**Java Interview Questions by interviewBit.**

**1. Why is Java a platform independent language?**

[**Java**](https://www.interviewbit.com/blog/features-of-java/) [**compiler**](https://www.interviewbit.com/online-java-compiler/) compiles the source code and then converts it to platform-independent byte code which can be run on multiple platform.

**2. Why is Java not a pure object oriented language?**

Java supports primitive data types - byte, short, char, int, long, float, double, boolean, and For an object-oriented programming language, data should be represented in the form of objects. hence it is not a pure/100% [**object oriented language**](https://www.interviewbit.com/oops-interview-questions/).

­**3. Difference between Heap and Stack Memory in java. And how java utilizes this.**

: stack is a static and fixed memory which is used to store the references for the threads and local variables.

: heap is a dynamic memory which is used to store the global variables/objects and JRE classes at runtime.

New objects are always created in heap memory, and the references to those objects are stored in stack memory.

Person p = new Person();

// 'p' is a reference on the stack, the actual object Person is on the heap.

**4. what is User-Defined Classes**

The Classes that are used to create user-defined data types and implement Abstract Data is user-defined class.

: Abstract Data type (ADT) is a type for objects whose action is defined by a set of values and a set of operations.

three ADTs namely List ADT, Stack ADT, Queue ADT.

: A user-defined data type (UDT) is a data type that is created by user/programmar not avaiable in Java default data type.

**Example**:

int i = 10; // premitive data type.

Student s1 = new Student(); // user-defined data type.

**6. what is compiler ?**

The compiler is nothing but a translator of source code(java code) into machine-executable code (byte code).

**7. What do you understand by an instance variable and a local variable?**

**Instance variables** are those variables which are accessible by all the methods in the class. They are declared outside the methods and inside the class.

**Example:**

**class** **Athlete** {

**public** String athleteName;

**public** **double** athleteSpeed;

**public** **int** athleteAge;

}

**Local variables** are those variables present within a block, function, or constructor and can be accessed only inside them. Whenever a local variable is declared inside a method, then other class methods don’t know about the local variable.

**Example:**

**public** **void** **athlete**() {

String athleteName;

**double** athleteSpeed;

**int** athleteAge;

}

**8. What are the default values assigned to variables and instances in java?**

* There are no default values assigned to the variables in java. We need to initialize the value before using it.
* But for instance (global variable), when we create an instance object, then a default value will be initialized by the default constructor based on the object type.
* If it is a reference, then it will be assigned to null.
* If it is numeric, then it will assign to 0.
* If it is a boolean, then it will be assigned to false. Etc.

There are three different types of variables a class can have in Java are **local variables, instance variables**, and **class/static** variables.

i) public class LocalVariableTest {

   public void show() {

      int num = 100; **// local variable**

      System.out.println("The number is : " + num);

   }

ii) public class InstanceVariableTest {

   int num; **// instance variable (global variable)**

   InstanceVariableTest(int n) {

      num = n;

   }

iii) public class StaticVaribleTest {

   int num;

static int count; **// static variable, can be accessed by calling with the class name.**

   StaticVaribleTest(int n) {

      num = n;

      count ++;

   }

**9. What do you mean by data encapsulation?**

* Encapsulation is data protection.
* Data Encapsulation is a process to Encapsulate/wrap the variables (data) & methods (code) as a single unit.
* the variables of a class will be restricted/hidden from direct access from other classes, and can be accessed only through the methods of the same class. Done by setter/getter methods.

//A Java class can be a fully encapsulated class by declairing all variables as private.

//It has a private data member and getter and setter methods.

**package** com.javatpoint;

**public** **class** Student{

//private data member

**private** String name;

//getter method for name

**public** String getName(){

**return** name;

}

//setter method for name

**public** **void** setName(String name){

**this**.name=name

}

}

**9.1. show me OOPS concept in your framework.**

Ans: **OOP** stands for Object Oriented Programming, where data should be represented in the form of object. and the Concepts are…

**Inheritance:** where a class inherits properties from parent class

**Example:**

class BaseClass{

public void startBrowser(){

WebDriver driver = new ChromeDriver();

} }

class Test extends BaseClass {

public static void main(String args[ ]){

startBrowser()

}}

**Polimorphism:** means many forms. In java it’s a mechanism for an object to take many forms.

There two types…

i. **method overloading**. / compile time / early binding / static binding polimorphism:

Example:

a. implecitlyWait method: we can use different argument like\_ seconds, or minutes, or hours etc.

b. read excel data: we use same method name for reading excel with different sheet name or number and with different column name or number.

ii. **method overriding.** / runtime binding / late binding / dynamic binding polimorpism:

Example: when we start a browser then we override the webDriver interface for cross-browser testing… like

**public** **class** BrowserFactory {

**public** **static** WebDriver startBrowser(WebDriver driver, String browser, String url) {

**if**(browser.equalsIgnoreCase("firefox")) {

WebDriverManager.*firefoxdriver*().setup();

driver = **new** FirefoxDriver();

}

**else** **if**(browser.equalsIgnoreCase("chrome")) {

WebDriverManager.*chromedriver*().setup();

driver = **new** ChromeDriver();

}

**else** **if**(browser.equalsIgnoreCase("edge")) {

WebDriverManager.*edgedriver*().setup();

driver = **new** EdgeDriver();

**Encapsulation:** is about data protection.

**Example:**

in a page object class we declare all the web-Element variables as **private** then we can access it using a get mehod.

@FindBy(xpath = “xxxxx”)

private userName;

**Abstruction:** is about hiding data implementation details.

Example :

We can apply Data Abstraction in a Selenium framework by using the Page Object Model design pattern. We define all our locators and their methods in the page class.

We can use these locators in our tests but we cannot see the implementation.

In a page object class we declare web elements and implemented by the methods of the class then we call those implemented methods from a test class. So that way web element implementation is hidden from test class.

@FindBy(xpath = “xxxxx”)

private userName;

public void enterUserName(Srting un){

userName.sendKeys(un);

}

**: difference between Abstaction and Encaptulation**

: Abstraction focuses on what should get done, while encapsulation focuses on how it should get done.

: Abstraction solves problems at the design or interface level, while encapsulation solves problems at the implementation level

: abstraction focusing on what an object does and encapsulation focusing on how an object's data is accessed.

**Interface:**

**Interfaces:** An interface is a collection of methods. I can use an interface to define a set of actions that a class should implement. A class can implement multiple interfaces, and all the methods of an interface must be implemented by the class that implements it.

Webdriver driver = new ChromeDriver();

**10. Tell us something about JIT compiler.**

* JIT stands for Just-In-Time and it is used for improving the performance of the code during run time.
  + when the Java source code is converted into byte code then JIT converts the byte code into native code and then JVM can execute the code directly instead of interpreting the code again.

**11. Can you tell me the difference between equals() method and equality operator (==) in Java?**

**: equals()** method is used to compare the references of the objects.

**: equality(==) operator** is used tocompare the contents of the objects.

String a = "Hello"; // a,b are object variables.

String b = "Hello";

System.***out***.println(a == b); //true

System.***out***.println(a.equals(b)); //true

String c = **new** String ("Hello"); // c,d are reference variables.

String d = **new** String ("Hello");

System.***out***.println(c == d); //false

System.***out***.println(c.equals(d)); //true

**12. How is an infinite loop declared in Java?**

Infinite loops are those loops that run infinitely without any breaking conditions. Some examples of consciously declaring infinite loop is:

* Using For Loop:

**for** (;;)

{

// Business logic

// Any break logic

}

* Using while loop:

**while**(**true**){

// Business logic

// Any break logic

}

* Using do-while loop:

**Do**{ // Business logic

// Any break logic

}**while**(**true**);

**13. Briefly explain the concept of constructor overloading**

Constructor overloading is the process of creating multiple constructors in a class having the same name as class name with a different parameters.

**14. Define Copy constructor in java.**

a copy constructor is a copy of another constructor that passes the object of the class as parameter and creates a duplicate object of the class.

