**1) What is Java?**

[Java](https://www.javatpoint.com/java-tutorial) is the high-level, [object-oriented](https://www.javatpoint.com/java-oops-concepts), programming language. It was developed by [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java) in June 1991. It is known for its features like robustness, security, platform-independent, performance, Multithreaded nature, and portability programming language It can also be known as the platform as it provides its own JRE and API.

**2) What are the differences between C++ and Java?**

The differences between [C++](https://www.javatpoint.com/cpp-tutorial) and Java are given in the following table.

|  |  |  |
| --- | --- | --- |
| **Comparison Index** | **C++** | **Java** |
| **Platform-independent** | C++ is platform-dependent. | Java is platform-independent. |
| **Multiple inheritance** | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by [interfaces in java](https://www.javatpoint.com/interface-in-java). |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java doesn't support operator overloading. |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write pointer program in C++. | Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in Java. |
| **Compiler and Interpreter** | C++ uses compiler only. Compiler converts source code into machine code so, C++ is platform dependent. | Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent. |
| **Thread Support** | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in [thread](https://www.javatpoint.com/multithreading-in-java) support. |
| **Documentation comment** | C++ doesn't support documentation comment. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |
| **Programming Paradigm** | |  | | --- | | Supports both procedural and object-oriented programming |  |  | | --- | |  | | Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) language. However, everything (except fundamental types) is an object in Java. |
| |  | | --- | | **Memory Management** | | |  | | --- | | Manual memory management using pointers and destructors | | |  | | --- | | Automatic garbage collection | |

**What does java uses compiler or interpreter?**

Java uses both a compiler and an interpreter in its execution process, which is often referred to as the "Java execution model." Here's how it works:

**Compiler (javac):**

Java source code is written in .java files.The Java compiler (javac) compiles these source files into bytecode, which is stored in .class files. Bytecode is a platform-independent, intermediate representation of the code.

**Interpreter (JVM - Java Virtual Machine):**

The Java Virtual Machine (JVM) interprets and executes the bytecode. The JVM can either interpret the bytecode directly or use Just-In-Time (JIT) compilation to convert it into native machine code for the host system, improving performance.

**3) List the features of Java Programming language.**

* **Simple:** Java is easy to learn. The syntax of Java is based on C++ which makes easier to write the program in it.
* **Object-Oriented:** Java follows the object-oriented paradigm which allows us to maintain our code as the combination of different type of objects that incorporates both data and behavior.
* **Portable:** Java supports read-once-write-anywhere approach. We can execute the Java program on every machine. Java program (.java) is converted to bytecode (.class) which can be easily run on every machine.
* **Platform Independent:** Java is a platform independent programming language. It is different from other programming languages like C and C++ which needs a platform to be executed. Java comes with its platform on which its code is executed. Java doesn't depend upon the operating system to be executed.
* **Secured:** Java is secured because it doesn't use explicit pointers. Java also provides the concept of ByteCode and Exception handling which makes it more secured.
* **Robust:** Java is a strong programming language as it uses strong memory management. The concepts like Automatic garbage collection, Exception handling, etc. make it more robust.
* **Architecture Neutral:** Java is architectural neutral as it is not dependent on the architecture. In C, the size of data types may vary according to the architecture (32 bit or 64 bit) which doesn't exist in Java.
* **Interpreted:** Java uses the Just-in-time (JIT) interpreter along with the compiler for the program execution.
* **High Performance:** Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
* **Multithreaded:** We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.
* **Distributed:** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
* **Dynamic:** Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

**4) What do you understand by Java virtual machine?**

[Java Virtual Machine](https://www.javatpoint.com/jvm-java-virtual-machine) is a virtual machine that enables the computer to run the Java program on any platform. JVM acts like a run-time engine which calls the main method present in the Java code. JVM is the specification which must be implemented in the computer system. It executes Java bytecode making java machine independent and manages memory through garbage collection.

**5) What is the difference between JDK, JRE, and JVM?**

The Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM) are three related but distinct components of the Java ecosystem. Here’s a breakdown of each:



**JVM (Java Virtual Machine)**

Definition: The JVM is the engine that executes Java bytecode. [Java Virtual Machine](https://www.javatpoint.com/jvm-java-virtual-machine) is a virtual machine that enables the computer to run the Java program on any platform. JVM acts like a run-time engine which calls the main method present in the Java code.

Components: Bytecode interpreter, Just-In-Time compiler, Garbage Collector.

Purpose: To provide a platform-independent way of executing Java bytecode.

**JRE (Java Runtime Environment)**

Definition: A package that includes the JVM and libraries Java applications need to run.

Components: JVM, core libraries, and other components to support Java application execution.

Purpose: To provide the runtime environment necessary for executing Java applications.

**JDK (Java Development Kit)**

Definition: A comprehensive toolkit for developing Java applications, including the JRE, compilers, and tools.

Components: JRE, javac compiler, tools for Java application development.

Purpose: To provide developers with the tools needed to develop, compile, and debug Java applications.

Usage: Used by Java developers for application development.

**In summary:**

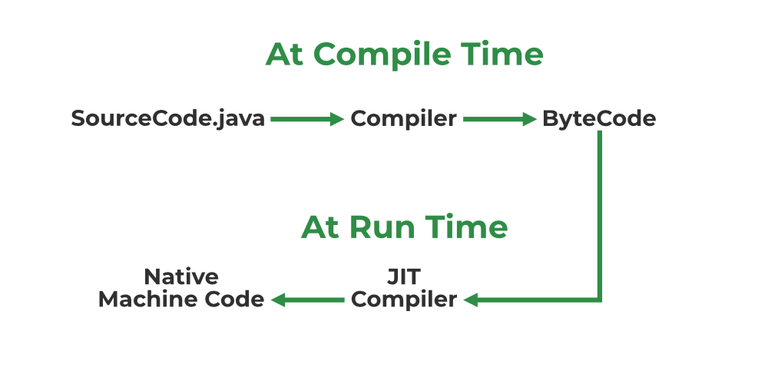
JVM is the engine that executes Java bytecode.

JRE provides the runtime environment for Java applications.

JDK is the development kit that includes the JRE and tools for developing Java applications.

**What is JIT?**

JIT stands for (Just-in-Time) compiler is a part of JRE(Java Runtime Environment), it is used for improve performance of the Java applications during run-time. The use of JIT is mentioned in step by step process mentioned below:



**Bytecode Execution**: When you write and compile a Java program, the Java compiler (javac) converts your source code into bytecode, which is platform-independent and runs on the Java Virtual Machine (JVM).

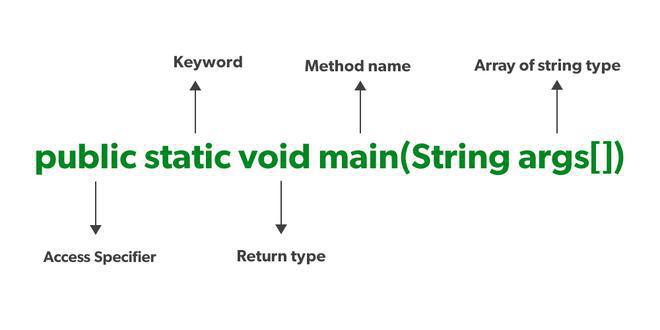
**Interpretation and JIT Compilation**:

* Initially, the JVM interprets the bytecode, executing it line by line. This interpretation is generally slower because each bytecode instruction is translated to machine code every time it is executed.
* The JIT compiler improves performance by identifying "hot spots" in the bytecode—sections of code that are executed frequently.
* The JIT compiler then compiles these hot spots into native machine code, which the CPU can execute directly. This compiled code is stored in memory, so future executions of the same code can be done quickly without further interpretation.

**Optimizations**: The JIT compiler also applies various optimizations during the compilation process, such as inlining methods, eliminating dead code, and optimizing loops, to make the generated machine code run more efficiently.

**Explain public static void main(String args[]) in Java.**

Unlike any other programming language like C, C++, etc. In Java, we declared the main function as a public static void main (String args[]). The meanings of the terms are mentioned below:



public: the public is the access modifier responsible for mentioning who can access the element or the method and what is the limit. It is responsible for making the main function globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

static: static is a keyword used so that we can use the element without initiating the class so to avoid the unnecessary allocation of the memory.

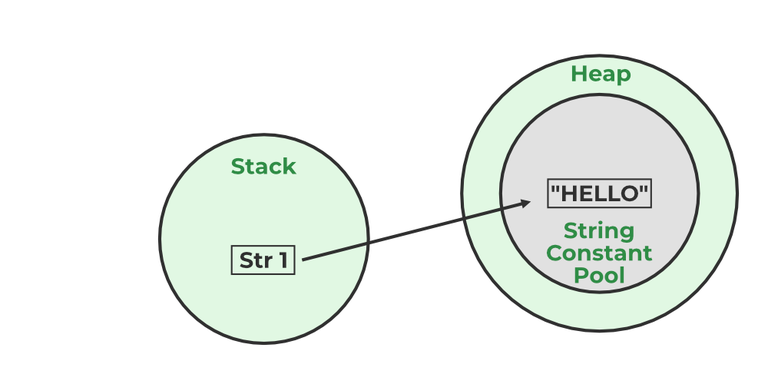
void: void is a keyword and is used to specify that a method doesn’t return anything. As the main function doesn’t return anything we use void.

main: main represents that the function declared is the main function. It helps JVM to identify that the declared function is the main function.

String args[]: It stores Java command-line arguments and is an array of type java.lang.String class.

**What is Java String Pool?**

A Java String Pool is a place in heap memory where all the strings defined in the program are stored. A separate place in a stack is there where the variable storing the string is stored. Whenever we create a new string object, JVM checks for the presence of the object in the String pool, If String is available in the pool, the same object reference is shared with the variable, else a new object is created.



//String str1="Hello";

// "Hello" will be stored in String Pool

// str1 will be stored in stack memory

**What will happen if we don’t declare the main as static?**

We can declare the main method without using static and without getting any errors. But, the main method will not be treated as the entry point to the application or the program.

**6) How many types of memory areas are allocated by JVM?**

**Heap Memory:**

This is where all the objects and their corresponding instance variables are allocated. The heap is shared among all threads in a Java application.

The heap is managed by the Garbage Collector, which reclaims memory used by objects that are no longer reachable.

**Stack Memory:**

Each thread in a Java application has its own stack memory. The stack memory stores method calls, local variables, and intermediate results. When a method is called, a new frame is created and pushed onto the stack. When the method execution is completed, the frame is popped from the stack.

**Method Area:**

Also known as the Class Area this memory area stores metadata about the classes and methods, such as bytecode, static variables, method information, and constant pool.

**Program Counter (PC) Register:**

Each thread has its own PC register. The PC register contains the address of the JVM instruction currently being executed.

**Native Method Stack:**

This memory area is used for native (non-Java) method execution. It is separate from the Java stack and is used for native method invocations. The memory used by this stack is allocated in the same manner as the Java stack.

**7) What is JIT compiler?**

It is used to improve the performance of java application.

The Java Virtual Machine (JVM) interpretes the bytecode. The JVM uses the HotSpot profiler to identify frequently executed or time-consuming sections of code, called "hot spots." Then JIT compiler compiles these hot spots into native machine code while the program is running (at runtime). The JVM then executes the native machine code directly, which is faster, and the JIT compiler continuously optimizes the code based on actual runtime behavior.

**10) What gives Java its 'write once and run anywhere' nature?**

The bytecode. Java compiler converts the Java programs into the class file (Byte Code) which is the intermediate language between source code and machine code. This bytecode is not platform specific and can be executed on any computer.

**14) If I don't provide any arguments on the command line, then what will the value stored in the String array passed into the main() method, empty or NULL?**

It is empty, but not null.

**15) What if I write static public void instead of public static void?**

The program compiles and runs correctly because the order of specifiers doesn't matter in Java.

**17) What are the various access specifiers in Java?**

In Java, access specifiers are the keywords which are used to define the access scope of the method, class, or a variable. In Java, there are four access specifiers given below.

* **Public** The classes, methods, or variables defined as public, can be accessed by any class or method.
* **Protected** Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.
* **Default** Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.
* **Private** The private class, methods, or variables defined as private can be accessed within the class only.

**18) What is the purpose of static methods and variables?**

The methods or variables defined as static are shared among all the objects of the class. The static variables are stored in the class area, and we do not need to create the object to access such variables. Therefore, static is used in the case, where we need to define variables or methods which are common to all the objects of the class.

For example, In the class simulating the collection of the students in a college, the name of the college is the common attribute to all the students. Therefore, the college name will be defined as **static**.

**20) What is the output of the following Java program?**

**class** Test

{

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

    {

        System.out.println(10 + 20 + "Javatpoint");

        System.out.println("Javatpoint" + 10 + 20);

    }

}

The output of the above code will be

30Javatpoint

Javatpoint1020

**Explanation**

In the first case, 10 and 20 are treated as numbers and added to be 30. Now, their sum 30 is treated as the string and concatenated with the string **Javatpoint**. Therefore, the output will be **30Javatpoint**.

In the second case, the string Javatpoint is concatenated with 10 to be the string **Javatpoint10** which will then be concatenated with 20 to be **Javatpoint1020**.

**21) What is the output of the following Java program?**

**class** Test

{

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

    {

        System.out.println(10 \* 20 + "Javatpoint");

        System.out.println("Javatpoint" + 10 \* 20);

    }

}

The output of the above code will be

200Javatpoint

Javatpoint200

**Explanation**

In the first case, The numbers 10 and 20 will be multiplied first and then the result 200 is treated as the string and concatenated with the string **Javatpoint** to produce the output **200Javatpoint**.

