Java Interview Questions based on Core Java Topics

Notion Link: https://kdjs.notion.site/Java-Interview-Questions-based-on-Core-Java-Topics-061bb57e0bfc4510be2a49c6d1abaf51

Core Java Topics

1. Garbage collection
2. Static , nonstatic, local variable
3. Methods - void return value n return
4. Data types
5. Var type
6. Constructor n types
7. New keyword
8. This keyword
9. Inheritance
10. Polymorphism - overriding and overloading
11. Method overloading
12. Access modifiers
13. Encapsulation
14. Type casting
15. Class casting
16. RTP
17. Interface || Functional Interface, Lock Interface, Runnable Interface **Imp**
18. Marker interface || Use of Marker interface **Imp**
19. Abstract keyword
20. Final keyword
21. Features of java in version 8
22. Abstraction || Two ways to achieve abstraction in Java i.e., Abstract class and interface
23. Interface
24. Difference b/w abstract and interface || Which is better interface or abstract class in Java? Imp
25. Exceptions
26. Exception hierarchy
27. Loops
28. Array
29. File handling
30. IIB

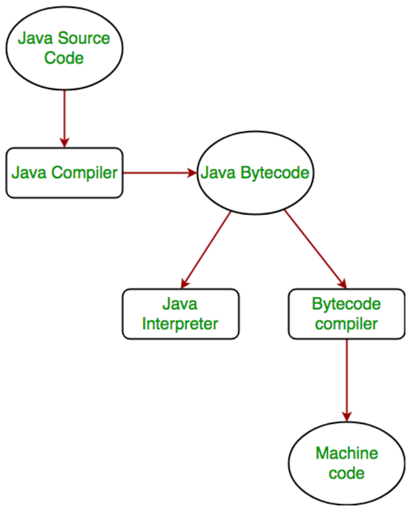
31. SIB
32. Super
33. Serialisation and deserialization
34. Transient keyword
35. Finally, practical example
36. Exception/ throwable
37. String
38. Thread || Which is better runnable interface or thread class
39. Wrapper class
40. Finalize
41. Throws()
42. Throw
43. Throwable
44. Regular expression in java
45. Tokenizer
46. Cloning
47. Collections
48. Generics

Q. How is Java platform independent?

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.**



Questions

1. Garbage Collection:

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

Simple Example of garbage collection in java

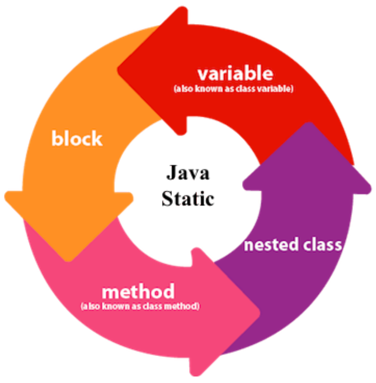
```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

2. Static, Non static/local variable

Top 14 Java Interview Questions on Static keyword

Static is a Non-Access Modifier. Static can be applied to variable, method, nested class and initialization blocks (static block). A Static variable gets memory allocated only once during the time of class loading. All the instance of the class share the same copy of the variable, a static variable can be accessed directly by calling " >.

https://www.javainterviewpoint.com/top-10-java-interview-questions-on-static-keyword/



Static Variable:

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

```
// Java program to demonstrate execution
// of static blocks and variables
class Test {
    // static variable
    static int a = m1();

    // static block
    static
    {
        System.out.println("Inside static block");
    }

    // static method
    static int m1()
    {
        System.out.println("from m1");
        return 20;
    }

    // static method(main !!)
    public static void main(String[] args)
    {
        System.out.println("Value of a : " + a);
        System.out.println("from main");
    }
}

Output:

from m1
Inside static block
Value of a : 20
from main
```

Non-static variable

- **Local Variables**: A variable defined within a block or method or constructor is called local variable.

- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
- Initialisation of Local Variable is Mandatory.

Q. What is static keyword in Java?

Ans. **Static** is a Non-Access Modifier. Static can be applied to variable, method, nested class and initialization blocks (static block).

Q. What is a static variable?

Ans. A **Static variable** gets memory allocated only once during the time of class loading.

Q. Can we have multiple static blocks in our code?

Ans. **Yes**, we can have more than one static block in our code. It will be executed in the same order it is written.

Q. What is a static block?

- A **static block** is a block of code inside a Java class that will be executed when a class is first loaded into the JVM. Mostly the static block will be used for initializing the variables.
- The static block will be called only one while loading and it **cannot have any return type**, or any keywords (**this** or **super**).

```
class test
{
  static int val;
  static
        {
            val = 100;
        }
}
```

Q. Why main() method is declared as static?

Ans. If our **main()** method is not declared as static then the JVM has to create an object first and call which causes the problem of having extra memory allocation.

Q. Can we Overload static methods in Java?

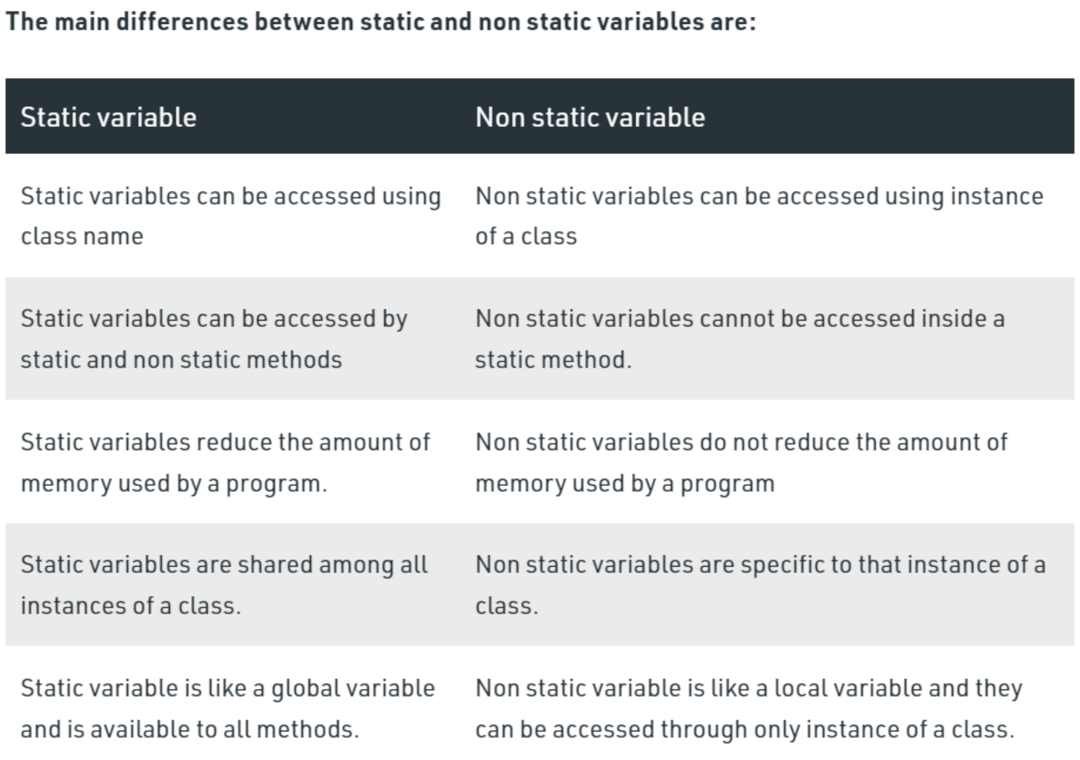
Ans. Yes, you can overload a static method in Java. Java Method Overloading is one of the feature in OOPs concept which allows you to have two or more methods with same method name with difference in the parameters in other words, we can also call this phenomena as Compile time polymorphism.

Q. Can we Override static methods in Java?

Ans. No, We cannot Override a static method in Java. Unlike Overloading of static method we cannot do overriding. When we declare a method with same signature and static in both Parent and Child class then it is not considered as Method Overriding as there will not be any Run-time Polymorphism happening.

- When the **Child** class also has defined the **same static method** like **Parent** class, then the method in the **Child** class **hides** the method in the **Parent** class. In the below code we can see that we are having a **static display() method** in both **Parent** and **Child** class.

```
package com.javainterviewpoint;

import java.io.IOException;

class Parent
{
    public static void display()
    {
        System.out.println("Welcome to Parent Class");
    }
}
public class Child extends Parent
{
    public static void display()
    {
        System.out.println("Welcome to Child class");
    }
    public static void main(String args[])
    {
        //Assign Child class object to Parent reference
        Parent pc = new Child();
        pc.display();
     }
}

Output:
Welcome to Parent Class
```



The main differences between static and non static variables are:

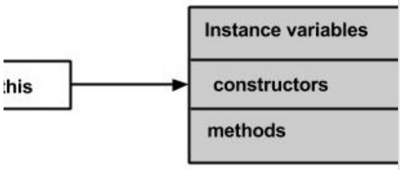
Static variable	Non static variable
Static variables can be accessed using class name	Non static variables can be accessed using instance of a class
Static variables can be accessed by static and non static methods	Non static variables cannot be accessed inside a static method.
Static variables reduce the amount of memory used by a program.	Non static variables do not reduce the amount of memory used by a program
Static variables are shared among all instances of a class.	Non static variables are specific to that instance of a class.
Static variable is like a global variable and is available to all methods.	Non static variable is like a local variable and they can be accessed through only instance of a class.

3. Methods - void, return, value, return

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out. println() method, for example, the system actually executes several statements in order to display a message on the console.

https://www.tutorialspoint.com/java/java_methods.htm



The void Keyword

The void keyword allows us to create methods which do not return a value.

4. Data Types

Q. What are the primitive data types in Java ?

There are eight primitive data types.

- byte.
- short.
- int.
- long.
- float.
- double.
- boolean.
- char.

Q. Is Java primitive data type stored on stack or heap?

Primitive types declared locally will be on the stack while primitive types that are defined as part of an object instance are stored on the heap.

Local variables are stored on stack while instance and static variables are stored on the heap.

Q. Can you compare a boolean with an int variable in Java?

Ans. No. you will get compilation error.

Q. Difference between double and float variables in Java.

In java, float takes 4 bytes in memory while Double takes 8 bytes in memory. Float is single precision floating point decimal number while Double is double precision decimal number.

Q. What is the default value of char data type in Java?

The default value of a char primitive type is '\u0000'(null character) as stated in the Java Language Specification.

The shortcut for 'u0000' is '\0', So the null can be represented either by 'u0000' or '\0'.

Q. How primitive variables passed to methods - by value or by reference?

In Java, primitive variables are passed to methods by value. If the passed value changes in the method, it does not change the original value.

Q. What is the output?

```
System.out.println(1.0/0);
```

Most of us may expect Arithmetic Exception, however, in this case, there will be no exception instead it prints **Infinity**.

1.0 is a double literal and double datatype supports infinity.

5. Variable types

Q. What is the difference between primitive variables and reference variables?

There are basically two different kinds of variables in Java programming language -

Primitive variables and Reference variables.

Primitive variables contain primitive literal values,

where as reference variables contain a reference to an Object.

```
class MyClass {
  //Primitive variable declaration - var1,
  //var1 contains literal value 123.
  int var1 = 123;

  //Reference variable declaration - var2,
  //var2 contains reference to object of type 'Box'.
  Box var2 = new Box();
}
```

Q. What are the different kinds of variables defined in java programming language?

Based on scope, variables can be of four different types - Class variables, Instance variables, Local variables and Parameters. Scope of a variable is determined based on where it is declared within a java class.

1. Class variable (Static fields) -

Class variables are variables declared within the class body, outside of any methods or blocks, and declared with 'static' keyword.

Class variables have the longest scope. They are created when the class is loaded, and remain in memory as long as the class remains loaded in JVM.

2. Instance variables (Non-static fields) -

Instance variable are variables declared within the class body, outside of any method or block, and declared without 'static' keyword.

Instance variables have the second highest scope. Instance variables are created when a new class instance is created, and live until the instance is removed from memory.

3. Local Variables -

Local variables are variables declared within a method body. They live only as long as the method in which it is declared remains on the stack.

4. Block variables -

Block variables are variables declared within a block such as an init block or within a for loop. They live only during the execution of the block and are the shortest living variables.

```
class MyClass {
  //Static variable
  static String string1 = 'test string 1';
  //Instance variable
  String string2 = 'test string 2';
  //Block variable in init block
  {String string3 = 'test string 3'}
  void perform() {
    //Local variable
    String string4 = 'test string 4'
    //Block variable in for loop
    for (int i=0; i < 4; i++) {...}
  }
}
```

Q. What are final variables in Java programming language?

Final variables are variables declared with keyword 'final'. Once a value is assigned to a final variable it cannot be changed.

If final variable is a primitive variable, the primitive literal cannot be changed once it is assigned to the primitive variable.

If the final variable is a reference variable and an object is assigned to it, it cannot be changed to refer to a different object. Please note that the attributes of the Object referred to by the final variable can change.

```
class MyClass {
  //final primitive variable var1,
  //value of var1 cannot change from 123
  final int var1 = 123;

  //final reference variable - var2
  //var2 cannot be changed to refer another Box object,
  //attributes of the Box object can change.
  final Box var2 = new Box();
}
```

Q. What are transient variables in Java programming language?

Transient variable is a variable whose value is not serialized during serialization of the object. During de-serialization of the object, transient primitive variables are initialized to their default values. Transient reference variables are initialized to null

```
class MyClass {
  // Transient variable
  transient int var1 = 123;
}
```

Q. What are volatile variables in Java programming language?

Volatile variables are relevant in multi-threaded Java programming, in which multiple threads access the variables of an object. A volatile variable is declared with the keyword 'volatile'.

Volatile variables tells the JVM that this variable will be updated by multiple threads, and to always take its value from the main memory. So all thread access and get the value of a volatile variable from the main memory and do not cache the value in the thread's memory cache.

```
class MyClass {
  // Volatile variable
  volatile int var1 = 123;
}
```

6. Constructor and Types

Top 17 Core Java Interview Questions on Constructors

Constructor is just like a method in that is used to initialize the state of an object and will be invoked during the time of object creation.

https://www.javainterviewpoint.com/top-14-core-java-interview-questions-constructors/

Q. What is a Constructor in Java? Imp

Constructor is just like a method in **Java** that is used to initialize the state of an object and will be invoked during the time of object creation.

OR

Q. **Why is constructor used** - Answer: Initializing variables

Q. What are the Rules for defining a constructor? Imp

1. Constructor **name** should be the same as the class name
2. It **cannot** contain any **return type**
3. It **can** have all **Access Modifiers** are allowed (private , public, protected, default)
4. It **Cannot** have any **Non Access Modifiers** (final ,static, abstract, synchronized)
5. **No return** statement is allowed
6. It **can** take any number of **parameters**
7. Constructor can **throw exception**, we can have **throws clause**

Q. Can we have a Constructor in an Interface?

No, We cannot have a Constructor defined in an **Interface**.

Q. What is Constructor Chaining in Java? Imp

Constructor Chaining is nothing but calling one Constructor from another. **this keyword** is used to call the **current** class constructor and **super keyword** is used to call the **parent** class constructor.

Q. Can we have this and super in the same constructor?

No, we **cannot** have have **this** and **super** in a same constructor as any one only can be in the first line of the constructor.

If we use we will get this error

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  Constructor call must be the first statement in a constructor
```

Q. Can a Constructor return any value ?

A Constructor cannot return any explicit value but implicitly it will be returning the instance of the class.

Q. Why constructor is allowed/available in abstract class? || Sai Ram 27 July

In abstract class, we have an instance variable, abstract methods, and non-abstract methods. **We need to initialize the non-abstract methods and instance variables**, therefore abstract classes have a constructor.

OR

Two reasons for this:

1) Abstract classes have `constructors` and those constructors are always invoked when a concrete subclass is instantiated. We know that when we are going to instantiate a class, we always use constructor of that class. Now every constructor invokes the constructor of its **super class with an implicit call to** `super()`.

2) We know constructor are also used to initialize fields of a class. We also know that abstract classes may contain fields and sometimes they need to be **initialized somehow by using constructor**.

```
abstract class Demo1 {
    String value;
    public Demo1( String value ) {
        this.value = value;
    }
    public String getValue()
    {
      return value;
    }
}
public class Test extends Demo1 {
    public Test() {
        super("CoreJava");
    }
}
```

Q. Why constructors cannot be final in Java?

When you set a method as final, then" The method cannot be overridden by any class", but **Constructor** by JLS (**Java Language Specification**) definition can't be overridden. A constructor is not inherited, so there is no need for declaring it as **final**.

Q. Why constructors cannot be abstract in Java?

When you set a method as abstract, then "The method doesn't or cannot have body". A constructor will be automatically called when object is created. It cannot lack a body moreover an abstract constructor could never be implemented.

Q. Why constructors cannot be static in Java?

When you set a method as static, it means "The Method belong to class and not to any particular object" but a constructor is always invoked with respect to an object, so it makes no sense for a constructor to be **static**.

Q. What is constructor overloading?

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Consider the following Java

program, in which we have used different constructors in the class.

```
public class Student {
//instance variables of the class
int id;
String name;

Student(){
System.out.println("this a default constructor");
}

Student(int i, String n){
id = i;
name = n;
}

public static void main(String[] args) {
//object creation
Student s = new Student();
System.out.println("\nDefault Constructor values: \n");
System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

System.out.println("\nParameterized Constructor values: \n");
Student student = new Student(10, "David");
System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name);
}
}

Output:
this a default constructor


Default Constructor values:


Student Id : 0
Student Name : null


Parameterized Constructor values:


Student Id : 10
Student Name : David
```

7. Java New Keyword

The Java new keyword is used to create an instance of the class. In other words, it instantiates a class by allocating memory for a new object and returning a reference to that memory. We can also use the new keyword to create the array object.

Syntax:

```
NewExample obj=new NewExample();
```

Points to remember

- It is used to create the object.
- It allocates the memory at runtime.
- All objects occupy memory in the heap area.

- It invokes the object constructor.
- It requires a single, postfix argument to call the constructor

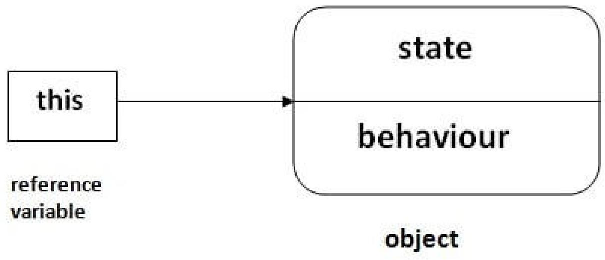
Simple Example for new keyword

```
public class NewExample1 {

    void display()
    {
        System.out.println("Invoking Method");
    }

    public static void main(String[] args) {
        NewExample1 obj=new NewExample1();
        obj.display();
    }

}

Output:

Invoking Method
```

8. Java This Keyword

Q. What is this keyword in java?

The **this** keyword is a reference variable that refers to the current object. There are the various uses of this keyword in Java. It can be used to refer to current class properties such as instance methods, variable, constructors, etc. It can also be passed as an argument into the methods or constructors. It can also be returned from the method as the current class instance.



Q. What are the main uses of this keyword?

There are the following uses of **this** keyword.

- **this** can be used to refer to the current class instance variable.
- **this** can be used to invoke current class method (implicitly)
- **this()** can be used to invoke the current class constructor.
- **this** can be passed as an argument in the method call.
- **this** can be passed as an argument in the constructor call.
- **this** can be used to return the current class instance from the method.

