would put code to initialize an instance variable in a constructor. • There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods. • Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

• The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

**What is singletone Class?**

Singleton class :--It is an design pattern of type creational design pattern

A class from which, we can create single instance is called singleton class.

A method which hides instantiation from end user is called **factory method.**

class Singleton{ private Singleton() {

}

private static Singleton instance = null;

public static Singleton getInstance( ) {

if( instance == null )

instance = new Singleton();

return instance; }

}

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

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

Singleton s3 = Singleton.getInstance();

} }

Purpose of Singleton class

The primary purpose of Single class is to restrict the limit of the number of object creation to only one. This often ensures that there is access control to resources, for example, socket or database connection.

The memory space wastage does not occur with the use of singleton class because it restricts the instance creation. As the object creation will take place only once instead of creating it each time a new request is made.

We can use this single object repeatedly as per the requirements. This is the reason why the multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings and much more.

For example, there is a license with us, and we have only one database connection or suppose if our JDBC driver does not allow us to do multithreading, then Singleton class comes into the picture and makes sure that at a time, only a single connection or a single thread can access the connection.

How to design a singleton class?

To design a singleton class, we need to do the following things:

* **Firstly, declare the constructor of the Singleton class with the private keyword. We declare it as private so that no other classes can instantiate or make objects from it.**
* A private static variable of the same class that is the only instance of the class.
* Declare a static factory method with the return type as an object of this singleton class.

Difference between Normal class and Singleton class

We can distinguish a Singleton class from the usual classes with respect to the process of instantiating the object of the class. **To instantiate a normal class, we use a java constructor. On the other hand, to instantiate a singleton class, we use the getInstance() method.**

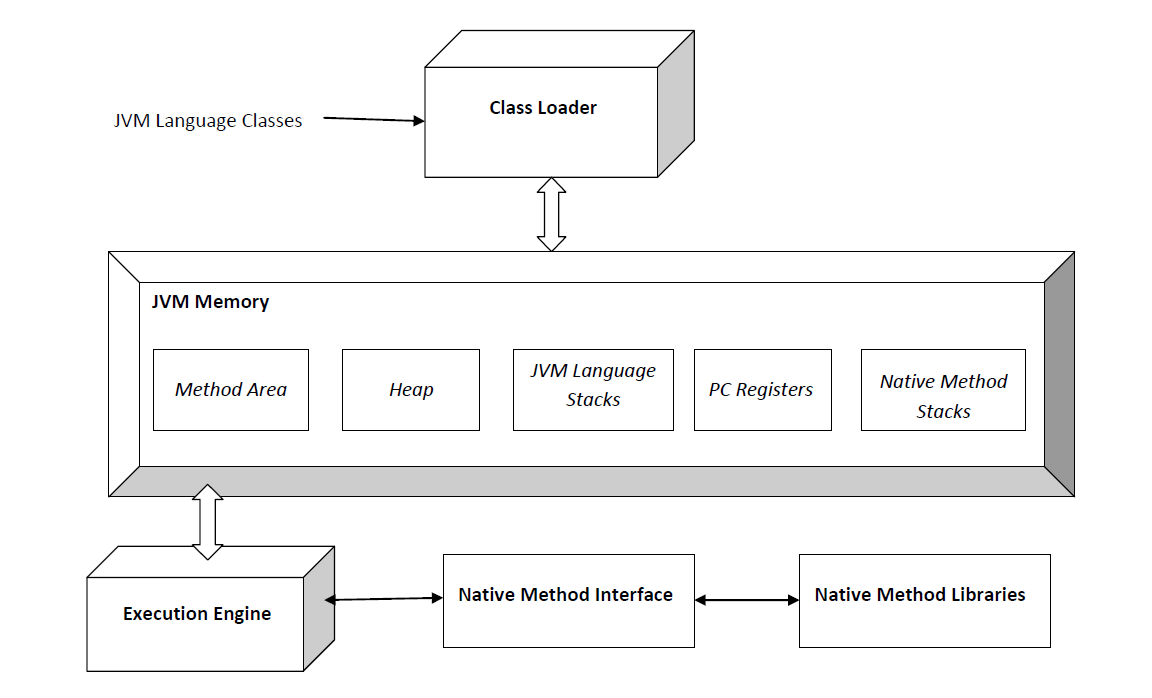
The other difference is that a normal class vanishes at the end of the lifecycle of the application while the singleton class does not destroy with the completion of an application.