In the second case, The numbers 10 and 20 will be multiplied first to be 200 because the precedence of the multiplication is higher than addition. The result 200 will be treated as the string and concatenated with the string **Javatpoint**to produce the output as **Javatpoint200**.

**22) What is the output of the following Java program?**

**class** Test

{

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

    {

**for**(**int** i=0; 0; i++)

        {

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

        }

    }

}

The above code will give the compile-time error because the for loop demands a boolean value in the second part and we are providing an integer value, i.e., 0.

**23) What is object-oriented paradigm?**

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects". Objects are the instances of classes. OOP focuses on using these objects to design and build applications. The main principles of OOP are encapsulation, inheritance, polymorphism, and abstraction

**What is the need for OOPs?**

There are many reasons why OOPs is mostly preferred, but the most important among them are:

* OOPs helps users to understand the software easily, although they don’t know the actual implementation.
* With OOPs, the readability, understandability, and maintainability of the code increase multifold.
* Even very big software can be easily written and managed easily using OOPs.

**What are some advantages of using OOPs?**

* OOPs is very helpful in solving very complex level of problems.
* Highly complex programs can be created, handled, and maintained easily using object-oriented programming.
* OOPs, promote code reuse, thereby reducing redundancy.
* OOPs also helps to hide the unnecessary details with the help of Data Abstraction.
* OOPs, are based on a bottom-up approach, unlike the Structural programming paradigm, which uses a top-down approach.
* Polymorphism offers a lot of flexibility in OOPs.

**Java Classes**

Classes are fundamentals of the object-oriented programming (OOP) paradigm used by Java. In Java, a class is a blueprint for creating objects . For example, Student is a class while a particular student named Ravi is an object.

**Properties of Java Classes**

* Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
* Class does not occupy memory.
* Class is a group of variables of different data types and a group of methods.
* A Class in Java can contain:
* Data member, Method, Constructor, Nested Class, Interface

**what are objects**

In object-oriented programming (OOP), an object is an instance of a class, that are created to access/use the variables and methods of a class. It is real world entity that has properties like which are identity(name), state (variables) and behavior (methods). They interact with one another to perform tasks within a program.

* State: It is represented by attributes/variable of an object. It also reflects the properties of an object.
* Behavior: It is represented by the methods of an object. It also reflects the response of an object with other objects.
* Identity: It gives a unique name to an object and enables one object to interact with other objects.

**What is method**

In Java a method is a reusable block of code that performs a specific task. Methods define the behaviors that an object can perform.

**25) What is the difference between an object-oriented programming language and object-based programming language?**

* Object-oriented languages follow all the concepts of OOPs whereas, the object-based language doesn't follow all the concepts of OOPs like inheritance and polymorphism.
* Object-oriented languages do not have the inbuilt objects whereas Object-based languages have the inbuilt objects, for example, JavaScript has window object.
* Examples of object-oriented programming are Java, C#, Smalltalk, etc. whereas the examples of object-based languages are JavaScript, VBScript, etc.

26) What will be the initial value of an object reference which is defined as an instance variable?

All object references are initialized to null in Java.

Core Java - OOPs Concepts: Constructor Interview Questions

27) What is the constructor?

The constructor can be defined as the special type of method that is used to initialize the state of an object. It is invoked when the class is instantiated, and the memory is allocated for the object. Every time, an object is created using the **new** keyword, the default constructor of the class is called. The name of the constructor must be similar to the class name. The constructor must not have an explicit return type.

[More Details.](https://www.javatpoint.com/constructor)

28) How many types of constructors are used in Java?

Based on the parameters passed in the constructors, there are two types of constructors in Java.

1. Default Constructor

If you do not create any constructor in the class, Java provides a default constructor. It takes no arguments and initializes the object with default values (e.g., null for reference types, 0 for numeric types, false for boolean).

2. No-Args (No Argument) Constructor

As the name specifies, the No-argument constructor does not accept any argument. By using the No-Args constructor you can initialize the class data members and perform various activities that you want on object creation.

3. Parameterized Constructor

A constructor with one or more arguments is called a parameterized constructor. A parameterized constructor is used when you have a class with multiple instance variables, and you want to initialize them with specific values at the time of object creation.

29) What is the purpose of a default constructor?

The purpose of the default constructor is to assign the default value to the objects. The java compiler creates a default constructor implicitly if there is no constructor in the class.

**class** Student3{

**int** id;

String name;

**void** display(){System.out.println(id+" "+name);}

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

Student3 s1=**new** Student3();

Student3 s2=**new** Student3();

s1.display();

s2.display();

}

}

Output:

0 null

0 null

**Explanation:** In the above class, you are not creating any constructor, so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

  
[More Details.](https://www.javatpoint.com/constructor)

30) Does constructor return any value?

In Java, constructors do not return a value in the traditional sense. They do not have a return type, not even void. However, when a constructor is called, it implicitly returns the newly created object instance of the class it belongs to. The primary role of a constructor is to initialize the object's state, not to return a value.

31)Is constructor inherited?

No, The constructor is not inherited.

32) Can you make a constructor final?

No, the constructor can't be final.

33) Can we overload the constructors?

Yes, the constructors can be overloaded by changing the number of arguments accepted by the constructor or by changing the data type of the parameters. Consider the following example.

**class** Test

{

**int** i;

**public** Test(**int** k)

    {

        i=k;

    }

**public** Test(**int** k, **int** m)

    {

        System.out.println("Hi I am assigning the value max(k, m) to i");

**if**(k>m)

        {

            i=k;

        }

**else**

        {

            i=m;

        }

    }

}

**public** **class** Main

{

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

    {

        Test test1 = **new** Test(10);

        Test test2 = **new** Test(12, 15);

        System.out.println(test1.i);

        System.out.println(test2.i);

    }

}

In the above program, The constructor Test is overloaded with another constructor. In the first call to the constructor, The constructor with one argument is called, and i will be initialized with the value 10. However, In the second call to the constructor, The constructor with the 2 arguments is called, and i will be initialized with the value 15.

34) What do you understand by copy constructor in Java?

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

//Java program to initialize the values from one object to another

**class** Student6{

**int** id;

    String name;

    //constructor to initialize integer and string

    Student6(**int** i,String n){

    id = i;

    name = n;

    }

    //constructor to initialize another object

    Student6(Student6 s){

    id = s.id;

    name =s.name;

    }

**void** display(){System.out.println(id+" "+name);}

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

    Student6 s1 = **new** Student6(111,"Karan");

    Student6 s2 = **new** Student6(s1);

    s1.display();

    s2.display();

   }

111 Karan

111 Karan

35) What are the differences between the constructors and methods?

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as class name. |

36) What is the output of the following Java program?

**public** **class** Test

{

    Test(**int** a, **int** b)

    {

        System.out.println("a = "+a+" b = "+b);

    }

    Test(**int** a, **float** b)

    {

        System.out.println("a = "+a+" b = "+b);

    }

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

    {

**byte** a = 10;

**byte** b = 15;

        Test test = **new** Test(a,b);

    }

}

a = 10 b = 15

Here, the data type of the variables a and b, i.e., byte gets promoted to int, and the first parameterized constructor with the two integer parameters is called.

37) What is the output of the following Java program?

**class** Test

{

**int** i;

}

**public** **class** Main

{

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

    {

        Test test = **new** Test();

        System.out.println(test.i);

    }

}

The output of the program is 0 because the variable i is initialized to 0 internally. As we know that a default constructor is invoked implicitly if there is no constructor in the class, the variable i is initialized to 0 since there is no constructor in the class.

38) What is the output of the following Java program?

**class** Test

{

**int** test\_a, test\_b;

    Test(**int** a, **int** b)

    {

    test\_a = a;

    test\_b = b;

    }

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

    {

        Test test = **new** Test();

        System.out.println(test.test\_a+" "+test.test\_b);

    }

}

There is a **compiler error** in the program because there is a call to the default constructor in the main method which is not present in the class. However, there is only one parameterized constructor in the class Test. Therefore, no default constructor is invoked by the constructor implicitly.

Core Java - OOPs Concepts: static keyword Interview Questions

39) What is the static variable?

Static variable belongs to the class rather than the object. The static variable is used to refer to the common property of all objects or we can say that static variables are shared among all the objects (that is not unique for each object), e.g., The company name of employees, college name of students, etc. Static variable gets memory only once in the class area at the time of class loading. Using a static variable makes your program more memory efficient (it saves memory).

//Program of static variable

**class** Student8{

**int** rollno;

   String name;

**static** String college ="ITS";

   Student8(**int** r,String n){

   rollno = r;

   name = n;

   }

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

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

 Student8 s1 = **new** Student8(111,"Karan");

 Student8 s2 = **new** Student8(222,"Aryan");

 s1.display();

 s2.display();

 }

}

111 Karan ITS

222 Aryan ITS

  
[More Details.](https://www.javatpoint.com/static-keyword-in-java)

40) What is the static method?

* A static method belongs to the class rather than the object.
* There is no need to create the object to call the static methods.
* A static method can access and change the value of the static variable.

[More Details.](https://www.javatpoint.com/static-keyword-in-java)

41) What are the restrictions that are applied to the Java static methods?

Two main restrictions are ap plied to the static methods.

* The static method can not use non-static data member or call the non-static method directly.
* this and super cannot be used in static context as they are non-static.

42) Why is the main method static?

Because the object is not required to call the static method. If we make the main method non-static, JVM will have to create its object first and then call main() method which will lead to the extra memory allocation.[More Details.](https://www.javatpoint.com/static-keyword-in-java)

43) Can we override the static methods?

No, we can't override static methods.

44) What is the static block?

Static block is used to initialize the static data member. It is executed before the main method, at the time of classloading.

**class** A2{

**static**{System.out.println("static block is invoked");}

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

   System.out.println("Hello main");

  }

}

static block is invoked

Hello main

[More Details.](https://www.javatpoint.com/static-keyword-in-java)

45) Can we execute a program without main() method?

Ans) No, It was possible before JDK 1.7 using the static block. Since JDK 1.7, it is not possible. [More Details.](https://www.javatpoint.com/static-keyword-in-java)

46) What if the static modifier is removed from the signature of the main method?

Program compiles. However, at runtime, It throws an error "NoSuchMethodError."

47) What is the difference between static (class) method and instance method?

|  |  |
| --- | --- |
| **static or class method** | **instance method** |
| 1)A method that is declared as static is known as the static method. | A method that is not declared as static is known as the instance method. |
| 2)We don't need to create the objects to call the static methods. | The object is required to call the instance methods. |
| 3)Non-static (instance) members cannot be accessed in the static context (static method, static block, and static nested class) directly. | Static and non-static variables both can be accessed in instance methods. |
| 4)For example: public static int cube(int n){ return n\*n\*n;} | For example: public void msg(){...}. |

48) Can we make constructors static?

As we know that the static context (method, block, or variable) belongs to the class, not the object. Since Constructors are invoked only when the object is created, there is no sense to make the constructors static. However, if you try to do so, the compiler will show the compiler error.

49) Can we make the abstract methods static in Java?

In Java, if we make the abstract methods static, It will become the part of the class, and we can directly call it which is unnecessary. Calling an undefined method is completely useless therefore it is not allowed.

50) Can we declare the static variables and methods in an abstract class?

Yes, we can declare static variables and methods in an abstract method. As we know that there is no requirement to make the object to access the static context, therefore, we can access the static context declared inside the abstract class by using the name of the abstract class. Consider the following example.

**abstract** **class** Test

{

**static** **int** i = 102;

**static** **void** TestMethod()

    {

        System.out.println("hi !! I am good !!");

    }

}

**public** **class** TestClass **extends** Test

{

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

    {

        Test.TestMethod();

        System.out.println("i = "+Test.i);

    }

}

**Output**

hi !! I am good !!

i = 102

Core Java - OOPs Concepts: Inheritance Interview Questions

51) 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.

  
[More Details.](https://www.javatpoint.com/this-keyword)

52) 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.

53) 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**

Test.java:5: error: cannot assign a value to final variable this

this = null;

^

1 error

54) 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

55) 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

**56) What are the advantages of passing this into a method instead of the current class object itself?**

As we know, that this refers to the current class object, therefore, it must be similar to the current class object. However, there can be two main advantages of passing this into a method instead of the current class object.

* this is a final variable. Therefore, this cannot be assigned to any new value whereas the current class object might not be final and can be changed.
* this can be used in the synchronized block.

57) What is the Inheritance?

Inheritance is a mechanism by which one class acquires all the properties and behavior of another class It is used for Code Reusability and Method Overriding. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

There are five types of inheritance

* Single-level inheritance
* Multi-level inheritance
* Multiple Inheritance
* Hierarchical Inheritance
* Hybrid Inheritance

Java supports single, multilevel and hierachical

[More Details.](https://www.javatpoint.com/inheritance-in-java)

In java programming, multiple and hybrid inheritance is supported through interface only.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



**Single Inheritance Example**

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** TestInheritance{

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

Dog d=**new** Dog();

d.bark();

d.eat();

}}

Output:

barking...

eating...

**Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** BabyDog **extends** Dog{

**void** weep(){System.out.println("weeping...");}

}

**class** TestInheritance2{

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

BabyDog d=**new** BabyDog();

d.weep();

d.bark();

d.eat();

}}

Output:

weeping...

barking...

eating...