It is used if we want to create a deep copy of an existing object. It is easier to implement in comparison to the clone() method. **It can not be inherited.**

**public** **class** Fruits

{

**private** **double** price;

**private** String name;

//copy constructor

**public** Fruits(Fruits fruits)

{

//copying each filed

**this**.price = fruits.price; //getter

**this**.name = fruits.name; //getter

}   }

**-----------------------------------------**

**public** **class** Fruit

{

**private** **double** fprice;

**private** String fname;

//constructor to initialize roll number and name of the student

**public**  Fruit(**double** fprice, String fname)  {

**this** .fprice = fprice;

**this** .fname = fname;

}

//creating a copy constructor

**public**  Fruit(Fruit fruit)  {

System.out.println("\nAfter invoking the Copy Constructor:\n");

**this** .fprice = fruit.fprice;

**this** .fname = fruit.fname;

}

//creating a method that returns the price of the fruit

**double** showPrice()  {

**return** fprice;

}

//creating a method that returns the name of the fruit

String showName()  {

**return** fname;

}

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

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

Fruit f1 = **new** Fruit(399, "Ruby Roman Grapes");

System.out.println("Name of the first fruit: "+ f1.showName());

System.out.println("Price of the first fruit: "+ f1.showPrice());

//passing the parameters to the copy constructor

Fruit f2 = **new** Fruit(f1);

System.out.println("Name of the second fruit: "+ f2.showName());

System.out.println("Price of the second fruit: "+ f2.showPrice());

}  }

**Output:**

Name of the first fruit: Ruby Roman Grapes

Price of the first fruit: 399.0

After invoking the Copy Constructor:

Name of the second fruit: Ruby Roman Grapes

Price of the second fruit: 399.0

**15**. **Can the main method be Overloaded?**

Yes, It is possible to overload the main method. We can create as many overloaded main methods as we want. Just using the different arguments in parameters instead of **String args[ ]**, use int, double, float, char etc. –

**public** **static** **void** **main**(string[] args)

Consider the below code snippets:

**class** **Main** {

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

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

}

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

System.out.println("Overloaded Integer array Main Method");

}

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

System.out.println("Overloaded Character array Main Method");

}

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

System.out.println("Overloaded Double array Main Method");

}

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

System.out.println("Overloaded float Main Method");

}

}

**17. A single try block and multiple catch blocks can co-exist in a Java Program ? Explain.**

Yes, A single **try** block can be followed by multiple **catch** blocks. If **try** block throws exception then matching block will be executed. first it will try to macth first **catch** block and if not then 2nd , if still not then it will throw the exception.

**public** **class** **MultipleCatch** {

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

**try** {

**int** n = 1000, x = 0;

**int** arr[] = **new** **int**[n];

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

arr[i] = i / x;

}

}

**catch** (ArrayIndexOutOfBoundsException exception) {

System.out.println("1st block = ArrayIndexOutOfBoundsException");

}

**catch** (ArithmeticException exception) {

System.out.println("2nd block = ArithmeticException");

}

**catch** (Exception exception) {

System.out.println("3rd block = Exception");

}

}

}

Here, the second catch block will be executed because of division by 0 (i / x). In case x was greater than 0 then the first catch block will execute because for loop runs till i = n and array index are till n-1.

**18. Explain the use of final keyword in variable, method and class.**

In Java, the final keyword is used as non-access modifier.

* **final variable:**
  + in the final variable, we can not change the value.
  + If the value has not been assigned, then it can be assigned by the constructor only.
* **final method:**
  + a final method cannot be overridden by sub-classes.
  + final constructor is not allowed in java.
* **final class:**
  + a final class can not be inherited by other classes But can extend to other classes. This means the final class can not be a superclass but can be a subclass.

we can create an object in a final class

**19. Do final, finally and finalize keywords have the same function?**

All three keywords have their own utility while programming.

**Final:** If any restriction is required for classes, variables, or methods, the final keyword comes in handy. Inheritance of a final class and overriding of a final method is restricted by the use of the final keyword. The variable value can be fixed using the final keyword. Example:

**final** **int** a=100;

a = 0; // error

The second statement will throw an error.

**Finally:** finally keyword is used in the try-catch block, when we want to execute some codes regardless try block passes or fails.

**try** {

**int** variable = 5;

}

**catch** (Exception exception) {

System. out.println("Exception occurred");

}

**finally** {

System.out.println("Execution of finally block");

}

**Finalize():** by default, every class extends Object class and that Object class has a method is finalize(),

Prior to the garbage collection of an object finalize() method is called by System.gc(); to implement the clean-up activity.

It is automatically called when the memory gets overloaded. We also can call the finalize() method manually by using System.gc();

then it will destroy all the un-reachable/un-usable objects right away instead of waiting for the memory to get overloaded.

**Example:**

**public** **class** JavafinalizeExample {

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

        JavafinalizeExample obj;

For(int i=0; i<10000000000; i++){

  obj = **new** JavafinalizeExample(); // creating Objects too many times

        // finalize() method will be called automatically when memory get overloaded.

    }

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

        JavafinalizeExample obj = **new** JavafinalizeExample();

        System.out.println(obj.hashCode());

        obj = **null**;

        // calling garbage collector manually

        System.gc();

        System.out.println("end of garbage collection");

    }

    @Override

**protected** **void** finalize() {

        System.out.println("finalize method is called");

    }

}

**20. Is it possible that the ‘finally’ block will not be executed? If yes then list the case.**

Yes. It is possible that the ‘finally’ block will not be executed. The cases are-

* if we use **System.exit()** in the above statement.
* If there are any error happens like Stack overflow, Memory access error, etc.

**22. When can you use super keyword?**

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

**Why we need it:**

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass that is overriden in the subclass.
3. To explicitly call superclass default () or parameterized constructor from the subclass constructor.

**\*\*\* to access super class attributes**

**class** Animal{

String color="white";

}

**class** Dog **extends** Animal{

String color="black";

**void** printColor(){

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

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

}

}

**class** Test{

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

Dog d=**new** Dog();

d.printColor();

}}

**--------------------------------------------------------------------**

**\*\*\* to call methods from super class**

**class** Animal{

**void** eat(){

System.out.println(" animal eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){

System.out.println("dog eating bread...");}

**void** work(){

eat();

**super**.eat();

}  }

**class** TestSuper2{

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

Dog d=**new** Dog();

d.work();

}}

**-------------------------------------------------------------------**

**\*\*\* to access super class constructor**

As we know, when we create an object of a class, its default constructor is automatically called. If we create an object of a sub class then super class constructor also will be called automatically. default constructor has no-argument.

To explicitly call the superclass constructor from the subclass constructor, we use super(). It's a special form of the super keyword.

super() can be used only inside the subclass constructor and must be the first

statement line.

**class** Animal{

Animal(){

System.out.println("animal is created");}

}

**class** Dog **extends** Animal{

Dog(){

**super**();

System.out.println("dog is created");

}

}

**class** TestSuper3{

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

Dog d=**new** Dog();

}}

**Output:**

animal is created

dog is created

**class** Animal{

Animal(){System.out.println("animal is created");}

}

**class** Dog **extends** Animal{

Dog(){

System.out.println("dog is created");

}

}

**class** TestSuper4{

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

Dog d=**new** Dog();

}}

**Output:**

animal is created

dog is created

**22.1** **What is the difference between this() and super() in Java?**

**This()** keyword is a reference variable which is used to refer current class object.

**super()** keyword is a reference variable which is used to refer immediate super class object.

**23. Can the static methods be overloaded?**

Yes! There can be two or more static methods in a class with the same name but differing input parameters.

**What is the pupose of static method ?**

A. to save memory. First thing static method belongs to the class not to object of class.

We don’t need to create an object of the class using **new** keyword to call static method. We can call the static methods and variables directly by using class name.

**24. Why is the main method static in Java?**

The main method is always static because static method is belong to the class, not an individual object. JVM calls the main method based on the class name. Not by creating an object. That’s why main method is static.

**25. Can the static methods be overridden?**

* No! we can not override a static method. We can create a static method with same name & signature in sub class but it will not be considerd as overriden. Because static method belong to a class. And static methods are loaded and looked up at the compile time statically. So Polimorphism will not take place here.

**26. Difference between static methods, static variables, and static classes in java.**

* **Static Methods and Static variables** are belong to a class, not to the object of a class. and will be called directly with the class names.
* **Static classes**- A class cannot be static. Only the inner class be static, then static inner class will be called with the outer class name.