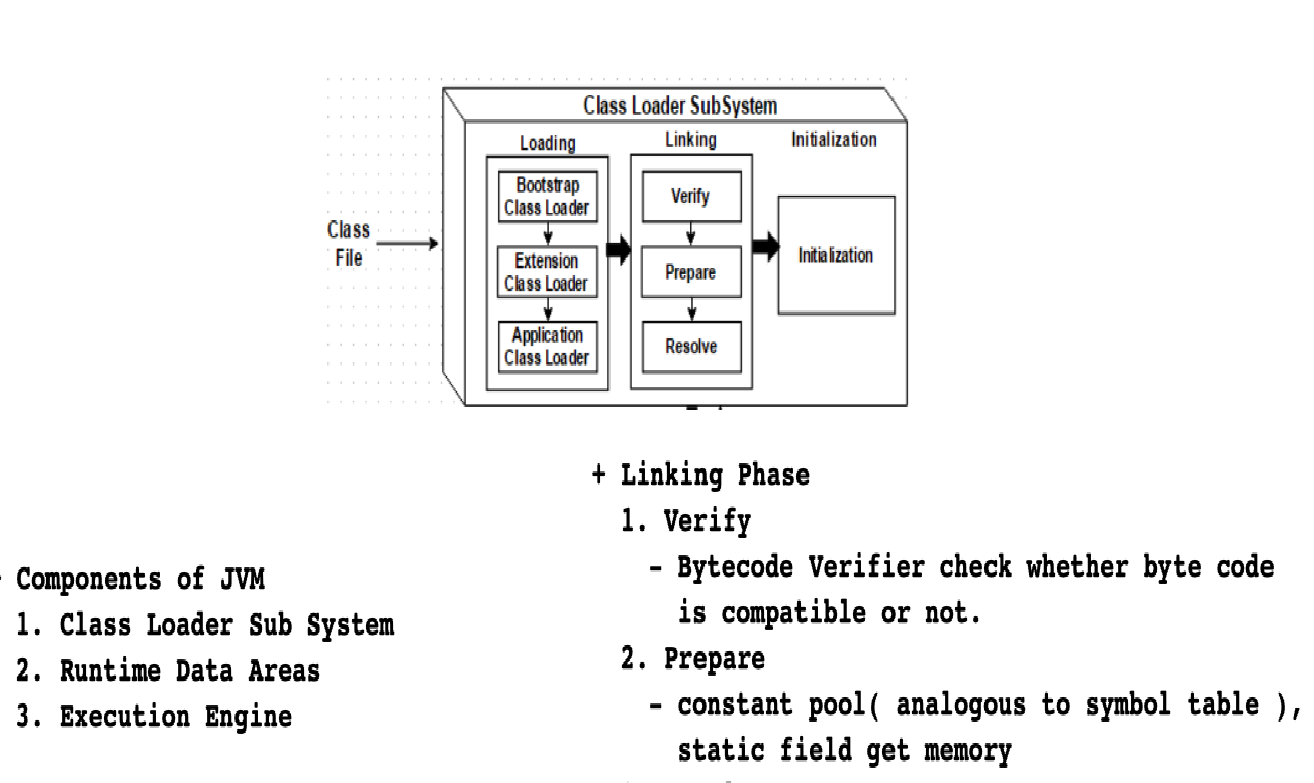
Forms of Singleton Design pattern

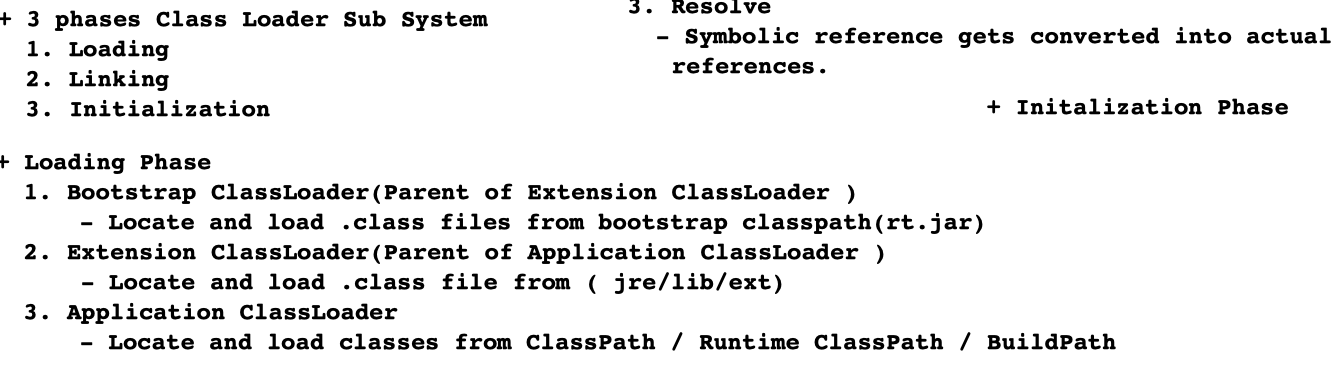
There are two forms of singleton design pattern, which are:

* **Early Instantiation:** The object creation takes place at the load time.
* **Lazy Instantiation:** The object creation is done according to the requirement.

**Que :- JVM Architecture :**







**JVM Memory**

1. **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
3. **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.

**Execution Engine**

Execution engine executes the “*.class”* (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

* *Interpreter*: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* *Just-In-Time Compiler(JIT)* : It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.
* *Garbage Collector*: It destroys un-referenced objects. For more on Garbage Collector, refer [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/).

**Java Native Interface (JNI) :**

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**Que What is diff between object and reference ?**

Reference : is used to refer or to point the object it do not contain any values and it is stored on stack area

Object : it stores the state or values and it get space in heap memory. We can have diff references pointing to the same object

A reference is an entity which provides a way to access object of its type. An object is an entity which provides a way to access the members of it's class or type.

--------------------------------------------------------------------------------------------------

**Que … this reference**

This is implicit ref variable available in every non static method class which is used to store ref of calling object.

It acts as a link between method and fields

**Que Diff between default and protected ?**

**Access Modifier in Java**

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

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

**Que. Advantages of JAVA :**

1. Simple 2. Object Oriented 3. Architecture Neutral 4. Portable 5. Robust 6. Multithreaded 7. Dynamic 8. Secure 9. High Performance 10.Distributed