**Hierarchical Inheritance Example**

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** Cat **extends** Animal{

**void** meow(){System.out.println("meowing...");}

}

**class** TestInheritance3{

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

Cat c=**new** Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}}

Output:

meowing...

eating...

58) Why is Inheritance used in Java?

There are various advantages of using inheritance in Java that is given below.

* Inheritance provides code reusability. The derived class does not need to redefine the method of base class unless it needs to provide the specific implementation of the method.
* Runtime polymorphism cannot be achieved without using inheritance.
* We can simulate the inheritance of classes with the real-time objects which makes OOPs more realistic.
* Inheritance provides data hiding. The base class can hide some data from the derived class by making it private.
* Method overriding cannot be achieved without inheritance. By method overriding, we can give a specific implementation of some basic method contained by the base class.

59) Which class is the superclass for all the classes?

The object class is the superclass of all other classes in Java.

60) Why is multiple inheritance not supported in java?

To avoid ambiguity, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

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

**class** A{

**void** msg(){System.out.println("Hello");}

}

**class** B{

**void** msg(){System.out.println("Welcome");}

}

**class** C **extends** A,B{//suppose if it were

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

   C obj=**new** C();

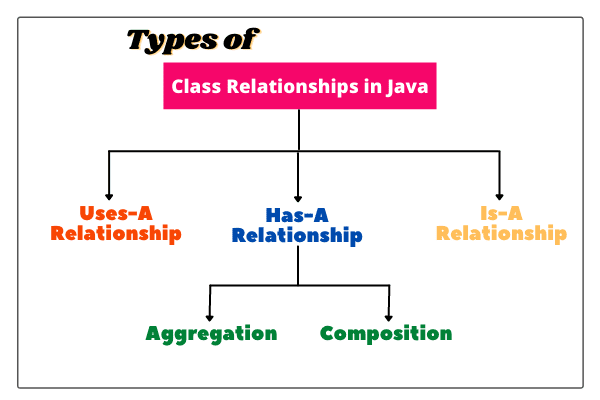
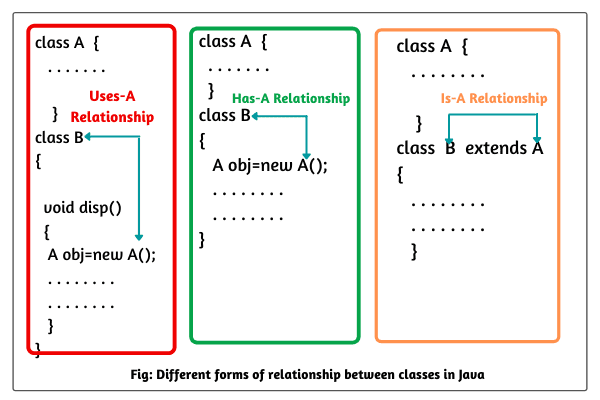
   obj.msg();//Now which msg() method would be invoked?

}

}

Compile Time Error

**Types of Relationship among Classes in Java**

There are three most common relationships among classes in Java that are as follows:

a. Dependence (“Uses-A”)

b. Association (“Has-A”)

c. Inheritance (“Is-A”)

**Dependence (Uses-A) Relationship in Java**

When a method of a class uses an object of another class, it is called dependence or Uses-A relationship in java.

Look at the below figure where a method display() of class B uses an object of class A. So, we can say that class A depends on another class B if it uses an object of class A.

**Association (“Has-A”) Relationship in Java**

Association is another relationship also known as “Has-A” relationship. When an object of one class is created as data member/ or used inside another class, it is called association relationship in java or simply Has-A relationship.

Look at the above fig. where an object of class A is created as data member/used inside another class B.

**There are two types of Has-A relationship that are as follows:**

**Aggregation**: Aggregation focuses on establishing a loose Has-A relationship between two classes. In other words, two objects have their own life cycle. That is, this objects can exist on its own. For example Music system and car can exists on their own.

**Composition**: Composition focuses on establishing a strong Has-A relationship between the two classes. In other words, two objects cannot have their own life cycle. That is, a object cannot exist on its own. If one composite object is destroyed, all its parts are also be deleted. For example, Car and car engine cannot exists on their own.

**Inheritance (“Is-A”) Relationship in Java**

In Is-A relationship one class is obtains the features and attributes of another class by using concept of inheritance. In other words, Is-A relationship defines the relationship between two classes, where one class extends another class.

Look at the above figure where a class B makes a relationship with class A by the keyword “extends” and can inherit data members from class A.

64) Why does Java not support pointers?

The pointer is a variable that refers to the memory address. They are not used in Java because they are unsafe(unsecured) and complex to understand.

65) What is super in java?

The **super** keyword in Java is a reference variable that is used to refer to the immediate parent class object. Whenever you create the instance of the subclass, an instance of the parent class is created implicitly which is referred by super reference variable. The super() is called in the class constructor implicitly by the compiler if there is no super or this.

**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

[More Details.](https://www.javatpoint.com/super-keyword)

66) How can constructor chaining be done by using the super keyword?

**class** Person

{

    String name,address;

**int** age;

**public** Person(**int** age, String name, String address)

    {

**this**.age = age;

**this**.name = name;

**this**.address = address;

    }

}

**class** Employee **extends** Person

{

**float** salary;

**public** Employee(**int** age, String name, String address, **float** salary)

    {

**super**(age,name,address);

**this**.salary = salary;

    }

}

**public** **class** Test

{

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

    {

        Employee e = **new** Employee(22, "Mukesh", "Delhi", 90000);

        System.out.println("Name: "+e.name+" Salary: "+e.salary+" Age: "+e.age+" Address: "+e.address);

    }

}

**Output**

Name: Mukesh Salary: 90000.0 Age: 22 Address: Delhi

67) What are the main uses of the super keyword?

There are the following uses of super keyword.

* super can be used to refer to the immediate parent class instance variable.
* super can be used to invoke the immediate parent class method.
* super() can be used to invoke immediate parent class constructor.

68) What are the differences between this and super keyword?

There are the following differences between this and super keyword.

* The super keyword always points to the parent class contexts whereas this keyword always points to the current class context.
* The super keyword is primarily used for initializing the base class variables within the derived class constructor whereas this keyword primarily used to differentiate between local and instance variables when passed in the class constructor.
* The super and this must be the first statement inside constructor otherwise the compiler will throw an error.

69) What is the output of the following Java program?

**class** Person

{

**public** Person()

    {

        System.out.println("Person class constructor called");

    }

}

**public** **class** Employee **extends** Person

{

**public** Employee()

    {

        System.out.println("Employee class constructor called");

    }

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

    {

        Employee e = **new** Employee();

    }

}

**Output**

Person class constructor called

Employee class constructor called

**Explanation**

The super() is implicitly invoked by the compiler if no super() or this() is included explicitly within the derived class constructor. Therefore, in this case, The Person class constructor is called first and then the Employee class constructor is called.

70) Can you use this() and super() both in a constructor?

No, because this() and super() must be the first statement in the class constructor.

**Example:**

**public** **class** Test{

    Test()

     {

**super**();

**this**();

         System.out.println("Test class object is created");

     }

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

     Test t = **new** Test();

     }

}

Output:

Test.java:5: error: call to this must be first statement in constructor

71) What is object cloning?

The object cloning is used to create the exact copy of an object. The clone() method of the 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.

Core Java - OOPs Concepts: Method Overloading Interview Questions

**Constructor overloading**

In Java, constructor overloading means to define multiple constructors but with different signatures. Constructor overloading is a technique of having more than one constructor in the same class with different parameter lists.

In other words, defining two or more constructors with the same name in a class but with different signatures is called constructor overloading. We arrange them in a such a way that each constructor performs a different task./

**What is polymorphism**

Polymorphism refers to the ability of an entity (method, or object) to operate in different ways based on the context. That is, when a single entity behaves differently in different cases, it is called polymorphism.

**Basically, there are two types of polymorphism in Java. They are:**

* Static polymorphism
* Dynamic polymorphism

**Static/Compile-time Polymorphism**

A polymorphism that exhibited during compilation is called static polymorphism in Java. In static polymorphism, the behavior of a method is decided at compile-time.

Hence, Java compiler binds method calls with method definition/body during compilation. Therefore, this type of polymorphism is also called compile-time polymorphism in Java.

Since binding is performed at compile-time, it is also known as early binding. Compile-time polymorphism can be achieved/implemented by method overloading in Java.

Method overloading is a mechanism in which a class has multiple methods having the same name but different signatures. It is one of the ways that Java implements polymorphism.

**Dynamic/Runtime-time Polymorphism/ dynamic method dispatch**

A polymorphism that is exhibited at runtime is called dynamic polymorphism in java. In dynamic polymorphism, the behavior of a method is decided at runtime. Therefore, the JVM (Java Virtual Machine) binds the method call with the method definition/body at runtime and invokes the relevant method during runtime when the method is called.

This happens because objects are created at runtime and the method is called using an object of the class. The Java compiler has no awareness of the method to be called on an instance during compilation. Therefore, JVM invokes the relevant method during runtime.

Dynamic or runtime polymorphism can be achieved/implemented in Java using the method overriding.

Method overriding is a mechanism where a method of Base class is overridden in the derived class to provide a more specific implementation. The signature of method in both base and derived classes is the same but they only differ in their implementation.

**Method Overloading**

When a class has more than one method having the same name but with different parameter lists, this feature is called method overloading in Java.

In other words, when we declare multiple methods with the same name but with different method signatures, all these methods are overloaded methods in Java. We refer this feature to as method overloading.

**Method Overloading Rules in Java**

1. The method name must be the same.

2. Parameters must be different, i.e. each overloaded method must take a unique list of parameter types. We can change the parameters in one of the following three ways:

* Data type of parameters
* Number of parameters
* Sequence of data type of parameters

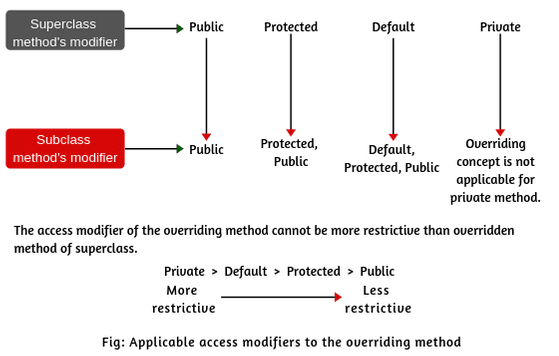
**Method overriding**

Method overriding method of superclass is overridden by the nethod of subclass to provide more specific implementation, it is called method overriding. That is child class method overrides the parent class method.

To override a method in a subclass, the method must be defined in the subclass using the same signature and same return type as in its superclass as shown in the below figure.

**Method Overriding Rules in Java**

1. Subclass method name must be the same as the superclass method name.
2. The parameters of the subclass method must be the same as the superclass method parameters. i.e. In overriding, method name and argument types must be matched. In other words, the method signature must be the same or matched.
3. Must be Is-A relationship (Inheritance).
4. Subclass method’s return type must be the same as the superclass method return type. But this rule is applicable until the Java 1.4 version only. From the Java 1.5 version onwards, covariant return types are also allowed.
5. Subclass method’s access modifier must be the same or less than the superclass method access modifier. Look at the below figure to understand better.



### 101) 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. |

### 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.

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.

### 103) Can you achieve Runtime Polymorphism by data members?

No, because method overriding is used to achieve runtime polymorphism and data members cannot be overridden. We can override the member functions but not the data members. Consider the example given below.

**class** Bike{

**int** speedlimit=90;

}

**class** Honda3 **extends** Bike{

**int** speedlimit=150;

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

  Bike obj=**new** Honda3();

  System.out.println(obj.speedlimit);//90

   }

Output:

90`

### 104) 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();

 }

}

### 105) What is the output of the following Java program?

**class** BaseTest

{

**void** print()

  {

    System.out.println("BaseTest:print() called");

  }

}

**public** **class** Test **extends** BaseTest

{

**void** print()

  {

    System.out.println("Test:print() called");

  }

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

  {

    BaseTest b = **new** Test();

    b.print();

  }

}

**Output**

Test:print() called

**Explanation**

It is an example of Dynamic method dispatch. The type of reference variable b is determined at runtime. At compile-time, it is checked whether that method is present in the Base class. In this case, it is overridden in the child class, therefore, at runtime the derived class method is called.

### 106) 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

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

73) Why is method overloading not possible by changing the return type in java?

In Java, method overloading is not possible by changing the return type of the program due to avoid the ambiguity.

**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

[More Details.](https://www.javatpoint.com/method-overloading-in-java)

74) Can we overload the methods by making them static?

No, We cannot overload the methods by just applying the static keyword to them(number of parameters and types are the same). Consider the following example.

**public** **class** Animal

{

**void** consume(**int** a)

    {

        System.out.println(a+" consumed!!");

    }

**static** **void** consume(**int** a)

    {

        System.out.println("consumed static "+a);

    }

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

    {

        Animal a = **new** Animal();

        a.consume(10);

        Animal.consume(20);

    }

}

**Output**

Animal.java:7: error: method consume(int) is already defined in class Animal

static void consume(int a)

^

Animal.java:15: error: non-static method consume(int) cannot be referenced from a static context

Animal.consume(20);

^

2 errors

75) Can we overload the main() method?

Yes, we can have any number of main methods in a Java program by using method overloading.

76) What is method overloading with type promotion?