Q. Can we assign the reference to this variable?

No, this cannot be assigned to any value because it always points to the current class object and this is the final reference in Java. However, if we try to do so, the compiler error will be shown. Consider the following example.

```
public class Test
{
    public Test()
    {
        this = null;
        System.out.println("Test class constructor called");
    }
    public static void main (String args[])
    {
        Test t = new Test();
    }
}

Output [Error]
Test.java:5: error: cannot assign a value to final variable this
        this = null;
        ^
1 error
```

Q. Can this keyword be used to refer static members?

Yes, It is possible to use this keyword to refer static members because this is just a reference variable which refers to the current class object. However, as we know that, it is unnecessary to access static variables through objects, therefore, it is not the best practice to use this to refer static members. Consider the following example.

```
public class Test
{
    static int i = 10;
    public Test ()
    {
        System.out.println(this.i);
    }
    public static void main (String args[])
    {
        Test t = new Test();
    }
}

Output

10
```

Q. How can constructor chaining be done using this keyword?

Constructor chaining enables us to call one constructor from another constructor of the class with respect to the current class object. We can use this keyword to perform constructor chaining within the same class. Consider the following example which illustrates how can we use this keyword to achieve constructor chaining.

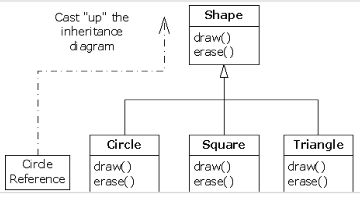
```
public class Employee
{
    int id,age;
    String name, address;
    public Employee (int age)
    {
        this.age = age;
    }
    public Employee(int id, int age)
    {
        this(age);
        this.id = id;
    }
    public Employee(int id, int age, String name, String address)
    {
        this(id, age);
        this.name = name;
        this.address = address;
    }
    public static void main (String args[])
    {
        Employee emp = new Employee(105, 22, "Vikas", "Delhi");
        System.out.println("ID: "+emp.id+" Name:"+emp.name+" age:"+emp.age+" address: "+emp.address);
    }

}

Output:
ID: 105 Name:Vikas age:22 address: Delhi
```

9.Inheritance

Top 21 Java Inheritance Interview Questions and Answers

It's expected from Java developers to know about these OOP concepts and have an understanding of when to use them as many times Composition is the better choice than Inheritance because of flexibility it offers but when it comes to leverage polymorphism of type,

https://www.java67.com/2016/03/top-21-java-inheritance-interview-Questions-Answer-Programming.html#ixzz7EHS6LU3v

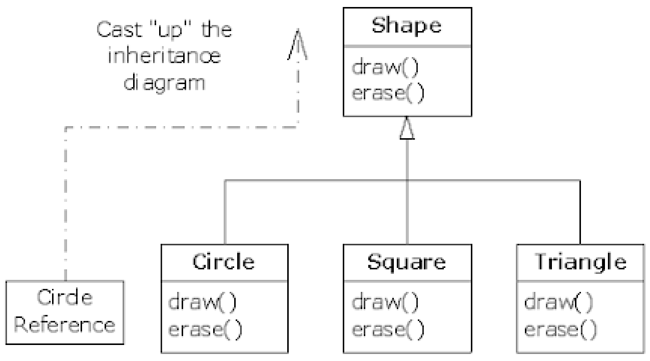


Q. What is Inheritance in Java?

Inheritance is an Object oriented feature which allows a class to inherit behavior and data from other class. For example, a class Car can extend basic feature of Vehicle class by using Inheritance. One of the most intuitive examples of Inheritance in the real world is Father-Son relationship, where Son inherit Father's property.

Q. What are different types of Inheritance supported by Java?

Java supports single Inheritance, multi-level inheritance and at some extent multiple inheritances because Java allows a class to only extend another class, but an interface in Java can extend multiple inheritances.



Q. Why multiple Inheritance is not supported by Java? || Why complex designing happened in multiple inheritance? Imp

Java is introduced after C++ and Java designer didn't want to take some C++ feature which is confusing and not essential. They think multiple inheritances is one of them which doesn't justify complexity and confusion it introduces.

Allowing multiple inheritance makes **the rules about function overloads and virtual dispatch decidedly more tricky**, as well as the language implementation around object layouts. These impact language designers/implementors quite a bit, and raise the already high bar to get a language done, stable and adopted.

Q. What is the syntax of Inheritance?

You can use either extends of implements keyword to implement Inheritance in Java. A class extends another class using extends keyword, an interface can extend another interface using extend keyword, and a class can implement an interface using implements keyword in Java.

Q. What is the difference between Inheritance and Encapsulation?

Inheritance is an object oriented concept which creates a parent-child relationship. It is one of the ways to reuse the code written for parent class but it also forms the basis of Polymorphism. On the other hand, Encapsulation is an object oriented concept which is used to hide the internal details of a class e.g. HashMap encapsulate how to store elements and how to calculate hash values.

Q. Can we override static method in Java?

No, you cannot override a static method in Java because it's resolved at compile time. In order for overriding to work, a method should be virtual and resolved at runtime because objects are only available at runtime. This is one of the tricky Java questions, where interviewer tries to confuse you. A programmer is never sure about whether they can override or overload a static method in Java.

Q. Can we overload a static method in Java?

Yes, you can overload a static method in Java. Overloading has nothing to do with runtime but the signature of each method must be different. In Java, to change the method signature, you must change either number of arguments, type of arguments or order of arguments.

Q. Can we override a private method in Java?

No, you cannot override a private method in Java because the private method is not inherited by the subclass in Java, which is essential for overriding. In fact, a private method is not visible to anyone outside the class and, more importantly, a call to the private method is resolved at compile time by using Type information as opposed to runtime by using the actual object.

Q. Can a class implement more than one interface in Java?

Yes, A class can implement more than one interface in Java e.g. A class can be both Comparable and Serializable at the same time. This is why the interface should be the best use for defining Type as described in Effective Java. This feature allows one class to play a polymorphic role in the program.

Q. Can a class extends more than one class in Java?

No, a class can only extend just one more class in Java. Though Every class also, by default extend the java.lang.Object class in Java.

Q. Can an interface extends more than one interface in Java?

Yes, unlike classes, an interface can extend more than one interface in Java. There are several example of this behavior in JDK itself e.g. java.util.List interface extends both Collection and Iterable interface to tell that it is a Collection as well as it allows iteration via Iterator.

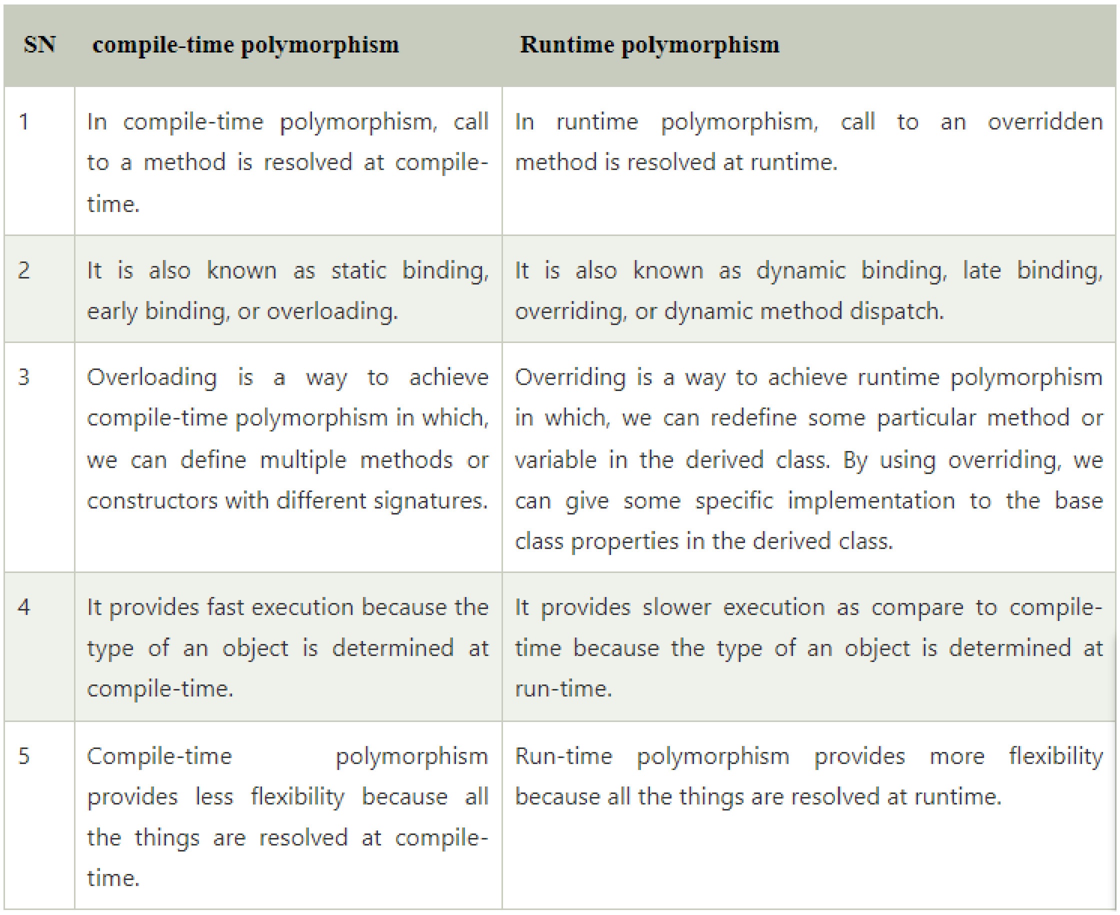
10. Polymorphism - Overriding and Overloading

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

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

Q. What is the difference between compile-time polymorphism and runtime polymorphism?



SN	compile-time polymorphism	Runtime polymorphism
1	In compile-time polymorphism, call to a method is resolved at compile-time.	In runtime polymorphism, call to an overridden method is resolved at runtime.
2	It is also known as static binding, early binding, or overloading.	It is also known as dynamic binding, late binding, overriding, or dynamic method dispatch.
3	Overloading is a way to achieve compile-time polymorphism in which, we can define multiple methods or constructors with different signatures.	Overriding is a way to achieve runtime polymorphism in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class.
4	It provides fast execution because the type of an object is determined at compile-time.	It provides slower execution as compare to compile-time because the type of an object is determined at run-time.
5	Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time.	Run-time polymorphism provides more flexibility because all the things are resolved at runtime.

Q. What is Runtime Polymorphism?

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}
  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}

Output:

running safely with 60km.
```

Q. What is the difference between static binding and dynamic binding?

In case of the static binding, the type of the object is determined at compile-time whereas, in the dynamic binding, the type of the object is determined at runtime.

```
Static Binding

class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}

Dynamic Binding

class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Q. What is Java instanceOf operator?

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 a null value, it returns false. Consider the following example.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}

Output:

true
```

11. Method overloading

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Q. 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 calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
Output:

main with String[]
```

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:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}

Output:

Compile Time Error: method add(int,int) is already defined in class Adder
```

12. Access Modifiers

Access modifiers in java - Javatpoint

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

https://www.javatpoint.com/access-modifiers

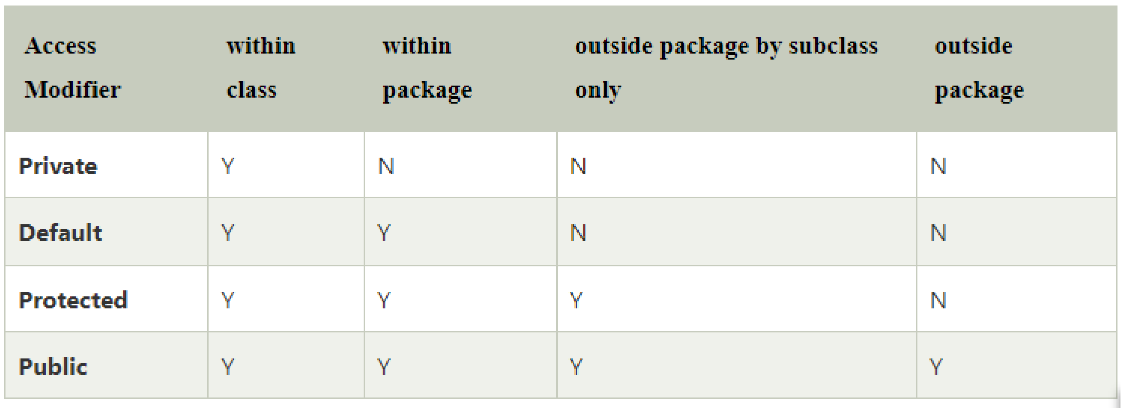
The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

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

Note: There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

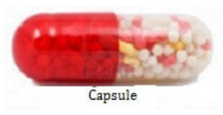
Understanding Java Access Modifiers



Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

13. Encapsulation

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is better for unit testing.
- The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Example:

```
//A Account class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
class Account {
//private data members
private long acc_no;
private String name,email;
private float amount;
//public getter and setter methods
public long getAcc_no() {
    return acc_no;
}
public void setAcc_no(long acc_no) {
    this.acc_no = acc_no;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public float getAmount() {
    return amount;
}
public void setAmount(float amount) {
    this.amount = amount;
}

}
```

```
//A Java class to test the encapsulated class Account.
public class TestEncapsulation {
public static void main(String[] args) {
    //creating instance of Account class
    Account acc=new Account();
    //setting values through setter methods
    acc.setAcc_no(7560504000L);
    acc.setName("Sonoo Jaiswal");
    acc.setEmail("sonoojaiswal@javatpoint.com");
    acc.setAmount(500000f);
    //getting values through getter methods
    System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
}
}

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

Q. What is Encapsulation in Java? Why is it called Data hiding?

he process of binding data and corresponding methods (behavior) together into a single unit is called encapsulation in Java.

If a field is declared private in the class then it cannot be accessed by anyone outside the class and hides the fields within the class. Therefore, Encapsulation is also called data hiding.

Q. How to achieve encapsulation in Java? Give an example.

There are two key points that should be kept in mind to achieve the encapsulation in Java. They are as follows:

- Declare the variable of the class as private.
- Provide public setter and getter methods to modify the values of variables.

```
public class EncapsulationTest{
 private String name;
 private String idNum;
 private int age;

public int getAge() {
   return age;
}
public String getName() {
   return name;
}
public String getIdNum() {
    return idNum;
}
public void setAge( int newAge) {
    age = newAge;
}
public void setName(String newName) {
    name = newName;
}
public void setIdNum( String newId) {
    idNum = newId;
  }
}
```

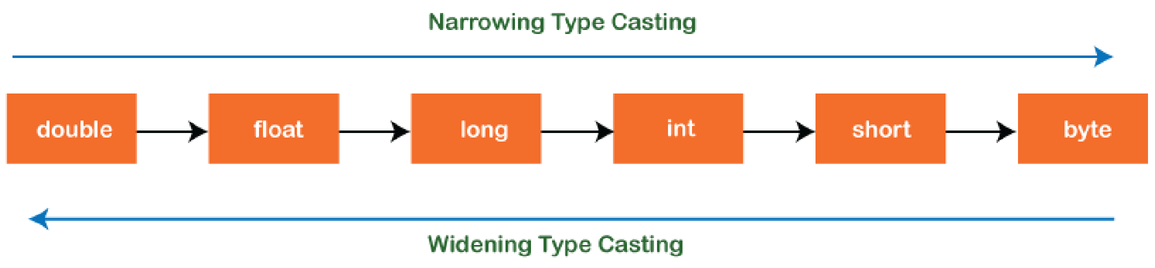
Q. What are getter and setter methods in Java?

In Java, setter method is a method that is used for updating the values of a variable. This method is also known as mutator method.

Getter method is a method that is used to retrieve the value of a variable or return the value of the private member variable. This method is also known as an accessor method.

14. Type Casting

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

Widening Type Casting:

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

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

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

Example:

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}

Output

Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

Narrowing Type Casting:

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

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

Example:

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
Output

Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

15.Class Casting:

1. **Upcasting** is casting a subtype to a super type in an upward direction to the inheritance tree. It is an automatic procedure for which there are no efforts poured in to do so where a sub-class object is referred by a superclass reference variable. One can relate it with dynamic polymorphism.
    - Implicit casting means class typecasting done by the compiler without cast syntax.
    - Explicit casting means class typecasting done by the programmer with cast syntax.
2. **Downcasting** refers to the procedure when subclass type refers to the object of the parent class is known as downcasting. If it is performed directly compiler gives an error as **ClassCastException** is thrown at runtime. It is only achievable with the use of **instanceof** operator The object which is already upcast, that object only can be performed downcast.

In order to perform class type casting we have to follow these two rules as follows:

1. Classes must be “IS-A-Relationship “
2. An object must have the property of a class in which it is going to cast.

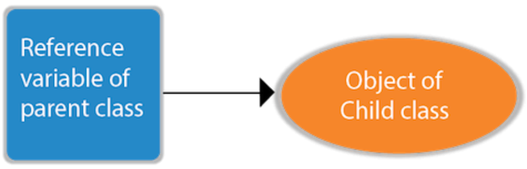
16. Runtime Polymorphism

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}
class B extends A{}

A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}

Output:

running safely with 60km.
```

17. Interface

Q. What is an interface in Java?

An interface in Java is a mechanism that is used to achieve complete abstraction. It is basically a kind of class that contains only constants and abstract methods.

Q. Can an interface extends another interface in Java?

Yes, an interface can extend another interface.

Q. Is it possible to define a class inside an interface?

Yes, we can define a class inside an interface.

Q. Which of the following is a correct representation of interface?

```
a) public abstract interface A {
    abstract void m1() {};
   }
b) public abstract interface A {
    void m1();
   }
c) abstract interface A extends B, C {
     void m1();
   }
d) abstract interface A extends B, C {
    void m1(){};
   }
e) abstract interface A {
    m1();
   }
f) interface A {
    void m1();
   }
g) interface A {
    int m1();
   }
h) public interface A {
    void m1();
   }
   public class B implements A {
   }
i) public interface A {
     void m1();
   }
   Public interface B {
     void m1();
   }
   public interface C extends A, B {
    void m1();
   }

Answer:
b, c, f, g, i.
```

Q. Identify the error in the following code.

```
interface A {
 void m1();
}
public class B implements A {
void m1(){
  System.out.println("One");
 }
}

Answer:
We cannot reduce the visibility of inherited method from interface A.
```

Q. Why an Interface method cannot be declared as final in Java?

Not possible. Doing so will result the compilation error problem. This is because a final method cannot be overridden in java. But an interface method should be implemented by another class.

So, the interface method cannot be declared as final. The modifiers such as public and abstract are only applicable for method declaration in an interface.

Q. Can an interface be final?

No. Doing so will result compilation error problem.

Q. What is the use of interface in Java?

There are many reasons to use interface in java. They are as follows:

a. An interface is used to achieve fully abstraction.

b. Using interfaces is the best way to expose our project’s API to some other project.

c. Programmers use interface to customize features of software differently for different objects.

d. By using interface, we can achieve the functionality of multiple inheritance.

Functional Interfaces:

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards,

lambda expressions

can be used to represent the instance of a functional interface. A functional interface can have any number of default methods.

Runnable

ActionListener

Comparable are some of the examples of functional interfaces.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
// Java program to demonstrate functional interface

class Test
{
  public static void main(String args[])
  {
    // create anonymous inner class object
    new Thread(new Runnable()
    {
      @Override
      public void run()
      {
        System.out.println("New thread created");
      }
    }).start();
  }
}

Output:

New thread created

<-------------------------------------------------------------------------------------->

// Java program to demonstrate Implementation of
// functional interface using lambda expressions

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

  // lambda expression to create the object
  new Thread(()->
  {System.out.println("New thread created");}).start();
}
}
```

Lock Interface:

A java.util.concurrent.locks.Lock interface **is used to as a thread synchronization mechanism similar to synchronized blocks**. New Locking mechanism is more flexible and provides more options than a synchronized block

Runnable Interface:

`java.lang.Runnable` is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass `Thread` and implement `Runnable`. There is no need of sub classing `Thread` when a task can be done by overriding only `run()` method of `Runnable`.

18. Marker Interface

Q. What is a Marker Interface in Java?

An Interface that doesn’t have any data members or methods is called marker interface in java. For example, Serializable, Cloneable, Remote, etc.

Q. Why do we need Functional interface in Java 8?

A Functional interface is mainly used in Lambda expressions, method reference, and constructor references.

Q. Why do you need marker interface in Java?

Marker interface is used **as a tag to inform a message to the Java compiler so that it can add special behavior to the class implementing it**. When a Java class is to be serialized, you should intimate the Java compiler in some way that there is a possibility of serializing this java class.