**• Java is object architecture neutral programming language.**

o Java technology is designed to support applications that will be deployed into heterogeneous network environments.

In such environments, applications must be capable of executing on a variety of hardware architectures.

Within this variety of hardware platforms, applications must execute on the top of a variety of operating systems.

To accommodate the diversity of operating environments, the Java Compiler product generates bytecodes--an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms.

***Java is portable programming language.***

o Java technology takes portability a stage further by being strict in its definition of the basic language.

o Java technology puts a stake in the ground and specifies the sizes of its basic data types and the behavior of its arithmetic operators.

o Your programs are the same on every platform--there are no data type incompatibilities across hardware and software architectures

***Java is robust programming language.***

o The Java programming language is designed for creating highly reliable software.

It provides extensive compile-time checking, followed by a second level of runtime checking.

Language features guide programmers towards reliable programming habits.

o Java is robust because of following features: 1. Architecture Neutral.

Ø Java developer is free from developing H/W or OS specific coding. 2. Object orientation.

Ø Reusability reduces developer’s effort. 3. Automatic memory management.

Ø Developer need not to worry about memory leakage / program crashes. 4. Exception handling.

Ø Java compiler helps developer to provide try-catch block.

**Java is multithreaded programming language.**

o When we start execution of Java application then JVM starts execution of two threads hence every Java is considered as multithreaded.

1. **Main thread**

Ø It is user thread / non daemon thread.

Ø It is responsible for invoking main method.

Ø Its default priority is 5( Thread.NORM\_PRIORITY ).

2. **Garbage Collector / Finalizer**

Ø It is daemon thread / background thread.

Ø It is responsible for releasing / deallocating memory of unused objects.

Ø Its default priority is 8( Thread.NORM\_PRIORITY + 3 ).