By Type promotion is method overloading, we mean that one data type can be promoted to another implicitly if no exact matching is found.



As displayed in the above diagram, the byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on. Consider the following example.

**class** OverloadingCalculation1{

**void** sum(**int** a,**long** b){System.out.println(a+b);}

**void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}

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

  OverloadingCalculation1 obj=**new** OverloadingCalculation1();

  obj.sum(20,20);//now second int literal will be promoted to long

  obj.sum(20,20,20);

  }

}

**Output**

40

60

77) What is the output of the following Java program?

**class** OverloadingCalculation3{

**void** sum(**int** a,**long** b){System.out.println("a method invoked");}

**void** sum(**long** a,**int** b){System.out.println("b method invoked");}

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

  OverloadingCalculation3 obj=**new** OverloadingCalculation3();

  obj.sum(20,20);//now ambiguity

  }

}

**Output**

OverloadingCalculation3.java:7: error: reference to sum is ambiguous

obj.sum(20,20);//now ambiguity

^

both method sum(int,long) in OverloadingCalculation3

and method sum(long,int) in OverloadingCalculation3 match

1 error

**Explanation**

There are two methods defined with the same name, i.e., sum. The first method accepts the integer and long type whereas the second method accepts long and the integer type. The parameter passed that are a = 20, b = 20. We can not tell that which method will be called as there is no clear differentiation mentioned between integer literal and long literal. This is the case of ambiguity. Therefore, the compiler will throw an error.

79) Can we override the static method?

No, you can't override the static method because they are the part of the class, not the object.

80) Why can we not override static method?

It is because the static method is the part of the class, and it is bound with class whereas instance method is bound with the object, and static gets memory in class area, and instance gets memory in a heap.

81) Can we override the overloaded method?

Yes.

82) Difference between method Overloading and Overriding.

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| 1) Method overloading increases the readability of the program. | Method overriding provides the specific implementation of the method that is already provided by its superclass. |
| 2) Method overloading occurs within the class. | Method overriding occurs in two classes that have IS-A relationship between them. |
| 3) In this case, the parameters must be different. | In this case, the parameters must be the same. |

83) Can we override the private methods?

No, we cannot override the private methods because the scope of private methods is limited to the class and we cannot access them outside of the class.

84) Can we change the scope of the overridden method in the subclass?

Yes, we can change the scope of the overridden method in the subclass. However, we must notice that we cannot decrease the accessibility of the method. The following point must be taken care of while changing the accessibility of the method.

* The private can be changed to protected, public, or default.
* The protected can be changed to public or default.
* The default can be changed to public.
* The public will always remain public.

85) Can we modify the throws clause of the superclass method while overriding it in the subclass?

Yes, we can modify the throws clause of the superclass method while overriding it in the subclass. However, there are some rules which are to be followed while overriding in case of exception handling.

* If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception, but it can declare the unchecked exception.
* If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

86) What is the output of the following Java program?

**class** Base

{

**void** method(**int** a)

    {

        System.out.println("Base class method called with integer a = "+a);

    }

**void** method(**double** d)

    {

        System.out.println("Base class method called with double d ="+d);

    }

}

**class** Derived **extends** Base

{

    @Override

**void** method(**double** d)

    {

        System.out.println("Derived class method called with double d ="+d);

    }

}

**public** **class** Main

{

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

    {

**new** Derived().method(10);

    }

}

**Output**

Base class method called with integer a = 10

**Explanation**

The method() is overloaded in class Base whereas it is derived in class Derived with the double type as the parameter. In the method call, the integer is passed.

87) Can you have virtual functions in Java?

Yes, all functions in Java are virtual by default.

89) What is the output of the following Java program?

**class** Base

{

**public** **void** baseMethod()

    {

        System.out.println("BaseMethod called ...");

    }

}

**class** Derived **extends** Base

{

**public** **void** baseMethod()

    {

        System.out.println("Derived method called ...");

    }

}

**public** **class** Test

{

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

    {

        Base b = **new** Derived();

        b.baseMethod();

    }

}

**Output**

Derived method called ...

**Explanation**

The method of Base class, i.e., baseMethod() is overridden in Derived class. In Test class, the reference variable b (of type Base class) refers to the instance of the Derived class. Here, Runtime polymorphism is achieved between class Base and Derived. At compile time, the presence of method baseMethod checked in Base class, If it presence then the program compiled otherwise the compiler error will be shown. In this case, baseMethod is present in Base class; therefore, it is compiled successfully. However, at runtime, It checks whether the baseMethod has been overridden by Derived class, if so then the Derived class method is called otherwise Base class method is called. In this case, the Derived class overrides the baseMethod; therefore, the Derived class method is called.

Core Java - OOPs Conepts: final keyword Interview Questions

90) What is the final variable?

In Java, the final variable is used to restrict the user from updating it. If we initialize the final variable, we can't change its value. In other words, we can say that the final variable once assigned to a value, can never be changed after that. The final variable which is not assigned to any value can only be assigned through the class constructor.



**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

[More Details.](https://www.javatpoint.com/final-keyword)

**Final**

Final is a keyword that is used to restrict the user in Java programming. It is a non-access modifier that is only applicable to a variable, method, or class.

* The value of the final variable cannot be changed.
* A final method cannot be overridden.
* A final class cannot be inherited.

**Final keyword has three different uses in Java. They are as:**

1. To declare a constant.

2. To prevent inheritance.

3. To prevent method from being overridden.

91) What is the final method?

If we change any method to a final method, we can't override it. [More Details.](https://www.javatpoint.com/final-keyword)

**class** Bike{

**final** **void** run(){System.out.println("running");}

}

**class** Honda **extends** Bike{

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

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

   Honda honda= **new** Honda();

   honda.run();

   }

}

Output:Compile Time Error

92) What is the final class?

If we make any class final, we can't inherit it into any of the subclasses.

**final** **class** Bike{}

**class** Honda1 **extends** Bike{

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

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

  Honda1 honda= **new** Honda1();

  honda.run();

  }

}

Output:Compile Time Error

[More Details.](https://www.javatpoint.com/final-keyword)

93) What is the final blank variable?

A final variable, not initialized at the time of declaration, is known as the final blank variable. We can't initialize the final blank variable directly. Instead, we have to initialize it by using the class constructor. It is useful in the case when the user has some data which must not be changed by others, for example, PAN Number. Consider the following example:

**class** Student{

**int** id;

String name;

**final** String PAN\_CARD\_NUMBER;

...

}

[More Details.](https://www.javatpoint.com/final-keyword)

94) Can we initialize the final blank variable?

Yes, if it is not static, we can initialize it in the constructor. If it is static blank final variable, it can be initialized only in the static block.[More Details.](https://www.javatpoint.com/final-keyword)

95) Can you declare the main method as final?

Yes, We can declare the main method as public static final void main(String[] args){}.

96) What is the output of the following Java program?

**class** Main {

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

**final** **int** i;

   i = 20;

   System.out.println(i);

 }

}

**Output**

20

**Explanation**

Since i is the blank final variable. It can be initialized only once. We have initialized it to 20. Therefore, 20 will be printed.

97) What is the output of the following Java program?

**class** Base

{

**protected** **final** **void** getInfo()

    {

        System.out.println("method of Base class");

    }

}

**public** **class** Derived **extends** Base

{

**protected** **final** **void** getInfo()

    {

        System.out.println("method of Derived class");

    }

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

    {

        Base obj = **new** Base();

        obj.getInfo();

    }

}

**Output**

Derived.java:11: error: getInfo() in Derived cannot override getInfo() in Base

protected final void getInfo()

^

overridden method is final

1 error

**Explanation**

The getDetails() method is final; therefore it can not be overridden in the subclass.

98) Can we declare a constructor as final?

The constructor can never be declared as final because it is never inherited. Constructors are not ordinary methods; therefore, there is no sense to declare constructors as final. However, if you try to do so, The compiler will throw an error.

99) Can we declare an interface as final?

No, we cannot declare an interface as final because the interface must be implemented by some class to provide its definition. Therefore, there is no sense to make an interface final. However, if you try to do so, the compiler will show an error.

## Core Java - OOPs Concepts: Abstraction Interview Questions

### 107) What is the abstraction?

Abstraction is a process of hiding the implementation details and showing only functionality to the user. It displays just the essential things to the user and hides the internal information, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery. Abstraction enables you to focus on what the object does instead of how it does it. Abstraction lets you focus on what the object does instead of how it does it.

In Java, there are two ways to achieve the abstraction.

* Abstract Class
* Interface

[More details.](https://www.javatpoint.com/abstract-class-in-java)

**Abstract Modifier in Java**

* Abstract is a keyword that can be applied with an outer class, inner class, method, outer interface, and inner interface. It cannot be applied with variable, constructor, block, and enum.
* When a class is declared with a keyword abstract, it is called abstract class in Java.
* An abstract keyword cannot be simultaneously declared with the final keyword.
* An interface in Java is by default abstract and does not need to be declared abstract.
* When a method is declared with an abstract keyword, it is known as abstract method. It contains only a signature and no body. If you declare a method as abstract in a class, the class must be declared as an abstract class.
* Abstract keyword cannot be declared simultaneously with final, private, native, static, or synchronized.

**Abstract Class in Java**

In Java, abstract class is declared with the abstract keyword. It has both abstract and non-abstract methods(methods with bodies). Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.

**Properties of an abstract class**

* An instance of an abstract class can not be created.
* Constructors are allowed.
* We can have an abstract class without any abstract method.
* There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
* We can define static methods in an abstract class
* We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract
* If a class contains at least one abstract method then compulsory should declare a class as abstract
* If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method

**Java Abstract Method**

A method declared using the abstract keyword and does not have a definition (method body) is called an abstract method. When we need just the method declaration in a super class, it can be achieved by declaring the methods as abstracts. A subclass must override it to provide the method definition.

The abstract Method is used for creating blueprints for classes or interfaces. Here methods are defined but these methods don’t provide the implementation. Abstract Methods can only be implemented using subclasses or classes that implement the interfaces.

**Important rules for abstract methods are mentioned below:**

* Any class that contains one or more abstract methods must also be declared abstract.
* If a class contains an abstract method it needs to be abstract and vice versa is not true.
* If a non-abstract class extends an abstract class, then the class must implement all the abstract methods of the abstract class else the concrete class has to be declared as abstract as well.
* The following are various illegal combinations of other modifiers for methods with respect to abstract modifiers:
* final, abstract native, abstract synchronized, abstract static, abstract private, abstract strictfp

**Interface**

An Interface in is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a class.It is a mechanism to achieve abstraction and multiple inheritance in Java.

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

* Java Interface also represents the IS-A relationship.
* It cannot be instantiated just like the abstract class.
* We can’t create an instance (interface can’t be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
* A class can implement more than one interface.
* An interface can extend to another interface or interface (more than one interface).
* A class that implements the interface must implement all the methods in the interface.
* All the methods are public and abstract. And all the fields are public, static, and final.
* It is used to achieve multiple inheritances.
* It is used to achieve loose coupling.
* Inside the Interface not possible to declare instance variables because by default variables are public static final.
* Inside the Interface, constructors are not allowed.
* Inside the interface main method is not allowed.
* Since Java 8, we can have default and static methods in an interface.
* Since Java 9, we can have private methods in an interface.

**Why use Java interface?**

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

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

**Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?**

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestInterface3 implements Printable, Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

### 108) What is the difference between abstraction and encapsulation?

Abstraction hides the implementation details whereas encapsulation wraps code and data into a single unit.

[More details.](https://www.javatpoint.com/abstract-class-in-java)

### 110) Can there be an abstract method without an abstract class?

No, if there is an abstract method in a class, that class must be abstract.

### 111) Is the following program written correctly? If yes then what will be the output of the program?

**abstract** **class** Calculate

{

**abstract** **int** multiply(**int** a, **int** b);

}

**public** **class** Main

{

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

    {

**int** result = **new** Calculate()

        {

            @Override

**int** multiply(**int** a, **int** b)

            {

**return** a\*b;

            }

        }

multiply(12,32);

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

    }

}

Yes, the program is written correctly. The Main class provides the definition of abstract method multiply declared in abstract class Calculation. The output of the program will be:

**Output**

384

### 112) Can you use abstract and final both with a method?

No, because we need to override the abstract method to provide its implementation, whereas we can't override the final method.

### 113) Is it possible to instantiate the abstract class?

No, the abstract class can never be instantiated even if it contains a constructor and all of its methods are implemented.

### 117) What is a marker interface?

A Marker interface can be defined as the interface which has no data member and member functions. Marker interfaces serve as a form of metadata, indicating that the implementing class has some specific property or should be treated in a particular way. Marker interfaces serve as a form of metadata, indicating that the implementing class has some specific property or should be treated in a particular way.

java.io.Serializable: Indicates that a class can be serialized, meaning its instances can be converted into a byte stream for storage or transmission.

java.lang.Cloneable: Indicates that a class supports cloning, meaning its objects can be duplicated using the clone() method.

**public** **interface** Serializable{

}

### 1. java.io.Serializable

* java.lang.String
* java.util.Date
* java.util.ArrayList
* java.util.HashMap
* java.lang.Integer (and other wrapper classes)
* Custom classes that you want to be serializable must also implement this interface.

### 2. java.lang.Cloneable

* java.util.ArrayList (which provides a clone method)
* java.lang.Object's subclass that overrides the clone() method should implement Cloneable if they wish to allow cloning.