19. Abstract Keyword

The abstract keyword is used to achieve abstraction in Java. It is a non-access modifier which is used to create abstract class and method.

The role of an abstract class is to contain abstract methods. However, it may also contain non-abstract methods. The method which is declared with abstract keyword and doesn't have any implementation is known as an abstract method.

```
abstract class Employee
{
abstract void work();
}
Note - We cannot declare abstract methods in non abstract class.
```

Rules of abstract keyword

Don'ts

- An abstract keyword cannot be used with variables and constructors.
- If a class is abstract, it cannot be instantiated.
- If a method is abstract, it doesn't contain the body.
- We cannot use the abstract keyword with the **final**.
- We cannot declare abstract methods as **private**.
- We cannot declare abstract methods as **static**.
- An abstract method can't be synchronized.

Do's

- An abstract keyword can only be used with class and method.
- An abstract class can contain constructors and static methods.
- If a class extends the abstract class, it must also implement at least one of the abstract method.
- An abstract class can contain the main method and the final method.
- An abstract class can contain overloaded abstract methods.
- We can declare the local inner class as abstract.
- We can declare the abstract method with a throw clause.

Example:

Abstract class containing the abstract method

```
abstract class Vehicle
{
    abstract void bike();

}
class Honda extends Vehicle
{

    @Override
    void bike() {
        System.out.println("Bike is running");

    }

}

public class AbstractExample1 {

    public static void main(String[] args) {

    Honda obj=new Honda();
    obj.bike();
    }
}

Output:

Bike is running
```

Abstract class containing the constructor

```
abstract class Vehicle
{
    String msg;

    Vehicle(String msg)
    {
    this.msg=msg;
    }

    void display()
    {
        System.out.println(msg);
    }

}
class Honda extends Vehicle
{
    Honda(String msg) {
        super(msg);

    }

}

public class AbstractExample3 {

    public static void main(String[] args) {

    Honda obj=new Honda("Constructor is invoked");
    obj.display();

    }
}

Output:

Constructor is invoked
```

20. Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

Example of Final Keyword:

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class

Output:Compile Time Error
```

Q. What is the use of the final keyword in Java?

Final keyword can be applied to **variable, method,** and **class**. Each of them has its own uses

- The **final variable** is a variable whose value **cannot** be changed at any time once assigned, it remains as a constant forever.
- The **final method cannot be overridden**
- A **final class cannot be subclassed (cannot be extended)**

Q. What is a blank final variable?

A **blank final variable** is a **final variable**, which is not initialized during declaration.

Q. Can we declare final variable without initialization?

Yes, We can declare a **final variable** without initialization and these final variables are called a blank final variable but must be initialized before usage.

Q. What is a final method?

When a method is declared as **final**, then it is called as a **final method,** The subclass can call the final method of the parent class but cannot **override** it.

Q. What is a final class? Imp

A class declared with a **final keyword** is called a **final class**, a final class **cannot be sub-classed**. This means a final class cannot be inherited by any class.

Q. Can a main() method be declared final?

Yes, the **main()** method can be declared as final and cannot be **overridden**.

Q. Can we declare constructor as final?

No, Constructor cannot be declared as **final**. Constructors are not inherited and so it cannot be overridden, so there is no use to have a final constructor.

You will get an error like **“Illegal modifier for the constructor in type Test; only public, protected & private are permitted”**

Q. Can we declare an interface as final?

The sole purpose of Interface is to have the subclass implement it if we make it final it cannot be implemented. Only **public & abstract** are permitted while creating an interface.

Q. Can Final Variable be serialized in Java?

Yes, the final variable can be serialized in Java

Q. Can final method be overloaded in Java?

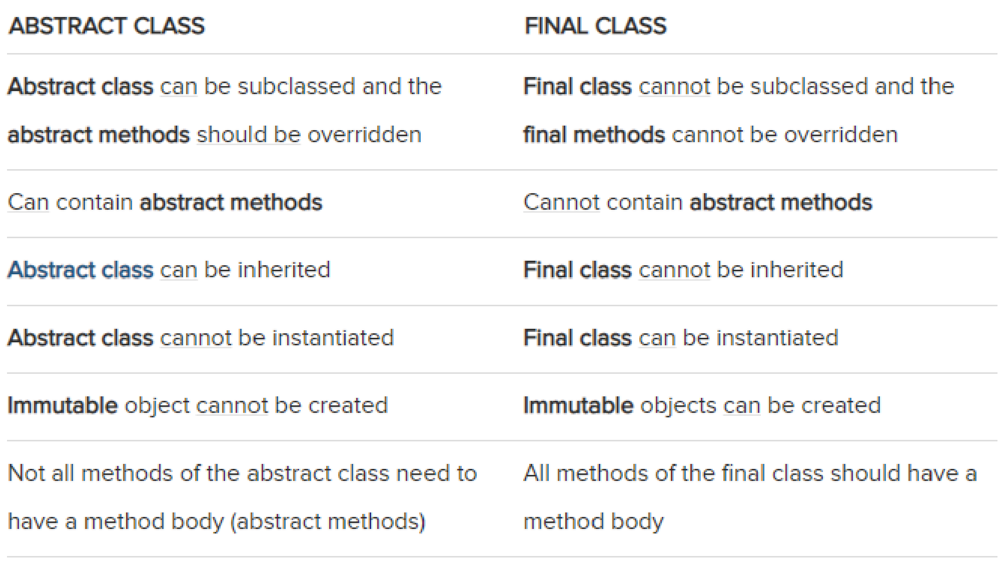
Yes, the final method can be **overloaded** but cannot be **overridden**. Which means you can have more than one final method with the same name with different parameters.

Q. What is the main difference between abstract methods and final methods?

Abstract methods are declared in **abstract classes** and cannot be implemented in the same class. They must be implemented in the **subclass**. The only way to use an **abstract method** is by **overriding** it

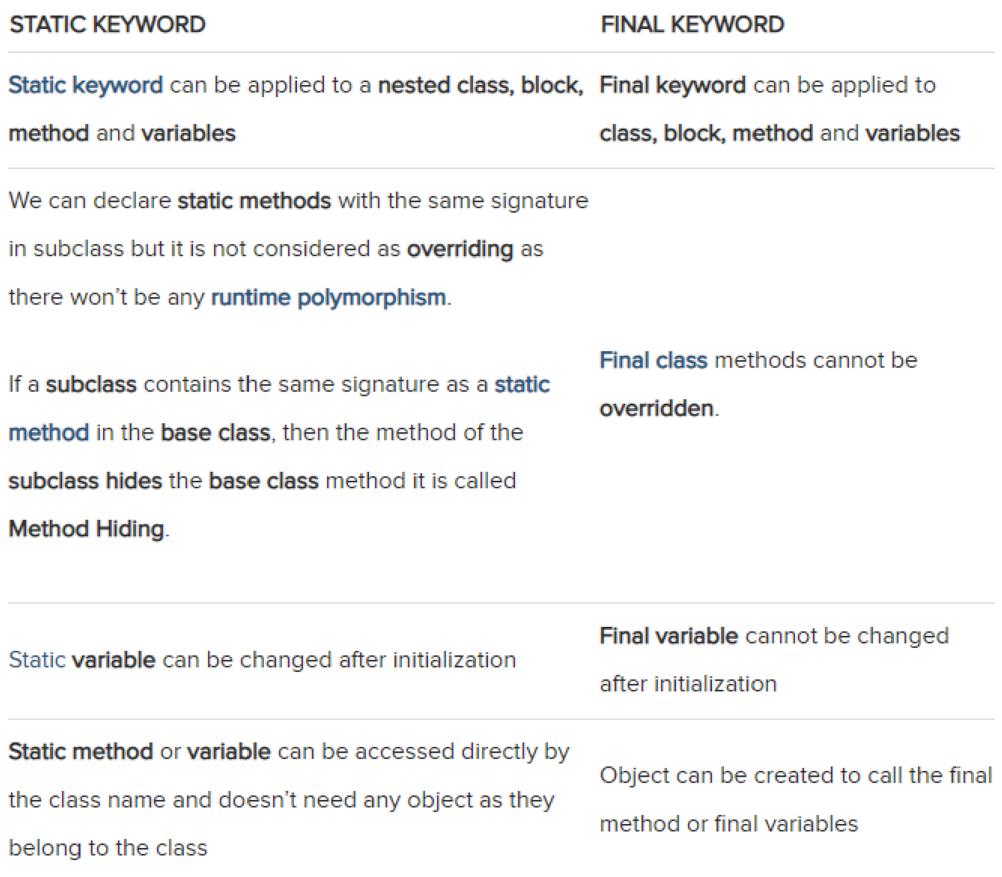
Final methods are quite opposite to **abstract**, final methods cannot be **overridden**.

Q. What is the difference between abstract class and final class?



ABSTRACT CLASS	FINAL CLASS
Abstract class can be subclassed and the **abstract methods** should be overridden	**Final class** cannot be subclassed and the **final methods** cannot be overridden
Can contain **abstract methods**	Cannot contain **abstract methods**
Abstract class can be inherited	**Final class** cannot be inherited
Abstract class cannot be instantiated	**Final class** can be instantiated
Immutable object cannot be created	**Immutable** objects can be created
Not all methods of the abstract class need to have a method body (abstract methods)	All methods of the final class should have a method body

Q. What is the difference between static and final in Java?



STATIC KEYWORD	FINAL KEYWORD
Static keyword can be applied to a **nested class, block, method** and **variables**	**Final keyword** can be applied to **class, block, method** and **variables**
We can declare **static methods** with the same signature in subclass but it is not considered as **overriding** as there won't be any **runtime polymorphism**. If a **subclass** contains the same signature as a **static method** in the **base class**, then the method of the **subclass hides** the **base class** method it is called **Method Hiding**.	**Final class** methods cannot be **overridden**.
Static **variable** can be changed after initialization	**Final variable** cannot be changed after initialization
Static method or **variable** can be accessed directly by the class name and doesn't need any object as they belong to the class	Object can be created to call the final method or final variables

Q. What is a Static Final variable in Java?

When have declared a variable as **static final** then the variable becomes a **CONSTANT**. Only one copy of variable exists which cannot be changed by any instance.

Q. What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
 }
}
Output: Compile Time Error
```

21. Features of Java 8

- Lambda expressions,
- Method references,
- Functional interfaces,
- Stream API,
- Default methods,
- Base64 Encode Decode,
- Static methods in interface,
- Optional class,
- Collectors class,
- ForEach() method,
- Nashorn JavaScript Engine,
- Parallel Array Sorting,
- Type and Repating Annotations,
- IO Enhancements,
- Concurrency Enhancements,
- JDBC Enhancements etc.

Lambda Expressions

Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface(Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

Method References

Java 8 Method reference is used to refer method of functional interface . It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Functional Interface

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods. It can also declare methods of object class.

Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces).

Optional

Java introduced a new class Optional in Java 8. It is a public final class which is used to deal with NullPointerException in Java application. We must import *java.util* package to use this class. It provides methods to check the presence of value for particular variable.

forEach

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interfaces.

It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach() method to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.

Stream API

Java 8 java.util.stream package consists of classes, interfaces and an enum to allow functional-style operations on the elements. It performs lazy computation. So, it executes only when it requires.

22.Abstraction

Q. How To Describe Abstraction In Interview?

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- A method that is declared as abstract and does not have implementation is known as abstract method.
- There are two ways to achieve abstraction in java1- By Abstract class (0 to 100%) , 2- By Interface (100%)

Q. What is abstraction and abstract class in Java?

Abstraction:

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

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Abstract class in Java:

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
abstract class A{}
```

23. Interface — Already Covered -17

24. Difference between Abstract 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();
	}

```
//Example
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}

Output:

       I am a
       I am b
       I am c
       I am d
```

Q. Which is better interface or abstract class in Java?

An abstract class allows you to create functionality that subclasses can implement or override. An interface only allows you to define functionality, not implement it. And whereas a class can extend only one abstract class, it can take advantage of multiple interfaces.

25. Exception

Q. What Is an Exception?

An exception is an abnormal event that occurs during the execution of a program and disrupts the normal flow of the program's instructions.

Q. How Can You Handle an Exception?

By using a *try-catch-finally* statement:

```
try {
    // ...
} catch (ExceptionType1 ex) {
    // ...
} catch (ExceptionType2 ex) {
    // ...
} finally {
    // ...
}
```

The block of code in which an exception may occur is enclosed in a *try* block. This block is also called “protected” or “guarded” code.

If an exception occurs, the *catch* block that matches the exception being thrown is executed, if not, all *catch* blocks are ignored.

The *finally* block is always executed after the *try* block exits, whether an exception was thrown or not inside it.

Q. How Can You Catch Multiple Exceptions?

There are three ways of handling multiple exceptions in a block of code.

The first is to use a *catch* block that can handle all exception types being thrown:

```
try {
    // ...
} catch (Exception ex) {
    // ...
}
```

Note : Exception handlers that are too broad can make your code more error-prone, catch exceptions that weren't anticipated, and cause unexpected behavior in your program.

The second way is implementing multiple catch blocks:

```
try {
    // ...
} catch (FileNotFoundException ex) {
    // ...
} catch (EOFException ex) {
    // ...
}
```

The third is to use a multi-catch block:

```
try {
    // ...
} catch (FileNotFoundException | EOFException ex) {
    // ...
}
```

Q. What Is the Difference Between a Checked and an Unchecked Exception? Imp

A checked exception must be handled within a *try-catch* block or declared in a *throws* clause; whereas an unchecked exception is not required to be handled nor declared.

Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively.

All exceptions are checked exceptions, except those indicated by *Error*, *RuntimeException*, and their subclasses.

Q. What Exception Will Be Thrown Executing the Following Code Block?

```
Integer[][] ints = { { 1, 2, 3 }, { null }, { 7, 8, 9 } };
System.out.println("value = " + ints[1][1].intValue());
```

It throws an *ArrayIndexOutOfBoundsException* since we're trying to access a position greater than the length of the array.

Q. What Is Exception Chaining?

Occurs when an exception is thrown in response to another exception. This allows us to discover the complete history of our raised problem:

```
try {
    task.readConfigFile();
} catch (FileNotFoundException ex) {
    throw new TaskException("Could not perform task", ex);
}
```

Q. Can You Throw Any Exception Inside a Lambda Expression's Body?

When using a standard functional interface already provided by Java, you can only throw unchecked exceptions because standard functional interfaces do not have a “throws” clause in method signatures:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> {
    if (i == 0) {
        throw new IllegalArgumentException("Zero not allowed");
    }
    System.out.println(Math.PI / i);
});
```

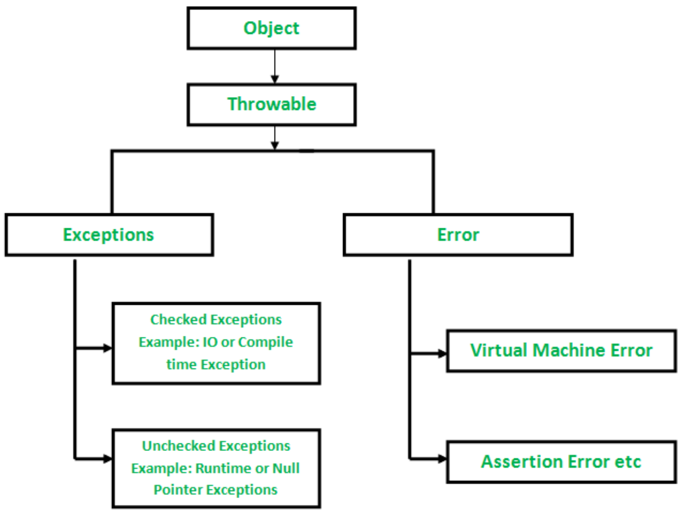
However, if you are using a custom functional interface, throwing checked exceptions is possible:

```
@FunctionalInterface
public static interface CheckedFunction<T> {
    void apply(T t) throws Exception;
}
```

```
public void processTasks(
  List<Task> taks, CheckedFunction<Task> checkedFunction) {
    for (Task task : taks) {
        try {
            checkedFunction.apply(task);
        } catch (Exception e) {
            // ...
        }
    }
}

processTasks(taskList, t -> {
    // ...
    throw new Exception("Something happened");
});
```

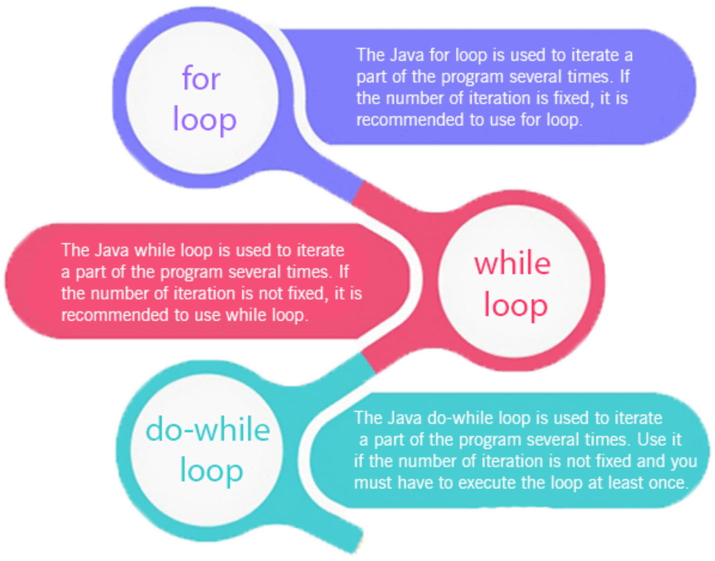
26. Exception hierarchy

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy.One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.Another branch,**Error** are used by the Java run-time system(**JVM**) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

27. Loops


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

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
4. **Statement**: The statement of the loop is executed each time until the second condition is false.

Example:

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}

Output:

1
2
3
4
5
6
7
8
9
10
```

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
        System.out.println(i+" "+j);
}//end of i
}//end of j
}
}

Output:
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Pyramid Example:

```
public class PyramidExample {
public static void main(String[] args) {
for(int i=1;i<=5;i++){
for(int j=1;j<=i;j++){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}

Output:
*
* *
* * *
* * * *
* * * * *
```

Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Example:

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
}
}

Output:

12
23
44
56
78
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

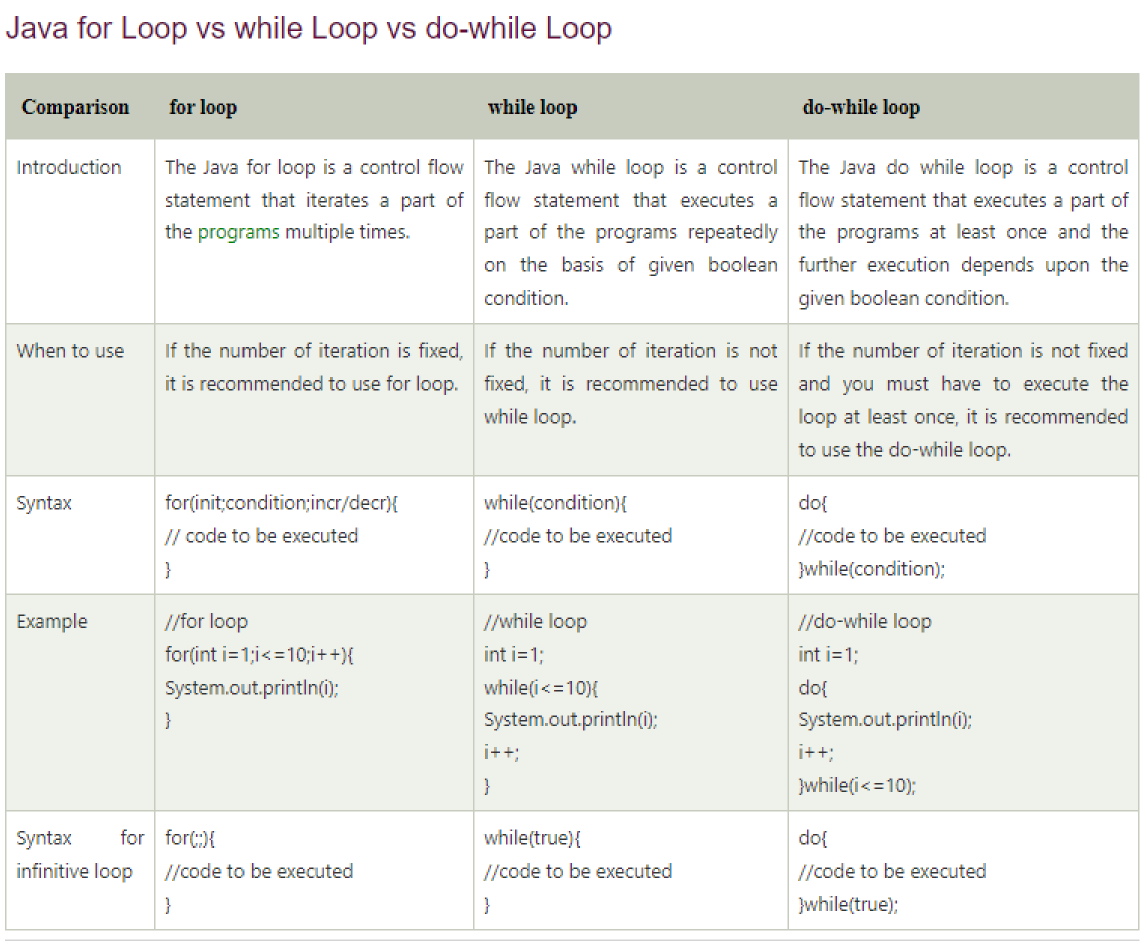
```
/A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
    //Using Label for outer and for loop
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break aa;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}

Output:
1 1
1 2
1 3
2 1
```

Java Infinitive for Loop

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

Example

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
    //Using no condition in for loop
    for(;;){
        System.out.println("infinitive loop");
    }
}
}

Output:
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c //press to exit from the program
```



Java for Loop vs while Loop vs do-while Loop

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax	for(init;condition;incr/decr){
	// code to be executed
	}
		while(condition){
		//code to be executed
		}
			do{
			//code to be executed
			}while(condition);

Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//while loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do-while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

28. Arrays

In Java, **Array** is the most important data structure. It is also used by other data structures such as HashTable, List, etc.

Q. On which memory arrays are created in Java?

Arrays are created on dynamic memory by JVM. There is no question of static memory in Java everything (variable, array, object, etc.) is created on dynamic memory only.

Q. Is it possible to declare array size as negative?

No, it is not possible to declare array size as negative. Still, if we declare the negative size, there will be no compile-time error. But we get the NegativeArraySizeException at run-time.

Q. What is the difference between int array[] and int[] array?

There is no difference between array[] and []array. Both array[] and []array are the ways to declare an array. The only difference between them is that if we are declaring more than one array in a line, we should use prefix []. If we are declaring a single array in a line, we should use postfix []. For example, consider the following declaration:

```
int array1[], array2;   //array1[] is an array while array2 is just a variable of type int
int[] arr1, arr2;  //both arr1 and arr2 are arrays of int type
```

Q. How to copy an array in Java?

We can create a copy of an array in two ways, first one is manually by iterating over the array and the second one is by using the arrayCopy() method. Using the arrayCopy() method of the System class is the fastest way to copy an array and also allows us to copy a part of the array. These two methods are the popular ways to copy an array.

The other two methods to copy an array is to use the Arrays.copyOf() method and using clone() method.

Q. What do you understand by the jagged array?

A jagged array is a multidimensional array in which member arrays are of different sizes. For example, int array[][]=new int[3][]. The statement creates a two-dimensional jagged array.

Q. What is an anonymous array also give an example?

Array reference that is not stored in a variable. It is used in the construction of other objects. Java's Polygon class has a constructor that parses anonymous array as a parameter.

```
Polygon(int[] xvalues, int[] yvalues, int n)
```

Q. What happens if we declare an array without assigning the size?

It is not possible to declare an array without size. When we declare an array without assigning the size, it throws the compile-time error. For example, height=new int[].

Q. What is the difference between Array and ArrayList?

Array: Array is static. It is of fixed size. Its size cannot be changed once it is declared. It contains both primitive data types and objects of a class. Array does not have generic features.

ArrayList: ArrayList is dynamic in size. Its size or capacity automatically grows when we add element into it. It contains only the object entries. It has a generic feature.

Q. Write a program to check whether two given Arrays are equal, given both contains same data type and same length ?

```
import java.util.*;
public class JavaHungry {

    public static void main(String args[]) {

    int[]  arr1 = {2, 3, 4};
    int[]  arr2 = {1, 2, 3};

    System.out.println(Arrays.equals(arr1 , arr2));

    int[]  arr3 = {2, 3, 4};

    System.out.println(Arrays.equals(arr1 , arr3));
    }
}
```

Q. What is the difference between Array and LinkedList in java ? Imp

Memory required for storing the same number of elements in Array is less as compared to LinkedList. Array only stores the data of the element whereas LinkedList stores data plus the address of the next node.

Array requires contiguous memory allocation where as LinkedList elements are present all over the heap memory. Unlike Array, LinkedList does not have limitation of contiguous memory.

Q. Find out smallest and largest number in a given Array?

Logic to find the smallest and largest number in a given Array is given below :

a. Create two variables for storing largest and smallest number.

b. Initialize smallest variable with value Integer.MAX_VALUE

c. Initialize largest variable with value Integer.MIN_VALUE

d. In each traversal of for loop, we will compare the current element with the largest and smallest number. We will update the value.

e. If a number is larger than largest, then it can not be smaller than the smallest. So we can skip if first condition is true.

```
import java.util.*;

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

    // Given Array
    int[] inputArr = {10,43,27,98,75,59,191};

    // Setting largest value
    int largest = inputArr[0];

    // Setting smallest value
    int smallest = inputArr[0];

    // Iterate through the Given Array
    for( int number : inputArr ) {
        if(number > largest) {
            largest = number;
        }
        else if (smallest > number) {
            smallest = number;
        }
    }
    System.out.println("Largest and Smallest numbers are " + largest +" "+smallest);
    }
}


Output :
Largest and Smallest numbers are 191 10
```

Q. How to reverse an Array in java ?

1. Swap the values of left indices with the right indices till the mid element.

```
import java.util.*;
public class ReverseArray
{
    public static void main (String[] args) throws java.lang.Exception
    {
        // Given input array
        int[] inputArray = {3,7,9,6,4};
        // Print array before reverse
        System.out.println("Array without reverse" +
                            Arrays.toString(inputArray));
        // Calling method to swap elements
        reverseArray(inputArray);
    }
    public static void reverseArray(int[] inputArray) {
        for (int left = 0, right = inputArray.length - 1;
                        left < right; left++, right--) {
            // swap the values at the left and right indices
            int temp = inputArray[left];
            inputArray[left]  = inputArray[right];
            inputArray[right] = temp;
        }
        // Printing the Array after reverse
        System.out.print("Reverse Array :");
        for(int val : inputArray)
            System.out.print(" "+val);
    }
}


Output :
Array without reverse[3, 7, 9, 6, 4]
Reverse Array : 4 6 9 7 3
```

Q. Code to Reverse String Array in Java

1. Convert the String **A**rray to the list using Arrays.asList() method.

2. Reverse the list using Collections.reverse() method

3. Convert the list back to the array using list.toArray() method.

```
import java.util.*;
public class ReverseArray
{
    public static void main (String[] args) throws java.lang.Exception
    {
        // Given input array
        String[] inputArray = {"India","USA","Germany","Australia"};
        // Print array before reverse
        System.out.println("Array without reverse : " +
                            Arrays.toString(inputArray));
        // Calling method to swap elements
        reverseArray(inputArray);
    }
   public static void reverseArray(String[] arr) {
        // Converting Array to List
        List<String> list = Arrays.asList(arr);
        // Reversing the list using Collections.reverse() method
        Collections.reverse(list);
        // Converting list back to Array
        String[] reversedArray = list.toArray(arr);
        // Printing the reverse Array
        System.out.print("Reverse Array : " + Arrays.toString(reversedArray));
   }
}

Output:
Array without reverse : [India, USA, Germany, Australia]
    Reverse Array : [Australia, Germany, USA, India]
```

Q. How to convert HashSet to Array in java ?

You can convert HashSet to Array using toArray() method.

Q. How to convert Array to TreeSet in java ?

To convert Array to TreeSet in java, first we need to convert Array to List using Arrays class asList() method. After converting Array to List, pass the list object to TreeSet constructor. That's it , Array has been converted to TreeSet. You can confirm by printing out the values of TreeSet object.

29. File Handling

Q. What is File Handling in Java?

Java FileWriter and FileReader classes are used to write and read data from text files (they are **Character Stream** classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

FileWriter:

FileWriter is useful to create a file writing characters into it.

- This class inherits from the OutputStream class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
- FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
- FileWriter creates the output file , if it is not present already.

Constructors:

- **FileWriter(File file) –** Constructs a FileWriter object given a File object.
- **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
- **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
- **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
- **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

Methods:

- **public void write (int c) throws IOException –** Writes a single character.
- **public void write (char [] stir) throws IOException –** Writes an array of characters.
- **public void write(String str)throws IOException –** Writes a string.
- **public void write(String str,int off,int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is number of character to write.
- **public void flush() throws IOException** flushes the stream
- **public void close() throws IOException** flushes the stream first and then closes the writer.

Reading and writing take place character by character, which increases the number of I/O operations and effects performance of the system.**BufferedWriter** can be used along with FileWriter to improve speed of execution.

```
// Creating a text File using FileWriter
import java.io.FileWriter;
import java.io.IOException;
class CreateFile
{
  public static void main(String[] args) throws IOException
  {
    // Accept a string
    String str = "File Handling in Java using "+
        " FileWriter and FileReader";

    // attach a file to FileWriter
    FileWriter fw=new FileWriter("output.txt");
    // read character wise from string and write
    // into FileWriter
    for (int i = 0; i < str.length(); i++)
      fw.write(str.charAt(i));

    System.out.println("Writing successful");
    //close the file
    fw.close();
  }
}
```

File Reader:

FileReader is useful to read data in the form of characters from a ‘text’ file.

- This class inherit from the InputStreamReader Class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
- FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Constructors:

- **FileReader(File file) –** Creates a FileReader , given the File to read from
- **FileReader(FileDescripter fd) –** Creates a new FileReader , given the FileDescripter to read from
- **FileReader(String fileName) –** Creates a new FileReader , given the name of the file to read from

Methods:

- **public int read () throws IOException –** Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
- **public int read(char[] cbuff) throws IOException –** Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
- **public abstract int read(char[] buff, int off, int len) throws IOException –**Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.Parameters:cbuf – Destination bufferoff – Offset at which to start storing characterslen – Maximum number of characters to read
- **public void close() throws IOException** closes the reader.
- **public long skip(long n) throws IOException –**Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.Parameters:n – The number of characters to skip

```
// Reading data from a file using FileReader
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
class ReadFile
{
  public static void main(String[] args) throws IOException
  {
    // variable declaration
    int ch;

    // check if File exists or not
    FileReader fr=null;
    try
    {
      fr = new FileReader("text");
    }
    catch (FileNotFoundException fe)
    {
      System.out.println("File not found");
    }

    // read from FileReader till the end of file
    while ((ch=fr.read())!=-1)
      System.out.print((char)ch);

    // close the file
    fr.close();
  }
}
```

30. IIB

In a Java program, operations can be performed on methods, constructors and initialization blocks. Instance Initialization Blocks or IIB are used to initialize instance variables . So firstly, constructor is invoked and the java compiler copies the instance initializer block in the constructor after the first statement super(). They run each time when object of the class is created.

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

```
// Java program to illustrate
// Instance Initialization Block
class GfG {
  // Instance Initialization Block
  {
    System.out.println("IIB block");
  }

  // Constructor of GfG class
  GfG() { System.out.println("Constructor Called"); }
  public static void main(String[] args)
  {
    GfG a = new GfG();
  }
}

Output:
IIB block
Constructor Called
```

Instance Initialization Block with parent class

You can have IIBs in parent class also. Instance initialization block code runs immediately after the call to super() in a constructor. The compiler executes parents class's IIB before executing current class's IIBs.

```
// Java program to illustrate
// Instance Initialization Block
// with super()

// Parent Class
class B {
  B() { System.out.println("B-Constructor Called"); }

  {
    System.out.println("B-IIB block");
  }
}

// Child class
class A extends B {
  A()
  {
    super();
    System.out.println("A-Constructor Called");
  }
  {
    System.out.println("A-IIB block");
  }

  // main function
  public static void main(String[] args)
  {
    A a = new A();
  }
}
```

31. SIB

While hearing ‘Static’ itself, we will be sure that the state of this will be static, i.e, it does not depends on object (values will not vary for each object). For accessing static things (variable, method, block, class) no need to create an instances.

SIB is used to initialize Static variables. It is execute when a class is first loaded, before loading main method. It is executed for every class loading.

This is normal block and all the code will be enclosed in {} and it should preceded with **static** keyword.

```
public class staticblock {

                static int i;
    int j;

    // start of static block
    static {
        i = 10;
  // j = 5; : compilation error :  cannot make a static reference to non-static field 'j'
        System.out.println("First static block called .........i="+i);
    }
    // end of static block

    staticblock(){
        System.out.println("Constructor called .......value of i....." +i);
    }
    void test() {
                staticblock.i = staticblock.i +5; // i = i+5 also correct
                System.out.println("inside method test  .......value of i....." +i);
    }
    static {
        System.out.println("Second static block .........i="+i);
    }
    public static void main(String args[]) {

    System.out.println(" Main is called .....");
    System.out.println("value of satic variable from main class ..i ="+i);
                staticblock s = new staticblock();
                s.test();
                staticblock s1 = new staticblock();
                s1.test();
                staticblock.staticMethod2();

    }
    static {
        staticMethod();
     //test(); : compilation error:can't make refernce to non-static method
```

```
        System.out.println("Third static block........i="+i);
    }

    public static void staticMethod() {
                i = i +5;
        System.out.println("This is static method...i="+i);
    }

    public static void staticMethod2() {
        System.out.println("This is static method2...i="+i);
    }
}
```

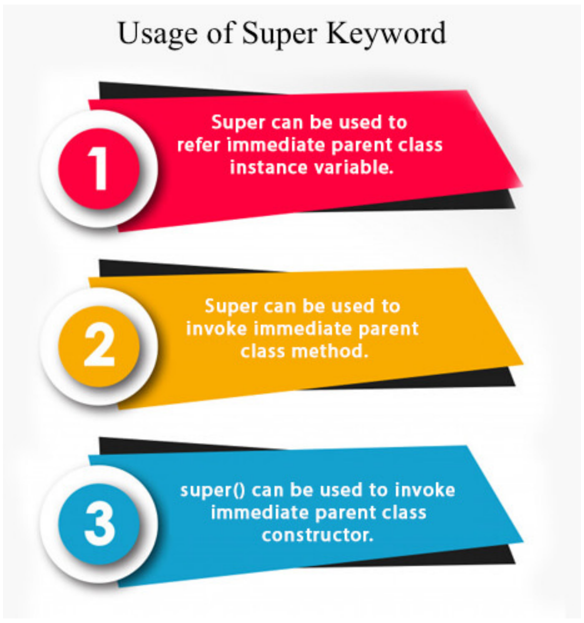
32. Super

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

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.


Usage of 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.

Real Use of Super Keyword:

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}

Output:

1 ankit 45000
```

Q. Can we call parent class method in sub class by using super keyword.

Yes, We can access parent class method in child class by using super keyword.

Example:

```
class Parent
{
void display()
{
System.out.println("parent");
}
}
class Child extends Parent
{
void display()
{
System.out.println("child");
super.display();
}
public static void main(String args[])
{
Child c = new Child();
c.display();
}
}

Output: child
        parent
```

Q. Can we call parent class constructor in sub class constructor i.e constructor chaining by using super keyword?

Yes, We can call parent class constructor in sub class constructor by using super keyword but super keyword must be the first statement in sub class constructor.

Q. Can we use both "this" and "super" in constructor?

No, It is not possible in java we cannot use both this and super keyword together in java constructor because this and super should be the first statement in any java constructor.

Q. What is difference between this() and super()?

Java this()

- It is used to access current class constructor.
- It can be used inside another constructor of same class.
- It can be used to remove ambiguity error when we have data members and local are same name.

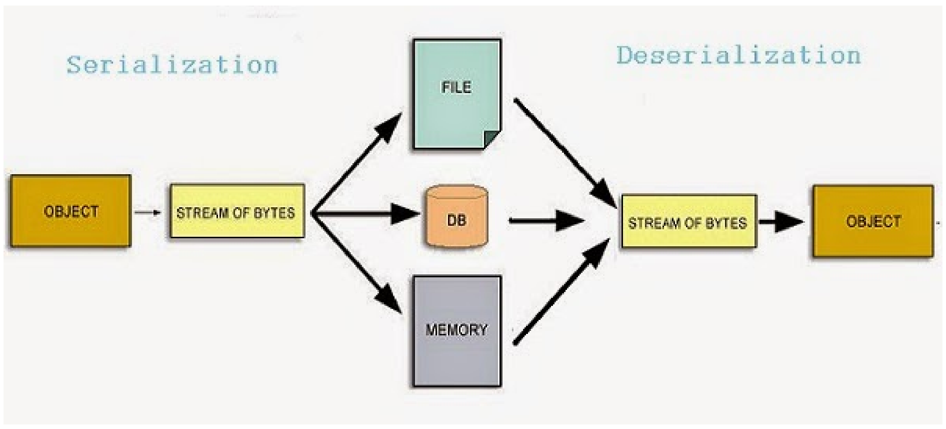
For example: Remove ambiguity error

```
class Demo
{
int a;
int b;
Demo(int a, int b)
{
this.a = a;
this.b = b;
}
}
}
```

Java super()

- It is used to access parent class constructor from sub class.
- It can be used to only inside child class constructor.
- It doesn't remove any ambiguity error from programs.

33. Serialization and deserialization



Q. What is Serialization in Java?

Object Serialization in Java is a process used to convert Object into a binary format which can be persisted into a disk or sent over the network to any other running Java virtual machine; the reverse process of creating object from the binary stream is called deserialization in Java. Java provides Serialization API for serializing and deserializing object which includes java.io.Serializable, java.io.Externalizable, ObjectInputStream and ObjectOutputStream etc.

Q. How to make a Java class Serializable? Imp

Making a class Serializable in Java is very easy, Your Java class just needs to implements java.io.Serializable interface and JVM will take care of serializing objects in the default format.

Q. Can you Serialize static variables?

No, you can't. As you know static variable are at class level not at object level and you serialize a object so you can't serialize static variables.

Q. Is it necessary to implement Serializable interface if you want to serialize any object?

Yes, it is necessary to implement Serializable interface if you want to serialize any object. Serializable is marker interface. Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface.

Q. What is deserialization in Java?

Deserialization is precisely the opposite of serialization. With deserialization, you have a byte stream and you recreate the object in the same state as when you serialized it. This means that you need to have the actual definition of the object to accomplish the recreation.

Q. How does Java deserialization work?

When deserializing a byte stream back to an object it does not use the constructor. It creates an empty object and uses reflection to write the data to the fields. Just like with serialization, private and final fields are also included.

Q. What is a Java deserialize vulnerability?

A Java deserialize vulnerability is a security vulnerability that occurs when a malicious user tries to insert a modified serialized object into the system that eventually compromises the system or its data. Think of arbitrary code execution that can be triggered when deserializing a serialized object. To better explain Java deserialize vulnerabilities, we first need to explore how deserialization works in Java.

34. Transient Keyword

Q. What is Transient keyword in Java?

Transient is basically a variables modifier that used for serialization. Now, what is Serialization? *Serialization in Java* is a mechanism that is used in converting the state of an object into a byte stream. At the time of serialization, if you don't want to save the value of a particular variable in a file, then use the transient keyword.

Syntax:

```
private transient <member variable>;
OR
transient private <member variable>;
```

In case you define any data member as transient, it will not be serialized. This is because every field marked as **transient** will not be serialized. You can use this transient keyword to indicate the Java virtual machine (JVM) that the transient variable is not part of the persistent state of an object.

```
class Demo implements Serializable
{
// Making human transient
private transient String human;
transient int age;
// serialize other fields
private String name, address;
Date dob;
// rest of the code
}
```

Q. Why is Transient modifier used?

Transient in Java is used to indicate that a field should not be part of the serialization process.

The modifier Transient can be applied to member variables of a class to turn off serialization on these member variables. Every field that is marked as transient will not be serialized. You can use this transient keyword to indicate to the Java virtual machine that the transient variable is not part of the persistent state of an object.

Q. How to use Transient with Final keyword?

Transient in Java can be used with the *final keyword* because it behaves differently in different situations which is not generally the case with other *keywords in Java*.

Have a look at this example.

```
private String
firstName;
private String
lastName;

//final field 1

public final transient String pass= "password";

//final field 2

public final transient Lock lock = Lock.getLock("demo");
```

Q. Difference between Transient and Volatile

Volatile and Transient are two completely different keywords that are used in *Java*. A Transient keyword is used during serialization of Java object. Volatile is related to the visibility of variables modified by multiple threads.

The only similarity between these keywords is that they are less used or uncommon keywords and not as popular as public, static or final.

35. Finally , Practical Example

The **finally** keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you just wish to execute, despite what happens within the protected code.

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

      try{
         int a = 10;
         int b = 0;
         int result = a/b;
      }catch(Exception e){
         System.out.println("Error: "+ e.getMessage());
      }
      finally{
           System.out.println("Finished.");
      }
   }
}
Output
Error: / by zero
Finished.
```

Difference between final, finally, finalize

Sr. no.	Key	final	finally	finalize
1.	Definition	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.	Applicable to	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.	Functionality	(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.	Execution	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 destroyed.

36. How to use the try/catch block to catch exceptions

How to call a method that throws an exception in catch block?

In my original class, I have lot of methods that have this format... I am trying to come up with a cleaner way to write the catch blocks. I think the issue is more to do with understanding how the exceptions are handled in applications; its a design issue, in general.

https://stackoverflow.com/questions/54701649/how-to-call-a-method-that-throws-an-exception-in-catch-block

Place any code statements that might raise or throw an exception in a `try` block, and place statements used to handle the exception or exceptions in one or more `catch` blocks below the `try` block. Each `catch` block includes the exception type and can contain additional statements needed to handle that exception type.

37. Strings

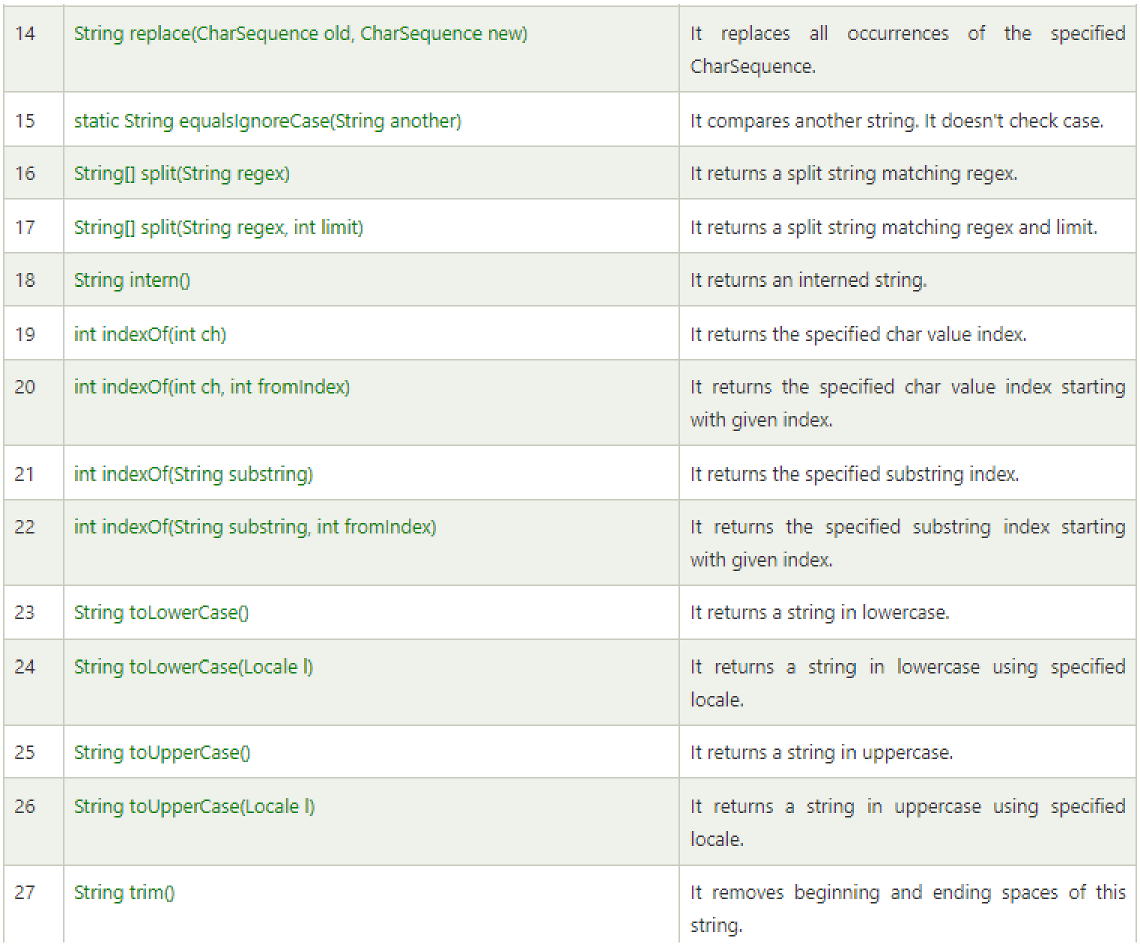
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);

is same as:

String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

String Methods

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.



14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.

Q. Is *String* a Primitive or a Derived Type?

A String is a derived type since it has state and behavior. For example, it has methods like *substring()*, *indexOf()*, and *equals(),* which primitives cannot have.

But, since we all use it so often, it has some special characteristics that make it feel like a primitive:

- While strings are not stored on the call stack like primitives are, they are **stored in a special memory region called the string pool**
- Like primitives, we can use the + operator on strings
- And again, like primitives, we can create an instance of a *String* without the *new* keyword

Q. What Are the Benefits of Strings Being Immutable?

According to an interview by James Gosling, strings are immutable to improve performance and security.

And actually, we see several benefits to having immutable strings:

- **The string pool is only possible if the strings, once created, are never changed,** as they are supposed to be reused
- The code can **safely pass a string to another method**, knowing that it can't be altered by that method
- Immutably **automatically makes this class thread-safe**
- Since this class is thread-safe, **there is no need to synchronize common data**, which in turn improves performance
- Since they are guaranteed to not change, **their hashcode can be easily cached**

Q. How Is a String Stored in Memory?

According to the JVM Specification, *String* literals are stored in a runtime constant pool, which is allocated from the JVM's method area.

Although the method area is logically part of the heap memory, the specification does not dictate the location, memory size, or garbage collection policies. It can be implementation-specific.

This runtime constant pool for a class or interface is constructed when the class or interface is created by the JVM.

Q. Are Interned Strings Eligible for Garbage Collection in Java?

Yes, all *String*s in the string pool are eligible for garbage collection if there are no references from the program.

Q. Is String Thread-Safe? How?

Strings are indeed completely thread-safe because they are immutable. Any class which is immutable automatically qualifies for thread-safety because its immutability guarantees that its instances won't be changed across multiple threads.

For example, if a thread changes a string's value, a new *String* gets created instead of modifying the existing one.

Q. How Can We Compare Two Strings in Java? What's the Difference Between *str1* == *str2* and *str1.equals(str2)*?

We can compare strings in two different ways: by using equal to operator (==) and by using the *equals()* method.

Both are quite different from each other:

- **The operator (*str1* == *str2*)** checks for referential equality
- **The method (*str1.equals(str2)*)** checks for lexical equality

Though, it's true that if two strings are lexically equal, then *str1.intern() == str2.intern()* is also *true*.

Typically, for comparing two *Strings* for their content, we should always use *String.equals*.

Q. Difference Between String, Stringbuffer and Stringbuilder? Imp

Strings are immutable. This means that **if we try to change or alter its values, then Java creates an absolutely new *String*.**

For example, if we add to a string *str1* after it has been created:

```
String str1 = "abc";
str1 = str1 + "def";
```

Then the JVM, instead of modifying *str1*, creates an entirely new *String*.

However, for most of the simple cases, the compiler internally uses *StringBuilder* and optimizes the above code.

But, for more complex code like loops, **it will create an entirely new *String*, deteriorating performance**. This is where *StringBuilder* and *StringBuffer* are useful.

Both *StringBuilder* and *StringBuffer* in Java create objects that hold a mutable sequence of characters. ***StringBuffer* is synchronized and therefore thread-safe whereas *StringBuilder* is not.**

Since the extra synchronization in *StringBuffer* is typically unnecessary, we can often get a performance boost by selecting *StringBuilder.*

Q. Why Is It Safer to Store Passwords in a Char[] Array Rather Than a String?

Since strings are immutable, they don't allow modification. **This behavior keeps us from overwriting, modifying, or zeroing out its contents, making *Strings* unsuitable for storing sensitive information.**

We have to rely on the garbage collector to remove a string's contents. Moreover, in Java versions 6 and below, strings were stored in PermGen, meaning that once a *String* was created, it was never garbage collected.

By using a *char[]* array, we have complete control over that information. We can modify it or wipe it completely without even relying on the garbage collector.

Using *char[]* over *String* doesn't completely secure the information; it's just an extra measure that reduces an opportunity for the malicious user to gain access to sensitive information.

Q. What Is the String Constant Pool? Imp

The string pool, also known as the *String* constant pool or the *String* intern pool, is a special memory region where the JVM stores *String* instances.

It optimizes application performance by reducing how often and how many strings are allocated:

- The JVM stores only one copy of a particular *String* in the pool
- When creating a new *String*, the JVM searches in the pool for a *String* having the same value
- If found, the JVM returns the reference to that *String* without allocating any additional memory
- If not found, then the JVM adds it to the pool (interns it) and returns its reference

Q. String objects stored in string constant pool or stack?

String literals are not stored on the stack. Never. In fact, no objects are stored on the stack.

String literals (or more accurately, the String objects that represent them) ~~are~~ were historically stored in a Heap called the "permgen" heap. (Permgen is short for permanent generation.)

Under normal circumstances, String literals and much of the other stuff in the permgen heap are "permanently" reachable, and are not garbage collected. (For instance, String literals are always reachable from the code objects that use them.) However, you can configure a JVM to attempt to find and collect dynamically loaded classes that are no longer needed, and this may cause String literals to be garbage collected.

Q. Convert a String to Character array in Java

```
// Java program to Convert a String
// to a Character array using Naive Approach

import java.util.*;

public class GFG {

  public static void main(String args[])
  {
    String str = "GeeksForGeeks";

    // Creating array of string length
    char[] ch = new char[str.length()];

    // Copy character by character into array
    for (int i = 0; i < str.length(); i++) {
      ch[i] = str.charAt(i);
    }

    // Printing content of array
    for (char c : ch) {
      System.out.println(c);
    }
  }
}
Output:
G
```

```
e
e
k
s
F
o
r
G
e
e
k
s
```

Method 2: Using **toCharArray()**

Method

- Step 1:Get the string.
- Step 2:Create a character array of same length as of string.
- Step 3:Store the array return by toCharArray() method.
- Step 4:Return or perform operation on character array.

```
// Java program to Convert a String
// to a Character array using toCharArray()

import java.util.*;

public class GFG {

  public static void main(String args[])
  {

    String str = "GeeksForGeeks";

    // Creating array and Storing the array
    // returned by toCharArray()
    char[] ch = str.toCharArray();

    // Printing array
    for (char c : ch) {
      System.out.println(c);
    }
  }
}

Output:
G
e
e
k
s
F
o
r
G
e
e
k
s
```

38. Thread

Q. What is Thread in Java?

Threads are basically the lightweight and smallest unit of processing that can be managed independently by a scheduler. Threads are referred to as parts of a process that simply let a program execute efficiently with other parts or threads of the process at the same time. Using threads, one can perform complicated tasks in the easiest way. It is considered the simplest way to take advantage of multiple CPUs available in a machine. They share the common address space and are independent of each other.

Q. What are the two ways of implementing thread in Java?

There are basically two ways of implementing thread in java as given below:

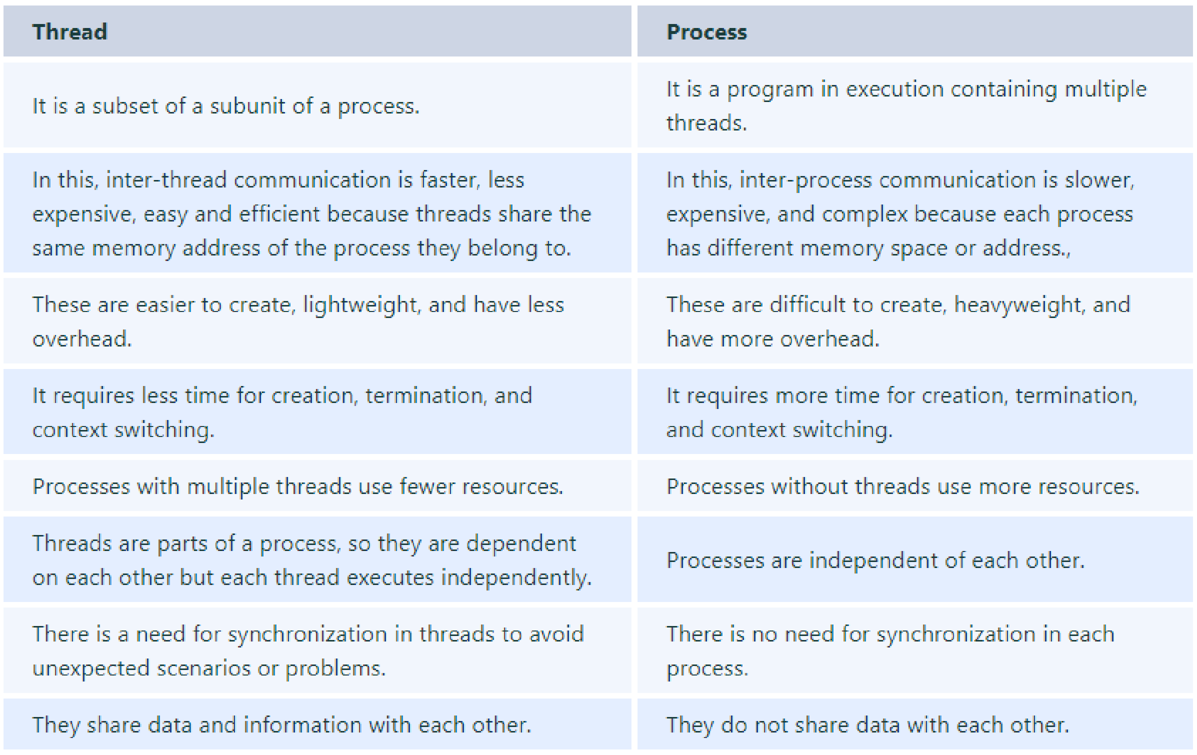
- Extending the **Thread** class

```
class MultithreadingDemo extends Thread
{
  public void run()
 {
     System.out.println("My thread is in running state.");
 }
  public static void main(String args[])
 {
    MultithreadingDemoobj=new MultithreadingDemo();
        obj.start();
  }
}
Output:

My thread is in running state.
```

- Implementing **Runnable** interface in Java

```
class MultithreadingDemo implements Runnable
{
   public void run()
 {
      System.out.println("My thread is in running state.");
  }
    public static void main(String args[])
 {
      MultithreadingDemo obj=new MultithreadingDemo();
      Threadtobj =new Thread(obj);       tobj.start();
 }
}
Output:

My thread is in running state.
```

Q. Thread and Process



Thread	Process
It is a subset of a subunit of a process.	It is a program in execution containing multiple threads.
In this, inter-thread communication is faster, less expensive, easy and efficient because threads share the same memory address of the process they belong to.	In this, inter-process communication is slower, expensive, and complex because each process has different memory space or address.,
These are easier to create, lightweight, and have less overhead.	These are difficult to create, heavyweight, and have more overhead.
It requires less time for creation, termination, and context switching.	It requires more time for creation, termination, and context switching.
Processes with multiple threads use fewer resources.	Processes without threads use more resources.
Threads are parts of a process, so they are dependent on each other but each thread executes independently.	Processes are independent of each other.
There is a need for synchronization in threads to avoid unexpected scenarios or problems.	There is no need for synchronization in each process.
They share data and information with each other.	They do not share data with each other.

Q. What's the difference between class lock and object lock?

Class Lock: In java, each and every class has a unique lock usually referred to as a class level lock. These locks are achieved using the keyword 'static synchronized' and can be used to make static data thread-safe. It is generally used when one wants to prevent multiple threads from entering a synchronized block.

```
public class ClassLevelLockExample
{
  public void classLevelLockMethod()
 {
     synchronized (ClassLevelLockExample.class)
       {
            //DO your stuff here
       }
 }
}
```

Object Lock: In java, each and every object has a unique lock usually referred to as an object-level lock. These locks are achieved using the keyword 'synchronized' and can be used to protect non-static data. It is generally used when one wants to synchronize a non-static method or block so that only the thread will be able to execute the code block on a given instance of the class.

```
public class ObjectLevelLockExample
{
  public void objectLevelLockMethod()
 {
     synchronized (this)
       {
            //DO your stuff here
       }
 }
}
```

User Thread vs Daemon Thread

User Thread	Daemon Thread
JVM waits for user threads to finish their tasks before termination.	JVM does not wait for daemon threads to finish their tasks before termination.
These threads are normally created by the user for executing tasks concurrently.	These threads are normally created by JVM.
They are used for critical tasks or core work of an application.	They are not used for any critical tasks but to do some supporting tasks.
These threads are referred to as high-priority tasks, therefore are required for running in the foreground.	These threads are referred to as low priority threads, therefore are especially required for supporting background tasks like garbage collection, releasing memory of unused objects, etc.

Q. What are the wait() and sleep() methods?

wait(): As the name suggests, it is a non-static method that causes the current thread to wait and go to sleep until some other threads call the notify () or notifyAll() method for the object's monitor (lock). It simply releases the lock and is mostly used for inter-thread communication. It is defined in the object class, and should only be called from a synchronized context.

```
synchronized(monitor)
{
monitor.wait();       Here Lock Is Released by Current Thread
}
```

sleep(): As the name suggests, it is a static method that pauses or stops the execution of the current thread for some specified period. It doesn't release the lock while waiting and is mostly used to introduce pause on execution. It is defined in thread class, and no need to call from a synchronized context.

```
synchronized(monitor)
{
Thread.sleep(1000);     Here Lock Is Held by The Current Thread
//after 1000 milliseconds, the current thread will wake up, or after we call that is interrupt() method
}
```

Q. What's the difference between notify() and notifyAll()?

notify(): It sends a notification and wakes up only a single thread instead of multiple threads that are waiting on the object's monitor.

notifyAll(): It sends notifications and wakes up all threads and allows them to compete for the object's monitor instead of a single thread.

Q. What is Runnable and Callable Interface? Write the difference between them.

Both the interfaces are generally used to encapsulate tasks that are needed to be executed by another thread. But there are some differences between them as given below:

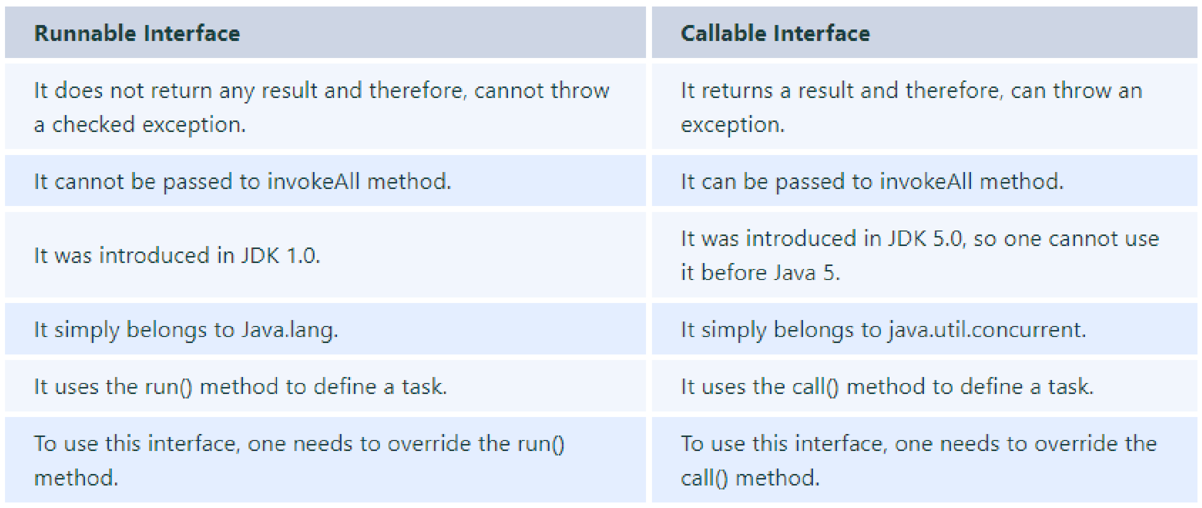
Running Interface

: This interface is basically available in Java right from the beginning. It is simply used to execute code on a concurrent thread.

Callable Interface

: This interface is basically a new one that was introduced as a part of the concurrency package. It addresses the limitation of runnable interfaces along with some major changes like generics, enum, static imports, variable argument method, etc. It uses generics to define the return type of object.

```
public interface Runnable
{
  public abstract void run();
}
public interface Callable<V>
{
V call() throws Exception;
}
```



Runnable Interface	Callable Interface
It does not return any result and therefore, cannot throw a checked exception.	It returns a result and therefore, can throw an exception.
It cannot be passed to invokeAll method.	It can be passed to invokeAll method.
It was introduced in JDK 1.0.	It was introduced in JDK 5.0, so one cannot use it before Java 5.
It simply belongs to Java.lang.	It simply belongs to java.util.concurrent.
It uses the run() method to define a task.	It uses the call() method to define a task.
To use this interface, one needs to override the run() method.	To use this interface, one needs to override the call() method.

Q. What is the start() and run() method of Thread class?

start()

: In simple words, the start() method is used to start or begin the execution of a newly created thread. When the start() method is called, a new thread is created and this newly created thread executes the task that is kept in the run() method. One can call the start() method only once.

run()

: In simple words, the run() method is used to start or begin the execution of the same thread. When the run() method is called, no new thread is created as in the case of the start() method. This method is executed by the current thread. One can call the run() method multiple times.

Q. Explain thread pool? Imp

A Thread pool is simply a collection of pre-initialized or worker threads at the start-up that can be used to execute tasks and put back in the pool when completed. It is referred to as pool threads in which a group of fixed-size threads is created. By reducing the number of application threads and managing their lifecycle, one can mitigate the issue of performance using a thread pool. Using threads, performance can be enhanced and better system stability can occur. To create the thread pools, java.util.concurrent.Executors class usually provides factory methods

Q. What’s the purpose of the join() method?

join() method is generally used to pause the execution of a current thread unless and until the specified thread on which join is called is dead or completed. To stop a thread from running until another thread gets ended, this method can be used. It joins the start of a thread execution to the end of another thread’s execution. It is considered the final method of a thread class.

Experienced candidate questions

Q. Can you start a thread twice?

No, it's not at all possible to restart a thread once a thread gets started and completes its execution. Thread only runs once and if you try to run it for a second time, then it will throw a runtime exception i.e., java.lang.IllegalThreadStateException.

```
public class TestThreadTwice1 extends Thread{
public void run(){
System.out.println(" thread is executing now........");
}
public static void main(String args[]){
TestThreadTwice1 t1=new TestThreadTwice1();
t1.start();
t1.start();
}
}

Output:

thread is executing now........
Exception in thread "main" java.lang.IllegalThreadStateException
```

Q. Explain context switching.?

Context switching is basically an important feature of multithreading. It is referred to as switching of CPU from one thread or process to another one. It allows multiple processes to share the same CPU. In context switching, the state of thread or process is stored so that the execution of the thread can be resumed later if required.

Q. What is Thread Scheduler and Time Slicing?

Thread Scheduler

: It is a component of JVM that is used to decide which thread will execute next if multiple threads are waiting to get the chance of execution. By looking at the priority assigned to each thread that is READY, the thread scheduler selects the next run to execute. To schedule the threads, it mainly uses two mechanisms: Preemptive Scheduling and Time slicing scheduling.

Time Slicing

: It is especially used to divide CPU time and allocate them to active threads. In this, each thread will get a predefined slice of time to execute. When the time expires, a particular thread has to wait till other threads get their chances to use their time in a round-robin fashion. Every running thread will get executed for a fixed time period.

Q. What is semaphore?

Semaphore is regarded as a thread synchronization construct that is usually required to control and manage the access to the shared resource using counters. It simply sets the limit of the thread. The semaphore class is defined within the package java.util.concurrent and can be used to send signals between threads to avoid missed signals or to guard critical sections. It can also be used to implement resource pools or bounded collection.

Q. Is it possible to call the run() method directly to start a new thread?

No, it's not possible at all. You need to call the start method to create a new thread otherwise run method won't create a new thread. Instead, it will execute in the current thread.

Q. Which is better runnable interface or thread class? Imp

- By extending Thread class
- By implementing Runnable interface

In the first approach, Our class always extends Thread class. There is no chance of extending any other class. 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.**

39. Wrapper Class

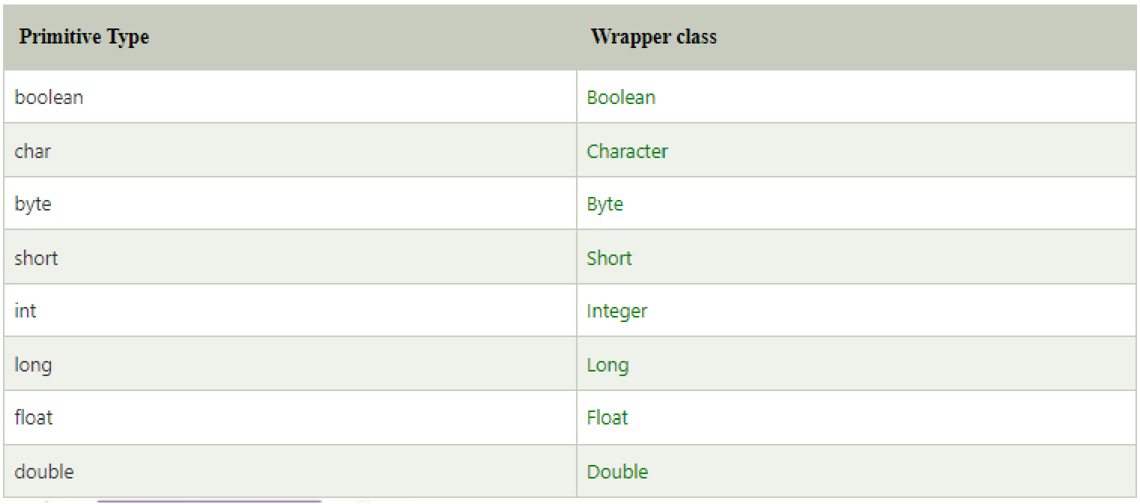
The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java:

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Q. Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

```
//Wrapper class Example: Primitive to Wrapper

//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
Output:

20 20 20
```

Q. Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
//Wrapper class Example: Wrapper to Primitive

//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

40. Finalize

Finalize() is the method of Object class. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

```
//Syntax
protected void finalize() throws Throwable
```

Throw

Throwable - the Exception is raised by this method

```
public class JavafinalizeExample1 {
     public static void main(String[] args)
    {
        JavafinalizeExample1 obj = new JavafinalizeExample1();
        System.out.println(obj.hashCode());
        obj = null;
        // calling garbage collector
        System.gc();
        System.out.println("end of garbage collection");

    }
    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}

Output:

2018699554
end of garbage collection
finalize method called
```

41. Throws

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

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

```
return_type method_name() throws exception_class_name{
//method code
}
```

Q. Which exception should be declared?

Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword:

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Example:

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

```
Output:

exception handled
normal flow...
```

42. Throw

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

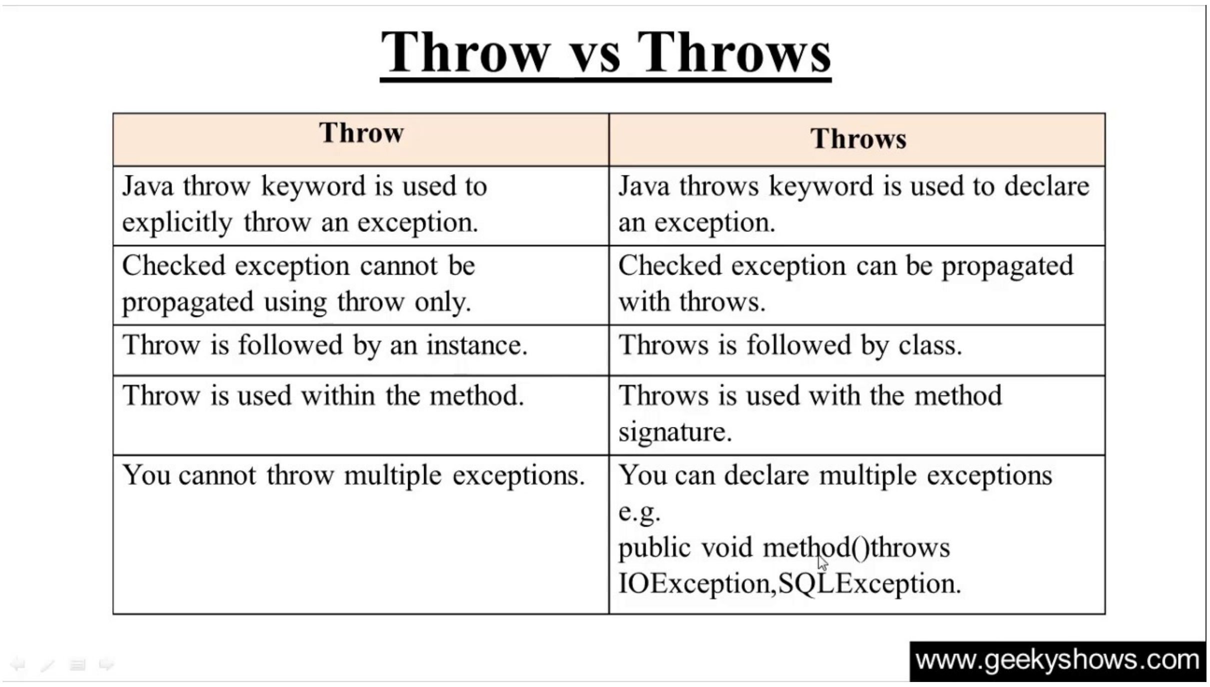
```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

43. Throwable

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.



Throw vs Throws

Throw	Throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

44. Regular Expressions

The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

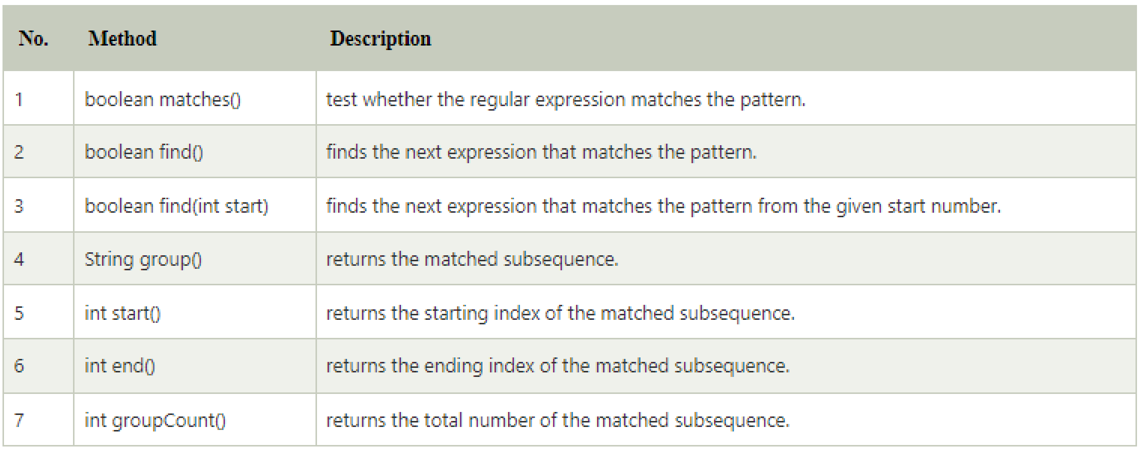
java.util.regex package

The Matcher and Pattern classes provide the facility of Java regular expression. The java.util.regex package provides following classes and interfaces for regular expressions.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

Matcher class

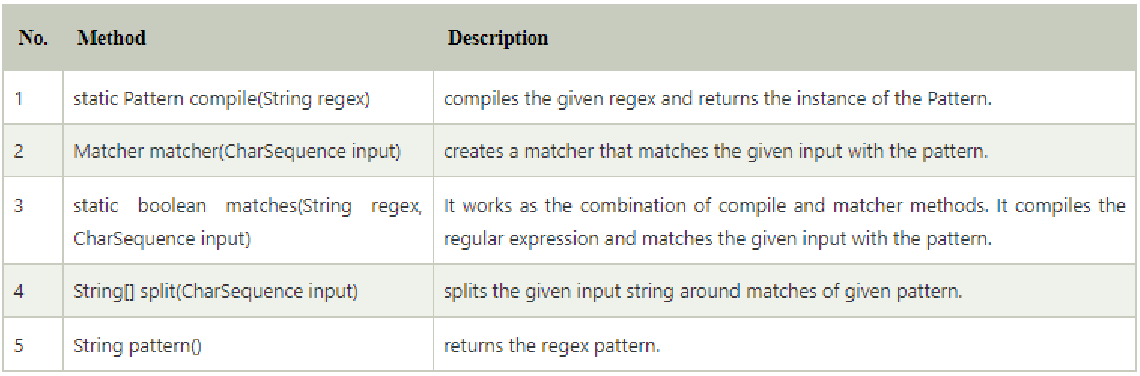
It implements the **MatchResult** interface. It is a *regex engine* which is used to perform match operations on a character sequence.



No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.
3	boolean find(int start)	finds the next expression that matches the pattern from the given start number.
4	String group()	returns the matched subsequence.
5	int start()	returns the starting index of the matched subsequence.
6	int end()	returns the ending index of the matched subsequence.
7	int groupCount()	returns the total number of the matched subsequence.

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.



No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Example:

```
import java.util.regex.*;
public class RegexExample1{
public static void main(String args[]){
//1st way
Pattern p = Pattern.compile(".s");//. represents single character
Matcher m = p.matcher("as");
boolean b = m.matches();

//2nd way
boolean b2=Pattern.compile(".s").matcher("as").matches();

//3rd way
boolean b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);
}}

Output
true true true
```

Regex Character classes

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

Q. Write a regex to split String by new line?

```
String lines[] = string.split("\\r?\\n");
```

Q. What is use of Dot(.) symbol in Java Regex?

The dot is used for matching any character. For example, the following regex represents "a number plus any other character":

```
[0-9].
```

Q. How to extract a substring using regex ?

Example - String test = "This is a test String and 'This is data we want'"

```
String data = "This is a test String and 'This is data we want'";
Pattern pattern = Pattern.compile("'(.*?)'");
Matcher matcher = pattern.matcher(data);
if (matcher.find())
{
    System.out.println(matcher.group(1));
}
```

Q. What is difference between matches() and find() in Java Regex?

Matches() returns true only if the whole string matches the specified pattern while find() returns trues even if a substring matches the pattern.

```
import java.util.regex.*;

public class RegexTutorial {
  public static void main(String[] args) {
        Pattern pattern = Pattern.compile("\\d");
        String test = "JavaInUse123";
        Matcher m = pattern.matcher(test);
        if (m != null){
            System.out.println(m.find());
            System.out.println(m.matches());
        }
    }
}
```

Q. How to replace all non-alphanumeric characters with empty strings?

`replaceAll("[^A-Za-z0-9]", "");`

Q. How to replace 2 or more spaces with single space in string and delete leading and trailing spaces?

Example - String test = " Test String " should be returned as "Test String"

```
replaceAll("\\s{2,}", " ").trim();
```

Q. Create a regular expression that accepts alphanumeric characters only. Its length must be five characters long only

```
import java.util.regex.*;

public class RegexTutorial {
  public static void main(String args[]) {
    System.out.println(Pattern.matches("[a-zA-Z0-9]{5}", "java1"));
    System.out.println(Pattern.matches("[a-zA-Z0-9]{5}", "java12"));
    System.out.println(Pattern.matches("[a-zA-Z0-9]{5}", "JA1Va"));
    System.out.println(Pattern.matches("[a-zA-Z0-9]{5}", "Java$"));
  }
}
```

Q. Create a regular expression that accepts 10 digit numeric characters starting with 1, 2 or 3 only.

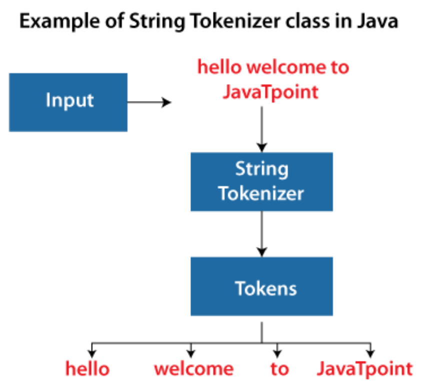
```
import java.util.regex.*;

public class RegexTutorial{

  public static void main(String args[]) {
    System.out.println("Regex Using character classes and quantifiers");

    System.out.println(Pattern.matches("[123]{1}[0-9]{9}", "1953038949"));
    System.out.println(Pattern.matches("[123][0-9]{9}", "1993038949"));

    System.out.println(Pattern.matches("[123][0-9]{9}", "9950389490"));
    System.out.println(Pattern.matches("[123][0-9]{9}", "695338949"));
    System.out.println(Pattern.matches("[123][0-9]{9}", "885338949"));

    System.out.println("Regex Using Metacharacters");
    System.out.println(Pattern.matches("[123]{1}\\d{9}", "2885338949"));
    System.out.println(Pattern.matches("[123]{1}\\d{9}", "685308949"));

  }
}
```

45. StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.


Example of String Tokenizer class in Java
Input
hello welcome to JavaTpoint
String Tokenizer
Tokens
hello
welcome
to
JavaTpoint

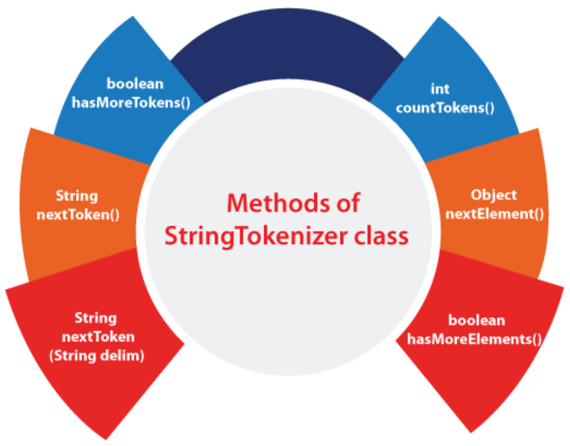
Constructors of the StringTokenizer Class

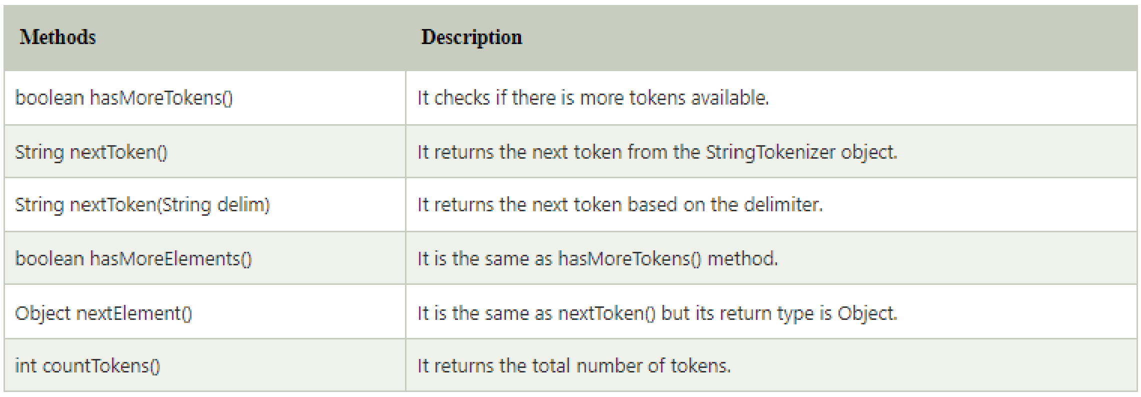
There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	It creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of the StringTokenizer Class

The six useful methods of the StringTokenizer class are as follows:


boolean hasMoreTokens()
int countTokens()
String nextToken()
Methods of StringTokenizer class
Object nextElement()
String nextToken (String delim)
boolean hasMoreElements()



Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.
int countTokens()	It returns the total number of tokens.

Example1 of StringTokenizer Class

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
Output:

my
name
is
khan
```

Example2 of StringTokenizer Class

```
import java.util.StringTokenizer;
public class StringToken {
public static void main(String[] args) {
String strname = new String(" Java is a high-level programming language originally developed by sun microsystem.");
StringTokenizer st1 = new StringTokenizer(strname," ") ;      //by using space” “ we are splitting string
while (st1.hasMoreTokens()) {
System.out.println(st1.nextElement());
}
}
}
output:

Java
is
a
high-level
programming
language
originally
developed
 by
 sun
 microsystem.
```

46. Cloning

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

```
protected Object clone() throws CloneNotSupportedException
```

Q. Why use clone() method ?

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

Advantage of Object cloning

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

Disadvantage of Object cloning

Following is a list of some disadvantages of clone() method:

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

```
class Student18 implements Cloneable{
int rollno;
String name;

Student18(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}
}

Output:101 amit
       101 amit
```

47. Collections

Java Collections Interview Questions

In Java, a collection is a framework that provides an architecture for storing and manipulating a collection of objects. In JDK 1.2, a new framework called "Collection Framework" was created, which contains all of the collection classes and interfaces.

https://www.interviewbit.com/java-collections-interview-questions/

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

A Collection represents a single unit of objects, i.e., a group.

Q. What is a framework in Java

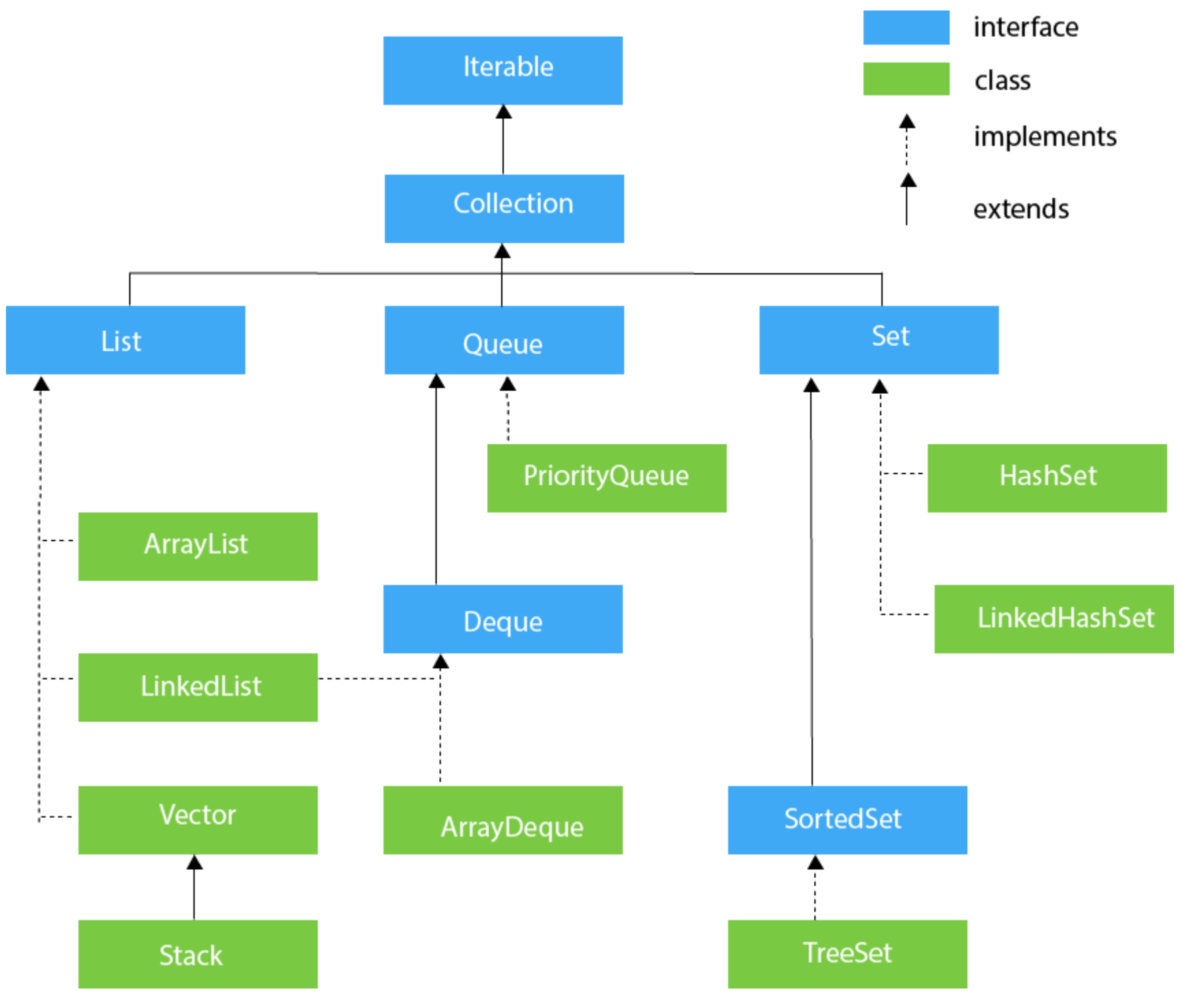
- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

Q. What is Collection framework

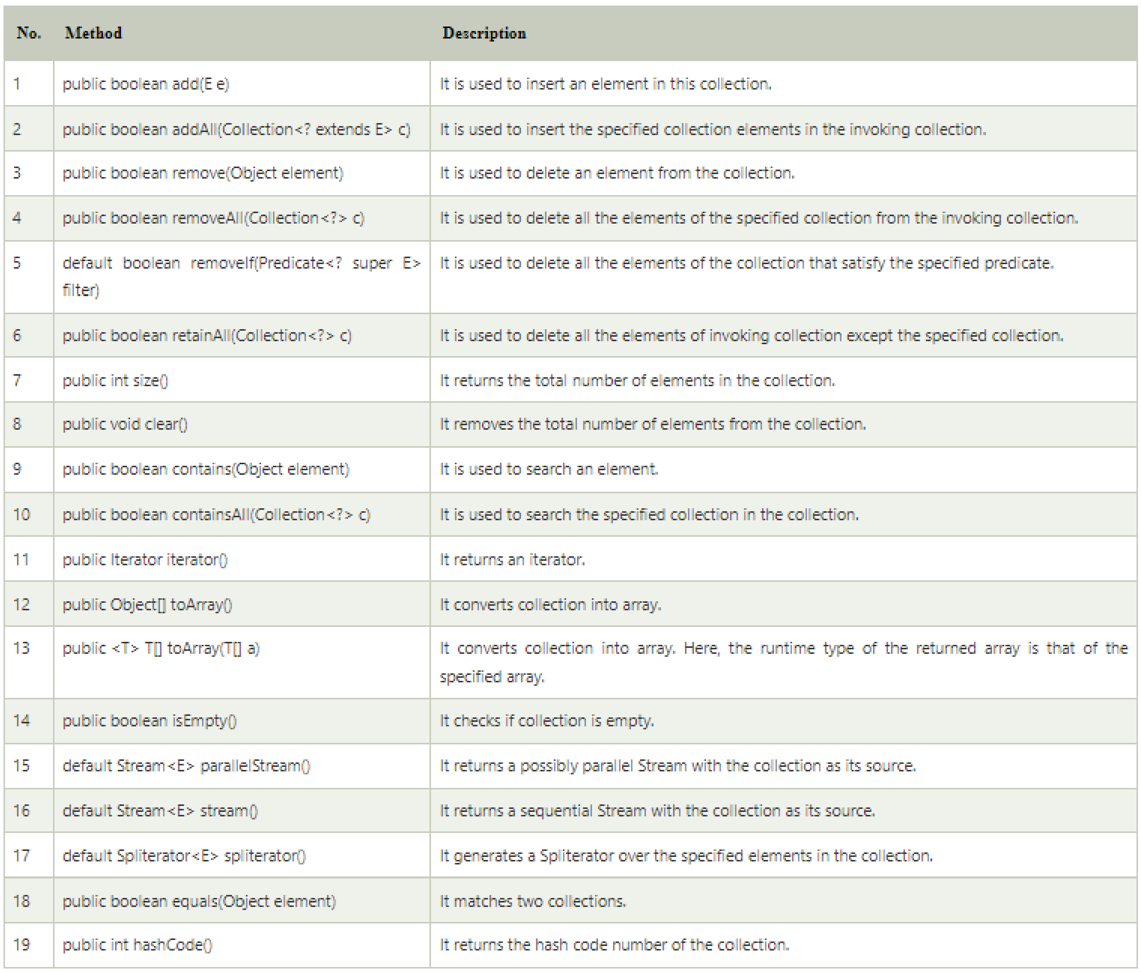
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework



Methods of Collection interface

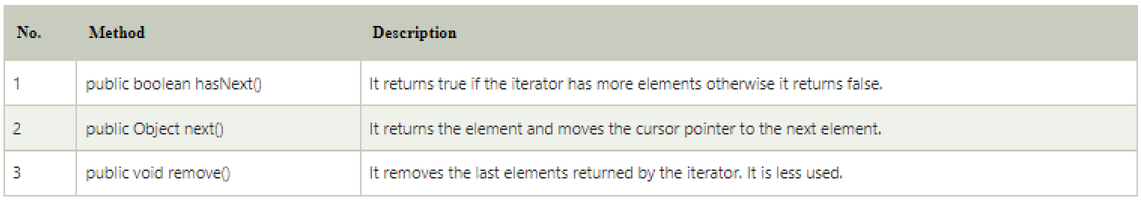


No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

Methods of Iterator interface

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



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

Iterable Interface

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

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T.

Collection Interface

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

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
```

```
List <data-type> list2 = new LinkedList();
```

```
List <data-type> list3 = new Vector();
```

```
List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
Output:

Ravi
Vijay
```

```
Ravi
Ajay
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
Output:

Ravi
Vijay
Ravi
Ajay
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}

Output:

Ayush
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
Output:

Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}

Output:

head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
Output:

Vijay
Ravi
Ajay
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}

Output:

Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}

Output:
```

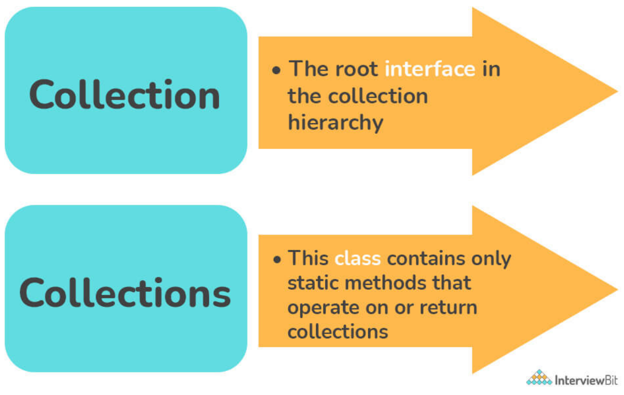
```
Ajay
Ravi
Vijay
```

Q. What is the difference between Array and Collection in java?

Array and Collection are equivalent in terms of storing object references and manipulating data, but they differ in a number of ways. The following are the primary distinctions between an array and a Collection:

Array	Collection
Arrays have a set size, which means that once we build one, we can't change it to meet our needs.	Collection are naturally grow-able and can be customized to meet our needs. We can change its size as per our requirement.
When it comes to performance, Arrays are the preferred to Collection.	Considering performance, Collection are not preferred to Arrays.
Only homogeneous data type elements can be stored in arrays.	Both homogeneous and heterogeneous components can be stored in a collection.
Because arrays have no underlying data structure, there is no ready-made method support.	Any collection class is built on a standard data structure, and so there is ready-made method support for every demand as a performance. These methods can be used directly, and we are not responsible for their implementation.
Objects and primitives can both be stored in arrays.	Only object types can be stored in a collection.
When it comes to memory, Arrays are not preferred to Collection.	Considering memory, Collection are preferred to Arrays.

Q. Differentiate between Collection and collections in the context of Java.


Collection
• The root interface in the collection hierarchy
Collections
• This class contains only static methods that operate on or return collections
InterviewBit

Collection	Collections
It's used to represent a collection of separate objects as a single entity.	It defines a number of useful methods for working with collections.
It is an interface.	It is a utility class.
Since Java 8, the Collection is an interface with a static function. Abstract and default methods can also be found in the Interface.	It only has static methods in it.

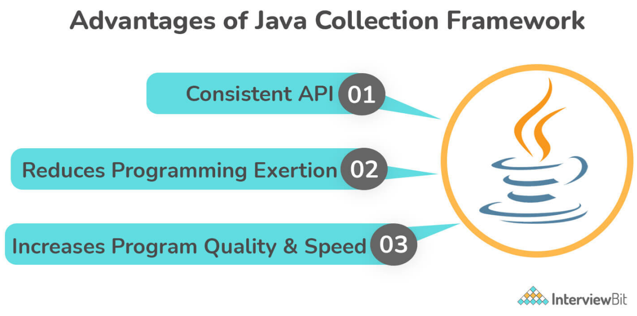
Q. What are the advantages of the Collection framework?

Following are the advantages of the Collection framework:-

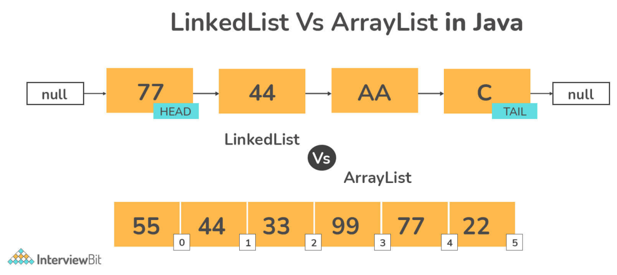
Consistent API: The API has a core set of interfaces like Collection, Set, List, or Map, and all the classes (ArrayList, LinkedList, Vector, and so on) that implement these interfaces have some common set of methods.

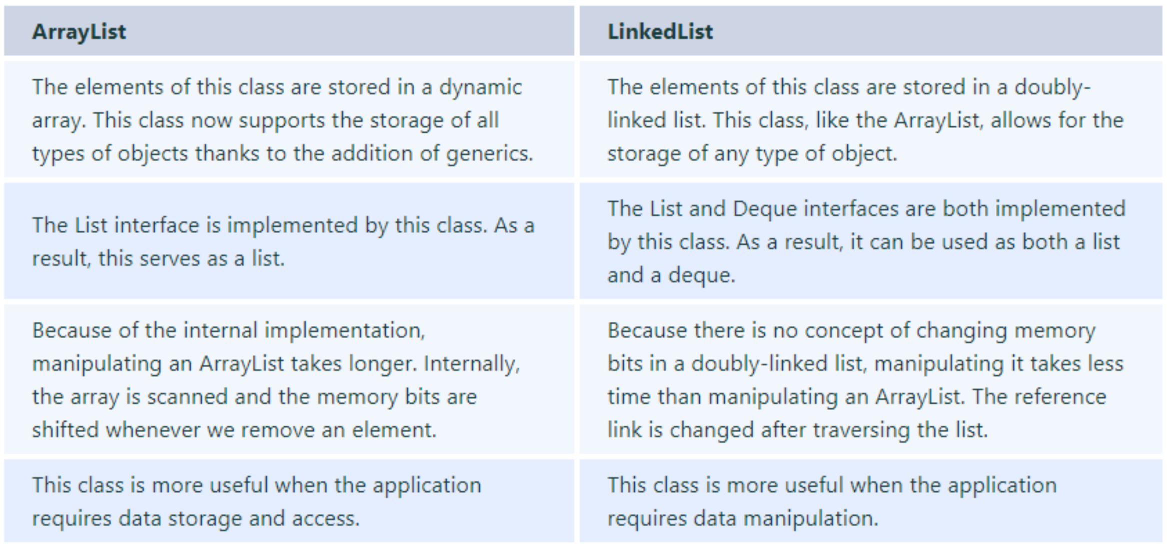
Cuts programming effort: Instead of worrying about the Collection's design, a programmer may concentrate on how best to use it in his program. As a result, the fundamental principle of Object-oriented programming (i.e. abstraction) has been applied successfully.

Improves program speed and quality by offering high-performance implementations of useful data structures and algorithms, as the programmer does not have to worry about the optimum implementation of a certain data structure in this scenario. They can simply use the best implementation to improve the performance of their program significantly.


Advantages of Java Collection Framework
Consistent API 01
Reduces Programming Exertion 02
Increases Program Quality & Speed 03
InterviewBit

Q. Difference between ArrayList and LinkedList.



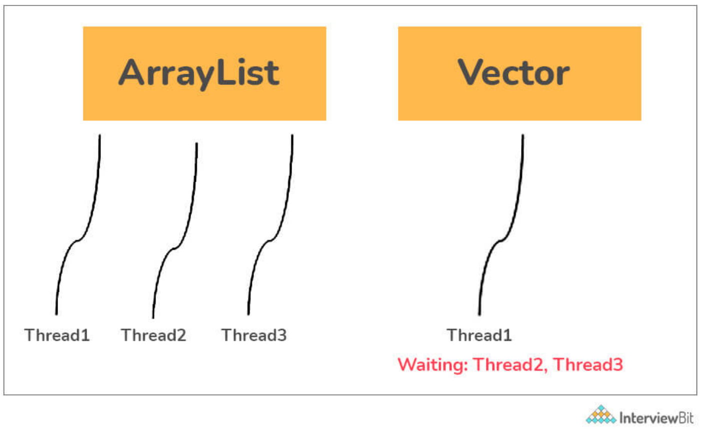


ArrayList	LinkedList
The elements of this class are stored in a dynamic array. This class now supports the storage of all types of objects thanks to the addition of generics.	The elements of this class are stored in a doubly-linked list. This class, like the ArrayList, allows for the storage of any type of object.
The List interface is implemented by this class. As a result, this serves as a list.	The List and Deque interfaces are both implemented by this class. As a result, it can be used as both a list and a deque.
Because of the internal implementation, manipulating an ArrayList takes longer. Internally, the array is scanned and the memory bits are shifted whenever we remove an element.	Because there is no concept of changing memory bits in a doubly-linked list, manipulating it takes less time than manipulating an ArrayList. The reference link is changed after traversing the list.
This class is more useful when the application requires data storage and access.	This class is more useful when the application requires data manipulation.

Q. Differentiate between ArrayList and Vector in java.

Following are the differences between ArrayList and Vector in java :

- Vector is synchronized, which means that only one thread can access the code at a time, however, ArrayList is not synchronized, which means that multiple threads can operate on ArrayList at the same time. In a multithreading system, for example, if one thread is executing an add operation, another thread can be performing a removal action. If multiple threads access ArrayList at the same time, we must either synchronize the code that updates the list fundamentally or enable simple element alterations. The addition or deletion of element(s) from the list is referred to as structural change. It is not a structural change to change the value of an existing element.



- **Data Growth**: Both ArrayList and Vector dynamically expand and shrink to make the most use of storage space, but the manner they do it is different. If the number of elements in an array exceeds its limit, ArrayList increments 50% of the current array size, while vector increments 100%, thereby doubling the current array size.
- **Performance**: ArrayList is faster than vector operations because it is non-synchronized, but vector operations are slower since they are synchronized (thread-safe). When one thread works on a vector, it acquires a lock on it, requiring any other threads working on it to wait until the lock is released.
- Vector can traverse over its elements using both Enumeration and Iterator, whereas ArrayList can only traverse using Iterator.

Q. Differentiate between List and Set in Java.

The List interface is used to keep track of an ordered collection. It is the Collection's child interface. It is an ordered collection of objects that allows for the storage of duplicate values. The insertion order is preserved in a list, which enables positional access and element insertion.

Set	List
It is an unordered sequence.	It is an ordered sequence.
Duplicate elements are not permitted in Set.	Duplicate elements are allowed in the list
Access to items from a certain position is not permitted.	Elements can be accessed based on their position.
A null element can only be stored once.	It is possible to store several null elements.

Q. Differentiate between Iterator and ListIterator in Java.

In Java's Collection framework, iterators are used to obtain elements one by one. It can be used on any type of Collection object. We can execute both read and remove operations using Iterator. Iterator must be used whenever we want to iterate elements in all Collection framework implemented interfaces, such as Set, List, Queue, and Deque, as well as all Map interface implemented classes. The only cursor accessible for the entire collection framework is the iterator.

ListIterator is only useful for classes that implement List collections, such as array lists and linked lists. It can iterate in both directions. When we wish to enumerate List elements, we must use ListIterator. This cursor has additional methods and capabilities than the iterator.

Iterator	ListIterator
Only has the ability to traverse components in a Collection in a forward direction.	In both forward and backward orientations, can traverse components in a Collection.
Iterators cannot be used to obtain indexes.	It offers methods to get element indexes at any time while traversing List, such as next Index() and previous Index().
It aids in the traversal of Maps, Lists, and Sets.	Only List may be traversed, not the other two.
It throws a Concurrent Modification Exception since it can't add elements.	At any time, you can quickly add elements to a collection.
next(), remove(), and has Next are some of the Iterator's functions ().	next(), previous(), has Next(), has Previous(), and add() are some of the List Iterator's methods

Q. Differentiate between HashSet and TreeSet. When would you prefer TreeSet to HashSet?

Following are the differences between HashSet and TreeSet:-

- **Internal implementation and speed**
    - **HashSet:** For search, insert, and remove operations, it takes constant time on average. TreeSet is slower than HashSet. A hash table is used to implement HashSet.
    - **TreeSet:** For search, insert, and delete, TreeSet takes O(Log n), which is higher than HashSet. TreeSet, on the other hand, preserves ordered data. Higher() (Returns the least higher element), floor(), ceiling(), and other operations are also supported. In TreeSet, these operations are likewise O(Log n), and HashSet does not implement them. A Self-Balancing Binary Search Tree is used to implement TreeSet (Red Black Tree). In Java, TreeSet is backed by TreeMap.
- **Way of storing elements** The elements of a HashSet are not ordered. In Java, the TreeSet class keeps objects in a Sorted order defined by the Comparable or Comparator methods. By default, TreeSet components are sorted in ascending order. It has a number of methods for dealing with ordered sets, including first(), last(), headSet(), tailSet(), and so on.
- **Allowing Null values** Null objects are allowed in HashSet. TreeSet does not allow null objects and throws a NullPointerException. This is because TreeSet compares keys using the compareTo() method, which throws java.lang. NullPointerException.
- **Comparison**HashSet compares two objects in a Set and detects duplicates using the equals() method. For the same purpose, TreeSet employs the compareTo() method. If equals() and compareTo() are not consistent, that is, if equals() returns true for two equal objects but compareTo() returns zero, the contract of the Set interface will be broken, allowing duplicates in Set implementations like TreeSet.

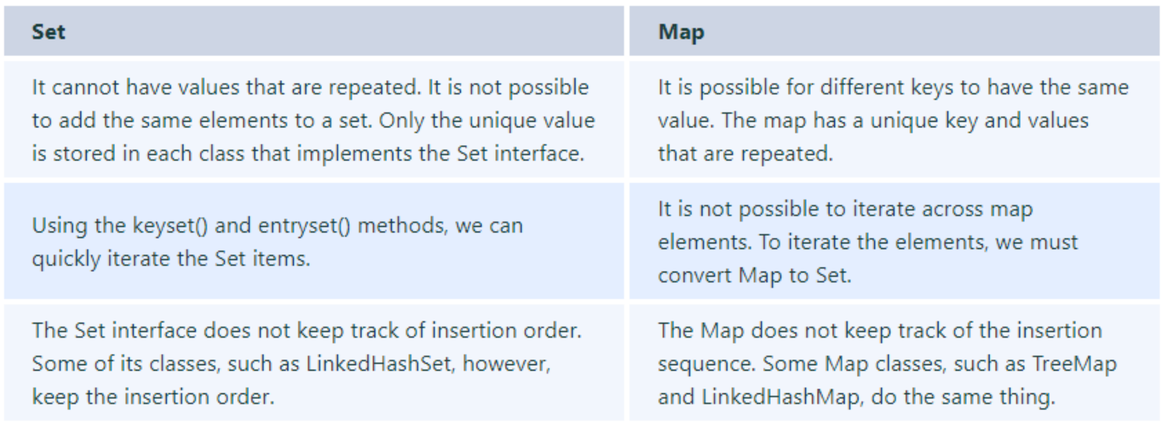
Following are the cases when TreeSet is preferred to HashSet :

1. Instead of unique elements, sorted unique elements are required. TreeSet returns a sorted list that is always in ascending order.
2. The locality of TreeSet is higher than that of HashSet. If two entries are close in order, TreeSet places them in the same data structure and hence in memory, but HashSet scatters the entries over memory regardless of the keys to which they are linked.
3. To sort the components, TreeSet employs the Red-Black tree method. TreeSet is a fantastic solution if you need to do read/write operations regularly.

Q. Can you add a null element into a TreeSet or HashSet?

We can add null elements in a HashSet but we cannot add null elements in a TreeSet. The reason is that TreeSet uses the compareTo() method for comparing and it throws a NullPointerException when it encounters a null element.

Q. Differentiate between Set and Map in Java.

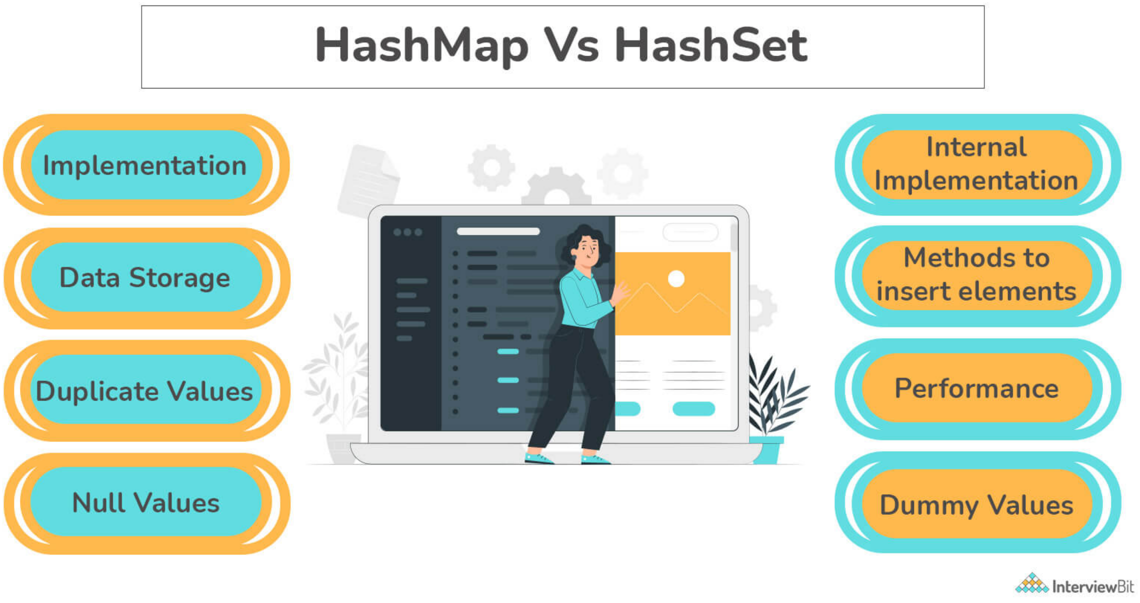


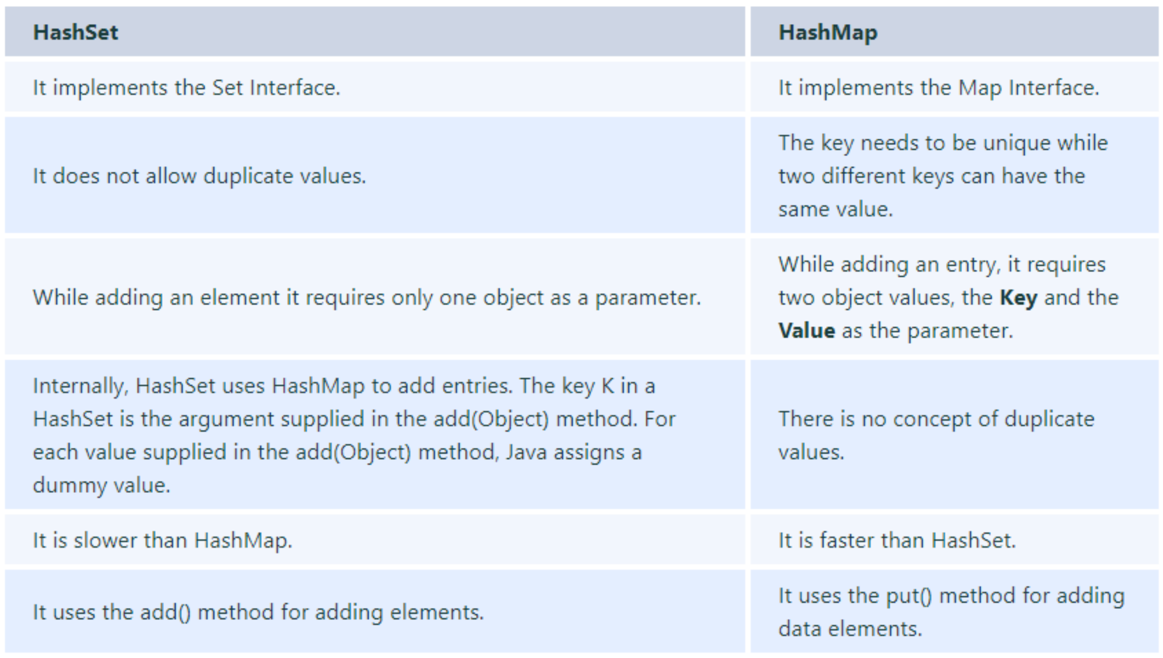
Set	Map
It cannot have values that are repeated. It is not possible to add the same elements to a set. Only the unique value is stored in each class that implements the Set interface.	It is possible for different keys to have the same value. The map has a unique key and values that are repeated.
Using the keyset() and entryset() methods, we can quickly iterate the Set items.	It is not possible to iterate across map elements. To iterate the elements, we must convert Map to Set.
The Set interface does not keep track of insertion order. Some of its classes, such as LinkedHashSet, however, keep the insertion order.	The Map does not keep track of the insertion sequence. Some Map classes, such as TreeMap and LinkedHashMap, do the same thing.

Q. Differentiate between HashSet and HashMap.

HashSet is a Set Interface implementation that does not allow duplicate values. The essential point is that objects stored in HashSet must override equals() and hashCode() methods to ensure that no duplicate values are stored in our set.

HashMap is a Map Interface implementation that maps a key to a value. In a map, duplicate keys are not permitted.





HashSet	HashMap
It implements the Set Interface.	It implements the Map Interface.
It does not allow duplicate values.	The key needs to be unique while two different keys can have the same value.
While adding an element it requires only one object as a parameter.	While adding an entry, it requires two object values, the **Key** and the **Value** as the parameter.
Internally, HashSet uses HashMap to add entries. The key K in a HashSet is the argument supplied in the add(Object) method. For each value supplied in the add(Object) method, Java assigns a dummy value.	There is no concept of duplicate values.
It is slower than HashMap.	It is faster than HashSet.
It uses the add() method for adding elements.	It uses the put() method for adding data elements.

Q. What is the default size of the load factor in hashing based collection?

The default load factor size is **0.75**. The default capacity is calculated by multiplying the initial capacity by the load factor

Q. Differentiate between Array and ArrayList in Java.

- Java provides arrays as a fundamental functionality. ArrayList is a component of Java's collection system. As a result, It is used to access array members, while ArrayList provides a set of methods for accessing and modifying components.
- ArrayList is not a fixed-size data structure, but Array is. When creating an ArrayList object, there is no need to provide its size. Even if we set a maximum capacity, we can add more parts afterward.
- Arrays can include both primitive data types and class objects, depending on the array's definition. ArrayList, on the other hand, only accepts object entries and not primitive data types. Note that when we use arraylist.add(1);, the primitive int data type is converted to an Integer object.
- Members of ArrayList are always referencing to objects at various memory locations since ArrayList can't be constructed for primitive data types As a result, the actual objects in an ArrayList are never kept in the same place. The references to the real items are maintained in close proximity. Whether an array is

primitive or an object depends on the type of the array. Actual values for primitive kinds are continuous regions, whereas allocation for objects is equivalent to ArrayList.

- Many other operations, such as indexOf() and delete(), are supported by Java ArrayList. Arrays do not support these functions.

Q. How can you make an ArrayList read-only in Java?

With the help of Collections.unmodifiableList() method, we can easily make an ArrayList read-only. This function takes a changeable ArrayList as an input and returns the ArrayList's read-only, unmodified view.

Q. Differentiate between HashMap and HashTable.

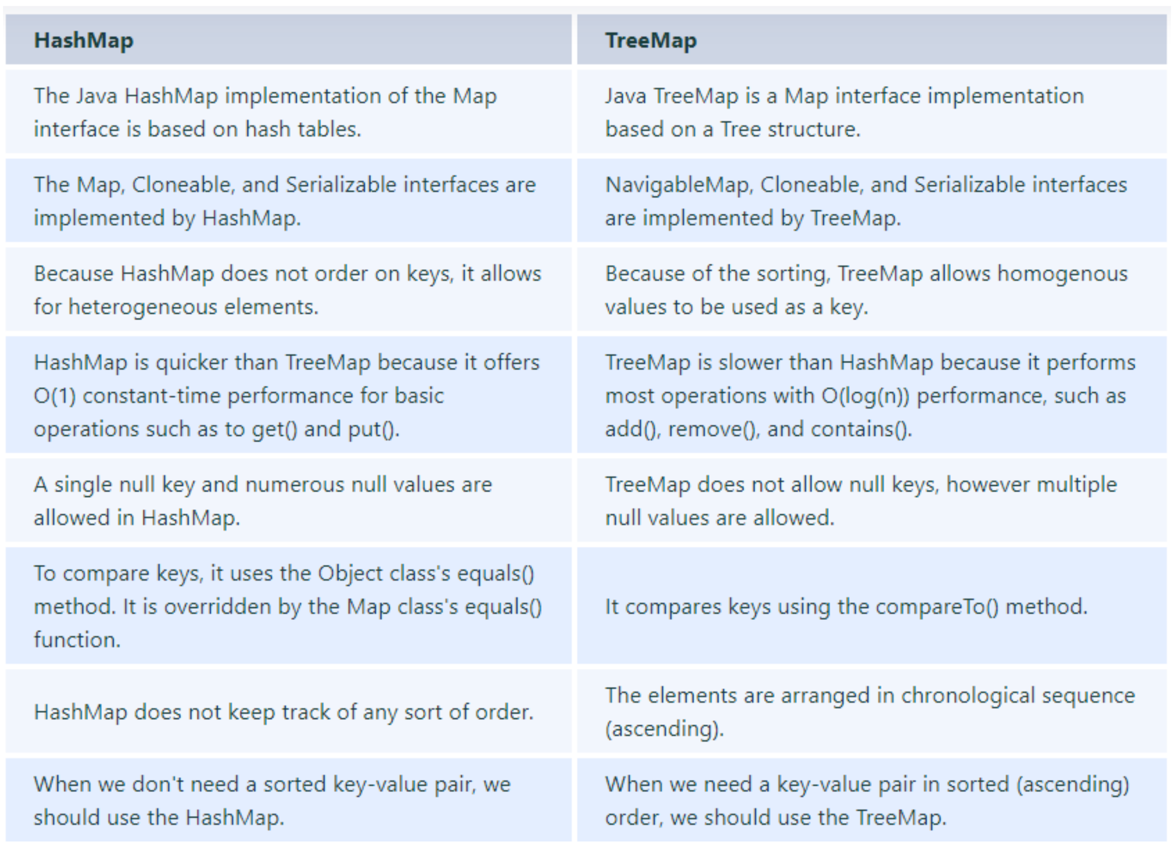
- HashMap is a non-synchronized data structure. It is not thread-safe and cannot be shared across many threads without the use of synchronization code, while Hashtable is synchronized. It's thread-safe and can be used by several threads.
- HashMap supports one null key and numerous null values, whereas Hashtable does not.
- If thread synchronization is not required, HashMap is often preferable over HashTable.

Q. How can you synchronize an ArrayList in Java?

An ArrayList can be synchronized using the following two ways :

- **Using Collections.synchronizedList() method:**All access to the backup list must be done through the returning list in order to perform serial access. When iterating over the returned list, it is critical that the user manually synchronizes

Q. Differentiate between HashMap and TreeMap in the context of Java.



HashMap	TreeMap
The Java HashMap implementation of the Map interface is based on hash tables.	Java TreeMap is a Map interface implementation based on a Tree structure.
The Map, Cloneable, and Serializable interfaces are implemented by HashMap.	NavigableMap, Cloneable, and Serializable interfaces are implemented by TreeMap.
Because HashMap does not order on keys, it allows for heterogeneous elements.	Because of the sorting, TreeMap allows homogenous values to be used as a key.
HashMap is quicker than TreeMap because it offers O(1) constant-time performance for basic operations such as to get() and put().	TreeMap is slower than HashMap because it performs most operations with O(log(n)) performance, such as add(), remove(), and contains().
A single null key and numerous null values are allowed in HashMap.	TreeMap does not allow null keys, however multiple null values are allowed.
To compare keys, it uses the Object class's equals() method. It is overridden by the Map class's equals() function.	It compares keys using the compareTo() method.
HashMap does not keep track of any sort of order.	The elements are arranged in chronological sequence (ascending).
When we don't need a sorted key-value pair, we should use the HashMap.	When we need a key-value pair in sorted (ascending) order, we should use the TreeMap.

Q. Which collection will use to sort the value by default?

By default, the **sort() method** sorts a given list into ascending order (or natural order). We can use Collections. reverseOrder() method, which returns a Comparator, for reverse sorting.

Q. Concurrent Hashmap

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details..

48. Generics

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects

Advantage of Java Generics

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

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
With Generics, it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0)
```

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

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```

Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{

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

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
Output

Printing Integer Array
10
20
30
40
50
Printing Character Array
J
A
V
A
T
P
O
I
N
T
```