**27. What is the main objective of garbage collection?**

The main objective of the garbage collection is to free up the memory space which is occupied by the unnecessary and unreachable objects during the program execution. Syntax: System.gc();

**28. What is a ClassLoader?**

* in java Classloader is a program belongs to JRE (Java Runtime Environment). ClassLoader loads the classes and interfaces to the JVM when it is required.
* **Example-** when want to get some input from the console/user, then we need a scanner class. And that Scanner class will be loaded by the ClassLoader to the JVM.

**28.1. What is Java String Pool?**

Java String pool is a collection of Strings, stored in heap memory. When we create a literal String object ex- **String s = “cat”;**, then String pool will look for the String with same value in the pool already present or not. If it is present, then it will return the same reference to the variable. If not, then it will create a new object in the heap space and will return the reference to the variable.

**String s1 = “cat”; [ s1 is a reference in pool ]**

**String s2 = new String(“cat”); [ created new space in heap ]**

In the above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so a total of 2 string objects will be created.

**: Difference between Java Heap Space and Stack Memory**

When any object is created, it's always stored in the Heap space and stack memory contains the reference of it.

Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.

**29. What part of memory - Stack or Heap - is cleaned in garbage collection process?**

Heap memory.

**29.1. What are the main concepts of OOPs in Java?**

Java - What is OOP? OOP stands for **Object-Oriented Programming**. It is a Process of creating objects for data, methods and classes.

And the OOPs concepts are like:

1. **Inheritance**: Inheritance is a process where one class acquires the properties of another.
2. **Encapsulation**: Data Encapsulation is a process to Encapsulate/wrap the variables (data) & methods (code) as a single unit.

the variables of a class will be restricted from accessing from other classes, and can be accessed only through the methods of their class. Done by setter/getter methods.

1. **Abstraction**: Abstraction is the mechanism of hiding the data implementation details from the user and only show the functionality to the users.
2. **Polymorphism**: Polymorphism is a way for a reference object to take multiple forms.

Exampl-

**class** A{}

**class** B **extends** A{}

A a=**new** B(); //upcasting

**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

**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();

  }

}

**29.2 What are the different types of inheritance in Java?**

Java supports four types of inheritance which are:

1. **Single Inheritance:** In single inheritance, one class inherits the properties of another i.e there will be only one parent as well as one child class.
2. **Multilevel Inheritance:**When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.
3. **Hierarchical Inheritance:**When a class has more than one child class (subclasses) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as hierarchical.
4. **Hybrid Inheritance:**Hybrid inheritance is a combination of twoormoretypes of inheritance. It happens in polymorphism.

**29.3 What is runtime polymorphism or dynamic method dispatch?**

runtime polymorphism is a process where an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method will be called using the superclass reference variable.

Example:

Class A{ }

Class B extends A{ }

Class C extends A{ }

Class D extends A{ }

A a = new A();

a = new B();

a = new C();

a = new D(); // a can call methods from both sub-class & super-class.

**Note #** compile time polymorphism is method overloading.

**30. What are shallow copy and deep copy in java?**

To create a copy of a object's data, we have several methods like deep copy and shallow copy.

: **shallow copy** – when we create a copy of an existing object by copying its reference then its called shallow copy. Any changes to the shallow copy will effect the original.

**int**[] x = {10, 20, 30, 40};

//shallow copy

**int**[] y = x;

//y[0] = 60; //change will effect original

System.***out***.println(Arrays.*toString*(x)); //[60, 20, 30, 40]

: **deep copy** – when we create a copy of any existing object by copying its content / values then its called deep copy. Any changes to the deep copy will not effect the original.

**int**[] x = {10, 20, 30, 40};

//deep copy

**int**[] z = **new** **int**[x.length];

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

z[i] = x[i];

}

z[0] = 70; //change will not effect original

System.***out***.println(Arrays.*toString*(x)); //[10, 20, 30, 40]

* **Shallow copy** - The shallow copy just creates a new copy of an existing object/reference variable. It doesn’t allocate new memory. Any update of new object effects original object.

**class** ABC  {

**int** x = 30;

}

**public** **class** ShallowCopy  {

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

ABC obj1 = **new** ABC();

ABC obj2 = obj1;

obj2.x = 6;

System.out.println("The value of x is: " + obj1.x);

}  }

**Output:**

The value of x is: 6

**Deep Copy** - deep copy is just a new objects/instance exactly like an existing object/instance. It is a complete object itself. It allocates new memory. Any update to any object does not effect other objects.

**class** ABC  {

**int** x = 30;

}

**public** **class** DeepCopy  {

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

ABC obj1 = **new** ABC();

ABC obj2 = **new** ABC();

obj2.x = 6;

System.out.println("The value of x is: " + obj1.x);

}

}   **Output:**

The value of x is: 30.

**30.1. What is the difference between a local variable and an instance variable?**

**local variable** is typically used inside a method, constructor, or a **block** and has only local scope. Thus, this variable can be used only within the scope of a block. The best benefit of having a local variable is that other methods in the class won’t be even aware of that variable.

Example..

if(x > 100)

{

String test = "Edureka";

}

Whereas, an **instance variable** in Java, is a variable which is bounded to its object itself. These variables are declared within a **class**, but outside a method. Every object of that class will create it’s own copy of the variable while using it. Thus, any changes made to the variable won’t reflect in any other instances of that class and will be bound to that particular instance only.

class Test{

public String EmpName;

public int empAge;

}

**Call by Value and Call by Reference in Java:**

If we call a method passing a **value** as parameter, it is known as call by value. any change made in the called method do not effect the original.

**public** **class** CallByValue {

**static** **int** *x*=10;

//value of data will be passed as parameter when method is called.

//but here we make change to data & will be in the local variable only

**public** **void** change(**int** data){

data=20;

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

}

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

CallByValue cv = **new** CallByValue();

System.***out***.println("before change "+CallByValue.*x*);

// passed x as value for data parameter.

cv.change(*x*);

System.***out***.println("after change "+CallByValue.*x*);

}

}

/\*

output:

before change 10

20

after change 10

\*/

**if we call a method passing the reference instance/object of the class as parameter then it is called call by reference. In case of call by reference original value is changed if we made changes in the called method.**

**public** **class** CallByReference {

**int** x=10;

//passed instance of class as parameter.

//but here we make change to x & will be in the local & original variable too.

**public** **void** change(CallByReference r1){

r1.x=20;

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

}

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

CallByReference r2 = **new** CallByReference();

System.***out***.println("before change "+r2.x);

// passed object r2 as value for parameter.

r2.change(r2);

System.***out***.println("after change "+r2.x);

}

}

/\*

output:

before change 10

20

after change 20

\*/

**Java Intermediate Interview Questions.**

**31. what is string immutable ?**

The string is immutable means that we cannot change the String data itself, but we can change the String data to the reference/object.

**class** Testimmutablestring{

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

String s=" Hello ";

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

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

}  }  **Output:**

Sachin

**-----------------------------------------------------------**

**class** Testimmutablestring1{

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

String s="Hello ";

s=s.concat(" World");

System.out.println(s);

}  }

**Output:**

Hello World

**32. What is a singleton class in Java? And How to implement a singleton class?**

Singleton classes are those classes, whose instance/objects are created only once. And with only that object the class members can be accessed.

**32.1 . What is constructor chaining in Java?**

In Java, constructor chaining is the process of calling one constructor from another with respect to the current object. Constructor chaining is possible only through legacy where a subclass constructor is responsible for invoking the superclass’ constructor first. There could be any number of classes in the constructor chain. Constructor chaining can be achieved in two ways:

1. Within the same class using **this()**
2. From sub class using **super()**

**32.2** **Why Java Strings are immutable in nature?**

Java string objects are immutable in nature which simply means once a String object is created its state cannot be modified.

**34. How would you differentiate between a String, StringBuffer, and a StringBuilder?**

**String :** In Java, a string is a sequence of characters. String is immutable means when we create a String object, we can not change the value. We use double quotes to represent a string. For example, "hello" is a sequence of characters 'h' , 'e' , 'l' , 'l' , and 'o' . in Java.

**StringBuffer :** Java StringBuffer class is used to create mutable (modifiable) String objects. StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.   
StringBuffer is synchronized i.e. thread safe. It means more than one thread can not be called at a time. It is not fast.

**StringBuilder :** Java StringBuilder class is used to create mutable (modifiable) String. StringBuilder class is same as StringBuffer class except that it is non-synchronized, it is not thread safe. It means more than one thread can be called at a time. It is fast.

**35. Using relevant properties highlight the differences between interfaces and abstract classes.**

* **Variable types:** Static and final variables can only be declared in the case of interfaces, whereas abstract classes can also have non-static and non-final variables.
* **Availability of methods:** Only abstract methods are available in interfaces, whereas non-abstract methods can be present along with abstract methods in abstract classes.
* **Inheritance:**Multiple inheritances are facilitated by interfaces, whereas abstract classes do not promote multiple inheritances.
* **Data member accessibility:** By default, the class data members of interfaces are of the public- type. Conversely, the class members for an abstract class can be protected or private also.
* **Implementation:** With the help of an abstract class, the implementation of an interface is easily possible. However, the converse is not true;

**Abstract class example:**

**public** **abstract** **class** **Athlete** {

**public** **abstract** **void** **walk**();

}

**Interface example:**

**public** **interface** **Walkable** {

**void** **walk**();

}

**36. How to create an instance/object of a class as return type ?**

Use- after login to Home page we need to call homePage class to work on that, so we can create an object of a class like that. Constructor will be called automatically.

**public** **class** HomePage {

**void** play() {

System.***out***.println("play at home");

}

}

**\*\*\*\*\*\*\*\*\*\*\*\*\***

**public** **class** TestClass {

// creating object of HomePage class as variable name home() with this method. Then we can call play method using the home(objec of HomePage).

**public** HomePage home() {

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

**return** **new** HomePage();

}

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

TestClass tc = **new** TestClass();

tc.home().play();

// **new** TestClass().home().play();

}

}

**Output:**

home method

play at home

**37. What is a Comparator in java?**

**Java Comparator** is an interface- it is used to sort the elements by comparing data members of a same user-defined class. This interface contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). For instance, it may be on roll no, name, age, or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List given by the comparator.

// Java Program to Demonstrate Working of

// Comparator Interface

// Importing required classes

// Class 1

// A class to represent a Student

class Student {

// Attributes of a student

int rollno;

String name, address;

// Constructor

public Student(int rollno, String name, String address)

{

// This keyword refers to current instance itself

this.rollno = rollno;

this.name = name;

this.address = address;

}

// Method of Student class

// To print student details in main()

public String toString()

{

// Returning attributes of Student

return this.rollno + " " + this.name + " "+ this.address;

}

}

// Class 2

// Helper class implementing Comparator interface

class Sortbyroll implements Comparator<Student> {

// Method

// Sorting in ascending order of roll number

public int compare(Student a, Student b)

{

return a.rollno - b.rollno;

}

}

// Class 3

// Helper class implementing Comparator interface

class Sortbyname implements Comparator<Student> {

// Method

// Sorting in ascending order of name

public int compare(Student a, Student b)

{

return a.name.compareTo(b.name);

}

}

// Class 4

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

{

// Creating an empty ArrayList of Student type

ArrayList<Student> ar = new ArrayList<Student>();

// Adding entries in above List

// using add() method

ar.add(new Student(111, "Mayank", "london"));

ar.add(new Student(131, "Anshul", "nyc"));

ar.add(new Student(121, "Solanki", "jaipur"));

ar.add(new Student(101, "Aggarwal", "Hongkong"));

// Display message on console for better readability

System.out.println("Unsorted");

// Iterating over entries to print them

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

System.out.println(ar.get(i));

// Sorting student entries by roll number

Collections.sort(ar, new Sortbyroll());

// Display message on console for better readability

System.out.println("\nSorted by rollno");

// Again iterating over entries to print them

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

System.out.println(ar.get(i));

// Sorting student entries by name

Collections.sort(ar, new Sortbyname());

// Display message on console for better readability

System.out.println("\nSorted by name");

// Again iterating over entries to print them

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

System.out.println(ar.get(i));

}

}

**Output:**

Unsorted

111 Mayank london

131 Anshul nyc

121 Solanki jaipur

101 Aggarwal Hongkong

Sorted by rollno

101 Aggarwal Hongkong

111 Mayank london

121 Solanki jaipur

131 Anshul nyc

Sorted by name

101 Aggarwal Hongkong

131 Anshul nyc

111 Mayank london

121 Solanki jaipur

**38. In Java, static as well as private method overriding is possible. Comment on the statement.**

The statement in the context is completely False. The static methods have no relevance with the objects, and these methods are of the class level. In the case of a child class, a static method with a method signature exactly like that of the parent class can exist without even throwing any compilation error.

The phenomenon mentioned here is popularly known as method hiding, and overriding is certainly not possible. Private method overriding is unimaginable because the visibility of the private method is restricted to the parent class only. As a result, only hiding can be facilitated and not overriding.

**39. What makes a HashSet different from a TreeSet?**

\* HashSet allowes heterogeneous data, but TreeSet doesn’t allowes heterogeneous data and null.

Although both HashSet and TreeSet are not synchronized and ensure that duplicates are not present, there are certain properties that distinguish a HashSet from a TreeSet.

* **Implementation:** For a HashSet, the hash table is utilized for storing the elements in an unordered manner. However, TreeSet makes use of the red-black tree to store the elements in a sorted manner.
* **Complexity/ Performance:** For adding, retrieving, and deleting elements, the time amortized complexity is O(1) for a HashSet. The time complexity for performing the same operations is a bit higher for TreeSet and is equal to O(log n). Overall, the performance of HashSet is faster in comparison to TreeSet.
* **Methods:**hashCode() and equals() are the methods utilized by HashSet for making comparisons between the objects. Conversely, compareTo() and compare() methods are utilized by TreeSet to facilitate object comparisons.
* **Objects type:** Heterogeneous and null objects can be stored with the help of HashSet. In the case of a TreeSet, runtime exception occurs while inserting heterogeneous objects or null objects.

**40. Why is the character array preferred over string for storing confidential information?**

In Java, a string is basically immutable i.e. it cannot be modified. After its declaration, it continues to stay in the string pool as long as it is not removed in the form of garbage. In other words, a string resides in the heap section of the memory for an unregulated and unspecified time interval after string value processing is executed.

As a result, vital information can be stolen for pursuing harmful activities by hackers if a memory dump is illegally accessed by them. Such risks can be eliminated by using mutable objects or structures like character arrays for storing any variable. After the work of the character array variable is done, the variable can be configured to blank at the same instant. Consequently, it helps in saving heap memory and also gives no chance to the hackers to extract vital data.

**41. What do we get in the JDK file?**

* **JDK**- For making java programs, we need some tools that are provided by JDK (Java Development Kit). JDK is the package that contains various tools, Compiler, Java Runtime Environment, etc.
* **JRE**-  To execute the java program we need an environment. (Java Runtime Environment) JRE contains a library of Java classes +  JVM.
* **What are JAVA Classes?**  It contains some predefined methods that help Java programs to use the feature, build and execute. ***For example*** - there is a system class in java that contains the print-stream method, and with the help of this, we can print something on the console.
* **JVM** - (Java Virtual Machine) JVM  is a part of JRE that executes the Java program at the end.  Actually, it is part of JRE, but it is software that converts bytecode into machine-executable code to execute on hardware.

**42. What are the differences between JVM, JRE and JDK in Java?**

| **Criteria** | **JDK** | **JRE** | **JVM** |
| --- | --- | --- | --- |
| **Abbreviation** | Java Development Kit | Java Runtime Environment | Java Virtual Machine |
| **Definition** | JDK is a complete software development kit for developing Java applications. It comprises JRE, JavaDoc, compiler, debuggers, etc. | JRE is a software package providing Java class libraries, JVM and all the required components to run the Java applications. | JVM is a platform-dependent, abstract machine comprising of 3 specifications - document describing the JVM implementation requirements, computer program meeting the JVM requirements and instance object for executing the Java byte code and provide the runtime environment for execution. |
| **Main Purpose** | JDK is mainly used for code development and execution. | JRE is mainly used for environment creation to execute the code. | JVM provides specifications for all the implementations to JRE. |
| **Tools provided** | JDK provides tools like compiler, debuggers, etc for code development | JRE provides libraries and classes required by JVM to run the program. | JVM does not include any tools, but instead, it provides the specification for implementation. |
| **Summary** | JDK = (JRE) + Development tools | JRE = (JVM) + Libraries to execute the application | JVM = Runtime environment to execute Java byte code. |

**43. What are the differences between HashMap and HashTable in Java?**

| **HashMap** | **HashTable** |
| --- | --- |
| HashMap is not synchronized thereby making it better for non-threaded applications. | HashTable is synchronized and hence it is suitable for threaded applications. |
| Allows only one null key but any number of null in the values. | This does not allow null in both keys or values. |
| Supports order of insertion by making use of its subclass LinkedHashMap. | Order of insertion is not guaranteed in HashTable. |

**44. What is the importance of reflection in Java?**

* The term reflection is used for describing the inspection capability of a code on other code either of itself or of its system and modify it during runtime.
* Consider an example where we have an object of unknown type and we have a method ‘fooBar()’ which we need to call on the object. The static typing system of Java doesn't allow this method invocation unless the type of the object is known beforehand. This can be achieved using reflection which allows the code to scan the object and identify if it has any method called “fooBar()” and only then call the method if needed.

Method methodOfFoo = fooObject.getClass().getMethod("fooBar", **null**);

methodOfFoo.invoke(fooObject, **null**);

* Using reflection has its own cons:
  + Speed — Method invocations due to reflection are about three times slower than the direct method calls.
  + Type safety — When a method is invoked via its reference wrongly using reflection, invocation fails at runtime as it is not detected at compile/load time.
  + Traceability — Whenever a reflective method fails, it is very difficult to find the root cause of this failure due to a huge stack trace. One has to deep dive into the invoke() and proxy() method logs to identify the root cause.
* Hence, it is advisable to follow solutions that don't involve reflection and use this method as a last resort.

**45. What are the different ways of threads usage?**

* We can define and implement a thread in java using two ways:
  + **Extending the Thread class**

**class** **InterviewBitThreadExample** **extends** **Thread**{

**public** **void** **run**(){

System.out.println("Thread runs...");

}

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

InterviewBitThreadExample ib = **new** InterviewBitThreadExample();

ib.start();

}

}

* **Implementing the Runnable interface**

**class** **InterviewBitThreadExample** **implements** **Runnable**{

**public** **void** **run**(){

System.out.println("Thread runs...");

}

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

Thread ib = **new** Thread(**new** InterviewBitThreadExample());

ib.start();

}

}

* Implementing a thread using the method of Runnable interface is more preferred and advantageous as Java does not have support for multiple inheritances of classes.
* start() method is used for creating a separate call stack for the thread execution. Once the call stack is created, JVM calls the run() method for executing the thread in that call stack.

**46. What are the different types of Thread Priorities in Java? And what is the default priority of a thread assigned by JVM?**

There are a total of 3 different types of priority available in Java.

**MIN\_PRIORITY:** It has an integer value assigned with 1.  
**MAX\_PRIORITY:** It has an integer value assigned with 10.  
**NORM\_PRIORITY:**It has an integer value assigned with 5.

In Java, Thread with MAX\_PRIORITY gets the first chance to execute. But the default priority for any thread is NORM\_PRIORITY assigned by JVM.

**47. What is the difference between the program and the process?**

* A program can be defined as a line of code written in order to accomplish a particular task. Whereas the process can be defined as the programs which are under execution.
* A program doesn't execute directly by the CPU. First, the resources are allocated to the program and when it is ready for execution then it is a process.

**48. What is the difference between the ‘throw’ and ‘throws’ keyword in java?**

* The ‘**throw**’ keyword is used inside a block of code to generate an custom exception manually while calling the method.

**Purpose**: if the outcome is not as expected then we can generate a custom exception manually using **throw** keyword and then we can provide an alternative solution in the catch block..

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

**int** result;

**try** {

result = 10/2;

**if**(result != 0) {

//throw new ArithmeticException();

**throw** **new** ArithmeticException("result is not as expected");

}

}

**catch**(ArithmeticException ar){

result = 10/10;

System.***out***.println("excption catched and provide alternative solution");

}

**catch**(NullPointerException ne) {

System.***out***.println("catched exception and it is NullPointerException");

}

**finally** {

System.***out***.println("finally block will be executed regardless try block pass or fail");

}

}

The **throws** keyword is used in the method signature to handle some checked exception which might be thrown by the function during execution of the code. So normal flow can be maintained. thows keyword does not fix the exception.

**What is checked exception ?**

The exception which is checked/caught by the compiler is called **checked exception**. the compiler will suggest the programmar to fix the mistakes or handle the exception then the programmar can fix the mistakes or handle the exception by using throws keyword or try catch block.

**Example**: All checked exceptions are child of exception class.

**Checked exceptions :**

FileNotFoundException

ClassNotFoundException,

IOException,

SQLException,

SocketException,

**What is unchecked exception ?**

The exception which was not checked by the compiler and occures at the run time is called **unchecked exception**. Unchecked exception is error/mistake made by programmar. after catch the exception programmar can fix it.

**Example**: All unchecked exceptions are child of runTimeException class.

**Unchecked exceptions :**

NullPointerException,

ArithmeticException,

ArrayIndexOutOfBoundsException

DateTimeException,

ClassCastException,

\* **Exception Handling** is mainly used to handle the checked exceptions. If there any unchecked exception occurs such as NullPointerException, it is programmers' fault that he is not checking the code before using the code. So it will throw an excepion and programmar can handle/fix it.

**Which exception should be declared?**

**Ans:** 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.

**There are two cases:**

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

**Case 1: Handle Exception Using try-catch block**

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

**Case 2: Declare Exception**

* In case we declare the exception, if exception does not occur, the code will be executed fine.
* In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not fix the exception. throws means I believe exeption will not happen, so I am throwing away the exception.

**49. What are the differences between constructor and method of a class in Java?**

| **Constructor** | **Method** |
| --- | --- |
| Constructor is used for initializing the object state. | Method is used for exposing the object's behavior. |
| Constructor has no return type. | Method should have a return type. Even if it does not return anything, return type is void. |
| Constructor gets invoked implicitly. | Method gets invoked explicitly. |
| If the constructor is not defined, then a default constructor is provided by the java compiler. | If a method is not defined, then the compiler does not provide it. |
| The constructor name should be equal to the class name. | The name of the method can have any name or have a class name too. |
| A constructor cannot be marked as final because whenever a class is inherited, the constructors are not inherited. Hence, marking it final doesn't make sense. Java throws compilation error saying - modifier final not allowed here | A method can be defined as final but it cannot be overridden in its subclasses. |
| Final variable instantiations are possible inside a constructor and the scope of this applies to the whole class and its objects. | A final variable if initialised inside a method ensures that the variable cant be changed only within the scope of that method. |

**53. Which among String or String Buffer should be preferred when there are lot of updates required to be done in the data?**

StringBuffer is mutable and dynamic in nature whereas String is immutable. Every updation / modification of String creates a new String thereby overloading the string pool with unnecessary objects. Hence, in the cases of a lot of updates, it is always preferred to use StringBuffer as it will reduce the overhead of the creation of multiple String objects in the string pool.

**54. How to not allow serialization of attributes of a class in Java?**

* In order to achieve this, the attribute can be declared along with the usage of transient keyword as shown below:

**public** **class** **InterviewBitExample** {

**private** **transient** String someInfo;

**private** String name;

**private** **int** id;

// :

// Getters setters

// :

}

* In the above example, all the fields except someInfo can be serialized.

**56. Consider the below program, identify the output, and also state the reason for that.**

**public** **class** **Main**{

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

System.out.println(" Hello. Main Method. ");

}

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

System.out.println(" Hello. Main Method2. ");

}

}

The output of the above program will be **Hello. Main Method**. This is because JVM will always call the main method based on the definition it already has. Doesn't matter how many main methods we overload it will only execute one main method based on its declaration in JVM.

**59. What do you understand by Object Cloning and how do you achieve it in Java?**

* It is the process of creating an exact copy of any object. In order to support this, a java class has to implement the Cloneable interface of java.lang package and override the clone() method provided by the Object class the syntax of which is:

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

**return** (Object)**super**.clone();

}

* In case the Cloneable interface is not implemented and just the method is overridden, it results in CloneNotSupportedException in Java.

**60. How does an exception propagate in the code?**

When an exception occurs, first it searches to locate the matching catch block. In case, the matching catch block is located, then that block would be executed. Else, the exception propagates through the method call stack and goes into the caller method where the process of matching the catch block is performed. This propagation happens until the matching catch block is found. If the match is not found, then the program gets terminated in the main method.

**61. How do exceptions affect the program if it doesn't handle them?**

Exceptions are runtime errors. Suppose we are making an android application with java. And it all works fine but there is an exceptional case when the application tries to get the file from storage and the file doesn’t exist (This is the case of exception in java). And if this case is not handled properly then the application will crash. This will be a bad experience for users.  This is the type of error that cannot be controlled by the programmer. But programmers can take some steps to avoid this so that the application won’t crash. The proper action can be taken at this step.

**62. Is it mandatory for a catch block to be followed after a try block?**

No, it is not necessary for a catch block to be present after a try block. - A try block should be followed either by a catch block or by a finally block. If the exceptions likelihood is more, then they should be declared using the throws clause of the method.

**63. Will the finally block get executed when the return statement is written at the end of try block and catch block as shown below?**

**public** **int** **someMethod**(**int** i){

**try**{

//some statement

**return** 1;

}**catch**(Exception e){

//some statement

**return** 999;

}**finally**{

//finally block statements

}

}

finally block will be executed irrespective of the exception or not. The only case where finally block is not executed is when it encounters ‘System.exit()’ method anywhere in try/catch block.

**64. Can you call a constructor of a class inside the another constructor?**

Yes, the concept can be termed as constructor chaining and can be achieved using this().

**Constructor chain**: a constructor is called from another constructor in the same class this process is known as **constructor chaining.** It occurs through inheritance. When we create an instance of a derived class, all the constructors of the inherited class (base class) are first invoked, after that the constructor of the calling class (derived class) is invoked.

We can achieve constructor chaining in two ways:

* **Within the same class:** If the constructors belong to the same class, we use **this**
* **From the base class:** If the constructor belongs to different classes (parent and child classes), we use the **super** keyword to call the constructor from the base class.

The Need of Constructor Chaining

Suppose, there are five tasks to perform. There are two ways to perform these tasks, either implement all the tasks in a single constructor or create separate tasks in separate constructor then chain them with each-other.

**Constructor Calling form another Constructor**

The calling of the constructor can be done in two ways:

* **By using this() keyword:** It is used when we want to call the current class constructor within the same class.
* **By using super() keyword:** It is used when we want to call the superclass constructor from the base class.

**65. Contiguous memory locations are usually used for storing actual values in an array but not in ArrayList. Explain.**

In the case of ArrayList, data storing in the form of primitive data types (like int, float, etc.) is not possible. The data members/objects present in the ArrayList have references to the objects which are located at various sites in the memory. Thus, storing of actual objects or non-primitive data types (like Integer, Double, etc.) takes place in various memory locations.

However, the same does not apply to the arrays. Object or primitive type values can be stored in arrays in contiguous memory locations, hence every element does not require any reference to the next element.

**67. Why is the remove method faster in the linked list than in an array?**

In the linked list, we only need to adjust the references when we want to delete the element from either end or the front of the linked list. But in the array, indexes are used. So to manage proper indexing, we need to adjust the values from the array So this adjustment of value is costlier than the adjustment of references.

**Example** - To Delete from the front of the linked list, internally the references adjustments happened like this

The only thing that will change is that the head pointer will point to the head’s next node. And delete the previous node. That is the constant time operation.

Whereas in the ArrayList, internally it should work like this-

For deletion of the first element, all the next element has to move to one place ahead. So this copying value takes time. So that is the reason why removing in ArrayList is slower than LinkedList.

**68. How many overloaded add() and addAll() methods are available in the List interface? Describe the need and uses.**

There are a total of 4 overloaded methods for add() and addAll() methods available in List Interface. The below table states the description of all.

| **Return Type** | **Method Description** |
| --- | --- |
| **boolean** | **add(Element e)**: This method is used for adding the element at the end of the List. The Datatype of the element is of any type it has been initially assigned with. It returns the boolean indicating successfully inserted or not. |
| **void** | **add(int index, Element e)**: This method is the overloaded version of add() method. In this, along with the element, the index is also passed to the method for the specific index the value needs to be inserted. |
| **boolean** | **addAll(Collection <extends ? Element > c)**: This method helps to add all elements at the end of collections from the list received in the parameter. It contains an iterator that helps to iterate the list and add the elements to the collection. |
| **boolean** | **addAll(int index, Collection <extends ? Element > c)**: This is the overloaded method for addAll() method. In this along with the list, we can pass the specified index from which the list elements need to be added. |

**69. How does the size of ArrayList grow dynamically? And also state how it is implemented internally.**

ArrayList is implemented in such a way that it can grow dynamically. We don't need to specify the size of ArrayList. For adding the values in it, the methodology it uses is -

1. Consider initially that there are 2 elements in the ArrayList. **[2, 3]**.

2. If we need to add the element into this. Then internally what will happen is-

* ArrayList will allocate the new ArrayList of Size (current size + half of the current size). And add the old elements into the new. Old - [2, 3],    New - [2, 3, null].

Then the new value will be inserted into it. [2, 3, 4, null]. And for the next time, the extra space will be available for the value to be inserted.

3. This process continues and the time taken to perform all of these is considered as the amortized constant time.

This is how the ArrayList grows dynamically. And when we delete any entry from the ArrayList then the following steps are performed -

1. It searches for the element index in the array. Searching takes some time. Typically it’s O(n) because it needs to search for the element in the entire array.

2. After searching the element, it needs to shift the element from the right side to fill the index.

So this is how the elements are deleted from the ArrayList internally. Similarly, the search operations are also implemented internally as defined in removing elements from the list (searching for elements to delete).

**Java Interview Questions for Experienced**

**71. What is the difference between ‘>>’ and ‘>>>’ operators in java?**

These 2 are the bitwise right shift operators. Although both operators look similar. But there is a minimal difference between these two right shift operators.

* **‘>>’ Bitwise Right Shift Operator**- This operator shifts each bit to its right position. And this maintains the signed bit.
* **‘>>>’ Bitwise Right Shift Operator with trailing zero**- This operator also shifts each bit to its right. But this doesn’t maintain the signed bit. This operator makes the Most significant bit to 0.

**Example-** Num1 = 8, Num2 = -8.

So the binary form of these numbers are -

**Num1 = 00000000 00000000 00000000 00001000**  
**Num2 = 11111111 11111111 11111111  11111000**

**‘>>’ Operator** : 8 >> 1 (Shift by one bit) :

**Num1 = 00000000 00000000 00000000 00000100**  
**Num2 = 11111111 11111111 11111111  11111100**

**‘>>>’ Operator**: 8 >>> 1 (Shift by one bit) =

**Num1 = 00000000 00000000 00000000 00000100**  
**Num2 = 01111111 11111111 11111111 11111100**

**72. What are Composition and Aggregation? State the difference.**

Composition, and Aggregation help to build (Has - A - Relationship) between classes and objects. But both are not the same in the end. **Let’s understand with the help of an example.**

* Consider the University as a class that has some departments in it. So the university will be the container object. And departments in it will contain objects. Now in this case, if the container object destroys then the contained objects will also get destroyed automatically.  So here we can say that there is a strong association between the objects. So this Strong Association is called **Composition**.
* Now consider one more example. Suppose we have a class department and there are several professors' objects there in the department. Now if the department class is destroyed then the professor's object will become free to bind with other objects. Because container objects (Department) only hold the references of contained objects (Professor’s). So here is the weak association between the objects. And this weak association is called **Aggregation**.

**73. How is the creation of a String using new() different from that of a literal?**

When a String is created using an assignment operator, it makes a path to a String into the String pool if the String is already available in the pool. If not available then it will create a new Object in the pool and will make a Path for that String.

same String content will be used by different String object just using different path to same String content.

**public** bool **checking**() {

String first = "InterviewBit";

String second = "InterviewBit";

**if** (first == second)

**return** **true**;

**else**

**return** **false**;

}

The checking() function will return true as the same content is referenced by both the variables.

Conversely, when a String formation takes place with the help of a new() operator, interning does not take place. The object gets created in the heap memory even if the same content object is present.

**public** bool **checking**() {

String first = **new** String("InterviewBit");

String second = **new** String("InterviewBit");

**if** (first == second)

**return** **true**;

**else**

**return** **false**;

}

The checking() function will return false as the same content is not referenced by both the variables.

**74. How is the ‘new’ operator different from the ‘newInstance()’ operator in java?**

Both ‘**new**’ and ‘**newInstance()**’ operators are used to creating objects. The difference is- that when we already know the class name for which we have to create the object then we use a new operator. But suppose we don’t know the class name for which we need to create the object, Or we get the class name from the command line argument, or the database, or the file. Then in that case we use the ‘**newInstance()**’ operator.

The ‘**newInstance()**’ keyword throws an exception that we need to handle. It is because there are chances that the class definition doesn’t exist, and we get the class name from runtime. So it will throw an exception.

**75. Is exceeding the memory limit possible in a program despite having a garbage collector?**

Yes, it is possible for the program to go out of memory in spite of the presence of a garbage collector. Garbage collection assists in recognizing and eliminating those objects which are not required in the program anymore, in order to free up the resources used by them.

In a program, if an object is unreachable, then the execution of garbage collection takes place with respect to that object. If the amount of memory required for creating a new object is not sufficient, then memory is released for those objects which are no longer in the scope with the help of a garbage collector. The memory limit is exceeded for the program when the memory released is not enough for creating new objects.

Moreover, exhaustion of the heap memory takes place if objects are created in such a manner that they remain in the scope and consume memory. The developer should make sure to dereference the object after its work is accomplished. Although the garbage collector endeavors its level best to reclaim memory as much as possible, memory limits can still be exceeded.

Let’s take a look at the following example:

List<String> example = **new** LinkedList<String>();

**while**(**true**){

example.add(**new** String("Memory Limit Exceeded"));

}

**76. Why is synchronization necessary? Explain with the help of a relevant example.**

Concurrent execution of different processes is made possible by synchronization. When a particular resource is shared between many threads, situations may arise in which multiple threads require the same shared resource.

Synchronization assists in resolving the issue and the resource is shared by a single thread at a time. Let’s take an example to understand it more clearly. For example, you have a URL and you have to find out the number of requests made to it. Two simultaneous requests can make the count erratic.

**No synchronization:**

**package** anonymous;

**public** **class** **Counting** {

**private** **int** increase\_counter;

**public** **int** **increase**() {

increase\_counter = increase\_counter + 1;

**return** increase\_counter;

}

}

If a thread Thread1 views the count as 10, it will be increased by 1 to 11. Simultaneously, if another thread Thread2 views the count as 10, it will be increased by 1 to 11. Thus, inconsistency in count values takes place because the expected final value is 12 but the actual final value we get will be 11.

Now, the function increase() is made synchronized so that simultaneous accessing cannot take place.

**With synchronization:**

**package** anonymous;

**public** **class** **Counting** {

**private** **int** increase\_counter;

**public** **synchronized** **int** **increase**() {

increase\_counter = increase\_counter + 1;

**return** increase\_counter;

}

}

If a thread Thread1 views the count as 10, it will be increased by 1 to 11, then the thread Thread2 will view the count as 11, it will be increased by 1 to 12. Thus, consistency in count values takes place.

**77. In the given code below, what is the significance of ... ?**

**public** **void** **fooBarMethod**(String... variables){

// method code

}

* Ability to provide ... is a feature called varargs (variable arguments) which was introduced as part of Java 5.
* The function having ... in the above example indicates that it can receive multiple arguments of the datatype String.
* For example, the fooBarMethod can be called in multiple ways and we can still have one method to process the data as shown below:

fooBarMethod("foo", "bar");

fooBarMethod("foo", "bar", "boo");

fooBarMethod(**new** String[]{"foo", "var", "boo"});

**public** **void** **myMethod**(String... variables){

**for**(String variable : variables){

// business logic

}

}

**78. What will be the output of the below java program and define the steps of Execution of the java program with the help of the below code?**

**class** **InterviewBit**{

**int** i;

**static** **int** j;

{

System.out.println(" Instance Block 1. Value of i = "+i);

}

**static**{

System.out.println(" Static Block 1. Value of j = "+j);

method\_2();

}

{

i = 5;

}

**static**{

j = 10;

}

InterviewBit(){

System.out.println(" Welcome to InterviewBit ");

}

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

InterviewBit ib = **new** InterviewBit();

}

**public** **void** **method\_1**(){

System.out.println(" Instance method. ");

}

**static**{

System.out.println(" Static Block 2. Value of j = "+j);

}

{

System.out.println(" Instance Block 2. Value of i = "+i);

method\_1();

}

**public** **static** **void** **method\_2**(){

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

}

}

The Output we get by executing this program will be

**Static Block 1. Value of j = 0**  
**Static method.**  
**Static Block 2. Value of j = 10**  
**Instance Block 1. Value of i = 0**  
**Instance Block 2. Value of i = 5**  
**Instance method.**  
**Welcome to InterviewBit**

This is a java tricky interview question frequently asked in java interviews for the experienced. The output will be like this because, when the java program is compiled and gets executed, then there are various steps followed for execution. And the steps are -

* Identification of Static Members from top to bottom.
* Execution of Static variable assignment and a Static block from top to bottom.
* Execution of the main method.
* Identification of Instance Members from top to bottom.
* Execution of Instance variable assignment and Instance block from top to bottom.
* Execution of Constructor.

In above steps from 4 to 6, will be executed for every object creation. If we create multiple objects then for every object these steps will be performed.

Now from the above code, the execution will happen like this -

1. In the step of identification of static members. It is found that -

* static int j.
* static block.
* main method.
* static method\_2.

During identification, the JVM will assign the default value in the static int j variable. Then it is currently in the state of reading and indirectly writing. Because the original value is not assigned.

2. In the next step, it will execute the static block and assign the value in static variables.

* First static block it will print and because execution from top to bottom and original value in j is not assigned. So it will print the default value of 0.
* After executing static block 1. It will execute the static method\_1 because it is called from the static block 1.
* Then it will assign the original value of 5 in the j variable. And executes the remaining static block.

3. Now it will execute the main method. In which it will create an object for the class InterviewBit. And then the execution of instances will happen.

4. Identify the instance variables and blocks from top to bottom.

* int i.
* Instance block 1.
* Instance method\_1.

Like a static variable, the instance variable also has been initialized with the default value 0 and will be in the state of reading and writing indirectly.

5. It will execute the instance methods and assign the original value to the instance variable.

* Prints the Instance block 1. And the current value of i is not assigned till now, so it will print 0.
* Assign the original value to i. Then print instance block 2. And after that instance method will be called and printed because it is being called in the instance block.

6. And at the last step, the constructor will be invoked and the lines will be executed in the constructor.

This is how the java program gets executed.

**79. Define System.out.println().**

**System.out.println()** is used to print the message on the console.

**System** - is a class present in **java.lang package**.

**Out** is the static variable of type PrintStream class present in the **System class. println()** is the method present in the PrintStream class.

So if we justify the statement, then we can say that if we want to print anything on the console then we need to call the **println()** method that was present in PrintStream class. And we can call this using the output object that is present in the System class.

**80. Can you explain the Java thread lifecycle?**

Java thread life cycle is as follows:

* **New** – When the instance of the thread is created and the start() method has not been invoked, the thread is considered to be alive and hence in the NEW state.
* **Runnable** – Once the start() method is invoked, before the run() method is called by JVM, the thread is said to be in RUNNABLE (ready to run) state. This state can also be entered from the Waiting or Sleeping state of the thread.
* **Running** – When the run() method has been invoked and the thread starts its execution, the thread is said to be in a RUNNING state.
* **Non-Runnable (Blocked/Waiting)** – When the thread is not able to run despite the fact of its aliveness, the thread is said to be in a NON-RUNNABLE state. Ideally, after some time of its aliveness, the thread should go to a runnable state.
  + A thread is said to be in a Blocked state if it wants to enter synchronized code but it is unable to as another thread is operating in that synchronized block on the same object. The first thread has to wait until the other thread exits the synchronized block.
  + A thread is said to be in a Waiting state if it is waiting for the signal to execute from another thread, i.e it waits for work until the signal is received.
* **Terminated** – Once the run() method execution is completed, the thread is said to enter the TERMINATED step and is considered to not be alive.

The following flowchart clearly explains the lifecycle of the thread in Java.

**81. What could be the tradeoff between the usage of an unordered array versus the usage of an ordered array?**

* The main advantage of having an ordered array is the reduced search time complexity of O(log n) whereas the time complexity in an unordered array is O(n).
* The main drawback of the ordered array is its increased insertion time which is O(n) due to the fact that its element has to reordered to maintain the order of array during every insertion whereas the time complexity in the unordered array is only O(1).
* Considering the above 2 key points and depending on what kind of scenario a developer requires, the appropriate data structure can be used for implementation.

**82. Is it possible to import the same class or package twice in Java and what happens to it during runtime?**

It is possible to import a class or package more than once, however, it is redundant because the JVM internally loads the package or class only once.

**83. In case a package has sub packages, will it suffice to import only the main package? e.g. Does importing of com.myMainPackage.\* also import com.myMainPackage.mySubPackage.\*?**

This is a big NO. We need to understand that the importing of the sub-packages of a package needs to be done explicitly. Importing the parent package only results in the import of the classes within it and not the contents of its child/sub-packages.

**84. Will the finally block be executed if the code System.exit(0) is written at the end of try block?**

NO. The control of the program post System.exit(0) is immediately gone and the program gets terminated which is why the finally block never gets executed.

**85. What do you understand by marker interfaces in Java?**

Marker interfaces, also known as tagging interfaces are those interfaces that have no methods and constants defined in them. They are there for helping the compiler and JVM to get run time-related information regarding the objects.

**86. Explain the term “Double Brace Initialization” in Java?**

This is a convenient means of initializing any collections in Java. Consider the below example.

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** **IBDoubleBraceDemo**{

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

Set<String> stringSets = **new** HashSet<String>()

{

{

add("set1");

add("set2");

add("set3");

}

};

doSomething(stringSets);

}

**private** **static** **void** **doSomething**(Set<String> stringSets){

System.out.println(stringSets);

}

}

In the above example, we see that the stringSets were initialized by using double braces.

* The first brace does the task of creating an anonymous inner class that has the capability of accessing the parent class’s behavior. In our example, we are creating the subclass of HashSet so that it can use the add() method of HashSet.
* The second braces do the task of initializing the instances.

Care should be taken while initializing through this method as the method involves the creation of anonymous inner classes which can cause problems during the garbage collection or serialization processes and may also result in memory leaks.

**87. Why is it said that the length() method of String class doesn't return accurate results?**

* The length method returns the number of Unicode units of the String. Let's understand what Unicode units are and what is the confusion below.
* We know that Java uses UTF-16 for String representation. With this Unicode, we need to understand the below two Unicode related terms:
  + Code Point: This represents an integer denoting a character in the code space.
  + Code Unit: This is a bit sequence used for encoding the code points. In order to do this, one or more units might be required for representing a code point.
* Under the UTF-16 scheme, the code points were divided logically into 17 planes and the first plane was called the Basic Multilingual Plane (BMP). The BMP has classic characters - U+0000 to U+FFFF. The rest of the characters- U+10000 to U+10FFFF were termed as the supplementary characters as they were contained in the remaining planes.
  + The code points from the first plane are encoded using **one** 16-bit code unit
  + The code points from the remaining planes are encoded using **two** code units.

Now if a string contained supplementary characters, the length function would count that as 2 units and the result of the length() function would not be as per what is expected.

In other words, if there is 1 supplementary character of 2 units, the length of that SINGLE character is considered to be TWO - Notice the inaccuracy here? As per the java documentation, it is expected, but as per the real logic, it is inaccurate.

**88. What is the output of the below code and why?**

**public** **class** **InterviewBit**{

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

{

System.out.println('b' + 'i' + 't');

}

}

“bit” would have been the result printed if the letters were used in double-quotes (or the string literals). But the question has the character literals (single quotes) being used which is why concatenation wouldn't occur. The corresponding ASCII values of each character would be added and the result of that sum would be printed.  
The ASCII values of ‘b’, ‘i’, ‘t’ are:

* ‘b’ = 98
* ‘i’ = 105
* ‘t’ = 116

98 + 105 + 116 = 319

Hence 319 would be printed.

**89. What are the possible ways of making object eligible for garbage collection (GC) in Java?**

**First Approach:** Set the object references to null once the object creation purpose is served.

**public** **class** **IBGarbageCollect** {

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

String s1 = "Some String";

// s1 referencing String object - not yet eligible for GC

s1 = **null**; // now s1 is eligible for GC

}

}

**Second Approach:** Point the reference variable to another object. Doing this, the object which the reference variable was referencing before becomes eligible for GC.

**public** **class** **IBGarbageCollect** {

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

String s1 = "To Garbage Collect";

String s2 = "Another Object";

System.out.println(s1); // s1 is not yet eligible for GC

s1 = s2; // Point s1 to other object pointed by s2

/\* Here, the string object having the content "To Garbage Collect" is not referred by any reference variable. Therefore, it is eligible for GC \*/

}

}

**Third Approach:**Island of Isolation Approach: When 2 reference variables pointing to instances of the same class, and these variables refer to only each other and the objects pointed by these 2 variables don't have any other references, then it is said to have formed an “Island of Isolation” and these 2 objects are eligible for GC.

**public** **class** **IBGarbageCollect** {

IBGarbageCollect ib;

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

IBGarbageCollect ibgc1 = **new** IBGarbageCollect();

IBGarbageCollect ibgc2 = **new** IBGarbageCollect();

ibgc1.ib = ibgc2; //ibgc1 points to ibgc2

ibgc2.ib = ibgc1; //ibgc2 points to ibgc1

ibgc1 = **null**;

ibgc2 = **null**;

/\*

\* We see that ibgc1 and ibgc2 objects refer

\* to only each other and have no valid

\* references- these 2 objects for island of isolcation - eligible for GC

\*/

}

}

**90. In the below Java Program, how many objects are eligible for garbage collection?**

**class** **Main**{

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

**int**[][] num = **new** **int**[3][];

num[0] = **new** **int**[5];

num[1] = **new** **int**[2];

num[2] = **new** **int**[3];

num[2] = **new** **int**[5];

num[0] = **new** **int**[4];

num[1] = **new** **int**[3];

num = **new** **int**[2][];

}

}

In the above program, a total of 7 objects will be eligible for garbage collection. Let’s visually understand what's happening in the code.

In the above figure on line 3, we can see that on each array index we are declaring a new array so the reference will be of that new array on all the 3 indexes. So the old array will be pointed to by none. So these three are eligible for garbage collection. And on line 4, we are creating a new array object on the older reference. So that will point to a new array and older multidimensional objects will become eligible for garbage collection.

**92. How we can set the spring bean scope. And what supported scopes does it have?**

A scope can be set by an annotation such as the @Scope annotation or the "scope" attribute in an XML configuration file. Spring Bean supports the following five scopes:

* Singleton
* Prototype
* Request
* Session
* Global-session.

**93. What are the different categories of Java Design patterns?**

Java Design patterns are categorized into the following different types. And those are also further categorized as

**Structural patterns:**

* Adapter
* Bridge
* Filter
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

**Behavioral patterns:**

* Interpreter
* Template method/ pattern
* Chain of responsibility
* Command pattern
* Iterator pattern
* Strategy pattern
* Visitor pattern

**J2EE patterns:**

* MVC Pattern
* Data Access Object pattern
* Front controller pattern
* Intercepting filter pattern
* Transfer object pattern

**Creational patterns:**

* Factory method/Template
* Abstract Factory
* Builder
* Prototype
* Singleton.

**94. What is a Memory Leak? Discuss some common causes of it.**

The Java Garbage Collector (GC) typically removes unused objects when they are no longer required, but when they are still referenced, the unused objects cannot be removed. So this causes the memory leak problem. **Example**- Consider a linked list like the structure below -

In the above image, there are unused objects that are not referenced. But then also Garbage collection will not free it. Because it is referencing some existing referenced object. So this can be the situation of memory leak.

**Some common causes of Memory leaks are** -

* When there are Unbounded caches.
* Excessive page swapping is done by the operating system.
* Improper written custom data structures.
* Inserting into a collection object without first deleting it.  
  etc.

**95. Assume a thread has a lock on it, calling the sleep() method on that thread will release the lock?**

A thread that has a lock won't be released even after it calls sleep(). Despite the thread sleeping for a specified period of time, the lock will not be released.