### 118) What are the differences between abstract class and interface?

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| An abstract class can have a method body (non-abstract methods). | The interface has only abstract methods. |
| An abstract class can have instance variables. | An interface cannot have instance variables. |
| An abstract class can have the constructor. | The interface cannot have the constructor. |
| An abstract class can have static methods. | The interface cannot have static methods. |
| You can extend one abstract class. | You can implement multiple interfaces. |
| The abstract class **can provide the implementation of the interface**. | The Interface **can't provide the implementation of the abstract class**. |
| The **abstract keyword** is used to declare an abstract class. | The **interface keyword** is used to declare an interface. |
| An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| An **abstract class** can be extended using keyword **extends** | An **interface class** can be implemented using keyword **implements** |
| A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

### 119) Can we define private and protected modifiers for the members in interfaces?

No, they are implicitly public.

### 120) When can an object reference be cast to an interface reference?

An object reference can be cast to an interface reference when the object implements the referenced interface.

### 121) How to make a read-only class in Java?

A class can be made read-only by making all of the fields private. The read-only class will have only getter methods which return the private property of the class to the main method. We cannot modify this property because there is no setter method available in the class. Consider the following example.

//A Java class which has only getter methods.

**public** **class** Student{

//private data member

**private** String college="AKG";

//getter method for college

**public** String getCollege(){

**return** college;

}

}

### 122) How to make a write-only class in Java?

A class can be made write-only by making all of the fields private. The write-only class will have only setter methods which set the value passed from the main method to the private fields. We cannot read the properties of the class because there is no getter method in this class. Consider the following example.

  //A Java class which has only setter methods.

**public** **class** Student{

//private data member

**private** String college;

//getter method for college

**public** **void** setCollege(String college){

**this**.college=college;

}

}

### Encapsulation

### The process of binding data and corresponding methods (behavior) together into a single unit is called encapsulation in Java. In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes. Thereby, we can keep variables and methods safes from outside interference and misuse.

### School bag is one of the most real examples of Encapsulation. School bag can keep our books, pens, etc.

### There are two important points whereby we can achieve or implement encapsulation in Java program.

### Declaring the instance variable of the class as private so that it cannot be accessed directly by anyone from outside the class.

### Provide the public setter and getter methods in the class to set/modify and read/retrieve the values of the variable/fields.

### There are the following advantages of encapsulation in Java. They are as follows:

### The encapsulated code is more flexible and easy to change with new requirements.

### It prevents the other classes from accessing the private fields.

### Encapsulation allows modifying the implemented code without breaking other code that has implemented the code.

### It keeps the data and codes safe from external inheritance. Thus, encapsulation helps to achieve security.

### It improves the maintainability of the application.

### 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.

### Disadvantage of Encapsulation in Java

### The major disadvantage of encapsulation in Java is it increases the length of the code and slows shutdown execution.

### Simple Example of Encapsulation in Java

### File: Student.java

### //A Java class which is a fully encapsulated class.

### //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

### }

### }

### File: Test.java

### //A Java class to test the encapsulated class.

### package com.javatpoint;

### class Test{

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

### //creating instance of the encapsulated class

### Student s=new Student();

### //setting value in the name member

### s.setName("vijay");

### //getting value of the name member

### System.out.println(s.getName());

### }

### }

### Output:

### vijay

## Java: Exception Handling Interview Questions

### 133) Explain the hierarchy of Java Exception classes?

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



### What is Exception in Java?

### Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions.

### Major reasons why an exception Occurs

### Invalid user input

### Device failure

### Loss of network connection

### Physical limitations (out-of-disk memory)

### Code errors

### Opening an unavailable file

### 131) How many types of exception can occur in a Java program?

* **Checked Exception:** Checked exceptions are the one which are checked at compile-time. For example, SQLException, ClassNotFoundException, etc.
* **Unchecked Exception:** Unchecked exceptions are the one which are handled at runtime because they can not be checked at compile-time. For example, ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
* **Error:** Error cause the program to exit since they are not recoverable. For Example, OutOfMemoryError, AssertionError, etc.

### What Is the Difference Between an Exception and Error?

### An exception is an event that represents a condition from which is possible to recover, whereas error represents an external situation usually impossible to recover from.

### All errors thrown by the JVM are instances of Error or one of its subclasses, the more common ones include but are not limited to:

### OutOfMemoryError – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available

### StackOverflowError – occurs when the stack space for a thread has run out, typically because an application recurses too deeply

### ExceptionInInitializerError – signals that an unexpected exception occurred during the evaluation of a static initializer

### NoClassDefFoundError – is thrown when the classloader tries to load the definition of a class and couldn’t find it, usually because the required class files were not found in the classpath

### UnsupportedClassVersionError – occurs when the JVM attempts to read a class file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java

### Although an error can be handled with a try statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown.

### what are checked and unchecked exceptions in java?

### Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions. These exceptions are also called compile time exceptions. Because, these exceptions will be known during compile time itself.

### Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions. All sub classes of java.lang.RunTimeException and java.lang.Error are unchecked exceptions.

### 132) What is Exception Handling?

Exception Handling is a mechanism that is used to handle exceptions It is used primarily to handle checked exceptions. Exception handling maintains the normal flow of the program. There are mainly two types of exceptions: checked and unchecked. Here, the error is considered as the unchecked exception.

[More details.](https://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception)

**How the exceptions are handled in Java? OR Explain exception handling mechanism in Java?**

Exceptions in Java are handled using try, catch and finally blocks.

* try block : The code or set of statements which are to be monitored for exception are kept in this block.
* catch block : This block catches the exceptions occurred in the try block.
* finally block : This block is always executed whether exception is occurred in the try block or not and occurred exception is caught in the catch block or not.

**Can we keep other statements in between try, catch and finally blocks?**

No. We shouldn’t write any other statements in between try, catch and finally blocks.

**Can we write only try block without catch and finally blocks?**

No, it shows compilation error. The try block must be followed by either catch block or finally block.

Note : From Java 7, with the introduction of try-with resources blocks, we can write only try block without catch and finally blocks provided resources must be AutoCloseable.

try (Connection connection = DriverManager.getConnection(url, username, password);

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery("SELECT \* FROM yourtable"))

{

}

135) What is the base class for Error and Exception?

The Throwable class is the base class for Error and Exception.

### 137) What is the output of the following Java program?

**public** **class** ExceptionHandlingExample {

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

{

**try**

    {

**int** a = 1/0;

        System.out.println("a = "+a);

    }

**catch**(Exception e){System.out.println(e);}

**catch**(ArithmeticException ex){System.out.println(ex);}

}

}

**Output**

ExceptionHandlingExample.java:10: error: exception ArithmeticException has already been caught

catch(ArithmeticException ex){System.out.println(ex);}

^

1 error

**Explanation**

ArithmaticException is the subclass of Exception. Therefore, it can not be used after Exception. Since Exception is the base class for all the exceptions, therefore, it must be used at last to handle the exception. No class can be used after this.

### 138) What is finally block?

The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. In other words, we can say that finally block is the block which is always executed. Finally block follows try or catch block. If you don't handle the exception, before terminating the program, JVM runs finally block, (if any). The finally block is mainly used to place the cleanup code such as closing a file or closing a connection. Here, we must know that for each try block there can be zero or more catch blocks, but only one finally block. The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).



### 139) Can finally block be used without a catch?

Yes, According to the definition of finally block, it must be followed by a try or catch block, therefore, we can use try block instead of catch.[More details.](https://www.javatpoint.com/finally-block-in-exception-handling)

### 140) Is there any case when finally will not be executed?

Finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).[More details.](https://www.javatpoint.com/finally-block-in-exception-handling)

### What does JVM do when an exception occurs in a program?

### Ans: When JVM faces an exception in a program, it creates an exception object and throws it to inform us that an error has occurred. If the exception object is not caught and handled properly, JVM will display an error message and will terminate the rest of the program abnormally.

### What is throwing an exception in Java?

### Ans: When an exception occurs inside a method in java program, the method in which exception has occurred creates an exception object (i.e, an object of exception class) internally with the help of JVM and hands it over to the java runtime system (JVM). This process is called throwing an exception in java.

### What is catching an exception in Java?

### Ans: The process of finding a handler by JVM to handle thrown exception is called catching an exception.

### What are the keywords to handle an exception in Java?

### Ans: Java provides five essential keywords to handle an exception. They are:

### try

### catch

### finally

### throw

### throws

### What happens when an exception is thrown by the main method?

### Ans: When an exception is thrown by the main() method, JVM terminates the program and prints the exception message and stack trace in system console.

### What is try block in Java?

### Ans: A try is a block of code or statements that might throw an exception. That’s why a try block is also known as exception generated block. An exception generated code (risky code) must be placed within a try block.

### What is catch block in Java?

### Ans: A catch is a block of code that handles the exception thrown by the try block. That’s why it is also known as exception handler block.

### Java Nested try block

### In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

### For example, the inner try block can be used to handle ArrayIndexOutOfBoundsException while the outer try block can handle the ArithemeticException (division by zero)

### Why use nested try block

### Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### //main try block

### try

### {

### statement 1;

### statement 2;

### //try catch block within another try block

### try

### {

### statement 3;

### statement 4;

### //try catch block within nested try block

### try

### {

### statement 5;

### statement 6;

### }

### catch(Exception e2)

### {

### //exception message

### }

### 

### }

### catch(Exception e1)

### {

### //exception message

### }

### }

### //catch block of parent (outer) try block

### catch(Exception e3)

### {

### //exception message

### }

### Java Nested try Example

### public class NestedTryBlock{

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

### //outer try block

### try{

### //inner try block 1

### try{

### System.out.println("going to divide by 0");

### int b =39/0;

### }

### //catch block of inner try block 1

### catch(ArithmeticException e)

### {

### System.out.println(e);

### }

### 

### 

### //inner try block 2

### try{

### int a[]=new int[5];

### 

### //assigning the value out of array bounds

### a[5]=4;

### }

### 

### //catch block of inner try block 2

### catch(ArrayIndexOutOfBoundsException e)

### {

### System.out.println(e);

### }

### 

### 

### System.out.println("other statement");

### }

### //catch block of outer try block

### catch(Exception e)

### {

### System.out.println("handled the exception (outer catch)");

### }

### 

### System.out.println("normal flow..");

### }

### }

### When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

### If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

### Java throw keyword

### 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.

### Custom Exceptions

### Custom exceptions are those exceptions that are created by users or programmers according to their own needs. The custom exceptions are also called user-defined exceptions that are created by extending the exception class.

### import java.util.Scanner;

### class IllegalAgeException extends Exception{

### public IllegalAgeException(String message) {

### super(message);

### }

### }

### public class Main {

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

### Scanner scanner = new Scanner(System.in);

### System.out.print("Enter your age: ");

### int age = scanner.nextInt();

### try {

### if (age < 18) {

### throw new IllegalAgeException("Age is less than 18. Access denied.");

### } else {

### System.out.println("Age is 18 or above. Access granted.");

### }

### } catch (IllegalAgeException e) {

### e.printStackTrace();

### }

### scanner.close();

### }

### }

### 141) What is the difference between throw and throws?

|  |  |
| --- | --- |
| **throw keyword** | **throws keyword** |
| 1) The **throw** keyword is used to throw an exception explicitly. | The **throws** keyword is used to declare an exception. |
| 2) The checked exceptions cannot be propagated with throw only. | The checked exception can be propagated with throws |
| 3) The **throw** keyword is followed by an instance. | The **throws** keyword is followed by class. |
| 4) The **throw** keyword is used within the method. | The **throws** keyword is used with the method signature. |
| 5) You cannot throw multiple exceptions. | You can declare multiple exceptions, e.g., public void method()throws IOException, SQLException. |

[More details.](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws)

### 142) What is the output of the following Java program?

**public** **class** Main{

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

**try**

        {

**throw** 90;

        }

**catch**(**int** e){

            System.out.println("Caught the exception "+e);

        }

    }

}

**Output**

Main.java:6: error: incompatible types: int cannot be converted to Throwable

throw 90;

^

Main.java:8: error: unexpected type

catch(int e){

^

required: class

found: int

2 errors

**Explanation**

In Java, the throwable objects can only be thrown. If we try to throw an integer object, The compiler will show an error since we can not throw basic data type from a block of code.

### 143) What is the output of the following Java program?

**class** Calculation **extends** Exception

{

**public** Calculation()

    {

        System.out.println("Calculation class is instantiated");

    }

**public** **void** add(**int** a, **int** b)

    {

        System.out.println("The sum is "+(a+b));

    }

}

**public** **class** Main{

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

**try**

        {

**throw** **new** Calculation();

        }

**catch**(Calculation c){

            c.add(10,20);

        }

    }

}

**Output**

Calculation class is instantiated

The sum is 30

**Explanation**

The object of Calculation is thrown from the try block which is caught in the catch block. The add() of Calculation class is called with the integer values 10 and 20 by using the object of this class. Therefore there sum 30 is printed. The object of the Main class can only be thrown in the case when the type of the object is throwable. To do so, we need to extend the throwable class.

### 144) Can an exception be rethrown?

Yes.

### 145) Can subclass overriding method declare an exception if parent class method doesn't throw an exception?

Yes but only unchecked exception not checked.