**Java is high performance programming language.**

o The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment.

o The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.

o Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform.

o In general, users perceive that interactive applications respond quickly even though they're interpreted.

***Que Types of Herirchey ?***

• Types of hierarchy: 1. Has-a / Part-of => Association 2. Is-a / Kind-of => Inheritance / Generalization 3. Use-a => Dependency 4. Creates-a => Instantiation.

• If has-a relationship is exist between the types then we should use association.

• Example 1. Car has a engine 2. Room has a chair • Let us consider example of car and engine: 1. Car has a engine 2. Engine is part of Car.

• If object/instance is a part/component of another instance then it is called as association.

• To implement association, we should declare instance of a class as a field inside another class.

If "is-a" relationship is exist between the types then we should use inheritance.

• Inheritance is also called as generalization.

• Example 1. Manager is a employee 2. Book is a product 3. Triangle is a shape 4. SavingAccount is a account.

**Que what is path and classpath ?**

**PATH**

a) An OS platforms environment variable which is used by the operating system to find the executables/ java language tools

b) PATH is nothing but setting up an environment for operating system. Operating System will look in this PATH for executables.

c) Refers to the system

**CLASSPATH**

a) An environment variable which is used by the Java compiler to find the path, of classes i.e in J2EE we give the path of jar files.

b) Classpath is nothing but setting up the environment for Java. Java will use to find compiled classes.

c) Refers to the Developing Environment.

**Absolute Path** :- path from root directory

**Relative Path** :- path from current directory

**Que what do u know about package?**

To group functionally related items together (like classes, interfaces) we should use package

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql

**Advantage of Java Package**

To group functionally related items together (like classes, interfaces) we should use package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision/ ambiguity

4) To make types easier to find from the perspective of java classes

Package can contains : - sub package, class, interface, enum, exceptions, error, annotations , type

**Que Why we can not write multiple public classes in java?**

as per java standard the (filename i.e .java file) and public classname should be same .

**Que What are methods of object class ?**

Methods Of Object class :

**5 Non final Methods :**

1. public String toString();

2. public boolean equals(Object obj);

3. public native int hashCode( );

4. protected native Object clone( )throws CloneNotSupportedException

5. protected void finalize( void )throws Throwable

**6 Final Methods :**

6. public final native Class<?> getClass( );

7. public final void wait( )throws InterruptedException

8. public final native void wait( long timeout)throws InterruptedException

9. public final void wait( long timeout, int nanos)throws InterruptedException

10.public final native void notify( );

11.public final native void notifyAll( );

What is Wrapper Class?what is their use ?

in java primitive data types are not objects but java has provided class for every datatypes which are called as wrapper class .

Need of Wrapper Classes

They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

The classes in java.util package handles only objects and hence wrapper classes help in this case also.

Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

An object is needed to support synchronization in multithreading.

**Que Java 8 features ?**

Functional Interfaces and Lambda Expressions A functional interface is **an interface that contains only one abstract method**.

forEach() method in Iterable interface

Optional class,

default and static methods in Interfaces

Method references

Java Stream API for Bulk Data Operations on Collections

Java Date Time API

Collection API improvements

Concurrency API improvements

Java IO improvements

Nashorn JavaScript engine

Base64 Encode Decode

Miscellaneous Core API improvements

**Que Advantages and disadvantages of generics ?**

**Advantages of Generics**:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.

2. Type Safety: Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time).

Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of a string, the compiler allows it.

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example every time we retrieve data from ArrayList,

we have to typecast it.

Typecasting at every retrieval operation is a big headache.

4. Generics promotes code reusability.

5. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same, they are type safe too.

**5. Disadvantage of generics**

According to oracle documentation, the following points are the disadvantage of generics:

Cannot instantiate Generic types with primitive types.

Cannot create instances of type parameters.

Cannot declare static fields whose types are type parameters.

Cannot use casts or instanceof with parameterized types.

Cannot create arrays of parameterized types.

Cannot create, catch, or throw Objects of parameterized types.

**Que Diff between start() and run method ()**

**start()**

Creates a new thread and the run() method is executed on the newly created thread.

Can’t be invoked more than one time otherwise throws java.lang.IllegalStateException

Defined in java.lang.Thread class.

**run()**

No new thread is created and the run() method is executed on the calling thread itself.

Multiple invocation is possible

Defined in java.lang.Runnable interface and must be overriden in the implementing class.

<https://www.geeksforgeeks.org/difference-between-thread-start-and-thread-run-in-java/>

| start() | run() |
| --- | --- |
| Creates a new thread and the run() method is executed on the newly created thread. | No new thread is created and the run() method is executed on the calling thread itself. |
| Can’t be invoked more than one time otherwise throws *java.lang.IllegalStateException* | Multiple invocation is possible |
| Defined in *java.lang.Thread* class. | Defined in *java.lang.Runnable* interface and must be overriden in the implementing class. |

**Que :When we should create thread using by extending runnable and by implementating runnable?**

we can define a thread in the following two ways:

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

In the first approach, Our class always extends Thread class. There is no chance of extending any other class because java support single inheritane. Hence we are missing Inheritance benefits. In the second approach, while implementing Runnable interface we can extends any other class. Hence we are able to use the benefits of Inheritance.

Because of the above reasons, implementing Runnable interface approach is recommended than extending Thread class.

The significant differences between extending Thread class and implementing Runnable interface:

When we extend Thread class, we can’t extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.

**When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same thread to multiple objects.**

or

**Difference between Runnable vs Thread**

There has been a good amount of debate on which is better way. Well, I also tried to find out and below is my learning.

Implementing Runnable is the preferred way to do it. Here, you’re not really specializing or modifying the thread’s behavior. You’re just giving the thread something to run. That means composition is the better way to go.

Java only supports single inheritance, so you can only extend one class.

Instantiating an interface gives a cleaner separation between your code and the implementation of threads.

Implementing Runnable makes your class more flexible.

If you extend Thread then the action you’re doing is always going to be in a thread. However, if you implement Runnable it doesn’t have to be.

You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.

**Que Lambda Expression in java 8**

# Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code**. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.**

lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* 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.

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample3{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

**Que significance of map ,filter , reduce and flatmap?**

**map()** can be used where we have to map the elements of a particular collection to a certain function, and then we need to return the stream which contains the updated results.

The function passed to map() operation returns a single value for a single input.

One-to-one mapping occurs in map().

Only perform the mapping.

Produce a stream of value.

map() is used only for transformation.

**flatMap()** can be used where we have to flatten or transform out the string, as we cannot flatten our string using map().

The function you pass to flatmap() operation returns an arbitrary number of values as the output.

One too many mapping occurs in flatMap().

Perform mapping as well as flattening.

Produce a stream of stream value.

flatMap() is used both for transformation and mapping.

## Stream.map() vs Stream.flatMap() in Java 8

In short, here are the key **difference between map() vs flatMap() in Java 8:**

* The function you pass to the map() operation returns a single value.
* The function you pass to flatMap() operation returns a Stream of value.
* flatMap() is a combination of map and flat operation.
* map() is used for transformation only, but [flatMap()](http://www.java67.com/2016/03/how-to-use-flatmap-in-java-8-stream.html)is used for both transformation and flattening.

Read more: <https://javarevisited.blogspot.com/2016/03/difference-between-map-and-flatmap-in-java8.html#ixzz77XnoMuvQ>

**filter()**

filter() is a intermediate Stream operation.

It returns a Stream consisting of the elements of the given stream that match the given predicate.

The filter() argument should be stateless predicate which is applied to each element in the stream to determine if it should be included or not.

Predicate is a functional interface. So, we can also pass lambda expression also.

It returns a new Stream so we can use other operations applicable to any stream.

ust like the map method, the filter method expects a lambda expression as its argument. However, the lambda expression passed to it must always return a boolean value, which determines whether or not the processed element should belong to the resulting Stream object

eg.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main

{

public static void main(String[] args)

{

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> evenNumbers = list.stream()

.filter(n -> n % 2 == 0)

.map(n -> n \* n)

.collect(Collectors.toList());

System.out.println(evenNumbers);

}

}

## Reduction Operations

A reduction operation is one which allows you to compute a result using all the elements present in a stream. Reduction operations are also called terminal operations because they are always present at the end of a chain of Stream methods. We’ve already been using a reduction method in our previous examples: the toArray method. It’s a terminal operation because it converts a Stream object into an array.

Java 8 includes several reduction methods, such as sum, average and count, which allow to perform arithmetic operations on Stream objects and get numbers as results. For example, if you want to find the sum of an array of integers, you can use the following code:

int myArray[] = { 1, 5, 8 };

int sum = Arrays.stream(myArray).sum();

If you want to perform more complex reduction operations, however, you must use the reduce method. Unlike the map and filter methods, the reduce method expects two arguments: an identity element, and a lambda expression. You can think of the identity element as an element which does not alter the result of the reduction operation. For example, if you are trying to find the product of all the elements in a stream of numbers, the identity element would be the number 1.

The lambda expression you pass to the reduce method must be capable of handling two inputs: a partial result of the reduction operation, and the current element of the stream. If you are wondering what a partial result is, it’s the result obtained after processing all the elements of the stream that came before the current element.

The following is a sample code snippet which uses the reduce method to concatenate all the elements in an array of String objects:

String[] myArray = { "this", "is", "a", "sentence" };

String result = Arrays.stream(myArray)

.reduce("", (a,b) -> a + b);

**Que JDBC**

If we want to access and process data stored in relational database

management system using Java programming language then we should use

**JDBC.**

• It minimizes database vendor dependency in the Java application

Steps to connect Java Application To Database

1. Include database (MySQL) connector into runtime classpath/build path.

2. Import java.sql package.

3. Load and register Driver.

4. Establish connection using users credential(username and password).

5. Create Statement/PreparedStatement/CallableStatement to execute query.

6. Execute query and process result.

7. Close resources.

//Step 1 : include .jar file in classpath/buildpath

package org.sunbeam.test;

//Step 2 : Import sql package;

import java.sql.\*;

public class Program {

    public static final String DRIVER\_CLASS = "com.mysql.cj.jdbc.Driver";

    public static final String URL = "jdbc:mysql://localhost:3306/sunbeam\_db";

    public static final String USERNAME = "sunbeam";

    public static final String PASSWORD = "sunbeam";

    public static void main(String[] args) {

        Connection connection = null;

        Statement statement = null;

        try {

**//Step 3 : Load and register driver**

            Class.forName(DRIVER\_CLASS);

**//Step 4 : Establish connection using users credential**

            connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);

**//Step 5 : Create Statement**

            statement = connection.createStatement();

**//Step 6 : Prepare and execute query**

            String sql = "SELECT \* FROM books";

            //DML : INSERT, UPDATE, DELETE => int executeUpdate( );

            //DQL : SELECT => ResultSet executeQuery( )

            ResultSet rs = statement.executeQuery(sql);

            while( rs.next()) {

                int bookId = rs.getInt(1);

                String subjectName = rs.getString(2);

                String bookName = rs.getString(3);

                String authorName = rs.getString(4);

                float price = rs.getFloat(5);

                System.out.printf("%-5d%-10s%-65s%-40s%-10.2f\n", bookId, subjectName, bookName, authorName, price );

            }

            rs.close();

        }catch( Exception ex ) {

            ex.printStackTrace();

        }finally {

            try {

                //Step 7 : Close resources

                statement.close();

                connection.close();

            } catch (SQLException e) {

                e.printStackTrace();

            }

        }

    }

}

***Driver***

1. Driver is interface declared in java.sql package.

2. Every driver class must implement this interface.

3. Driver implementation handles the communication with database.

4. When a Driver class is loaded, it should create an instance of itself

and register it with the DriverManager. This means that a user can

load and register a driver by calling:

Class.forName(”com.mysql.cj.JDBC.Driver")

# DriverManager class

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

# Connection interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

# Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

# ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

#### By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE\_SCROLL\_INSENSITIVE or TYPE\_SCROLL\_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

A ResultSet object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method next is called. When a ResultSet object is first created, the cursor is positioned before the first row, so the first call to the next method puts the cursor on the first row, making it the current row. ResultSet rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method next.

# PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

1. String sql="insert into emp values(?,?,?)";

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

### Why use PreparedStatement?

**Improves performance**: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

# Java CallableStatement Interface

CallableStatement interface is used to call the **stored procedures and functions**.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.