[More details.](https://www.javatpoint.com/exception-handling-with-method-overriding)

### 146) What is exception propagation?

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

**class** TestExceptionPropagation1{

**void** m(){

**int** data=50/0;

  }

**void** n(){

    m();

  }

**void** p(){

**try**{

    n();

   }**catch**(Exception e){System.out.println("exception handled");}

  }

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

   TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();

   obj.p();

   System.out.println("normal flow...");

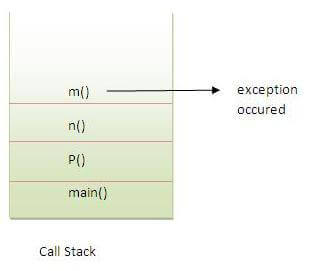
  }

}

**Output:**

exception handled

normal flow...

  
[More details.](https://www.javatpoint.com/exception-propagation)

### 147) What is the output of the following Java program?

**public** **class** Main

{

**void** a()

    {

**try**{

        System.out.println("a(): Main called");

        b();

        }**catch**(Exception e)

        {

            System.out.println("Exception is caught");

        }

    }

**void** b() **throws** Exception

    {

**try**{

         System.out.println("b(): Main called");

         c();

     }**catch**(Exception e){

**throw** **new** Exception();

     }

**finally**

     {

         System.out.println("finally block is called");

     }

    }

**void** c() **throws** Exception

    {

**throw** **new** Exception();

    }

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

    {

        Main m = **new** Main();

        m.a();

    }

}

**Output**

a(): Main called

b(): Main called

finally block is called

Exception is caught

**Explanation**

In the main method, a() of Main is called which prints a message and call b(). The method b() prints some message and then call c(). The method c() throws an exception which is handled by the catch block of method b. However, It propagates this exception by using **throw Exception()** to be handled by the method a(). As we know, finally block is always executed therefore the finally block in the method b() is executed first and prints a message. At last, the exception is handled by the catch block of the method a().

### 148) What is the output of the following Java program?

**public** **class** Calculation

{

**int** a;

**public** Calculation(**int** a)

    {

**this**.a = a;

    }

**public** **int** add()

    {

        a = a+10;

**try**

        {

            a = a+10;

**try**

            {

                a = a\*10;

**throw** **new** Exception();

            }**catch**(Exception e){

                a = a - 10;

            }

        }**catch**(Exception e)

        {

            a = a - 10;

        }

**return** a;

    }

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

    {

        Calculation c = **new** Calculation(10);

**int** result = c.add();

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

    }

}

**Output**

result = 290

**Explanation**

The instance variable a of class Calculation is initialized to 10 using the class constructor which is called while instantiating the class. The add method is called which returns an integer value result. In add() method, a is incremented by 10 to be 20. Then, in the first try block, 10 is again incremented by 10 to be 30. In the second try block, a is multiplied by 10 to be 300. The second try block throws the exception which is caught by the catch block associated with this try block. The catch block again alters the value of a by decrementing it by 10 to make it 290. Thus the add() method returns 290 which is assigned to result. However, the catch block associated with the outermost try block will never be executed since there is no exception which can be handled by this catch block.

## Java: String Handling Interview Questions

## How to Create String Object in Java?

To handle string data in Java, we need an object of string class. Basically, there are three ways to create a string object in Java.

1. By string literal.
2. By new keyword.
3. By converting character arrays into strings

Let’s understand all three ways one by one.

## String Literal in Java

String literal in Java is created by using double quotes. For example:

String s = "Hello";

The string literal is always created in the string constant pool. In Java, **String constant pool** is a special area that is used for storing string objects.

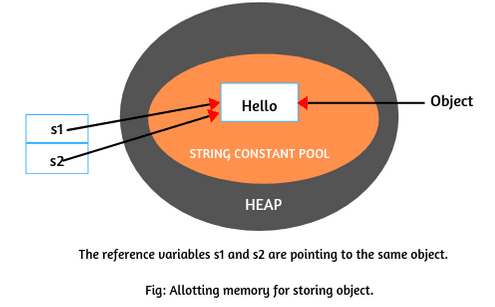
SCP is a part of heap memory. Whenever we create a string literal in Java, JVM checks string constant pool first. If the string already exists in string constant pool, no new string object will be created in the string pool by JVM.

JVM uses the same string object by pointing a reference to it to save memory. But if string does not exist in the string pool, JVM creates a new string object and placed it in the pool. For example:

String s1 = "Hello";

String s2 = "Hello";

Look at the below figure where we have represented string objects with contents in the memory.

[](https://www.scientecheasy.com/2020/05/string-in-java.html/)

In the above example, when JVM will execute the first statement, it will not find any string with value “Hello” in the string constant pool. So, it will create an object in string pool and store string “Hello” in that object as shown in the above figure. This object is referenced by the reference variable s1. In other words, if literal is not available in the pool, a new object will be created in the pool and the address of that object will be assigned to a reference variable s1.

When [JVM](https://www.scientecheasy.com/2021/03/what-is-jvm.html/) will execute the second statement, it will first check to string constant pool to know whether string object with the same content is already available there or not. Since string with value “Hello” is already available in the string pool.

So, JVM will not create a new string object in the pool, and the address of the available object will assign to reference variable s2. That is, it will point a reference variable s2 to the old existing object as shown in the above figure.

**Key points:**

1. Remember that creating an object means allocating memory for storing data.  
2. A string object is created for every string literal.

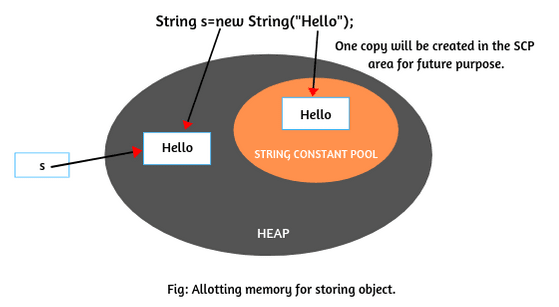
## Creating String Object by new Keyword

The second way of creating an object to string class is by using new operator. It is just like creating an object of any class. It can be declared as follows:

String s = new String("Hello");

Whenever we will create an object of string class using the new operator, JVM will create two objects. First, it will create an object in the heap area and store string “Hello” into the object.

After storing data into memory, JVM will point a reference variable s to that object in the heap. Allocating memory for string objects is shown in the below figure.

[](https://www.scientecheasy.com/2020/05/string-in-java.html/)

Now JVM will create the second object as a copy for literal “Hello” in string constant pool for future purposes. There is no explicit reference variable pointing to the copy object in the pool but internally, JVM maintains the reference variable implicitly.

Remember that the object created in the SCP area is not eligible for garbage collection because implicitly, the reference variable will be maintained by JVM itself.

**Key point:**For every string literal, one copy will be created in the SCP area.

## By Converting Character Arrays into String

The third way to create strings is by converting the character arrays into string. Let’s take a character type array: arr[ ] with some characters as given below:

char arr[ ] = {'j','a','v','a'};

Now create a string object by passing array name to string constructor like this:

String s = new String(arr);

Now string object ‘s’ contains string “java”. It means that all the characters of the array are copied into string.

If you do not want all the characters of the array into string then you can also mention which character you need, like this:

String s = new String(arr, 1,3);

From the above statement, total of three characters will be copied into string s. Since the original characters are j-a-v-a. So, the counting will start from 0 i.e. 0th character in the array is ‘j’ and the first character is ‘a’. Starting from ‘a’, total of three characters ‘aja’ will copy into string s.

## How many total objects will be created in memory for following string objects?

String s1 = new String("Scientech");

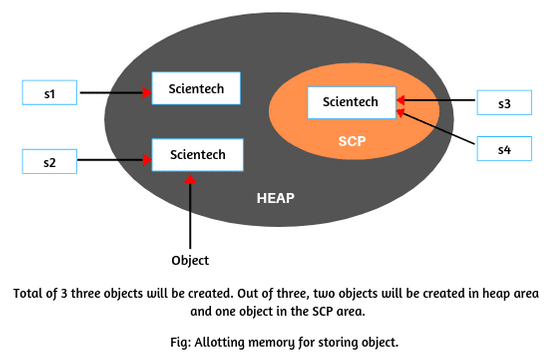
String s2 = new String("Scientech");

String s3 = "Scientech";

String s4 = "Scientech";

In the preceding code, we are creating three string objects using two new operators and 2 string literal.

**How many objects will be created in the heap and SCP area?**

[](https://www.scientecheasy.com/2020/05/string-in-java.html/)

1. During the execution of first statement using new operator, JVM will create two objects, one with content in heap area and another as a copy for literal “Scientech” in the SCP area for future purposes as shown in the figure.

The reference variable s1 is pointing to the first object in the heap area.

2. When the second statement will be executed, for every new operation, JVM will create again one new object with content “Scientech” in the heap area.

But in the SCP area, no new object for literal will be created by JVM because it is already available in the SCP area. The s2 is pointing to the object in the heap area as shown in the figure.

3. During the execution of third and fourth statements, JVM will not create a new object with content “Scientech” in the SCP area because it is already available in string constant pool.

It simply points the reference variables s3 and s4 to the same object in the SCP. They are shown in the above figure.

Thus, three objects are created, two in the heap area and one in string constant pool.

### Key Points of String:

1. Strings are objects in Java`.

2. When we create a String object, we are creating a string that cannot be changed. In other words, once a String object has been created, we cannot change any characters in string. Therefore, string object is immutable in Java.

3. The number of characters in a string is called length of string. For example, the length of “Hello” is 5.

4. String class in Java has numerous methods for string manipulation.

5. String is not a primitive data type. It is a reference data type.

### 149) What is String Pool?

String pool is the space reserved in the heap memory that can be used to store the strings. The main advantage of using the String pool is whenever we create a string literal; the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. Therefore, it saves the memory by avoiding the duplicacy.



### 150) What is the meaning of immutable regarding String?

The simple meaning of immutable is unmodifiable or unchangeable. In Java, String is immutable, i.e., once string object has been created, its value can't be changed. Consider the following example for better understanding.

**class** Testimmutablestring{

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

   String s="Sachin";

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

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

 }

}

**Output:**

Sachin

[More details.](https://www.javatpoint.com/immutable-string)

### 151) Why are the string objects immutable in java?

Because Java uses the concept of the string literal. Suppose there are five reference variables, all refer to one object "sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why string objects are immutable in java.



### 153) How many objects will be created in the following code?

String s1="Welcome";

String s2="Welcome";

String s3="Welcome";

Only one object will be created using the above code because strings in Java are immutable.

[More details.](https://www.javatpoint.com/string-handling-in-java)

### 154) Why java uses the concept of the string literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

[More details.](https://www.javatpoint.com/string-handling-in-java)

### 155) How many objects will be created in the following code?

1. String s = **new** String("Welcome");

Two objects, one in string constant pool and other in non-pool(heap).

[More details.](https://www.javatpoint.com/string-handling-in-java)

### 156) What is the output of the following Java program?

**public** **class** Test

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

  {

      String a = **new** String("Sharma is a good player");

      String b = "Sharma is a good player";

**if**(a == b)

      {

          System.out.println("a == b");

      }

**if**(a.equals(b))

      {

          System.out.println("a equals b");

      }

  }

**Output**

a equals b

**Explanation**

The operator **==** also check whether the references of the two string objects are equal or not. Although both of the strings contain the same content, their references are not equal because both are created by different ways(Constructor and String literal) therefore, **a == b** is unequal. On the other hand, the equal() method always check for the content. Since their content is equal hence, **a equals b** is printed.

### Java StringBuffer Class

### Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed. Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

### It provides much of the functionality & flexibility than String. As we know that a string is fixed-length, immutable, and cannot be modified. To overcome this, Java language introduced another class called StringBuffer.

### We create a StringBuffer class object, we can perform any required changes in the object. i.e, its data can be modified. That’s why StringBuffer class objects are mutable in Java.

### StringBuffer class defines four constructors in Java. They are as follows:

### StringBuffer(): This constructor creates an empty string buffer object with an initial capacity of 16 characters.

### StringBuffer(int size): This constructor creates an empty string buffer object and accepts an integer value that specifies the size or length of the string object.

### StringBuffer(String str): This constructor creates a string buffer object with the specified string whose size is equal to the string specified in this constructor.

### StringBuffer(CharSequence seq): This form of constructor creates a string buffer that contains the same characters as the specified CharSequence.

### A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

### class StringBufferExample{

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

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

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

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

### }

### }

### Output:

### Hello Java

### Java StringBuilder Class

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

### StringBuffer is thread-safe and StringBuilder is not thread-safe. Methods provided by StringBuffer for modifying the buffer are synchronized, which means that only one thread is allowed to access StringBuffer object at a time. Most of the time, we do not require thread safety. In such as case, if we use StringBuffer, the performance of the application will be down. Performance wise, string buffer is not recommended to use.

### This is the reason that StringBuilder class was added later in J2SE 5. The main advantage of using StringBuilder is faster performance because multiple threads have not to wait to access the same string builder object at a time. That’s why, performance is relatively high.

### Note: In StringBuffer and StringBuilder, almost everything is the same, including constructors and methods.

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| **Introduction** | Introduced in JDK 1.0 | Introduced in JDK 1.5 | Introduced in JDK 1.0 |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread Safety** | Thread Safe | Not Thread Safe | Thread Safe |
| **Memory Efficiency** | High | Efficient | Less Efficient |
| **Performance** | High(No-Synchronization) | High(No-Synchronization) | Low(Due to Synchronization) |
| **Usage** | This is used when we want immutability. | This is used when Thread safety is not required. | This is used when Thread safety is required. |

### 160) How can we create an immutable class in Java?

We can create an immutable class by defining a final class having all of its members as final. Consider the following example.

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

**final** String pancardNumber;

**public** Employee(String pancardNumber){

**this**.pancardNumber=pancardNumber;

}

**public** String getPancardNumber(){

**return** pancardNumber;

}

}

[More details.](https://www.javatpoint.com/how-to-create-immutable-class)

### 161) What is the purpose of toString() method in Java?

The toString() method returns the string representation of an object. If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object, etc. depending upon your implementation. By overriding the toString() method of the Object class, we can return the values of the object, so we don't need to write much code. Consider the following example.

**class** Student{

**int** rollno;

 String name;

 String city;

 Student(**int** rollno, String name, String city){

**this**.rollno=rollno;

**this**.name=name;

**this**.city=city;

 }

**public** String toString(){//overriding the toString() method

**return** rollno+" "+name+" "+city;

 }

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

   Student s1=**new** Student(101,"Raj","lucknow");

   Student s2=**new** Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()

   System.out.println(s2);//compiler writes here s2.toString()

 }

}

**Output:**

101 Raj lucknow

102 Vijay ghaziabad

[More details.](https://www.javatpoint.com/understanding-toString()-method)

### 162) Why CharArray() is preferred over String to store the password?

String stays in the string pool until the garbage is collected. If we store the password into a string, it stays in the memory for a longer period, and anyone having the memory-dump can extract the password as clear text. On the other hand, Using CharArray allows us to set it to blank whenever we are done with the password. It avoids the security threat with the string by enabling us to control the memory.

### 163) write a Java program to count the number of words present in a string?

**Program:**

**public** **class** Test

{

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

    {

        String s = "Sharma is a good player and he is so punctual";

String words[] = s.split(" ");

System.out.println("The Number of words present in the string are : "+words.length);

}

}

**Output**

The Number of words present in the string are : 10

## Java: Multithreading Interview Questions

### Multitasking

### Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

### Process-based Multitasking (Multiprocessing)

### Thread-based Multitasking (Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

### Each process has an address in memory. In other words, each process allocates a separate memory area.

### A process is heavyweight.

### Cost of communication between the process is high.

### Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

### Threads share the same address space.

### A thread is lightweight.

### Cost of communication between the thread is low.

### 1) What is multithreading?

Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking. It consumes less memory and gives the fast and efficient performance. Its main advantages are:

* Threads share the same address space.
* The thread is lightweight.
* The cost of communication between the processes is low.

[More details.](https://www.javatpoint.com/multithreading)

### 2) What is the thread?

A thread is the smallest unit of execution within a process. A thread has its own call stack, program counter, and local variables, but it shares resources like memory and file handles with other threads within the same process. In Java, threads can be created by implementing the Runnable interface or extending the Thread class.

[More details.](https://www.javatpoint.com/multithreading)

### Thread creation by extending the Thread class

### We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

### // Java code for thread creation by extending

### // the Thread class

### class MultithreadingDemo extends Thread {

### public void run()

### {

### try {

### // Displaying the thread that is running

### System.out.println(

### "Thread " + Thread.currentThread().getId()

### + " is running");

### }

### catch (Exception e) {

### // Throwing an exception

### System.out.println("Exception is caught");

### }

### }

### }

### 

### // Main Class

### public class Multithread {

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

### {

### int n = 8; // Number of threads

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

### MultithreadingDemo object

### = new MultithreadingDemo();

### object.start();

### }

### }

### }

### Output

### Thread 15 is running

### Thread 14 is running

### Thread 16 is running

### Thread 12 is running

### Thread 11 is running

### Thread 13 is running

### Thread 18 is running

### Thread 17 is running

### Thread creation by implementing the Runnable Interface

### We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

### // Java code for thread creation by implementing

### // the Runnable Interface

### class MultithreadingDemo implements Runnable {

### public void run()

### {

### try {

### // Displaying the thread that is running

### System.out.println(

### "Thread " + Thread.currentThread().getId()

### + " is running");

### }

### catch (Exception e) {

### // Throwing an exception

### System.out.println("Exception is caught");

### }

### }

### // Main Class

### class Multithread {

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

### {

### int n = 8; // Number of threads

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

### Thread object

### = new Thread(new MultithreadingDemo());

### object.start();

### }

### }

### }

### Output

### Thread 13 is running

### Thread 11 is running

### Thread 12 is running

### Thread 15 is running

### Thread 14 is running

### Thread 18 is running

### Thread 17 is running

### Thread 16 is running

### Thread Class vs Runnable Interface

### If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

### We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

### Using runnable will give you an object that can be shared amongst multiple threads.

### 3) Differentiate between process and thread?

| **Feature** | **Process** | **Thread** |
| --- | --- | --- |
| Definition | An independent program running in its own memory space. | A thread is a subset of the process |
| Memory | Each process has its own separate memory space. | Threads within the same process share the same memory space. |
|  | Lightweight | Heavyweigth |
| Communication | Inter-process communication is slower and expensive | Inter-thread communication is fast and inexpensive |
| Context Switching | More overhead due to separate memory space and resources. | Less overhead since threads share the same memory space. |
| Creation | More resource-intensive and slower to create. | Less resource-intensive and faster to create. |
| Isolation | Processes are isolated from each other; a crash in one process doesn’t affect others. | Threads are not isolated; a crash in one thread can affect the entire process. |
| Concurrency | Process-based multitasking involves running multiple processes concurrently. | Multithreading involves running multiple threads concurrently within a single process. |
| Example | Running a web server and a database server simultaneously. | A web server handling multiple client requests using separate threads. |



## Java Thread Methods

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |

### 4) What do you understand by inter-thread communication?

* The process of communication between synchronized threads is termed as inter-thread communication.
* Inter-thread communication is used to avoid thread polling in Java.
* The thread is paused running in its critical section, and another thread is allowed to enter (or lock) in the same critical section to be executed.
* It can be obtained by wait(), notify(), and notifyAll() methods.

### 7) What are the advantages of multithreading?

* Multithreading allows an application/program to be always reactive for input, even already running with some background tasks.
* Multithreading allows the faster execution of tasks, as threads execute independently.
* Multithreading provides better utilization of cache memory as threads share the common memory resources.
* Multithreading reduces the number of the required server as one server can execute multiple threads at a time.

### 8) What are the states in the lifecycle of a Thread?

A thread can have one of the following states during its lifetime:

1. **New:** In this state, a Thread class object is created using a new operator, but the thread is not alive. Thread doesn't start until we call the start() method.
2. **Runnable:** In this state, the thread is ready to run after calling the start() method. However, the thread is not yet selected by the thread scheduler.
3. **Running:** In this state, the thread scheduler picks the thread from the ready state, and the thread is running.
4. **Waiting/Blocked:** In this state, a thread is not running but still alive, or it is waiting for the other thread to finish.
5. **Dead/Terminated:** A thread is in terminated or dead state when the run() method exits.



### 10) What is context switching?

In Context switching the state of the process (or thread) is stored so that it can be restored and execution can be resumed from the same point later. Context switching enables the multiple processes to share the same CPU.

### 12) What does join() method?

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task. Join method is overloaded in Thread class in the following ways.

* public void join()throws InterruptedException
* public void join(long milliseconds)throws InterruptedException

[More details.](https://www.javatpoint.com/join()-method)

### 13) Describe the purpose and working of sleep() method.

The sleep() method in java is used to block a thread for a particular time, which means it pause the execution of a thread for a specific time. There are two methods of doing so.

**Syntax:**

* public static void sleep(long milliseconds)throws InterruptedException
* public static void sleep(long milliseconds, int nanos)throws InterruptedException

**Working of sleep() method**

When we call the sleep() method, it pauses the execution of the current thread for the given time and gives priority to another thread(if available). Moreover, when the waiting time completed then again previous thread changes its state from waiting to runnable and comes in running state, and the whole process works so on till the execution doesn't complete.

### 15) Is it possible to start a thread twice?

No, we cannot restart the thread, as once a thread started and executed, it goes to the Dead state. Therefore, if we try to start a thread twice, it will give a runtimeException "java.lang.IllegalThreadStateException". Consider the following example.

**public** **class** Multithread1 **extends** Thread

{

**public** **void** run()

    {

**try** {

          System.out.println("thread is executing now........");

      } **catch**(Exception e) {

      }

    }

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

        Multithread1 m1= **new** Multithread1();

        m1.start();

        m1.start();

    }

}

**Output**

thread is executing now........

Exception in thread "main" java.lang.IllegalThreadStateException

at java.lang.Thread.start(Thread.java:708)

at Multithread1.main(Multithread1.java:13)

[More details.](https://www.javatpoint.com/can-we-start-a-thread-twice)

### 16) Can we call the run() method instead of start()?

Yes, calling run() method directly is valid, but it will not work as a thread instead it will work as a normal object. There will not be context-switching between the threads. When we call the start() method, it internally calls the run() method, which creates a new stack for a thread while directly calling the run() will not create a new stack.

### Daemon Thread in Java

### Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

### There are many java daemon threads running automatically e.g. gc, finalizer etc.

### Points to remember for Daemon Thread in Java

### It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

### Its life depends on user threads.

### It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

### The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or  user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

### public class TestDaemonThread1 extends Thread{

### public void run(){

### if(Thread.currentThread().isDaemon()){//checking for daemon thread

### System.out.println("daemon thread work");

### }

### else{

### System.out.println("user thread work");

### }

### }

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

### TestDaemonThread1 t1=new TestDaemonThread1();//creating thread

### TestDaemonThread1 t2=new TestDaemonThread1();

### TestDaemonThread1 t3=new TestDaemonThread1();

### 

### t1.setDaemon(true);//now t1 is daemon thread

### 

### t1.start();//starting threads

### t2.start();

### t3.start();

### }

### }

### Output:

### daemon thread work

### user thread work

### user thread work

### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

### class TestDaemonThread2 extends Thread{

### public void run(){

### System.out.println("Name: "+Thread.currentThread().getName());

### System.out.println("Daemon: "+Thread.currentThread().isDaemon());

### }

### 

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

### TestDaemonThread2 t1=new TestDaemonThread2();

### TestDaemonThread2 t2=new TestDaemonThread2();

### t1.start();

### t1.setDaemon(true);//will throw exception here

### t2.start();

### }

### }

### Output:

### exception in thread main: java.lang.IllegalThreadStateException

### 18)Can we make the user thread as daemon thread if the thread is started?

No, if you do so, it will throw IllegalThreadStateException. Therefore, we can only create a daemon thread before starting the thread.

**class** Testdaemon1 **extends** Thread{

**public** **void** run(){

          System.out.println("Running thread is daemon...");

}

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

      Testdaemon1 td= **new** Testdaemon1();

      td.start();

      setDaemon(**true**);// It will throw the exception: td.

   }

}

**Output**

Running thread is daemon...

Exception in thread "main" java.lang.IllegalThreadStateException

at java.lang.Thread.setDaemon(Thread.java:1359)

at Testdaemon1.main(Testdaemon1.java:8)

[More details.](https://www.javatpoint.com/daemon-thread)

### 19)What is shutdown hook?

The shutdown hook is a thread that is invoked implicitly before JVM shuts down. So we can use it to perform clean up the resource or save the state when JVM shuts down normally or abruptly. We can add shutdown hook by using the following method:

**public** **void** addShutdownHook(Thread hook){}

Runtime r=Runtime.getRuntime();

r.addShutdownHook(**new** MyThread());

Some important points about shutdown hooks are :

* Shutdown hooks initialized but can only be started when JVM shutdown occurred.
* Shutdown hooks are more reliable than the finalizer() because there are very fewer chances that shutdown hooks not run.
* The shutdown hook can be stopped by calling the halt(int) method of Runtime class.

[More details.](https://www.javatpoint.com/ShutdownHook-thread)

### 20)When should we interrupt a thread?

We should interrupt a thread when we want to break out the sleep or wait state of a thread. We can interrupt a thread by calling the interrupt() throwing the InterruptedException.

[More details.](https://www.javatpoint.com/interrupting-a-thread)

### 21) What is the synchronization?

Synchronization is the capability to control the access of multiple threads to any shared resource. It is used:

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

When the multiple threads try to do the same task, there is a possibility of an erroneous result, hence to remove this issue, Java uses the process of synchronization which allows only one thread to be executed at a time. Synchronization can be achieved in three ways:

* by the synchronized method
* by synchronized block
* by static synchronization

Syntax for synchronized block

**synchronized**(object reference expression)

    {

        //code block

    }

[More details.](https://www.javatpoint.com/synchronization)

### 22) What is the purpose of the Synchronized block?

The Synchronized block can be used to perform synchronization on any specific resource of the method. Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.

* Synchronized block is used to lock an object for any shared resource.
* The scope of the synchronized block is limited to the block on which, it is applied. Its scope is smaller than a method.

[More details.](https://www.javatpoint.com/synchronized-block-example)

### 23)Can Java object be locked down for exclusive use by a given thread?

Yes. You can lock an object by putting it in a "synchronized" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it.

### 24) What is static synchronization?

If you make any static method as synchronized, the lock will be on the class not on the object. If we use the synchronized keyword before a method so it will lock the object (one thread can access an object at a time) but if we use static synchronized so it will lock a class (one thread can access a class at a time). [More details.](https://www.javatpoint.com/static-synchronization-example)

### 25)What is the difference between notify() and notifyAll()?

The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

### 26)What is the deadlock?

Deadlock is a situation in which every thread is waiting for a resource which is held by some other waiting thread. In this situation, Neither of the thread executes nor it gets the chance to be executed. Instead, there exists a universal waiting state among all the threads. Deadlock is a very complicated situation which can break our code at runtime.

[More details.](https://www.javatpoint.com/deadlock-in-java)

### 27) How to detect a deadlock condition? How can it be avoided?

We can detect the deadlock condition by running the code on cmd and collecting the Thread Dump, and if any deadlock is present in the code, then a message will appear on cmd.

**Ways to avoid the deadlock condition in Java:**

* **Avoid Nested lock:** Nested lock is the common reason for deadlock as deadlock occurs when we provide locks to various threads so we should give one lock to only one thread at some particular time.
* **Avoid unnecessary locks:** we must avoid the locks which are not required.
* **Using thread join:** Thread join helps to wait for a thread until another thread doesn't finish its execution so we can avoid deadlock by maximum use of join method.

### 28) What is Thread Scheduler in java?

In Java, when we create the threads, they are supervised with the help of a Thread Scheduler, which is the part of JVM. Thread scheduler is only responsible for deciding which thread should be executed. Thread scheduler uses two mechanisms for scheduling the threads: Preemptive and Time Slicing.

Java thread scheduler also works for deciding the following for a thread:

* It selects the priority of the thread.
* It determines the waiting time for a thread
* It checks the Nature of thread

### 29) Does each thread have its stack in multithreaded programming?

Yes, in multithreaded programming every thread maintains its own or separate stack area in memory due to which every thread is independent of each other.

### 30) How is the safety of a thread achieved?

If a method or class object can be used by multiple threads at a time without any race condition, then the class is thread-safe. Thread safety is used to make a program safe to use in multithreaded programming. It can be achieved by the following ways:

* Synchronization
* Using Volatile keyword
* Using a lock based mechanism
* Use of atomic wrapper classes

### 31) What is race-condition?

A Race condition is a problem which occurs in the multithreaded programming when various threads execute simultaneously accessing a shared resource at the same time. The proper use of synchronization can avoid the Race condition.

### 32) What is the volatile keyword in java?

The volatile keyword in Java is used to indicate that a variable's value may be modified by different threads. It ensures that changes to the variable are always visible to all threads, providing a lightweight synchronization mechanism. Here’s how it works and when it’s used

### 33) What do you understand by thread pool?

A thread pool in Java is a collection of reusable threads that can be used to execute multiple tasks concurrently. Instead of creating a new thread for each task, a thread pool reuses existing threads, which helps in managing resources more efficiently and improving performance, especially in applications that require frequent thread creation and destruction.

* Java Thread pool represents a group of worker threads, which are waiting for the task to be allocated.
* Threads in the thread pool are supervised by the service provider which pulls one thread from the pool and assign a job to it.
* After completion of the given task, thread again came to the thread pool.
* The size of the thread pool depends on the total number of threads kept at reserve for execution.

The advantages of the thread pool are :

* Using a thread pool, performance can be enhanced.
* Using a thread pool, better system stability can occur.

**Why JAVA is no 100% OOP language?**

Java is not considered 100% Object-Oriented because it includes primitive data types like int, char, and boolean, which are not objects. It helps in optimizing performance but deviates from the pure OOP principle where everything should be represented as objects. Additionally, Java supports static methods and variables, which can be accessed without using objects, which also deviates from purely object-oriented paradigm.

**Wrapper Class**

A class that wraps a [primitive data type](https://www.scientecheasy.com/2020/06/data-types-in-java.html/) into an object is called **wrapper class in Java**. In simple words, wrapper class provides a mechanism to convert primitive data type value into an object and vice-versa. For example, wrapping int into Integer class, wrapping double into Double class, and wrapping char into Character class.

## Code for Wrapping and Unwrapping an int value

Wrapping an int value:

int x = 25;

Integer a = new Integer(x); // Pass the primitive type to the wrapper constructor.

Unwrapping an int value:

int b = a.intValue();

Two concepts are there in the wrapper classes namely autoboxing and unboxing.

Autoboxing is a procedure of converting a primitive value into an object of the corresponding [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/). For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
* Assigned to a variable of the corresponding **wrapper class**.

Unboxingis a procedure of converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

* Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
* Assigned to a variable of the corresponding **primitive type**

**Now let us land on discussing the useful features of wrapper classes, they are listed as follows:**

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in [*java.util package*](https://www.geeksforgeeks.org/java-util-package-java/) handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

One of the major important features provided by wrapper classes is a lot of utility methods. Say when we have a float value, and we want to find the integer value of that float, then we have a specific method for that which is depicted from the illustration given below.

**Illustration:**

If we want to create an integer value from a string or a boolean value from a string. We can do it with the help of wrapper classes.

**Syntax:** Creation from other data types

Integer hundred = Integer.valueOf("100");

Boolean value = Boolean.valueOf("True");

**Simple Autoboxing of wrapper class example**

|  |
| --- |
| import java .util.\*;  class gfg {        public static void main(String args[]) {          int x = 5;          float y = 3.14f;          long z=6000;            // autoboxing          Integer intobj = x;          Float floatobj=y;          Long longobj=z;          // print          System.out.println(intobj);          System.out.println(floatobj);          System.out.println(longobj);      }  } |

**Output**

5

3.14

6000

**Example:**

|  |
| --- |
| // Java Program to Show Wrapper class concept    // Importing input output classes  import java.io.\*;    // Main Class  class GFG {        // Main driver method      public static void main(String[] args)      {            // In java, in case of floating values          // they are stored as x = (y)f            // Conversion of float value to int          Float floatWrap = Float.valueOf(45.158f);            // Invoking the intValue() method over the stored          // float value to store          int floatToInt = floatWrap.intValue();            // Print the non-primitive(Integer) value          System.out.println(floatToInt);            // Now for another number N          // Say N = 5            // Convert the binary number to the integer value          Integer five = Integer.valueOf("101", 2);            // Print the number          System.out.println(five);      }  } |

**Output**

45

5

In java, we have 7 non-access modifiers. They are used with classes, methods, variables, constructors, etc to provide information about their behavior to JVM. They are as follows:

[static](https://www.geeksforgeeks.org/static-keyword-java/)

[final](https://www.geeksforgeeks.org/final-keyword-java/)

abstract

[synchronized](https://www.geeksforgeeks.org/synchronized-in-java/)

[transient](https://www.geeksforgeeks.org/transient-keyword-java/)

[volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/)

[native](https://www.geeksforgeeks.org/native-keyword-java/)

[strictfp](https://www.geeksforgeeks.org/strictfp-keyword-java/)

**What is serialization?**

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

**JAVA Inner Class?**

A class within another class is called a nested class or an **inner class**. In other words, the inner class is a member of a class just as a class has a member as variables and methods; it can also have another class as its member.

Such class, which has other classes as its members is called a top-level class or outer class. A top-level class may contain any number of inner classes.

### Need for Inner Class in Java

Now, you might be thinking about why to use an inner class rather than using separate classes? The following points will help you understand the purpose of using inner class along with its importance:

* **It helps in the logical grouping of classes that belong together:**
* **It helps to increase the encapsulation:**
* **It helps to increase the readability and maintainability of the code:**

**Syntax of writing inner class:**

Following is the syntax to write an inner class or nested class. Here, the class OuterClassDemo is the outer class or the top-level class and the class InnerClassDemo is the nested or inner class.

class OuterClassDemo

{

//code of the outer class

class InnerClassDemo

{

//code of inner class

}

}

### Types of Inner Classes in Java

There are four types of inner classes:

1. Nested Inner Class
2. Static Inner Class
3. Method Local Inner Class
4. Anonymous Inner Class

We will discuss each of them in detail along with their syntax and code:

#### **1. Nested Inner Class**

Nested Inner class is an inner class that can access other instance variables of the outer class, even if they are declared as private. We can use any access modifier for the nested inner class – public, private, protected, or default.

**Code to understand Nested Inner Class:**

package com.techvidvan.innerclass;

public class JavaOuterClass

{

// private variable of the outer class

private int value = 30;

// inner class

class JavaInnerClass

{

// public variable of the inner class

public int getValue()

{

System.out.println("This is the getValue method of the inner class:");

return value;

}

} //inner class end here

public static void main(String args[])

{

//Creating object of outer class

JavaOuterClass outer = new JavaOuterClass();

// Creating object of inner class

JavaOuterClass.JavaInnerClass inner = outer.new JavaInnerClass();

System.out.println("Value:" inner.getValue());

}

}

**Output:**

This is the getValue method of the inner class:  
Value: 30

#### **2. Method Local Inner class**

Method Local Inner Class allows us to declare a class inside a **method** body that will be of a **local** type. The scope of the inner class is restricted within the method, similar to the local variables.

We can initialize a local inner class only inside the method where the inner class is defined. We can not declare Method Local Class as private, protected, static and transient but we can declare it as abstract and final, but not both at the same time.

**Code to understand Method Local Inner Class:**

package com.techvidvan.innerclass;

//outer class

public class OuterClass

{

void outerMethod()

{

System.out.println("Inside outer method");

//inner class inside a method of outer class

class InnerClass

{

void innerMethod()

{

System.out.println("Inside inner method");

}

} //inner class ends here

//initializing object of inner class inside the method

InnerClass innerObj = new InnerClass();

innerObj.innerMethod();

} //method ends here

public static void main(String[] args)

{

OuterClass outerObj = new OuterClass();

outerObj.outerMethod();

}

}

**Output:**

Inside outer method  
Inside inner method

**Note:** A Method Local class cannot access a local variable from the outer class. To access the local variable from the outer class, we must define it as final.

**For example, the below code generates an error if we do not declare the variable as final:**

package com.techvidvan.innerclass;

//outer class

public class OuterClass

{

void outerMethod()

{

final int var = 60; //declaring variable as final

System.out.println("Inside outer method");

//inner class inside a method of outer class

class InnerClass

{

void innerMethod()

{

System.out.println("\nInside inner method");

System.out.println("Value of variable is: "+var);

}

}//inner class ends here

//initializing object of inner class inside the method

InnerClass innerObj = new InnerClass();

innerObj.innerMethod();

} //method ends here

public static void main(String[] args)

{

OuterClass outerObj = new OuterClass();

outerObj.outerMethod();

}

}

**Output:**

Inside outer method  
Inside inner method  
Value of variable is: 60

#### **3. Static Inner class**

A static inner class acts as a static member of an outer class. As it is a static member, we can access it without initializing the outer class with the help of a static method. So, we can say that technically Static Inner classes are not a Java inner class.

Similar to static members, a static nested class cannot access the instance variables and methods of the outer class.

**Code to understand Static Inner Class:**

package com.techvidvan.innerclass;

public class OuterClassDemo

{

static class NestedDemo

{

public void myMethod()

{

System.out.println("This is a static nested class");

}

public static void main(String args[])

{

//Accessing the static nested class without initializing the object //of Outer class

OuterClassDemo.NestedDemo nested = new

OuterClassDemo.NestedDemo();

nested.myMethod();

}

}

}

**Output:**

This is a static nested class

#### **4. Anonymous Inner class**

Anonymous inner class is an inner class that is declared without a name. It helps you to make a more concise code. Generally, they are used when there is a need to override the method of a class or an interface.

We can also use them if we need to use a local class only once. They are similar to local inner classes with the exception that they do not have a name.

**Code to understand Static Inner Class:**

package com.techvidvan.innerclass;

interface AnonymousAnimal

{

void type();

}

public class AnonymousInnerClass

{

public static void main(String args[])

{

AnonymousAnimal animal = new AnonymousAnimal(){

public void type()

{

System.out.println("Anonymous Anteater");

System.out.println("Anonymous Unicorn");

System.out.println("Anonymous Capybara");

System.out.println("Anonymous Beaver");

}

};

animal.type();

}

}

**Output:**

Anonymous Anteater  
Anonymous Unicorn  
Anonymous Capybara  
Anonymous Beaver

### What is Garbage Collection?

Garbage collection is a process of reclaiming the unused runtime objects. It is performed for memory management. In other words, we can say that It is the process of removing unused objects from the memory to free up space and make this space available for Java Virtual Machine. Due to garbage collection java gives 0 as output to a variable whose value is not set, i.e., the variable has been defined but not initialized. For this purpose, we were using free() function in the C language and delete() in C++. In Java, it is performed automatically. So, java provides better memory management.

### What is the purpose of the finalize() method?

The finalize() method is invoked just before the object is garbage collected. It is used to perform cleanup processing. The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created an object without new, you can use the finalize method to perform cleanup processing (destroying remaining objects). The cleanup processing is the process to free up all the resources, network which was previously used and no longer needed. It is essential to remember that it is not a reserved keyword, finalize method is present in the object class hence it is available in every class as object class is the superclass of every class in java. Here, we must note that neither finalization nor garbage collection is guaranteed. Consider the following example.

**public** **class** FinalizeTest {

**int** j=12;

**void** add()

    {

        j=j+12;

        System.out.println("J="+j);

    }

**public** **void** finalize()

    {

        System.out.println("Object is garbage collected");

    }

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

**new** FinalizeTest().add();

        System.gc();

**new** FinalizeTest().add();

    }

}

### What is the difference between final, finally and finalize?

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method, and variable. The final class can't be inherited, final method can't be overridden, and final variable value can't be changed. | Finally is used to place important code, it will be executed whether an exception is handled or not. | Finalize is used to perform clean up processing just before an object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |