1. **What is Java?**

Java is a platform-independent high-level programming language. It is platform-independent because its byte codes can run on any system regardless of its operating system.

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

The differences between C++ and Java are given in the following table.

Aspect	**C++**	**Java**
Platform-Independent	C++ is platform-dependent	Java is platform-independent
Used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows, web-based, enterprise, and mobile applications.
Multiple Inheritance	C++ supports multiple inheritance.	Java does not support for multiple inheritance through a class. Interface in java can achieve it
Operator Overloading	C++ supports operator overloading.	Java does not support operator overloading.
Pointers	C++ supports pointers. We can write a pointer program in C++.	Java supports pointers internally, How eve, we cannot write a pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses a compiler only.	Java uses a compiler and

	C++ is compiled and run using the compiler, which converts source code into machine code, so C++ is platform-dependent.	an interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted, which is why it is platform independent.
Call by Value and Call by Reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in Java.
Structure and Union	C++ supports structures and unions.	Java does not support structures and unions.
Thread Support	C++ does not have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Object-Oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.
Memory Management	Manual memory management using new and delete keywords.	Automatic memory management through garbage collection.

Garbage Collection	Manual memory management, no built-in garbage collection.	Automatic garbage collection to manage memory.

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

There are the following features in the Java Programming Language.

a. **Simple**: Java is easy to learn. The syntax of Java is based on C++, which makes it easier to write programs in it.
b. **Object-Oriented Programming (OOP):** Java follows the object-oriented paradigm, which allows us to maintain our code as a combination of different types of objects that incorporate both data and behavior. Everything in Java is treated as an object, promoting modularity and code reusability.
c. **Portable**: Java supports a read-once-write-anywhere approach. We can execute the Java program on every machine. A Java program (.java) is converted to bytecode (.class), which can be easily run on every machine.
d. **Platform Independent:** Java is a platform-independent programming language. It is different from other programming languages like C and C++, which need a platform to be executed. Java comes with its own platform on which its code is executed. Java doesn't depend upon the operating system to be executed.
e. **Secured**: Java is secured because it does not use explicit pointers. Java also provides the concept of ByteCode and Exception handling, which makes it more secured. The Java Runtime Environment (JRE) includes a security manager and a set of APIs to control access to resources and prevent malicious activities.
f. **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.
g. **Architecture Neutral**: Java is architecture-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 does not exist in Java.

h. **Interpreted**: Java uses the Just-in-time (JIT) interpreter along with the compiler for the program execution.

i. **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 (for example, C++). While interpreted, Java's Just-In-Time (JIT) compilation translates bytecode into machine code for improved performance.

j. **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 does not occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc. Java's bytecode can be executed on any device with a JVM, ensuring portability across different platforms.

k. **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.

l. **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 (JVM)?

Java Virtual Machine is a virtual machine that enables a computer to run Java programs. It is a virtualized execution environment that allows Java programs to run on any device with a compatible JVM. JVM is the specification that must be implemented in the computer system. It interprets Java bytecode and translates it into machine code for the underlying hardware.

5. What are the differences between JDK, JRE, and JVM?

Feature	JDK	JRE	JVM

Definition	Java Development Kit (JDK) is a software development kit used to develop Java applications.	Java Runtime Environment (JRE) is a software package that provides Java Virtual Machine (JVM), class libraries and other components to run applications in Java.	Java Virtual Machine (JVM) is an abstract machine that provides an environment for the execution of Java ByteCodes.
Functions	Provides tools for developing Java applications. It includes compiler (javac), JRE, and debugging tools	It provides an environment for executing Java applications. It includes JVM and libraries required to run Java programs, but does not include development tools.	The virtual machine that runs the bytecode generated by the compiler. It's a part of JRE and is platform-dependent but provides a platform-independent execution environment.
Use Cases	Used by developers to write, compile, and debug Java programs.	Used by end-users to execute Java applications without needing development tools.	Converts bytecode into machine code and executes it on the underlying hardware.
Platform Dependency	It is platform dependent.	It is platform dependent.	It is platform dependent.
Implementation	JDK = JRE +	JRE = JVM + Class	JVM = provides a

	Development tools	libraries	runtime environment.

6. **How many types of memory areas are allocated by the JVM?**
   a. **Class (Method) Area**: Class Area stores per-class structures such as the runtime constant pool, field, method data, and the code for methods.
   b. **Heap**: It is the runtime data area in which memory is allocated to the objects
   c. **Stack**: Java Stack stores frames. Holds local variables and partial results, supporting method execution. Each thread has a private JVM stack, created at the same time as the thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
   d. **PC Register (Non-Heap)**: Keeps track of the current execution position for each thread.
   e. **Native Method Stack (Non-Heap):** It contains native method information and supports native code execution.

JVM Memory Struture

7. **What is JIT compiler?**
   - **Just-In-Time (JIT) Compiler**: It is used to improve the performance. The compiler is part of the Java Runtime Environment (JRE).

- It translates Java bytecode into native machine code at runtime. Therefore, reduces the amount of time needed for compilation.
- Here, the term "compiler" refers to a translator from the instruction set of a Java Virtual Machine (JVM) to the instruction set of a specific CPU.

8. What gives Java its 'write once and run anywhere' nature?

Java compiler converts the Java programs into a class file (bytecode), which is the intermediate language between source code and machine code. Code is compiled to bytecode, which is platform-independent, and the JVM executes this bytecode on any device with a compatible JVM.

9. What is a ClassLoader?

Classloader is a subsystem of the JVM that is used to load class files. It loads Java classes into memory during runtime. Responsible for finding and loading class files needed by a Java program.

There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader:**This is the first classloader, which is the superclass of Extension classloader. It loads the *jar* file, which contains all class files of Java Standard Edition, like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes, etc.
2. **Extension ClassLoader:**This is the child classloader of Bootstrap and the parent classloader of the System classloader. It loads the jar files located inside *$JAVA_HOME/jre/lib/ext*

3. **System/Application ClassLoader:**This is the child classloader of the Extension classloader. It loads the class files from the classpath. By default, the classpath is set to the current directory. You can change the classpath using the "-cp" or "-classpath" switch. It is also known as the Application classloader.

10. Is an empty .java file name a valid source file name?

Yes, in Java, the source file can be saved with the .java extension, and then it can be compiled using the javac command followed by the file name (including the .java extension). Finally, the program can be executed using the Java command followed by the name of the class that contains the main() method.

Let's take a simple example:

```
//save by .java only
class Main {
public static void main(String args[]){
System.out.println("Hello Java");
}
}
```

Compile the code using the javac command:

```
javac .java
```

Run the program using the java command:

```
java Main
```

11. Is delete, next, main, exit, or null keyword in Java?

No, in Java, delete, next, main, exit, and null are not keywords.

12. If I do not 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. If we do not provide any command-line arguments when running a Java program, the String array passed into the main() method will be an empty array, not null. The args parameter of the main() method is an array of strings, and it is always initialized by Java, even if no command-line arguments are provided.

13. What if we write static public void instead of public static void?

The program compiles and runs correctly because the order of specifiers does not matter in Java.

In Java, the placement of access modifiers (public, private, protected) and other specifiers (static, final, etc.) before the return type (void in this case) is flexible. Both public static void and static public void are accepted and considered correct syntax.

Here are the two equivalent valid declarations for the main() method:

```
public static void main(String[] args) {
    // Code goes here
}
static public void main(String[] args) {
    // Code goes here
}
```

14. What is the default value of the local variables?

In Java, local variables are not assigned default values by the compiler. They must be explicitly initialized before they are used, or a compilation error will occur.

```
public class Main {
```

```
    public static void main(String[] args) {
        int x;
            System.out.println(x); // Compilation error: The local variable x may not have been initialized
    }
}
```

15. What are the various access specifiers in Java?

In Java, access specifiers are the keywords that are used to define the access scope of a method, class, or variable. There are two types of modifiers in Java: access modifiers and non-access modifiers.

Access Specifiers

public: public classes, methods, or variables that are defined as public can be accessed by any class or method.

protected: The protected classes, methods, or variables can be accessed by the class of the same package, or by a subclass of this class, or within the same class.

private: The private class, methods, or variables defined as private can be accessed within the class only.

default (package-private): If no access specifier is used, it is considered package-private. Members with default access are accessible only within the same package.

Non-Access Specifier

It includes static, synchronized, abstract, native, volatile, transient.

16. 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. Static is the part of the class and not of the object. Static methods and variables in Java are associated with the class rather than with an instance of the class. Therefore, static is used in the case where we need to define variables or methods that 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 a common attribute to all the students. Therefore, the college name will be defined as static.

Class-level Access: Static members belong to the class itself rather than to instances of the class. It means they can be accessed using the class name without creating an instance of the class.

Constants: Static variables are commonly used to define constants in a class. These values are shared among all instances of the class and can be accessed using the class name.

17. What are the advantages of packages in Java?

Packages in Java provide a way to organize and structure classes and interfaces into a hierarchical structure. They offer several advantages:

- Packages avoid the name clashes.
- The Package provides easier access control.
- We can also have the hidden classes that are not visible outside and used by the package.

- It is easier to locate the related classes.

18. What is the output of the following Java program?

```
class Test
{
    public static void main (String args[])
    {
        System.out.println(10 + 20 + "Tpointtech");
        System.out.println("Tpointtech" + 10 + 20);
    }
}
```

Output:

======

30Tpointtech

Tpointtech1020

19. What is the output of the following Java program?

```
class Test
{
    public static void main (String args[])
    {
        System.out.println(10 * 20 + "Tpointtech");
        System.out.println("Tpointtech" + 10 * 20);
    }
}
```

Output:

200Tpointtech

Tpointtech200

20. 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 Tpointtech");
        }
    }
}
```

The above code will result in a compilation error due to the non-boolean expression in the second part of the for loop, and given is an integer value, i.e., 0.

Core Java - OOP Concepts: Initial OOP Interview Ques

21. What is an object-oriented paradigm?

It is a programming paradigm based on objects having data and methods defined in the class to which it belongs. Key principles of the object-oriented paradigm include encapsulation, inheritance, and polymorphism, providing a framework for designing and structuring software in a way that models real-world entities and promotes code reusability and maintainability. Objects are the instances of classes that interact with one another to design applications and programs.

There are the following features of the object-oriented paradigm.

- OOPs follows the bottom-up approach in program design.
- Focus on data with methods to operate upon the object's data
- Includes concepts like Encapsulation and abstraction, which hide the complexities from the user and show only functionality.
- Implements the real-time approach like inheritance, abstraction, etc.

The examples of the object-oriented paradigm are C++, Simula, Smalltalk, Python, C#, etc.

22. What is an object?

The Object is a real-time entity having some state and behavior. In Java, an Object is an instance of a class that has instance variables as the state of the object and methods as the behavior of the object. The object of a class can be created by using the new keyword.

For example, if we have a class called Car, an object of that class could represent a specific car with attributes like color, model, year, and methods like startEngine() or drive(). Each instance of the Car class would be a distinct object with its own unique state.

23. What is the difference between an object-oriented programming language and an object-based programming language?

There are the following basic differences between object-oriented languages and object-based languages.

Object-oriented languages follow all the concepts of OOP, whereas object-based languages do not follow all the concepts of OOP, like inheritance and polymorphism. It allows the creation of classes and objects, and it supports the defining and implementation of methods within those classes.

Object-oriented languages do not have the inbuilt objects, whereas Object-based languages have the inbuilt objects; for example, JavaScript has the window object. It allows the creation of objects and encapsulation of data, but it might not provide the complete set of features associated with full OOP.

Examples of object-oriented programming are Java, C#, Smalltalk, etc., whereas the examples of object-based languages are JavaScript, VBScript, etc.

24. What will be the initial value of an object reference that is defined as an instance variable?

In Java, when an object reference is declared as an instance variable (a member variable of a class). It is automatically initialized to null by default. It is true for all object references, regardless of the class type.

Core Java - OOPs Concepts: Constructor Interview Questions

25. What is the constructor?

The constructor can be defined as a 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.

26. 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.

Default Constructor: The default constructor is the one that does not accept any value. The default constructor is mainly used to initialize the instance variable with the default values. The default constructor initializes instance variables to their default values (numeric types to 0, booleans to false, and object references to null). A default constructor is invoked implicitly by the compiler if there is no constructor defined in the class.

```
public class Main {
    // Default constructor (automatically provided if not defined explicitly)
}
```

Parameterized Constructor: The parameterized constructor is the one that can initialize the instance variables with the given values. It allows us to initialize the object with specific values during the object creation process. In other words, we can say that the constructors that can accept the arguments are called parameterized constructors.

```
public class Main {
    // Parameterized constructor
```

```
    public Example(int value) {
        // Initialization logic here
    }
}
```

27. What is the purpose of a default constructor?

The default constructor in Java is a constructor with no parameters. The purpose of the default constructor is to assign the default value to the objects. The Java compiler automatically provides it if a class does not have any explicit constructor defined.

Example

```
class Main {
int id;
String name;
void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
Main m1=new Main ();
Main m2=new Main();
m1.display();
m2.display();
}
}
```

Output:

0 null

0 null

28. Does the constructor return any value?

Yes, in Java, a constructor does not have a return type, not even void. The purpose of a constructor is to initialize the object's state and prepare it for use. The constructor implicitly returns the current instance of the class (we cannot use an explicit return type with the constructor).

29. Is the constructor inherited?

No, Constructors are not inherited. The superclass constructor can be called from the first line of a subclass constructor by using the keyword super and passing appropriate parameters to set the private instance variables of the superclass.

30. Can you make a constructor final?

No, in Java, constructors cannot be declared as final. The final keyword is used to indicate that a class, method, or variable cannot be further subclassed, overridden, or modified. However, it does not apply to constructors.

31. 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.

Each constructor provides a different way to initialize an object of the class. The choice of which constructor to invoke is based on the arguments passed during object creation.

Consider the following example.

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);
        }
    }
```

Output:

Hi I am assigning the value max(k, m) to i

10

15

32. 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 a copy constructor in C++.

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

- By using a constructor
- By assigning the values of one object to another

- By using Object.clone() Method

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

Example

```
public class Main {
    int id;
    String name;
    // Constructor to initialize integer and string
    Main(int id, String name) {
        this.id = id;
        this.name = name;
    }
    // Copy constructor
    Main(Main s) {
        this.id = s.id;
        this.name = s.name;
    }
    void display() {
        System.out.println(id + " " + name);
    }
    public static void main(String[] args) {
        Main obj1 = new Main(111, "Karan");
        Main obj2 = new Main(obj1);
        obj1.display();
        obj2.display();
    }
}
```

Output:

111 Karan

111 Karan

33. What are the differences between the constructors and methods?

There are many differences between constructors and methods.

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 we do not have any constructor in a class.	The compiler does not provide the method in any case.
The constructor's name must be the same as the class name.	The method name may or may not be the same as the class name.

NEXT IS NOW

34. What is the output of the following Java program?

```
public class Main
{
    Main(int a, int b)
    {
        System.out.println("a = "+a+" b = "+b);
    }
    Main(int a, float b)
    {
        System.out.println("a = "+a+" b = "+b);
    }
    public static void main (String args[])
    {
        byte a = 10;
        byte b = 15;
        Main obj = new Main(a,b);
```

```
    }
}
```

Output:

a = 10 b = 15

35. 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);
    }
}
```

Output:

0

36. 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);
    }
}
```

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem: The constructor Test() is undefined

Core Java - OOPs Concepts: static Keyword Interview Questions

37. What is the static variable?

The static variable is used to refer to the common property of all objects (that is not unique for each object), for example, the company name of employees, the 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). A static variable belongs to the class rather than the object. The following Java program depicts the use of a static variable.

Example

```
public class Main {
    int rollno;
    String name;
    static final String COLLEGE = "ITS";
    Main(int r, String n) {
        rollno = r;
        name = n;
    }
    void display() {
        System.out.println(rollno + " " + name + " " + COLLEGE);
    }
```

NEXT IS NOW

```
    public static void main(String[] args) {
        Main obj1 = new Main (111, "Karan");
        Main obj2 = new Main(222, "Aryan");
        obj1.display();
        obj2.display();
    }
}
```

Output:

111 Karan ITS
222 Aryan ITS

38. What is the static method?

In Java, a static variable is a class-level variable that belongs to the class rather than to instances of the class. It is shared among all instances of the class and is initialized only once when the class is loaded into memory.

- 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.

39. What are the restrictions that are applied to the Java static methods?

Java static methods have certain restrictions and characteristics that differentiate them from instance methods. Here are the key restrictions and considerations for Java static methods:

The two main restrictions applied to the static methods are as follows:

- The static method cannot use a non-static data member or call a non-static method directly.
- This and super keyword cannot be used in a static context as they are non-static.

- Static methods in Java cannot be overridden. Even if a subclass declares a static method with the same signature as a static method in its superclass, it is considered method hiding, not overriding.

40. Why the main() method is static?

Because the object is not required to call the static method, if we make the main() method non-static, the JVM will have to create its object first and then call the main() method, which will lead to extra memory allocation.

Here are the main reasons:

The main() method is the entry point for the execution of a Java program. When we run a Java program, the Java Virtual Machine (JVM) looks for the main() method with the following signature:

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

When a Java program is executed, the JVM loads the class into memory and looks for the main() method to start the program.

41. Can we override the static methods?

No, static methods cannot be overridden in Java. While a subclass can declare a static method with the same signature as a static method in its superclass, it is considered method hiding, not method overriding.

42. What is the static block?

A static block is used to initialize the static data member. It is executed before the execution of the main() method, at the time of class loading.

Example

```
class Main {
static { System.out.println("static block is invoked"); }
public static void main(String args[]) {
System.out.println("Hello main");
```

```
    }
}
```

Output:

static block is invoked

Hello main

43. Can we execute a program without a main() method?

No, in Java, we cannot execute a program without the main() method. The main() method serves as the entry point for the execution of a Java program, and it must be defined in every Java class that serves as the starting point for an application. The JVM looks for the main() method with the following signature to begin program execution:

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

44. What if the static modifier is removed from the signature of the main method?

Program compiles. If the static keyword is removed from the signature of the main() method in Java, the program will compile successfully, but it will not run successfully. The main() method must be declared as static because it serves as the entry point for the Java program, and the Java Virtual Machine (JVM) needs to invoke it without creating an instance of the class. It throws an error "NoSuchMethodError."

45. What is the difference between a static (class) method and an instance method?

A method that is declared as static is known as a static method.	A method that is not declared as static is known as an instance method.
We do not need to create the objects to call the static methods.	The object is required to call the instance methods.
Non-static (instance) members cannot be accessed in the static context (static	Static and non-static variables can both be accessed in instance methods.

method, static block, and static nested class) directly.	
For example: public static int cube(int n){ return n*n*n;}	For example: public void msg(){...}.

46. Can we make constructors static?

No, constructors cannot be declared as static in Java, 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 in making the constructors static. Making a constructor static would imply that it belongs to the class and not to instances, which contradicts the fundamental purpose of a constructor. However, if we try to do so, the compiler will show the compiler error..

47. Can we make the abstract methods static in Java?

No, abstract methods cannot be declared as static in Java. In Java, if we make the abstract methods static, they will become part of the class, and we can directly call them, which is unnecessary. Since abstract methods are expected to be overridden by subclasses, making them static would be contradictory to their purpose. Calling an undefined method is completely useless; therefore, it is not allowed.

48. Can we declare the static variables and methods in an abstract class?

Yes, it is possible to declare static variables and methods in an abstract class in Java. As we know that there is no requirement to make the object 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.

Example

```
abstract class Test
{
   static int i = 102;
   static void display()
   {
```

```
        System.out.println("hi !! I am good !!");
    }
}
public class Main extends Test
{
    public static void main (String args[])
    {
        Test.display();
        System.out.println("i = "+Test.i);
    }
}
```

Output

hi !! I am good !!
i = 102

Core Java - OOPs Concepts: Inheritance Interview Questions

49. What is this keyword in Java?

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

50. What are the uses of this keyword?

There are the following uses of this keyword:

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

Let's understand through a Java program.

```
public class Main {
    private int x;
    public void setX(int x) {
        this.x = x; // Using 'this' to refer to the instance variable
    }
}
```

51. Can we assign the reference to this variable?

No, it is not possible to assign a new value to this variable in Java. this keyword is an implicit reference to the current instance of the class and is automatically set by the Java runtime system when a method is invoked. However, if we try to do so, the compiler error will be shown. Consider the following example.

Example

```
public class Main
{
    public Main()
    {
```

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

Output

Main.java:5: error: cannot assign to 'this'
this = null;
^
1 error

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

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

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

Output

10

53. 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. Constructor chaining in Java can be achieved using the this keyword. Consider the following example that illustrates how we can 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, "Peter", "Delhi");
```

```
            System.out.println("ID: "+emp.id+" Name: "+emp.name+" age: "+emp.age+" address: "+emp.address);
        }
    }
```

Output

ID: 105 Name: Peter age: 22 address: Delhi

54. What are the advantages of passing this into a method instead of the current class object itself?

As we know, 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 keyword can be used in the synchronized block.

Passing this into a method instead of the current class object itself provides several advantages:

- Flexibility
- Encapsulation
- Consistency

55. What is Inheritance?

Inheritance is a mechanism by which one object acquires all the properties and behavior of another object of another class. It is used for Code Reusability and Method Overriding. Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit attributes and behaviors from another class (superclass or base class). Moreover, we can add new methods and fields in your current class also. Inheritance represents the IS-A relationship that is also known as a parent-child relationship.

To read more Inheritance in Java

There are five types of inheritance in Java:

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

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

56. Why is inheritance used in Java?

Various advantages of using inheritance in Java are given below.

- Inheritance provides code reusability. Inheritance allows subclasses to inherit attributes and behaviors from a superclass, promoting code reuse. Common functionality can be defined in a superclass, and subclasses can extend or specialize it.
- Runtime polymorphism cannot be achieved without using inheritance.
- We can simulate the inheritance of classes with 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. Inheritance facilitates polymorphism, allowing objects of a subclass to be treated as objects of the superclass. By method overriding, we can give a specific implementation of some basic method contained by the base class.

57. Which class is the superclass for all the classes?

In Java, the Object class is the superclass of all other classes. Every class in Java directly or indirectly extends the Object class. The Object class provides fundamental methods that are inherited by all classes, such as toString(), equals().

58. Why is multiple inheritance not supported in Java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in Java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and we call it from a child class object, there will be ambiguity in calling the method of A or B class.

The Diamond Problem occurs when a class inherits from two classes that have a common ancestor, leading to ambiguity in method resolution.

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

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class Main extends A,B{//suppose if it were
 public Static void main(String args[]){
   Main obj=new Main();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

The above program gives a compile-time error.

59. What is aggregation?

Aggregation can be defined as the relationship between two classes where the aggregate class contains a reference to the class it owns.

Aggregation is a type of association in object-oriented programming where one class contains an object of another class, forming a relationship between them. It represents a "has-a" relationship, indicating that a class has an entity as a part of its structure.

For example, the aggregate class Employee, having various fields such as age, name, and salary, also contains an object of the Address class, having various fields such as Address-Line 1, City, State, and pin-code.

In other words, we can say that Employee (class) has an object of Address class. Consider the following example.

Address.java

```
public class Address {
String city, state, country;
public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
}
}
```

Employee.java

```
public class Emp {
int id;
String name;
Address address;
public Emp(int id, String name,Address address) {
   this.id = id;
   this.name = name;
   this.address=address;
}
void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}
public static void main(String[] args) {
Address address1 = new Address("gzb", "UP", "India");
Address address2 = new Address("gno", "UP", "India");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();
e2.display();
}
}
```

Output

111 varun
gzb UP India
112 arun
gno UP India

60. What is composition?

Composition is a stronger form of aggregation in object-oriented programming. Holding the reference of a class within another class is known as composition. When an object contains another object, if the contained object cannot exist without the existence of the

container object, then it is called composition. In other words, we can say that composition is a particular case of aggregation, which represents a stronger relationship between two objects. Example: A class contains students. A student cannot exist without a class. There exists a composition between the class and the students.

Composition is a way to create more robust and tightly integrated structures by combining objects to form a larger, more complex system.

61. What is the difference between aggregation and composition?

Aggregation represents a weak relationship, whereas composition represents a strong relationship. Aggregation represents a "has-a" relationship, where one class has a reference to another class, but the referenced class can exist independently. Composition also represents a "has-a" relationship, but it is a stronger form of association. For example, the bike has an indicator (aggregation), but the bike has an engine (composition).

62. Why does Java not support pointers?

Java does not support explicit pointers like those found in languages such as C or C++, as 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.

63. What is super in Java?

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

The super keyword is commonly used in scenarios where a class extends another class, and there is a need to differentiate between the members of the subclass and the superclass.

```
class Animal {
Animal() { System.out.println("An animal is created"); }
}
class Dog extends Animal {
Dog(){
System.out.println(" A dog is created");
}
}
class Main {
public static void main(String args[]) {
Dog d=new Dog();
}
}
```

Output:

An animal is created
A dog is created

64. How can constructor chaining be done by using the super keyword?

Constructor chaining in Java refers to the process of calling one constructor from another within the same class or in the parent class. The super keyword is used to invoke the constructor of the parent class.

Example

```
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 Main
{
    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

65. What are the main uses of the super keyword?

There are the following uses of the super keyword:

- It is used to refer to the immediate parent class instance variable.
- It is used to invoke the immediate parent class method.

- The constructor super() can be used to invoke the immediate parent class constructor.

66. What are the differences between this and super keyword?

These 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 instance of the class.

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.

67. 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

68. Can we use this() and super() in a constructor?

No, you cannot use both this() and super() in the same constructor because this() and super() must be the first statement in the class constructor. Using both in the same constructor would result in a compilation error.

Example:

```
public class Main {
    Main()
    {
        super();
        this();
        System.out.println("Main class object is created");
    }
    public static void main(String []args){
    Main t = new Main();
    }
}
```

Output:

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

69. What is object cloning?

Object cloning in Java refers to the process of creating an 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 do not implement Cloneable interface, clone() method generates CloneNotSupportedException. The clone() method creates a new object with the same state as the original object.

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

Core Java - OOP Concepts: Method Overloading Interview Questions

70. What is method overloading?

Method overloading is the polymorphism technique that allows us to create multiple methods with the same name but different signatures. Method overloading in Java occurs when a class has multiple methods with the same name but different parameter lists (number, type, or order of parameters). We can achieve method overloading in the following two ways:

- Different number of arguments
- Different data types of arguments

Method overloading increases the readability of the program. Method overloading is performed to figure out the program quickly.

71. 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 to avoid ambiguity. The method name and the parameter types determine the method signature. Changing only the return type does not provide enough information for the compiler to distinguish between overloaded methods.

```
class Adder{
```

```
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class Main {
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

72. 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

73. Can we overload the main() method?

Yes, we can have any number of main() methods in a Java program by using method overloading. However, the JVM looks for the standard public static void main(String[] args) signature when starting a Java program. Overloaded main() methods can be called from within the standard main() method, but the program execution begins from the standard main() method.

74. 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.

Java Method Overloading with Type Promotion

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.

NEXT IS NOW

```
class Main {
  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[]){
Main obj=new Main();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);
  }
}
```

Output

40

60

75. What is the output of the following Java program?

```
class Main {
  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[]){
Main obj = new Main();
  obj.sum(20,20);//now ambiguity
  }
}
```

Output

```
Main.java:7: error: reference to sum is ambiguous
obj.sum(20,20);//now ambiguity
^
both method sum(int,long) in Main
and method sum(long,int) in Main match
1 error
```

Core Java - OOP Concepts: Method Overriding Interview Questions

76. What is method overriding?

If a subclass provides a specific implementation of a method that is already provided by its parent class, it is known as Method Overriding. Method overriding in Java occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. It is used for runtime polymorphism and to implement the interface methods.

Rules for Method Overriding

- The method must have the same name as in the parent class.
- The method must have the same signature as in the parent class.
- Two classes must have an IS-A relationship between them.

77. Can we override the static method?

No, we cannot override the static method because they are a part of the class, not the object.

78. Why can we not override a static method?

Static methods cannot be overridden in the traditional sense of dynamic method dispatch that is associated with polymorphism. It is because the static method is a part of the class, and it is bound with the class, whereas the instance method is bound with the object, and static gets memory in the class area, and instance gets memory in the heap.

79. Can we override the overloaded method?

Yes, we can override the overloaded method in Java. Overriding: Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in a superclass (or base class).

80. What are the differences between method overloading and overriding?

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

81. Can we override the private methods?

No, we cannot override the private methods in Java because the scope of private methods is limited to the class, and we cannot access them outside of the class. But private methods are not accessible outside their class. Therefore, private methods cannot be overridden because subclasses do not inherit them.

82. 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 access specifier can be changed to protected, public, or default.
- The protected access specifier can be changed to public or default.
- The default access specifier can be changed to public.
- The public access specifier will always remain public.

83. 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 that are to be followed while overriding in case of exception handling.

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

84. What is the output of the following Java program?

Example

```
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

85. Can you have virtual functions in Java?

Yes, all functions in Java are virtual by default. It means that method invocation is determined at runtime based on the actual type of the object on which the method is called. So, while the term "virtual" is not explicitly used in Java, the concept is present through the mechanism of dynamic method dispatch and method overriding.

86. What is a covariant return type?

Since Java 5, it is possible to override any method by changing the return type if the return type of the subclass overriding method is a subclass type. It is known as a covariant return type. The covariant return type specifies that the return type may vary in the same direction as the subclass.

Example

```
class A {
A get(){return this;}
}
class Main extends A {
Main get() {
return this; }
void message() {
System.out.println("Welcome to the covariant return type");}
public static void main(String args[]){
new Main().get().message();
}
}
```

Output

Welcome to the covariant return type

87. What is the output of the following Java program?

Example

```
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 Main
{
    public static void main (String args[])
    {
        Base b = new Derived();
        b.baseMethod();
    }
}
```

Output

Derived method called ...

Core Java - OOPs Concepts: final keyword Interview Questions

88. 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 cannot change its value. The final keyword is used to declare a final variable. 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 Main {
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
Main obj=new Main();
 obj.run();
 }
}//end of class
```

Output:

Compile Time Error

89. What is the final method?

If we change any method to a final method, we cannot override it. In Java, a final method is a method that subclasses cannot override. When a method is declared as final in a superclass, it means that no subclass can provide a different implementation for that method.

```
class Bike{
 final void run(){System.out.println("running");}
```

```
}
class Main extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Main obj = new Main();
  obj.run();
  }
}
```

Output:

Compile Time Error

90. What is the final class?

If we make a class final, we cannot inherit it into any of the subclasses.

In Java, a final class is a class that cannot be subclassed. When a class is declared as final, it means that no other class can extend (inherit from) it.

```
final class Bike{}
class Main extends Bike{
 void run(){System.out.println("running safely with 100kmph");}
 public static void main(String args[]){
 Main obj = new Main();
 Obj.run();
 }
}
```

NEXT IS NOW

Output:

Compile Time Error

91. What is the final blank variable?

In Java, a final blank variable refers to a blank final variable that has not been initialized at the time of declaration but must be initialized exactly once before it is used. The term "blank" implies that the variable is declared without an initial value. It is useful in the case

when the user has some data that must not be changed by others, for example, the PAN Number. Consider the following example:

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

92. Can we initialize the final blank variable?

Yes, if it is not static, we can initialize it in the constructor. If it is a static, blank final variable, it can be initialized only in the static block.

93. Can we declare the main() method as final?

Yes, we can declare the main() method as public static final void main(String[] args){}. it is technically allowed to declare the main method as final, but it would not have any impact or special significance. The final keyword, when applied to a method, indicates that subclasses cannot override the method.

94. 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

95. 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

Main.java:10: error: getInfo() in Main cannot override getInfo() in Base

protected final void getInfo()

^

overridden method is final

1 error

96. Can we declare a constructor as final?

No, constructors cannot be declared as final in Java because it is never inherited. Constructors are not ordinary methods; therefore, there is no sense in declaring constructors as final. However, if we try to do so, the compiler will throw an error.

97. Can we declare an interface as final?

No, we cannot declare an interface as final because some class must implement the interface to provide its definition. The whole idea of an interface is to provide a contract that classes can implement. Making an interface final would defeat the purpose of allowing classes to implement it. However, if we try to do so, the compiler will show an error.

98. What are the differences between the final method and the abstract method?

Final method: A final method in a class cannot be overridden by subclasses. Once a method is declared final, it cannot be modified or overridden by any subclass.

Abstract method: An abstract method is a method declared in an abstract class or interface that has no implementation in the class/interface where it is declared. Subclasses or implementing classes must provide a concrete implementation for abstract methods.

The main difference between the final method and abstract method is that the abstract method cannot be final, as we need to override it in the subclass to give its definition.

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

There are the following differences between compile-time polymorphism and runtime polymorphism.

S.N	Compile time polymorphism	Run time

		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.

100. What is Runtime Polymorphism?

Runtime polymorphism also known as 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. It is achieved through method overriding in object-oriented programming, where a subclass provides a specific implementation for a method that is already defined in its superclass.

```
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.

101. Can we achieve runtime polymorphism by data members?

No, runtime polymorphism is primarily related to methods/functions, not data members 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. Data members (fields or properties) do not exhibit polymorphic behavior in the same way. 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
```

102. 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();
 }
}
```

103. 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

104. What is Java instanceof operator?

The instanceof operator in Java is also known as type comparison operator because it compares the instance with type. It is used to test whether an object is an instance of a particular class, interface, or 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

Core Java - OOPs Concepts: Abstraction Interview Questions

105. 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, it involves simplifying the system by modelling classes based on real-world entities and focusing on relevant attributes and behaviors. For example, sending SMS where you type the text and send the message. We do not know the internal processing about the message delivery. Abstraction enables us 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

106. What is the difference between abstraction and encapsulation?

Abstraction and encapsulation are related concepts but have distinct meanings. Abstraction is about showing only the essential features of an object while hiding the unnecessary details. Encapsulation, on the other hand, is the bundling of data and methods that operate on that data into a single unit, often a class. Abstraction is more about design and modelling, while encapsulation is about implementation and data hiding.

107. What is the abstract class?

A class that is declared as abstract is known as an abstract class. It needs to be extended and its method implemented. It cannot be instantiated. Abstract classes can have both abstract and concrete methods and can be extended by other classes. It can also have the final methods which will force the subclass not to change the body of the method.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Output

running safely

108. Can there be an abstract method without an abstract class?

No, an abstract method must belong to an abstract class. Abstract methods are defined in abstract classes and must be implemented by the subclasses that extend the abstract class.

109. Is the following program written correctly? If yes, 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

110. Can we use abstract and final both with a method?

No, we cannot use both abstract and final together for a method. An abstract method is meant to be overridden by subclasses, while a final method cannot be overridden.

111. Is it possible to instantiate the abstract class?

No, we cannot instantiate an abstract class directly in Java. Abstract classes are meant to be extended by concrete subclasses, and we can only create objects of concrete classes.

112. What is the interface?

The interface is a blueprint for a class that has static constants and abstract methods. It can be used to achieve full abstraction and multiple inheritance. An interface in Java is a collection of abstract methods. It is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. However, we need to implement it to define its methods. Since Java 8, we can have the default, static, and private methods in an interface.

113. Can we declare an interface method static?

No, interface methods cannot be declared as static prior to Java 8. However, starting from Java 8, we can declare static methods in interfaces.

114. Can the Interface be final?

No, because an interface needs to be implemented by the other class and if it is final, it cannot be implemented by any class.

115. What is a marker interface?

A marker interface in Java is an interface that does not declare any methods. Its purpose is to mark or tag a class, indicating that the class possesses certain behavior or characteristics. For example, Serializable, Cloneable are marker interfaces. The marker interface can be declared as follows.

```
public interface Serializable{
}
```

116. 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 cannot 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();
	}

117. Can we define private and protected modifiers for the members in interfaces?

No, In Java interfaces, all members (fields and methods) are implicitly public and abstract. Interface members are public because the interface defines a contract that implementing classes must adhere to, and abstract because they are meant to be implemented by the classes that implement the interface.

118. 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.

119. 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;
}
}
```

120. 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;
}
}
```

121. What are the advantages of Encapsulation in Java?

There are the following advantages of Encapsulation in Java:

By providing only the setter or getter method, we can make the class read-only or write-only. In other words, we can skip the getter or setter methods.

It provides control over the data. Suppose we want to set the value of id which should be greater than 100 only, we can write the logic inside the setter method. We can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

The encapsulate class is easy to test. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is easy and fast to create an encapsulated class in Java.

Core Java - OOPs Concepts: Package Interview Questions

122. What is the package?

A package is a group of similar type of classes, interfaces, and sub-packages. It provides access protection and removes naming collision. The packages in Java can be categorized into two forms, inbuilt package, and user-defined package. There are many built-in packages such as Java, lang, awt, javax, swing, net, io, util, sql, etc. Consider the following example to create a package in Java.

Simple.java

```
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

123. What are the advantages of defining packages in Java?

By defining packages, we can avoid the name conflicts between the same class names defined in different packages. Packages also enable the developer to organize the similar classes more effectively. For example, one can clearly understand that the classes present in java.io package are used to perform io related operations.

124. How to create packages in Java?

If you are using the programming IDEs like Eclipse, NetBeans, MyEclipse, etc. click on file->new->project and eclipse will ask you to enter the name of the package. It will create the project package containing various directories such as src, etc.

If we are using an editor like notepad for Java programming, use the following steps to create the package.

To create packages in Java, you use the package keyword at the beginning of your Java source file and organize your classes in directories based on the package name. Define a package package_name. Create the class with the name class_name and save this file with your_class_name.java.

Now compile the file by running the following command on the terminal.

```
javac -d . your_class_name.java
```

The above command creates the package with the name package_name in the present working directory.

Now, run the class file by using the absolute class file name, like following.

```
java package_name.class_name
```

125. How can we access some class in another class in Java?

There are two ways to access a class in another class.

By using the fully qualified name: To access a class in a different package, either we must use the fully qualified name of that class, or we must import the package containing that class.

```
// Using fully qualified name
com.example.otherpackage.OtherClass obj = new
com.example.otherpackage.OtherClass();
// Using import statement
import com.example.otherpackage.OtherClass;
// ...
OtherClass obj = new OtherClass();
```

By using the relative path, we can use the path of the class that is related to the package that contains our class. It can be the same or subpackage.

126. Do I need to import java.lang package any time? Why?

No. we do not need to import the java.lang package explicitly as it is automatically imported in every Java program. It is by default loaded internally by the JVM.

127. Can I import same package/class twice? Will the JVM load the package twice at runtime?

Yes, one can import the same package or the same class multiple times. Neither compiler nor JVM complains about it. However, the JVM will load the package or class only once at runtime.

128. What is the static import?

Static import is a feature introduced in Java 5 that allows members (fields and methods) of a class to be used in a Java program without qualifying them with the class name.

```
import static java.lang.Math.*;
public class MyMathClass {
    public static void main(String[] args) {
        double result = sqrt(25); // No need to use Math.sqrt here
        // ...
    }
}
```

Java: Exception Handling Interview Questions

129. How many types of exception occurs in a Java program?

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- Checked Exception: Checked exceptions are checked at compile-time, and the programmer must handle them using try-catch or declare them in the method's throws clause. For example, SQLException, ClassNotFoundException, etc.
- Unchecked Exception: Unchecked exceptions are the one which are handled at runtime because they cannot be checked at compile-time. For example, ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.

130. What is Exception Handling?

Exception Handling is a mechanism that is used to handle runtime errors (exceptions) to prevent the abnormal termination of a program. 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.

131. Explain the hierarchy of Java Exception classes?

The hierarchy of Java Exception classes includes the top-level class 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:

PARVAM
NEXT IS NOW

132. What is the difference between Checked Exception and Unchecked Exception?

1) Checked Exception

Checked exceptions are checked at compile-time, and the programmer must handle them using try-catch or declare them in the method's throws clause. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions, for example, IOException, SQLException, etc.

2) Unchecked Exception

Checked exceptions are not checked at compile-time and include runtime exceptions and errors. The classes that extend RuntimeException are known as unchecked exceptions, for example, ArithmeticException, NullPointerException, etc.

133. What is the base class for Error and Exception?

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

134. Is it necessary that each try block must be followed by a catch block?

No, it is not necessary to follow a try block with a catch block. We can use a try block with either catch, finally, or both. So whatever exceptions are likely to be thrown should be declared in the throws clause of the method. Consider the following example.

```
public class Main{
    public static void main(String []args){
      try{
        int a = 1;
        System.out.println(a/0);
      }
      finally
      {
        System.out.println("rest of the code...");
      }
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...

135. 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 cannot 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.

136. What is finally block?

In programming languages like Java, the finally block is a part of exception handling. It contains code that will be executed regardless of whether an exception is thrown or not. The finally block is often used to ensure that certain clean-up or resource release operations take place, whether an exception occurs or not.

Finally block follows try or catch block. If we do not handle the exception, before terminating the program, JVM runs finally block, (if any). The finally block is mainly used to place the clean-up 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).

137. Can finally block be used without a catch?

Yes, a finally block can be used without a catch block. In exception handling, we can have a try block followed by either a catch block, a finally block, or both. 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.

138. Is there any case when finally block will not be executed?

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). One such case is when the program is terminated abruptly, for example, by calling System.exit() or if there is a hardware failure or an error in the JVM that prevents normal program termination.

139. What is the difference between throw and throws?

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

140. 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 cannot throw basic data type from a block of code.

141.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.

142. Can an exception be rethrown?

Yes, an exception can be rethrown. In Java, we can catch an exception in one catch block and then throw the same exception (or a different one) again. It can be useful for logging or handling the exception at a higher level in the program.

143. Can subclass overriding method declare an exception if parent class method doesn't throw an exception?

Yes, in Java, if a subclass is overriding a method from its parent class, the subclass is allowed to declare an exception in its method signature even if the overridden method in the parent class does not throw an exception.

144. What is exception propagation?

Exception propagation refers to the process by which an exception is thrown in a program and then propagated up the call stack until it is caught and handled by an appropriate catch block or until it reaches the top level of the program.

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...

145. 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 the Main class is called that prints a message and call b(). The method b() prints some message and then call the method c(). The method c() throws an exception that 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().

146. 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 the 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

149. What is String Pool?

The String Pool is a special memory area in Java where string literals are stored. 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 does not exist in the pool, a new string instance is created and placed in the pool. This mechanism helps in memory optimization by reducing the number of duplicate string objects. Therefore, it saves the memory by avoiding the duplicacy.

148. 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 cannot 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

149. Why are the objects immutable in Java?

Objects are immutable in Java for several reasons, 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. Thread Safety, Immutable objects are inherently thread-safe. That is why string objects are immutable in Java.

150. What are the ways to create string object?

There are several ways to create a string object in Java:

1) Using String Literal

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

```
String s="welcome";
```

Every time when 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 does not exist in the pool, a new string instance is created and placed in the pool. String objects are stored in a special memory area known as the string constant pool. For example,

```
String s1="Welcome";
String s2="Welcome";    //it does not create a new instance
```

2) Using the new Keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the constant string pool. The variable s will refer to the object in a heap (non-pool).

151. 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

152. Why Java uses the concept of the string literal?

Java uses the concept of string literals for efficiency and memory optimization. To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool). It reduces memory consumption and improves performance.

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

```
String s = new String("Welcome");
```

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

154. 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.

155. What is the output of the following Java program?

```
public class Test
```

```
{
    public static void main (String args[])
    {
        String s1 = "Sharma is a good player";
        String s2 = new String("Sharma is a good player");
        s2 = s2.intern();
        System.out.println(s1 ==s2);
    }
}
```

Output

true

Explanation

Let's break down the code line by line and understand how String objects and the string pool work in Java:

1. **String s1 = "Sharma is a good player";**
    - When you create a String literal like this, Java first checks its **string pool**. The string pool is a special memory area within the heap that stores unique String objects.
    - If "Sharma is a good player" doesn't already exist in the string pool, a *new* String object is created in the pool with this value, and s1 is made to refer to this object.
    - If it *does* exist, s1 is simply made to refer to the existing object in the pool. In most cases, for the first declaration of a literal, it will be added to the pool.
2. **String s2 = new String("Sharma is a good player");**
    - When you use the new keyword to create a String object, you are explicitly telling Java to create a *new* object in the **heap**, regardless of whether an identical string exists in the string pool.
    - So, at this point, there are two distinct String objects in memory:
        - One in the string pool (referred to by s1).

    - One in the general heap (referred to by s2).
  - Even though their *content* is identical, they are different objects in different memory locations.

3. **s2 = s2.intern();**

   - This is the crucial line. The intern() method is used to "intern" a String object.
   - When s2.intern() is called, Java checks the string pool for a String object that has the same content as the current s2 ("Sharma is a good player").
   - **If a matching string is found in the string pool:** The intern() method returns a reference to that existing string object from the pool.
   - **If no matching string is found in the string pool:** The current String object (the one s2 refers to) is *added* to the string pool, and a reference to *that same object* is returned.
   - In our case, "Sharma is a good player" *already exists* in the string pool because of the s1 declaration. Therefore, s2.intern() returns a reference to the *same String object in the string pool that s1 is already pointing to*.
   - The line s2 = s2.intern(); then reassigns s2 to point to this String object from the string pool.

4. **System.out.println(s1 == s2);**

   - The == operator when used with objects in Java compares their **references** (i.e., their memory addresses), not their content.
   - After the s2 = s2.intern(); line, both s1 and s2 are now pointing to the *exact same String object* in the string pool.
   - Since they refer to the same object in memory, their references are equal.

Therefore, s1 == s2 evaluates to true.

156. What are the differences between String and StringBuffer?

String	StringBuffer
The String class is immutable.	The StringBuffer class is mutable.

The String is slow and consumes more memory when we concat too many strings because every time it creates a new instance.	The StringBuffer is fast and consumes less memory when you concat strings.
The String class overrides the equals() method of Object class. So we can compare the contents of two strings by equals() method.	The StringBuffer class does not override the equals() method of Object class

157. What are the differences between StringBuffer and StringBuilder?

StringBuffer	**StringBuilder**
StringBuffer is *synchronized*, i.e., thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized*,i.e., not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

158. 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.

a. Make the class final, so it cannot be subclassed.
b. Declare all fields private and final.
c. Do not provide setter methods for the fields.
d. Ensure that any mutable objects within the class are defensively copied to prevent external modification.
e. If the class has mutable fields, make sure to return defensive copies in getter methods.

```
public final class Employee{
```

```
final String pancardNumber;

public Employee(String pancardNumber){
this.pancardNumber=pancardNumber;
}

public String getPancardNumber(){
return pancardNumber;
}

}
```

159. What is the purpose of the toString() method in Java?

The toString() method is used to obtain a string representation of an object. It is defined in the Object class and can be overridden by custom classes to provide a meaningful representation of the object's state depending on implementation. By overriding the toString() method of the Object class, we can return the values of the object, so we do not 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

160. Why toCharArray() method is preferred over String to store the password?

Using the toCharArray() method over String to store a password is preferred for security reasons. 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. If we store a password as a string, it remains in the memory until the garbage collector clears it. On the other hand, using the toCharArray() 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.

161. 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

162. Name some classes present in java.util.regex package.

There are the following classes and interfaces present in java.util.regex package.

- Pattern Class: It is an abstract class that represents a compiled regular expression. It provides no public constructors.
- Matcher Class: It is an abstract class that performs matching operations on a character sequence against given string. It also provides no public constructors.
- PatternSyntaxException Class: It is an unchecked exception. Indicates syntax errors in a regular expression pattern.

NEXT IS NOW

163. How the metacharacters are different from the ordinary characters?

Metacharacters in regular expressions have special meanings and are used to construct patterns. The metacharacters are ^, $, ., *, +, etc. The regular expression engine does not consider them as the regular characters. To enable the regular expression engine treating the metacharacters as ordinary characters, we need to escape the metacharacters with the backslash.

164. Write a regular expression to validate a password. A password must start with an alphabet and followed by alphanumeric characters. It must 8 to 20 characters long.

The regular expression for the above criteria will be:

^[a-zA-Z][a-zA-Z0-9]{8,19}

Where,

- ^ represents the start of the regex.
- [a-zA-Z] represents that the first character must be an alphabet.
- [a-zA-Z0-9] represents the alphanumeric character.
- {8,19} represents that the length of the password must be in between 8 and 20.

165. What is the output of the following Java program?

```
import java.util.regex.*;
class RegexExample2{
public static void main(String args[]){
System.out.println(Pattern.matches(".s", "as")); //line 4
System.out.println(Pattern.matches(".s", "mk")); //line 5
System.out.println(Pattern.matches(".s", "mst")); //line 6
System.out.println(Pattern.matches(".s", "amms")); //line 7
System.out.println(Pattern.matches("..s", "mas")); //line 8
}}
```

Output

true
false
false
false
true

Explanation

line 4 prints true since the second character of string is s, line 5 prints false since the second character is not s, line 6 prints false since there are more than 3 characters in the string, line 7 prints false since there are more than 2 characters in the string, and it

contains more than 2 characters as well, line 8 prints true since the third character of the string is s.

Core Java: Nested classes and Interfaces Interview Questions

166. What are the advantages of Java inner classes?

There are following advantages of Java inner classes:

- Encapsulation of logic within a class.
- Improved organization and readability of code.
- Enhanced code modularity.
- Access to the private members of the outer class.
- Code Optimization: It requires less code to write.
- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of the outer class including private.
- Nested classes are used to develop a more readable and maintainable code because it logically groups classes and interfaces in one place only.

167. What is a nested class?

The nested class can be defined as the class which is defined inside another class or interface. It can be a static nested class or an inner class (non-static nested class). We use the nested class to logically group classes and interfaces in one place so that it can be more readable and maintainable. A nested class can access all the data members of the outer class including private data members and methods. The syntax of the nested class is defined below.

```
class Java_Outer_class{
 //code
 class Java_Nested_class{
  //code
 }
}
```

There are two types of nested classes, static nested class, and non-static nested class. The non-static nested class can also be called as inner-class.

168. What are the disadvantages of using inner classes?

There are the following disadvantages of using inner classes:

- Increased complexity and reduced readability.
- Larger bytecode size.
- Tight coupling with the outer class.
- Inner classes increase the total number of classes used by the developer and therefore increases the workload of JVM since it has to perform some routine operations for those extra classes which result in slower performance.
- IDEs provide less support to the inner classes as compare to the top level classes and therefore it annoys the developers while working with inner classes.

169. What are the types of inner classes (non-static nested class) used in Java?

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. Its name is decided by the java compiler.
Local Inner class	A class created within the method.

170. Is there any difference between nested classes and inner classes?

Yes, inner classes are non-static nested classes. In other words, we can say that inner classes are the part of nested classes.

171. Can we access the non-final local variable, inside the local inner class?

No, in Java, a local inner class can access final or effectively final local variables. It cannot access non-final local variables.

172. How many class files will be created when we compile the outer class of the following program?

```
public class Person {
String name, age, address;
class Employee{
  float salary=10000;
}
class BusinessMen{
  final String gstin="£4433drt3$";
}
public static void main (String args[])
{
  Person p = new Person();
}
}
```

On compiling the above code, we get three class-files named as Person.class, Person$BusinessMen.class, and Person$Employee.class.

173. What are anonymous inner classes?

Anonymous inner classes in Java are unnamed classes defined on the fly, often used for one-time use. They are declared and instantiated at the same time. We cannot apply different access modifiers to them. Anonymous class cannot be static, and cannot define any static fields, method, or class. In other words, we can say that it a class without the name and can have only one object that is created by its definition. Consider the following example.

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
```

```
  p.eat();
 }
}
```

Output:

nice fruits

Consider the following example for the working of the anonymous class using interface.

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Output:

nice fruits

174. What is the nested interface?

An Interface that is declared inside the interface or class is known as the nested interface. A nested interface in Java is an interface declared within another interface or a class. It is static by default. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The external interface or class must refer to the nested interface. It cannot be accessed directly. The nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class. The syntax of the nested interface is given as follows.

```
interface interface_name{
```

```
...
interface nested_interface_name{
 ...
}
}
```

175. Can a class have an interface?

Yes, an interface can be defined within the class. It is called a nested interface. A class in Java can implement multiple interfaces.

176. Can an Interface have a class?

Yes, we can define a class inside an interface. The class can be either inner or nested class. In general, if the methods of the interface use this class and if we are not using it anywhere else we will declare a class within an interface. Interfaces may contain member type declarations.

A member type declaration in an interface is implicitly static and public. It is permitted to redundantly specify either or both of these modifiers. It is not possible to declare non-static classes inside a Java interface.

```
interface UserChecker {
  Ticket validateUser(Credentials credentials);
  class Credentials {
    // user and password
  }
  class Ticket {
    // some obscure implementation
  }
}
```

NEXT IS NOW

Garbage Collection Interview Questions

177. What is Garbage Collection?

Garbage Collection is the automatic process in Java that identifies and removes unused objects, freeing up memory and preventing memory leaks. 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 the free() function in the C language and delete() in C++. In Java, it is performed automatically. So, Java provides better memory management.

178. What is gc()?

The gc() method is not a standard method in Java. The garbage collection process is automatic, and we generally do not need to explicitly call a method to trigger garbage collection. The JVM handles garbage collection as part of its memory management. This function explicitly makes the Java Virtual Machine free up the space occupied by the unused objects so that it can be utilized or reused. Consider the following example for the better understanding of how the gc() method invoke the garbage collector.

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

Output:

object is garbage collected
object is garbage collected

179. How garbage collection is controlled?

Garbage collection is managed by Java Virtual Machine (JVM). It is performed when there is not enough space in the memory and memory is running low. We can externally call the System.gc() method for the garbage collection. Developers can influence garbage collection behavior by adjusting parameters such as heap size, garbage collection

algorithms, and tuning options. However, it depends upon the JVM whether to perform it or not.

180.How can an object be unreferenced?

An object becomes unreferenced when there are no more references to it from any part of the program. In Java, objects are automatically garbage collected when they are no longer reachable, meaning there are no references pointing to them. It can happen in several ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

181. 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 clean-up processing. The Garbage collector of JVM collects only those objects that are created by new keyword. However, it is important to note that the finalize() method is rarely used and is considered somewhat unreliable for resource cleanup. So if we have created an object without new, we can use the finalize method to perform clean-up processing (destroying remaining objects). The clean-up 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();
    }
}
```

182. Can an unreferenced object be referenced again?

Yes, once an object becomes unreferenced, there's no direct way to reference it again in Java. However, it is possible to indirectly reference an unreferenced object if it is reachable through other objects that are still referenced.

183. What kind of thread is the Garbage collector thread?

Daemon thread. The Garbage Collector thread in Java is a special thread responsible for reclaiming memory by removing objects that are no longer reachable. It runs in the background and is managed by the Java Virtual Machine (JVM).

184. What is the difference between final, finally and finalize?

final	finally	finalize
Final is used to apply restrictions on class, method, and variable. The final class cannot be inherited, final method cannot be overridden, and final variable value cannot 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.
Final is a keyword	Finally is a block.	Finalize is a method.

NEXT IS NOW

185. What is the purpose of the Runtime class?

Java Runtime class is used to interact with a Java Runtime Environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory, etc. It allows Java applications to interact with the environment in which they are running, such as executing system commands, managing process execution, accessing environment variables, etc. There is only one instance of java.lang.Runtime class is available for one Java application. The Runtime.getRuntime() method returns the singleton instance of Runtime class.

186. How will you invoke any external process in Java?

We can invoke an external process in Java using the ProcessBuilder or Runtime class. Here's an example using ProcessBuilder:

```
public class Runtime1{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("notepad");//will open a new notepad
 }
}
```

I/O Interview Questions

187. Give the hierarchy of InputStream and OutputStream classes.

OutputStream Hierarchy

InputStream Hierarchy

188. What do you understand by an IO stream?

An I/O stream in Java represents a sequence of data. It is used for input and output operations, allowing data to be read from or written to various sources such as files, network connections, memory, etc. In Java, the following three streams are created automatically.

- System.out: standard output stream
- System.in: standard input stream
- System.err: standard error stream

189. What is the difference between the Reader/Writer class hierarchy and the InputStream/OutputStream class hierarchy?

The Reader/Writer class hierarchy is character-oriented, and the InputStream/OutputStream class hierarchy is byte-oriented. The ByteStream classes are used to perform input-output of 8-bit bytes whereas the CharacterStream classes are used to perform the input/output for the 16-bit Unicode system. There are many classes in the ByteStream class hierarchy, but the most frequently used classes are FileInputStream and FileOutputStream. The most frequently used classes CharacterStream class hierarchy is FileReader and FileWriter.

The Reader/Writer class hierarchy in Java is used for character-based input/output operations. These classes deal with Unicode characters and are suitable for handling text data.	The InputStream/OutputStream class hierarchy in Java is used for byte-based input/output operations. These classes deal with raw bytes and are suitable for handling binary data.

190. What are the super most classes for all the streams?

All the stream classes can be divided into two types of classes that are ByteStream classes and CharacterStream Classes. The ByteStream classes are further divided into InputStream classes and OutputStream classes. CharacterStream classes are also divided into Reader classes and Writer classes. The super-most classes for all streams are InputStream and OutputStream. These classes define the basic functionality for reading from and writing to streams. Similarly, for all the reader classes, the super-most class is java.io.Reader, and for all the writer classes, it is java.io.Writer.

191. What are the FileInputStream and FileOutputStream?

Java FileOutputStream is an output stream used for writing data to a file. If you have some primitive values to write into a file, use FileOutputStream class. They are subclasses of InputStream and OutputStream, respectively. We can write byte-oriented as well as character-oriented data through the FileOutputStream class. However, for character-oriented data, it is preferred to use FileWriter than FileOutputStream. Consider the following example of writing a byte into a file.

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
  public static void main(String args[]){
      try{
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        fout.write(65);
        fout.close();
        System.out.println("success...");
      }catch(Exception e){System.out.println(e);}
    }
```

```
}
```

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video, etc. we can also read character-stream data. However, for reading streams of characters, it is recommended to use FileReader class. Consider the following example for reading bytes from a file.

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=fin.read();
          System.out.print((char)i);

          fin.close();
        }catch(Exception e){System.out.println(e);}
      }
     }
```

192. What is the purpose of using BufferedInputStream and BufferedOutputStream classes?

In Java, the purpose of using BufferedInputStream and BufferedOutputStream classes is to improve I/O performance by reducing the number of disk or network accesses. It internally uses a buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. Whereas, Java BufferedInputStream class is used to read information from the stream. It internally uses the buffer mechanism to make the performance fast.

193. How to set the Permissions to a file in Java?

In Java, FilePermission class is used to alter the permissions set on a file. Java FilePermission class contains the permission related to a directory or file. We can set permissions to a file using the setReadable(), setWritable(), and setExecutable() methods

of the java.io.File class. These methods allow us to set the read, write, and execute permissions, respectively, for the specified file. All the permissions are related to the path. The path can be of two types:

- D:\\IO\\-: It indicates that the permission is associated with all subdirectories and files recursively.
- D:\\IO*: It indicates that the permission is associated with all directory and files within this directory excluding subdirectories.

Let's see the simple example in which permission of a directory path is granted with read permission and a file of this directory is granted for write permission.

```
package com.Tpointtech;
import java.io.*;
import java.security.PermissionCollection;
public class FilePermissionExample{
    public static void main(String[] args) throws IOException {
     String srg = "D:\\IO Package\\java.txt";
     FilePermission file1 = new FilePermission("D:\\IO Package\\-", "read");
     PermissionCollection permission = file1.newPermissionCollection();
     permission.add(file1);
        FilePermission file2 = new FilePermission(srg, "write");
        permission.add(file2);
       if(permission.implies(new FilePermission(srg, "read,write"))) {
        System.out.println("Read, Write permission is granted for the path "+srg
);
         }else {
         System.out.println("No Read, Write permission is granted for the path
"+srg);          }
    }
}
```

Output:

Read, Write permission is granted for the path D:\IO Package\java.txt

194. What are FilterStreams?

FilterStream classes are used to add additional functionalities to the other stream classes. FilterStream classes act like an interface which read the data from a stream, filters it, and pass the filtered data to the caller. These are streams that "filter" data as it is being read from or written to an underlying stream. The FilterStream classes provide extra functionalities like adding line numbers to the destination file, etc.

195. What is an I/O filter?

An I/O filter is an object that reads from one stream and writes to another. Usually altering the data in some way as it is passed from one stream to another. Filters can be applied to both input and output streams to perform tasks such as encryption, compression, buffering, etc. Many Filter classes that allows a user to make a chain using multiple input streams. It generates a combined effect on several filters.

196. What are the ways to take input from the console?

In Java, there are three ways by using which, we can take input from the console.

- Using the Scanner class
- Using the BufferedReader class along with System.in
- Using the Console class

Let's brief each.

- Using BufferedReader Class: We can take input from the console by wrapping System.in into an InputStreamReader and passing it into the BufferedReader. It provides an efficient reading as the input gets buffered. Consider the following example.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Person
{
   public static void main(String[] args) throws IOException
   {
```

```
    System.out.println("Enter the name of the person");
      BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
      String name = reader.readLine();
      System.out.println(name);
   }
}
```

- Using Scanner Class: The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values. Java Scanner class is widely used to parse text for string and primitive types using a regular expression. Java Scanner class extends Object class and implements Iterator and Closeable interfaces. Consider the following example.

```
import java.util.*;
public class ScannerClassExample2 {
    public static void main(String args[]){
      String str = "Hello/This is Tpointtech/My name is Abhishek.";
      //Create scanner with the specified String Object
      Scanner scanner = new Scanner(str);
      System.out.println("Boolean Result: "+scanner.hasNextBoolean());
      //Change the delimiter of this scanner
      scanner.useDelimiter("/");
      //Printing the tokenized Strings
      System.out.println("---Tokenizes String---");
     while(scanner.hasNext()){
       System.out.println(scanner.next());
     }
      //Display the new delimiter
      System.out.println("Delimiter used: " +scanner.delimiter());
      scanner.close();
      }
}
```

- Using Console Class: The Java Console class is used to get input from the console. It provides methods to read texts and passwords. If we read the password using the Console class, it will not be displayed to the user. The java.io.Console class is attached to the system console internally. The Console class is introduced since 1.5. Consider the following example.

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

Serialization Interview Questions

197. What is serialization?

Serialization in Java is a mechanism of writing the state of an object into a byte stream. It is used primarily in Hibernate, RMI, JPA, EJB and JMS technologies. It is mainly used to travel object's state on the network (which is known as marshalling). Serializable interface is used to perform serialization. It is helpful when you require to save the state of a program to storage such as the file. At a later point of time, the content of this file can be restored using deserialization. Deserialization is the reverse process of converting the byte stream back into an object. It is also required to implement RMI (Remote Method Invocation). With the help of RMI, it is possible to invoke the method of a Java object on one machine to another machine.

198. How to make a class serializable in Java?

A class can become serializable by implementing the Serializable interface. This interface acts as a marker interface, indicating to the Java runtime that instances of the class can be serialized.

199. How to avoid serialization in child class if the base class is implementing the Serializable interface?

It is very tricky to prevent serialization of child class if the base class is intended to implement the Serializable interface. However, we cannot do it directly, but the serialization can be avoided by implementing the writeObject() or readObject() methods in the subclass and throw NotSerializableException from these methods. Consider the following example.

If a base class implements the Serializable interface and you want to avoid serialization in a child class, you can mark the fields of the child class as transient. his will prevent those fields from being serialized when objects of the child class are serialized, even though the base class implements Serializable.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.NotSerializableException;
```

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
class Person implements Serializable
{
   String name = " ";
   public Person(String name)
   {
      this.name = name;
   }
}
class Employee extends Person
{
   float salary;
   public Employee(String name, float salary)
   {
      super(name);
      this.salary = salary;
   }
   private void writeObject(ObjectOutputStream out) throws IOException
   {
      throw new NotSerializableException();
   }
   private void readObject(ObjectInputStream in) throws IOException
   {
      throw new NotSerializableException();
   }

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

```
        throws Exception
    {
        Employee emp = new Employee("Sharma", 10000);

        System.out.println("name = " + emp.name);
        System.out.println("salary = " + emp.salary);

        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(emp);

        oos.close();
        fos.close();

        System.out.println("Object has been serialized");

        FileInputStream f = new FileInputStream("ab.txt");
        ObjectInputStream o = new ObjectInputStream(f);

        Employee emp1 = (Employee)o.readObject();

        o.close();
        f.close();

        System.out.println("Object has been deserialized");

        System.out.println("name = " + emp1.name);
        System.out.println("salary = " + emp1.salary);
    }
}
```

200. Can a Serialized object be transferred via network?

Yes, a serialized object can be transferred via a network. Once an object is serialized into a stream of bytes and can be transmitted over the network. We can also write the serialized object to the disk or the database.

201. What is Deserialization?

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Deserialization involves reading the serialized data from a file, network stream, or any other source, and reconstructing the original object from that data. An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

```
import java.io.*;
class Depersist{
 public static void main(String args[])throws Exception{

  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name);

  in.close();
 }
}
```

Output:

211 ravi

202. What is the use of transient keyword?

In Java, the transient keyword is used to indicate that a field should not be serialized when the object is serialized. By determining transient keyword, the value of variable need not persist when it is restored.

203. What is Externalizable?

The Externalizable interface is used to write the state of an object into a byte stream in a compressed format. It allows more control over the serialization and deserialization

process compared to Serializable. Classes that implement Externalizable must provide implementations for the writeExternal() and readExternal() methods, which are called during serialization and deserialization, respectively. It is not a marker interface.

204. What is the difference between Serializable and Externalizable interface?

Serializable	Externalizable
The Serializable interface does not have any method, i.e., it is a marker interface.	The Externalizable interface contains is not a marker interface, It contains two methods, i.e., writeExternal() and readExternal()
It is used to "mark" Java classes so that objects of these classes may get the certain capability.	The Externalizable interface provides control of the serialization logic to the programmer.
It is easy to implement but has the higher performance cost.	It is used to perform the serialization and often result in better performance.
No class constructor is called in serialization.	We must call a public default constructor while using this interface.

Reflection Interview Questions

205. What is the reflection?

Reflection refers to the ability of a program to inspect and modify its own structure and behavior at runtime. The java.lang.Class class provides various methods that can be used to get metadata, examine and change the runtime behavior of a class. It allows us to analyze the capabilities of classes, interfaces, and objects, as well as dynamically invoke methods, access fields, and create new objects. The java.lang and java.lang.reflect packages provide classes for Java reflection. It is used in:

- IDE (Integrated Development Environment). Like, Eclipse, MyEclipse, NetBeans.

- Debugger
- Test Tools, etc.

206. What is the purpose of using java.lang.Class class?

The java.lang.Class class performs mainly two tasks:

- Provides methods to get the metadata of a class at runtime.
- Provides methods to examine and change the runtime behavior of a class.

207. What are the ways to instantiate the Class class?

There are three ways to instantiate the Class class.

- Using the .class syntax: Class<?> clazz = MyClass.class;
- Using the getClass() method: Class<?> clazz = obj.getClass();
- Using the forName() method: Class<?> clazz = Class.forName("fully.qualified.ClassName");

Using Class.forName() Method: The forName() method is used to load the class dynamically. It returns the instance of Class class. It should be used if we know the fully qualified name of the class. This cannot be used for primitive types.

Using Object.getClass() Method: It returns the instance of Class class. It should be used if we know the type. Moreover, it can be used with primitives.

Using .class Syntax: If a type is available, but there is no instance then it is possible to obtain a Class by appending ".class" to the name of the type. It can be used for primitive data type also.

208. What is the output of the following Java program?

```
class Simple{
 public Simple()
 {
   System.out.println("Constructor of Simple class is invoked");
 }
 void message(){System.out.println("Hello Java");}
```

```
}

class Test1{
 public static void main(String args[]){
  try{
  Class c=Class.forName("Simple");
  Simple s=(Simple)c.newInstance();
  s.message();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Output

Constructor of Simple class is invoked
Hello Java

209. What is the purpose of using javap?

The javap command disassembles a class file. It prints information about the fields, methods, and constructors of a class in human-readable format. javap can be useful for understanding the internal structure of Java classes and analyzing bytecode.

Syntax

```
javap fully_class_name
```

210. Can we access the private method from outside the class?

No, we cannot access a private method from outside the class in Java. Private methods are accessible only within the class in which they are declared. They cannot be accessed from subclasses or from other classes in the same package.

Miscellaneous Interview Questions

211. What are wrapper classes?

In Java, wrapper classes are the classes that allow primitive types to be accessed as objects. In other words, we can say that wrapper classes are built-in Java classes that allows the conversion of objects to primitives and primitives to objects. The process of

converting primitives to objects is called auto-boxing, and the process of converting objects to primitives is called unboxing. There are eight wrapper classes present in java.lang package is given below.

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

212. What are auto-boxing and unboxing? When does it occur?

The auto-boxing is the process of converting primitive data type to the corresponding wrapper class object. For example, int to Integer.

The unboxing is the process of converting wrapper class object to primitive data type. For example, Integer to int.

Unboxing and auto-boxing occur automatically in Java. However, we can externally convert one into another by using the methods like valueOf() or xxxValue().

- It can occur whenever a wrapper class object is expected, and primitive data type is provided or vice versa.
- Adding primitive types into Collection like ArrayList in Java.
- Creating an instance of parameterized classes. For example, ThreadLocal which expect Type.

- Java automatically converts primitive to object whenever one is required and another is provided in the method calling.
- When a primitive type is assigned to an object type.

213. What is the output of the following Java program?

```
public class Test1
{
  public static void main(String[] args) {
  Integer i = new Integer(201);
  Integer j = new Integer(201);
  if(i == j)
  {
    System.out.println("hello");
  }
  else
  {
    System.out.println("bye");
  }
  }
}
```

Output

bye

Explanation

The Integer class caches integer values from -127 to 127. Therefore, the Integer objects can only be created in the range -128 to 127. The operator == will not work for the value greater than 127; thus bye is printed.

214. What is object cloning?

Object cloning refers to the process of creating an 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. Object cloning

creates a new object with the same state as the original object, but the two objects are independent of each other. If we do not implement Cloneable interface, the clone() method generates CloneNotSupportedException. The clone() method is defined in the Object class. The syntax of the clone() method is as follows:

protected Object clone() throws CloneNotSupportedException

215. What are the advantages and disadvantages of object cloning?

Advantage of Object Cloning

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

Disadvantage of Object Cloning

- To use the Object.clone() method, we have to change many syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone(), etc.
- We have to implement the Cloneable interface while it does not have any methods in it. We have to use it to tell the JVM that we can perform a clone() on our object.
- clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- clone() does not invoke any constructor, so we do not have any control over object construction.
- If we want to write a clone method in a child class, then all of its super classes should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

- clone() method supports only shallow copying, but we will need to override it if we need deep cloning.

216. What is a native method?

A native method is a method that is implemented in a language other than Java. Natives methods are sometimes also referred to as foreign methods. Native methods are declared using the native keyword, and their implementations are provided by native libraries or external sources.

217. What is the purpose of the strictfp keyword?

Java strictfp keyword ensures that we will get the same result on every platform if we perform operations in the floating-point variable. The precision may differ from platform to platform that is why Java programming language has provided the strictfp keyword so that we get the same result on every platform. It ensures consistent behavior of floating-point operations across different JVM implementations. Therefore, we have better control over the floating-point arithmetic.

218. What is the purpose of the System class?

The purpose of the System class is to provide access to system resources such as standard input and output. It cannot be instantiated. Facilities provided by System class are given below.

- Standard input
- Error output streams
- Standard output
- utility method to copy the portion of an array
- utilities to load files and libraries

There are the three fields of Java System class, i.e., static printstream err, static inputstream in, and standard output stream.

219. What comes to mind when someone mentions a shallow copy in Java?

Object cloning. When someone mentions a shallow copy in Java, they are referring to a copy operation that creates a new object and copies the values of the fields from the original object to the new object.

220. What is a singleton class?

A singleton class in Java is a class that is restricted to having only one instance throughout the lifetime of the application. To make a class singleton, we either make its constructor private or use the static getInstance() method. Consider the following example.

```
class Singleton{
   private static Singleton single_instance = null;
   int i;
    private Singleton ()
    {
       i=90;
    }
    public static Singleton getInstance()
    {
       if(single_instance == null)
       {
          single_instance = new Singleton();
       }
       return single_instance;
    }
}
public class Main
{
   public static void main (String args[])
   {
      Singleton first = Singleton.getInstance();
      System.out.println("First instance integer value:"+first.i);
      first.i=first.i+90;
      Singleton second = Singleton.getInstance();
      System.out.println("Second instance integer value:"+second.i);
   }
}
```

221. Write a Java program that prints all the values given at command-line.

Program

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

for(int i=0;i<args.length;i++)
System.out.println(args[i]);

}
}
compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc
```

Output

```
sonoo
jaiswal
1
3
abc
```

Java Bean Interview Questions

222. What is a JavaBean?

JavaBean is a reusable software component written in the Java programming language, designed to be manipulated visually by a software development environment, like JBuilder or VisualAge for Java. JavaBeans follow a set of conventions, such as providing getter and setter methods for accessing properties, being serializable, and providing a no-argument constructor. A JavaBean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides the easy maintenance. Consider the following example to create a JavaBean class.

Employee.java

```
//Employee.java
package mypack;
```

```
public class Employee implements java.io.Serializable{
private int id;
private String name;
public Employee(){}
public void setId(int id){this.id=id;}
public int getId(){return id;}
public void setName(String name){this.name=name;}
public String getName(){return name;}
}
```

Core Java: Data Structure Interview Questions

223. How to perform Bubble Sort in Java?

Consider the following program to perform Bubble sort in Java.

```
public class BubbleSort {
 public static void main(String[] args) {
 int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
 for(int i=0;i<10;i++)
 {
  for (int j=0;j<10;j++)
  {
   if(a[i]<a[j])
   {
    int temp = a[i];
    a[i]=a[j];
    a[j] = temp;
   }
  }
 }
 System.out.println("Printing Sorted List ...");
 for(int i=0;i<10;i++)
 {
  System.out.println(a[i]);
```

```
    }
  }
}
```

Output:

```
Printing Sorted List . . .
7
9
10
12
23
34
34
44
78
101
```

224. How to perform Binary Search in Java?

Consider the following program to perform the binary search in Java.

```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
  int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
  int item, location = -1;
  System.out.println("Enter the item which you want to search");
  Scanner sc = new Scanner(System.in);
  item = sc.nextInt();
  location = binarySearch(arr,0,9,item);
  if(location != -1)
  System.out.println("the location of the item is "+location);
  else
    System.out.println("Item not found");
  }
```

```
public static int binarySearch(int[] a, int beg, int end, int item)
{
  int mid;
  if(end >= beg)
  {
    mid = (beg + end)/2;
    if(a[mid] == item)
    {
      return mid+1;
    }
    else if(a[mid] < item)
    {
      return binarySearch(a,mid+1,end,item);
    }
    else
    {
      return binarySearch(a,beg,mid-1,item);
    }
  }
  return -1;
}
}
```

Output:

Enter the item which you want to search
45
the location of the item is 5

225. How to perform Selection Sort in Java?

Consider the following program to perform selection sort in Java.

```
public class SelectionSort {
```

```
public static void main(String[] args) {
  int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
  int i,j,k,pos,temp;
  for(i=0;i<10;i++)
  {
    pos = smallest(a,10,i);
    temp = a[i];
    a[i]=a[pos];
    a[pos] = temp;
  }
  System.out.println("\nprinting sorted elements...\n");
  for(i=0;i<10;i++)
  {
    System.out.println(a[i]);
  }
}
public static int smallest(int a[], int n, int i)
{
  int small,pos,j;
  small = a[i];
  pos = i;
  for(j=i+1;j<10;j++)
  {
    if(a[j]<small)
    {
      small = a[j];
      pos=j;
    }
  }
  return pos;
}
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

226. How to perform Linear Search in Java?

Consider the following program to perform Linear search in Java.

```
import java.util.Scanner;
public class Leniear_Search {
public static void main(String[] args) {
 int[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
 int item,flag=0;
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter Item?");
 item = sc.nextInt();
 for(int i = 0; i<10; i++)
 {
  if(arr[i]==item)
  {
   flag = i+1;
   break;
  }
  else
   flag = 0;
 }
```

```
  if(flag != 0)
  {
    System.out.println("Item found at location" + flag);
  }
  else
    System.out.println("Item not found");
}
}
```

Output:

Enter Item?
23
Item found at location 2
Enter Item?
22
Item not found

227. How to perform merge sort in Java?

Consider the following program to perform merge sort in Java.

```
public class MyMergeSort
{
void merge(int arr[], int beg, int mid, int end)
{

int l = mid - beg + 1;
int r = end - mid;

intLeftArray[] = new int [l];
intRightArray[] = new int [r];

for (int i=0; i<l; ++i)
LeftArray[i] = arr[beg + i];
```

```
for (int j=0; j<r; ++j)
RightArray[j] = arr[mid + 1+ j];


int i = 0, j = 0;
int k = beg;
while (i<l&&j<r)
{
if (LeftArray[i] <= RightArray[j])
{
arr[k] = LeftArray[i];
i++;
}
else
{
arr[k] = RightArray[j];
j++;
}
k++;
}
while (i<l)
{
arr[k] = LeftArray[i];
i++;
k++;
}

while (j<r)
{
arr[k] = RightArray[j];
j++;
k++;
}
```

```
}

void sort(int arr[], int beg, int end)
{
if (beg<end)
{
int mid = (beg+end)/2;
sort(arr, beg, mid);
sort(arr , mid+1, end);
merge(arr, beg, mid, end);
}
}
public static void main(String args[])
{
intarr[] = {90,23,101,45,65,23,67,89,34,23};
MyMergeSort ob = new MyMergeSort();
ob.sort(arr, 0, arr.length-1);

System.out.println("\nSorted array");
for(int i =0; i<arr.length;i++)
{
  System.out.println(arr[i]+"");
}
}
}
```

Output:

Sorted array
23
23
23
34
45

65

67

89

90

101

228. How to perform quicksort in Java?

Consider the following program to perform quicksort in Java.

```
public class QuickSort {
public static void main(String[] args) {
   int i;
   int[] arr={90,23,101,45,65,23,67,89,34,23};
   quickSort(arr, 0, 9);
   System.out.println("\n The sorted array is: \n");
   for(i=0;i<10;i++)
   System.out.println(arr[i]);
 }
 public static int partition(int a[], int beg, int end)
 {

   int left, right, temp, loc, flag;
   loc = left = beg;
   right = end;
   flag = 0;
   while(flag != 1)
   {
     while((a[loc] <= a[right]) && (loc!=right))
     right--;
     if(loc==right)
     flag =1;
     elseif(a[loc]>a[right])
     {
```

```
      temp = a[loc];
      a[loc] = a[right];
      a[right] = temp;
      loc = right;
    }
    if(flag!=1)
    {
      while((a[loc] >= a[left]) && (loc!=left))
      left++;
      if(loc==left)
      flag =1;
      elseif(a[loc] <a[left])
      {
        temp = a[loc];
        a[loc] = a[left];
        a[left] = temp;
        loc = left;
      }
    }
  }
  returnloc;
}
static void quickSort(int a[], int beg, int end)
{

  int loc;
  if(beg<end)
  {
    loc = partition(a, beg, end);
    quickSort(a, beg, loc-1);
    quickSort(a, loc+1, end);
  }
}
```

```
}
```

Output:

The sorted array is:
23
23
23
34
45
65
67
89
90
101

229. Write a program in Java to create a doubly linked list containing n nodes.

Consider the following program to create a doubly linked list containing n nodes.

```
public class CountList {

  //Represent a node of the doubly linked list

  class Node{
    int data;
    Node previous;
    Node next;

    public Node(int data) {
      this.data = data;
    }
  }

  //Represent the head and tail of the doubly linked list
  Node head, tail = null;
```

```
//addNode() will add a node to the list
public void addNode(int data) {
    //Create a new node
    Node newNode = new Node(data);

    //If list is empty
    if(head == null) {
        //Both head and tail will point to newNode
        head = tail = newNode;
        //head's previous will point to null
        head.previous = null;
        //tail's next will point to null, as it is the last node of the list
        tail.next = null;
    }
    else {
        //newNode will be added after tail such that tail's next will point to
newNode
        tail.next = newNode;
        //newNode's previous will point to tail
        newNode.previous = tail;
        //newNode will become new tail
        tail = newNode;
        //As it is last node, tail's next will point to null
        tail.next = null;
    }
}

//countNodes() will count the nodes present in the list
public int countNodes() {
    int counter = 0;
    //Node current will point to head
    Node current = head;
```

```
    while(current != null) {
        //Increment the counter by 1 for each node
        counter++;
        current = current.next;
    }
    return counter;
}

//display() will print out the elements of the list
public void display() {
    //Node current will point to head
    Node current = head;
    if(head == null) {
        System.out.println("List is empty");
        return;
    }
    System.out.println("Nodes of doubly linked list: ");
    while(current != null) {
        //Prints each node by incrementing the pointer.

        System.out.print(current.data + " ");
        current = current.next;
    }
}

public static void main(String[] args) {

    CountList dList = new CountList();
    //Add nodes to the list
    dList.addNode(1);
    dList.addNode(2);
    dList.addNode(3);
    dList.addNode(4);
```

```
        dList.addNode(5);

        //Displays the nodes present in the list
        dList.display();

        //Counts the nodes present in the given list
        System.out.println("\nCount of nodes present in the list: " +
dList.countNodes());
    }
}
```

Output:

Nodes of doubly linked list:
1 2 3 4 5
Count of nodes present in the list: 5

230. Write a program in Java to find the maximum and minimum value node from a circular linked list.

Consider the following program.

```
public class MinMax {
    //Represents the node of list.
    public class Node{
        int data;
        Node next;
        public Node(int data) {
            this.data = data;
        }
    }

    //Declaring head and tail pointer as null.
    public Node head = null;
    public Node tail = null;

    //This function will add the new node at the end of the list.
```

```
public void add(int data){
    //Create new node
    Node newNode = new Node(data);
    //Checks if the list is empty.
    if(head == null) {
        //If list is empty, both head and tail would point to new node.
        head = newNode;
        tail = newNode;
        newNode.next = head;
    }
    else {
        //tail will point to new node.
        tail.next = newNode;
        //New node will become new tail.
        tail = newNode;
        //Since, it is circular linked list tail will points to head.
        tail.next = head;
    }
}

//Finds out the minimum value node in the list
public void minNode() {
    Node current = head;
    //Initializing min to initial node data
    int min = head.data;
    if(head == null) {
        System.out.println("List is empty");
    }
    else {
        do{
            //If current node's data is smaller than min
            //Then replace value of min with current node's data
            if(min > current.data) {
```

```
                min = current.data;
            }
            current= current.next;
        }while(current != head);

        System.out.println("Minimum value node in the list: "+ min);
    }
}

//Finds out the maximum value node in the list
public void maxNode() {
    Node current = head;
    //Initializing max to initial node data
    int max = head.data;
    if(head == null) {
        System.out.println("List is empty");
    }
    else {
        do{
            //If current node's data is greater than max
            //Then replace value of max with current node's data
            if(max < current.data) {
                max = current.data;
            }
            current= current.next;
        }while(current != head);

        System.out.println("Maximum value node in the list: "+ max);
    }
}

public static void main(String[] args) {
    MinMax cl = new MinMax();
```

```
            //Adds data to the list
            cl.add(5);
            cl.add(20);
            cl.add(10);
            cl.add(1);
            //Prints the minimum value node in the list
            cl.minNode();
            //Prints the maximum value node in the list
            cl.maxNode();
        }
    }
```

Output:

```
Minimum value node in the list: 1
Maximum value node in the list: 20
```

Java Collections Interview Questions and Answers

231. What is the Collection framework in Java?

A Collection Framework is a combination of classes and interfaces that is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose. This framework offers an efficient and flexible method of handling data structures by providing a standardised approach to working with collections of items.

232.What are the main differences between array and collection?

Array and Collection are somewhat similar regarding storing the references of objects and manipulating the data, but they differ in many ways.

Aspect	Array	Collection
Size	Fixed size	Dynamically (resizable)
Type of elements	Homogeneous (same type)	Heterogeneous (different types)
Methods	Limited methods available	Abundance of methods for manipulation and iteration
Flexibility	Limited flexibility	High flexibility for manipulation and expansion
Usage	Basic data storage	Advanced data manipulation and storage

233. Explain various interfaces used in Collection framework?

Collection framework implements various interfaces, Collection interface and Map interface (java.util.Map) are the mainly used interfaces of Java Collection Framework. List of interfaces of Collection Framework is given below:

1. Collection interface:

This is the collection hierarchy's base interface. Its elements are a collection of items that it represents. Duplicate elements may be found in collections, and elemental orders may change. Commonly used classes like ArrayList, LinkedList, and HashSet are examples of implementations of this interface.

Syntax:

```
public interface Collection<E>extends Iterable
```

Where <E> represents that this interface is of Generic type

2. List interface:

The list extends the Collection interface and represents an ordered collection of elements. It allows duplicate elements and provides methods to access elements by their index. Implementations of the List interface include ArrayList, LinkedList, and Vector.

Syntax:

```
public interface List<E> extends Collection<E>
```

3. Set interface:

A collection that is incapable of having duplicate elements is represented by a set, which expands on the Collection interface. It does not ensure the components' order; instead, it simulates the abstraction of a mathematical set. HashSet, TreeSet, and LinkedHashSet implement the Set interface.

Syntax:

```
public interface Set<E> extends Collection<E>
```

4. Queue interface:

A queue is a collection intended to keep things before they are processed, and it extends the Collection interface. Its elements are arranged in FIFO (First-In-First-Out) order. The Queue interface is implemented by LinkedList, PriorityQueue, and ArrayDeque.

Syntax:

```
public interface Queue<E> extends Collection<E>
```

5. Deque Interface

Deque stands for "Double Ended Queue" and extends the Queue interface. It represents a linear collection that supports element insertion and removal at both ends. Deques can function as standard queues (FIFO) as well as stacks (LIFO). Implementations of the Deque interface include LinkedList and ArrayDeque.

Syntax:

```
public interface Dequeue<E> extends Queue<E>
```

5. Map interface:

The Map interface does not extend the Collection interface, in contrast to other collection interfaces. It shows a mapping of keys to values in which a single value is connected to each key. HashMap, TreeMap, and LinkedHashMap are examples of map implementations. Furthermore, a map that retains its keys in ascending order is provided by the SortedMap interface, which extends Map.

Syntax

```
public interface Map<K, V> {}
```

234. What are the difference between ArrayList and Vector?

Aspect	ArrayList	Vector
Synchronization	ArrayList is not synchronized	Vector is synchronized.
Legacy	ArrayList is not a legacy class.	Vector is a legacy class.
Resize Strategy	ArrayList increases its size by 50% of the array size.	Vector increases its size by doubling the array size.
Thread Safety	ArrayList is not ?thread-safe? as it is not synchronized	Vector list is ?thread-safe? as it?s every method is synchronized.
Performance	Generally faster due to lack of synchronization overhead.	Slower due to synchronization overhead.
Iterator	Fail-fast iterator (Throws ConcurrentModificationException if modified during iteration).	Fail-safe iterator (Doesn't throw ConcurrentModificationException if modified during iteration).

Usage	Preferred in non-concurrent environments.	Used in legacy or concurrent environments.

235. What are the difference between ArrayList and LinkedList?

Aspect	**ArrayList**	**LinkedList**
Data Structure	ArrayList uses a dynamic array.	LinkedList uses a doubly linked list.
Efficiency	ArrayList is not efficient for manipulation because too much is required.	LinkedList is efficient for manipulation.
Data Storage	ArrayList is better to store and fetch data.	LinkedList is better to manipulate data.
Access	ArrayList provides random access.	LinkedList does not provide random access.
Memory Overhead	ArrayList takes less memory overhead as it stores only object	LinkedList takes more memory overhead, as it stores the object as well as the address of that object.

236. What are the differences between Iterator and ListIterator?

Iterator traverses the elements in the forward direction only whereas ListIterator traverses the elements into forward and backward direction.

Aspect	Iterator	ListIterator
Traversal Direction	The Iterator traverses the elements in the forward direction only.	ListIterator traverses the elements in backward and forward directions both.
Applicable Collections	The Iterator can be used in List, Set, and Queue.	ListIterator can be used in List only.
Supported Operations	The Iterator can only perform remove operation while traversing the collection.	ListIterator can perform add, remove, and set operation while traversing the collection
Current Position Management	It does not maintain a current position pointer	It maintains a current position pointer.
Modification of Collection	It cannot add or modify elements during traversal.	It can add, remove, or modify elements during traversal.
Implementation Classes	java.util.Iterator	java.util.ListIterator

237. What are the differences between Iterator and Enumeration?

Aspect	Iterator	Enumeration
Traversal Capability	The Iterator can traverse legacy and non-legacy elements.	Enumeration can traverse only legacy elements.
Fail-Fast Behavior	The Iterator is fail-fast.	Enumeration is not fail-fast.
Speed	The Iterator is slower than Enumeration.	Enumeration is faster than Iterator.

Modification of Collection	The Iterator can perform remove operation while traversing the collection.	The Enumeration can perform only traverse operation on the collection.
Current Position Management	It maintains a current position pointer	It does not maintain a current position pointer
Support for Collection Types	List, Set, Queue, Map	Mainly supports legacy collection classes like Vector, Hashtable, and Properties.
Concurrent	Modification Detects concurrent modification.	No built-in mechanism for detecting modifications.
Methods	It has additional methods like hasNext(), next().	It has methods like hasMoreElements(), nextElement().

238. What are the differences between List and Set?

Comparison Point	**List**	**Set**
Duplication of Elements	It can contain duplicate elements.	It includes only unique items.
Ordering of Elements	It maintains insertion order.	It does not preserve insertion order.
Legacy Classes	It contains legacy class Vector.	It does not have any legacy class.
Null Values	It can allow multiple null values.	It allows a single null value.

Subinterfaces	It includes ListIterator, ListIterator.	It includes SortedSet, NavigableSet.
Example Classes	ArrayList, LinkedList, Vector.	HashSet, TreeSet.

239. What are the differences between HashSet and TreeSet?

Aspect	**Hashset**	**Treeset**
Ordering of Elements	Maintains no specific order	Maintains elements in ascending order
Implementation Structure	Implemented using a hash table	Implemented using a tree structure
Performance	Generally faster due to constant-time operations	Slightly slower due to log(n) time operations
Backing Data Structure	Backed by a HashMap	Backed by a TreeMap
Ordering of Elements	Unordered	Ordered

240. What are the differences between Set and Map?

Aspect	**Set**	**Map**
Eelements	It contains only values.	It contains key-value pairs.
Uniqueness of Elements	It contains unique values.	It can contain unique keys with duplicate values.
Handling of Null Values	It can hold a single null value	It can have a single null key with multiple null values.

Accessing Elements	It is accessed by iteration or specific methods.	It is accessed by keys.
Examples	HashSet, TreeSet, LinkedHashSet	HashMap, TreeMap, LinkedHashMap

241. What are the differences between HashSet and HashMap?

Aspect	HashSet	HashMap
Elements	It contains only values.	It contains key-value pairs.
Interface Implementation	It implements Set interface.	It implements Map interface.
Iteration	It can be directly iterated	It needs to be converted into Set for iteration
Uniqueness of Elements	It cannot have duplicate values.	It can contain duplicate values with unique keys.
Handling of Null Values	It can hold a single null value.	It can have a single null key with multiple null values.
Internal Data Structure	It implements using a hash table.	It implements using an array of buckets (hash table and linked list).

242. What are the differences between HashMap and TreeMap?

Aspect	HashMap	TreeMap
Ordering of Elements	There is no specific order.	It maintains elements in ascending order
Implementation Structure	It implements using a hash table.	It implements using a tree structure
Sorting Capability	It can be sorted by key or value.	It can be sorted by key only.
Null Key Handling	It can contain a null key with multiple null values.	It cannot have a null key but can have multiple null values.
Internal Data Structure	It uses hash table for storage and retrieval.	It uses a red-black tree for storage and retrieval.

NEXT IS NOW

243. What are the differences between HashMap and Hashtable?

Aspect	HashMap	Hashtable
Synchronization	It is not synchronized.	It is synchronized.

Null Key and Value Handling	It can have one null key and multiple null values.	It cannot have null keys or values.
Thread Safety	It is not thread-safe.	It is Tthread-safe.
Inheritance	It inherits AbstractMap class.	It inherits Dictionary class.
Performance	It generally offers better performance.	It may have performance overhead due to synchronization.
Iteration Behavior	Order of iteration may not be predictable	Iteration occurs in the order of insertion
Iterator Fail-Safety	It is fail-fast iterator.	It is fail-safe iterator.

244. What are the differences between Collection and Collections?

Aspect	Collection	Collections
Type	It is an interface.	It is a class.
Purpose	It defines a standard functionality for data structures like List, Set, and Queue.	It provides utility methods for sorting and synchronizing collections.

Methods	It defines methods for data structure manipulation.	It provides static methods for various collection operations.
Example Implementations	ArrayList, LinkedList, HashSet	Collections.sort(), Collections.synchronized List()
Nullability	It can contain null elements.	It does not have any direct impact on nullability.
Extensibility	It can be implemented by other interfaces and classes.	It cannot be extended or implemented.

245. What are the differences between Comparable and Comparator?

Aspect	Comparable	Comparator
Sort Sequences	It provides one sort sequence.	It provides multiple sort sequences.
Method	It requires implementing compareTo() method.	It requires implementing compare() method.

Location	It belongs to java.lang package.	It belongs to java.util package.
Class Change	It modifies the actual class.	It does not modify the actual class.
Flexibility	It is less flexible as it's tied to the object's class	It is more flexible, allows sorting based on different criteria

247. What do you understand by BlockingQueue?

BlockingQueue is an interface that extends the Queue interface. It provides concurrency in operations like retrieval, insertion, and deletion. When retrieving an element, it waits for the queue to be non-empty. When storing elements, it waits for available space. BlockingQueue cannot contain null elements, and its implementation is thread-safe.

It's crucial to remember that null components are not allowed in BlockingQueue implementations. These implementations are also thread-safe, which makes them appropriate in situations where several threads must concurrently access and change a shared queue. In concurrent programming, BlockingQueue is frequently used to manage data interchange between producer and consumer threads in a thread-safe way.

Syntax:

```
public interface BlockingQueue<E> extends Queue <E>
```

248. What is the advantage of Properties file?

If we change the value in the properties file, we do not need to recompile the Java class. So, it makes the application easy to manage. It is used to store information that is to be changed frequently. Consider the following example.

Test.java

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{
    FileReader reader=new FileReader("db.properties");
    Properties p=new Properties();
    p.load(reader);
    System.out.println(p.getProperty("user"));
    System.out.println(p.getProperty("password"));
}
}
```

Output

Tpointtech
12345

Explanation

The snippet of code that is provided loads the key-value pairs that indicate configuration settings from the db.properties file using the Properties class. The values linked to certain keys ("user" and "password") are then retrieved from the properties file using the getProperty() method, and they are subsequently shown on the console. By separating the configuration data from the application logic, this method facilitates flexibility and manageability.

db.properties

```
user= Tpointtech
password=12345
```

249. What does the hashCode() method?

Java's hashCode() function yields the object's hash code value. Many data structures, including HashMap, HashSet, and others, employ this value to organise and store things efficiently. The hashCode() method's main goal is to give an object's state a distinct integer representation.

1. Separate objects can share the same hash code, even if the hashCode() method seeks to generate unique hash codes for each object.
2. Two objects may not always generate the same hash code even if their equals() function indicates that they are equal. For consistency's sake, two objects should share the same hash code if the equals() method determines that they are equal.
3. The equals () function and the hashCode () technique are usually used together. To preserve consistency, it is advised that classes that override the equals() method also override the hashCode() function. This guarantees that items of the same kind have the same hash code, which is necessary for some data structures to function effectively.
4. Java's Object class provides the default implementation of the hashCode() method, that produces hash codes depending on the object's memory location. To offer a more meaningful hash code based on the object's state, user-defined classes typically override this function.

250. Why we override equals() method?

The equals method is used to check whether two objects are the same or not. It needs to be overridden if we want to check the objects based on the property.

For example, Employee is a class that has 3 data members: id, name, and salary. However, we want to check the equality of employee object by the salary. Then, we need to override the equals() method.

```
@Override
public boolean equals(Object obj) {
    // Check if the object being compared is the same instance
    if (this == obj) {
        return true;
    }
    // Check if the object being compared is null or of a different class
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    // Convert the object to the same class type
```

```
        Employee other = (Employee) obj;

        // Compare the properties for equality
        return this.salary == other.salary; // Example: Compare based on salary
    }
```

251. How to synchronize List, Set, and Map elements?

Yes, Collections class provides methods to make List, Set or Map elements as synchronized:

```
public static List synchronizedList(List l)
public static Set synchronizedSet(Set s)
public static SortedSet synchronizedSortedSet(SortedSet s)
public static Map synchronizedMap(Map m)
public static SortedMap synchronizedSortedMap(SortedMap m)
```

These methods provide thread-safe collection operations by enclosing the supplied collection (List, Set, or Map) in a synchronised wrapper. This avoids dangerous race circumstances and ensures data integrity, especially in situations where multiple threads may access or modify the collection concurrently.

252. What are the advantages of the generic collection?

Generic collections in Java provide several advantages over non-generic collections:

1. Elimination of Type Casting

A major benefit of utilising generic collections is not requiring explicit type casting. When retrieving elements from non-generic collections, we must cast them to the correct data type; if this is done incorrectly, runtime issues may result. By using generics, we can work directly with the proper data type without casting because the compiler guarantees type safety at compilation time.

2. Type Safety

The compiler guarantees that only the proper data types are utilised within the collection by using generic collections, that offer type safety. By doing this, runtime issues caused by

mismatched data types are avoided. When constructing a generic collection, we can reduce the possibility of errors resulting from erroneous data types by clearly defining the types of components that the collection can include by using the type argument.

3. Compile-Time Checking:

Compile-time checking is an additional benefit of generic collections. Type checks and problems relating to types can be found by the compiler prior to code execution because the type information is available at compile time. This enhances code stability and reliability by spotting any flaws early in the development process. Coders can lessen the chance of runtime mistakes and strengthen their code by proactively addressing errors found during compilation.

253. What are the differences between Array and ArrayList?

The main differences between the Array and ArrayList are given below.

Aspect	Array	ArrayList
Size flexibility	Arrays are of fixed size.	ArrayList is dynamic in size.
Type	It is static type.	It is dynamic type.
Data types	It can store primitive types and objects.	It can only store objects, not primitives.
Declaration syntax	It uses square brackets (for example, int[] arr).	It uses <> for declaration (for example, ArrayListlist).

Framework affiliation	It is not a part of Java Collections Framework.	It is a part of Java Collections Framework.
Access methods	Direct access using indices.	Access through methods like get() and set().
Multi-dimensionality	It supports multi-dimensional arrays.	It is inherently one-dimensional.

254. What are the differences between the length of an Array and the size of an ArrayList?

The length of an array can be obtained using the property of length whereas ArrayList does not support length property, but we can use size() method to get the number of objects in the list.

Finding the length of the array

```
Int [] array = new int[4];
System.out.println("The size of the array is " + array.length);
```

Finding the size of the ArrayList

```
ArrayList<String> list=new ArrayList<String>();
list.add("ankit");
list.add("nippun");
System.out.println(list.size());
```

NEXT IS NOW

255. How to convert ArrayList to Array and Array to ArrayList?

We can convert an Array to ArrayList by using the asList() method of Arrays class. asList() method is the static method of Arrays class and accepts the List object. Consider the following syntax:

```
Arrays.asList(item)
```

We can convert an ArrayList to Array using toArray() method of the ArrayList class. Consider the following syntax to convert the ArrayList to the List object.

```
List_object.toArray(new String[List_object.size()])
```

256. How to make Java ArrayList Read-Only?

We can obtain a Java ArrayList that is Read-only by calling the Collections.unmodifiableCollection() method. When we define an ArrayList as Read-only, we cannot modify the collection through the add(), remove(), or set() methods.

ReadOnlyArrayList.java

```
import java.util.*;
public class ReadOnlyArrayList {
    public static void main(String[] args) {
        // Create a normal ArrayList
        ArrayList<String> originalList = new ArrayList<>();
        originalList.add("Apple");
        originalList.add("Banana");
        originalList.add("Orange");
        // Make the ArrayList read-only
        List<String> readOnlyList = Collections.unmodifiableList(originalList);
        // Try to modify the read-only list (will throw
UnsupportedOperationException)
        try {
            readOnlyList.add("Grape"); // Trying to add an element
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot add elements to the read-only list.");
        }
        // Try to modify the original list (will be reflected in the read-only list)
        originalList.add("Grape");
        // Display the read-only list
        System.out.println("Read-only list: " + readOnlyList);
    }
}
```

Output:

Cannot add elements to the read-only list.
Read-only list: [Apple, Banana, Orange, Grape]

257. How to remove duplicates from ArrayList?

There are two ways to remove duplicates from the ArrayList.

- Using HashSet: By using HashSet we can remove the duplicate element from the ArrayList, but it will not then preserve the insertion order.
- Using LinkedHashSet: We can also maintain the insertion order by using LinkedHashSet instead of HashSet.

The Process to remove duplicate elements from ArrayList using the LinkedHashSet:

- Copy all the elements of ArrayList to LinkedHashSet.
- Empty the ArrayList using clear() method, which will remove all the elements from the list.
- Now copy all the elements of LinkedHashset to ArrayList.

257. How to reverse ArrayList?

To reverse an ArrayList, we can use reverse() method of Collections class. Consider the following example.

ReverseArrayList.java

NEXT IS NOW

```
import java.util.*;
public class ReverseArrayList {
public static void main(String[] args) {
    List list = new ArrayList<>();
    list.add(10);
    list.add(50);
    list.add(30);
    Iterator i = list.iterator();
    System.out.println("printing the list....");
```

```
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
        Iterator i2 = list.iterator();
        Collections.reverse(list);
        System.out.println("printing list in reverse order....");
        while(i2.hasNext())
        {
            System.out.println(i2.next());
        }
       }
    }
```

Output

printing the list....
10
50
30
printing list in reverse order....
30
50
10

258. How to sort ArrayList in descending order?

To sort the ArrayList in descending order, we can use the reverseOrder() method of Collections class. Consider the following example.

ReverseArrayList.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
```

```
import java.util.Iterator;
import java.util.List;

public class ReverseArrayList {
public static void main(String[] args) {
    List list = new ArrayList<>();
    list.add(10);
    list.add(50);
    list.add(30);
    list.add(60);
    list.add(20);
    list.add(90);

    Iterator i = list.iterator();
    System.out.println("printing the list....");
    while(i.hasNext())
    {
        System.out.println(i.next());
    }

   Comparator cmp = Collections.reverseOrder();
   Collections.sort(list,cmp);
    System.out.println("printing list in descending order....");
    Iterator i2 = list.iterator();
    while(i2.hasNext())
    {
        System.out.println(i2.next());
    }

}
}
```

Output

printing the list....

10

50

30

60

20

90

printing list in descending order....

90

60

50

30

20

10

259. How to synchronize ArrayList?

We can synchronize ArrayList in two ways.

1. Using Collections.synchronizedList() Method

With this method, the Collections class's synchronisedList() methods are wrapped around your ArrayList. It makes the list thread-safe by guaranteeing that only one thread can access it at a time.

2. Using CopyOnWriteArrayList<T>

This is a customised version of Java's built-in synchronization-supporting List interface. Every time an action is done that modifies the underlying array, a new copy of it is made. This eliminates the requirement for explicit synchronisation and permits safe concurrent access to the list.

260. When to use ArrayList and LinkedList?

LinkedLists are better to use for the update operations whereas ArrayLists are better to use for the search operations. While it's true that ArrayLists and LinkedLists have different performance characteristics that make them more suitable for certain operations, the

decision on when to use one over the other should consider various factors beyond just update and search operations.

ArrayList:

- Use ArrayList when you need fast access to elements by index, as it provides constant-time access to elements based on their position.
- ArrayList is efficient for read-heavy scenarios where elements are frequently accessed but less frequently modified.
- It's also suitable when memory overhead is a concern, as ArrayLists have a smaller memory footprint compared to LinkedLists due to their simple array-based structure.

LinkedList:

- Use LinkedList when you require frequent insertions or deletions at arbitrary positions within the list, as it provides constant-time insertion and deletion operations.
- LinkedList performs better than ArrayList for frequent modifications, especially when inserting or removing elements from the middle of the list, as it doesn't require shifting of elements.
- It's suitable for scenarios where you need to iterate over the entire list and perform modifications, as LinkedList's iterator performs better in such cases compared to ArrayList's iterator.

NEXT IS NOW

Java 8 Interview Questions And Answers

261. After Java 8, what do you think about Java?

Is it still an object oriented language or it has turned into functional programming language?

Java is still an object oriented language where everything is done keeping objects (data) in mind. But, with the introduction of new features in Java 8, you can use Java as a functional programming language also. You can treat it as as an added

advantage over the other languages which are either object oriented or functions oriented. From Java 8, you can use Java either in an object-oriented programming paradigm or in a functional programming paradigm. It supports both.

262. What are the three main features of Java 8 which make Java as a functional programming language?

Lambda expressions, functional interfaces and Stream API are the three main features of Java 8 which enables developers to write functional style of programming in Java also.

263. What are lambda expressions? How this feature has changed the way you write code in Java? Explain with some before Java 8 and after Java 8 examples?

Lambda Expressions can be defined as methods without names i.e anonymous functions. Like methods, they also have parameters, a body, a return type and possible list of exceptions that can be thrown. But unlike methods, neither they have names nor they are associated with any particular class.

Lambda expressions are used where an instance of functional interface is expected. Before Java 8, anonymous inner classes are used for this purpose. After Java 8, you can use lambda expressions to implement functional interfaces.

These lambda expressions have changed the style of programming in Java significantly. They have made the Java code more clear, concise and readable than before. For example,

Below code shows how Comparator interface is implemented using anonymous inner class before Java 8.

```
Comparator<Student> idComparator = new Comparator<Student>() {
```

```
        @Override
        public int compare(Student s1, Student s2) {
            return s1.getID()-s2.getID();
        }
    };
```

and after Java 8, above code can be written in a single line using Java 8 lambda expressions as below.

```
Comparator<Student> idComparator = (Student s1, Student s2) -> s1.getID()-s2.getID();
```

Another example,

Implementation of Runnable interface using anonymous inner class before Java 8 :

```
Runnable r = new Runnable() {
        @Override
        public void run() {
```

```
            System.out.println("Runnable Implementation Using Anonymous Inner Class");

        }

    };
```

Implementation of *Runnable* interface using lambda expressions after Java 8 :

```
Runnable r = () -> System.out.println("Runnable Implementation Using Lambda Expressions");
```

264. How the signature of lambda expressions are determined?

The signature of lambda expressions are derived from the signature of abstract method of functional interface. For example,

run() method of Runnable interface accepts nothing and returns nothing. Then signature of lambda expression implementing Runnable interface will be () -> void.

compare() method of Comparator interface takes two arguments of type Object and returns int. Then signature of lambda expression for implementing Comparator interface will be (Object, Object) -> int.

265. How the compiler determines the return type of a lambda expression?

Compiler uses target type to check the return type of a lambda expression.

For example,

```
Runnable r = () -> System.out.println("Runnable Implementation Using Lambda Expressions");
```

In this example, target type of lambda expression is Runnable. Compiler uses run() method of Runnable interface to check the return type of lambda expression.

266. Can we use non-final local variables inside a lambda expression?

No. Only final local variables are allowed to use inside a lambda expressions just like anonymous inner classes.

267. What are the advantages of lambda expressions?

- Lambda expressions let you to write more clear, concise and readable code.
- Lambda expressions removes verbosity and repetition of code.

268. What are the functional interfaces? Do they exist before Java 8 or they are the whole new features introduced in Java 8?

Functional interfaces are the interfaces which has exactly one abstract method. Functional interfaces provide only one functionality to implement.

There were functional interfaces exist before Java 8. It is not like that they are the whole new concept introduced only in Java 8. Runnable, ActionListener, Callable and Comaprator are some old functional interfaces which exist even before Java 8.

The new set of functional interfaces are introduced in Java 8 for writing lambda expressions. Lambda expressions must implement any one of these new functional interfaces.

269. What are the new functional interfaces introduced in Java 8? In which package they have kept in?

Below is the list of new functional interfaces introduced in Java 8. They have kept in java.util.function package.

ParvaM Consul-tech Pvt. Ltd.

Bengaluru-560090

Functional Interface And Its Abstract Method	Operation It Represents.	When To Use?	Related Functional Interfaces To Support Primitive Types
Predicate boolean test(T t)	Represents an operation which takes one argument and returns boolean.	Use this interface when you want to evaluate a boolean expression which takes an argument of type T.	IntPredicate LongPredicate DoublePredicate
Consumer void accept(T t)	Represents an operation that accepts single argument and returns nothing.	Use this interface when you want to perform some operations on an object.	IntConsumer LongConsumer DoubleConsumer
Function R apply(T t)	Represents an operation that accepts an argument of type T and returns a result of type R.	Use this interface when you want to extract a data from an existing data.	IntFunction LongFunction DoubleFunction ToIntFunction ToLongFunction ToDoubleFunction IntToLongFunction IntToDoubleFunction LongToDoubleFunction LongToIntFunction DoubleToIntFunction DoubleToLongFunction
Supplier T get()	Represents an operation which takes nothing but returns a result of type T.	Use this interface when you want to create new objects.	BooleanSupplier IntSupplier LongSupplier DoubleSupplier
BiPredicate boolean test(T t, U u)	Represents a predicate of two arguments.	Use this interface when you want to evaluate a boolean expression of two arguments.	
BiConsumer void accept(T t, U u)	Represents an operation that accepts two arguments and returns nothing.	Use this interface when you want to perform some operations on two objects.	ObjIntConsumer ObjLongConsumer ObjDoubleConsumer
BiFunction R apply(T t, U u)	Represents an operation which takes two arguments and produces a result.	Use this interface when you want to extract result data from two existing objects.	ToIntBiFunction ToLongBiFunction ToDoubleBiFunction
UnaryOperator (extends Function)	Same as Function but argument and result should be of same type.	Same as Function.	IntUnaryOperator LongUnaryOperatot DoubleUnaryOperator
BinaryOperator (extends BiFunction)	Same as BiFunction but argument and result should be of same type.	Same as BiFunction.	IntBinaryOperator LongBinaryOperator DoubleBinaryOperator

270. What is the difference between Predicate and BiPredicate?

Predicate is a functional interface which represents a boolean operation which takes one argument.

BiPredicate is also functional interface but it represents a boolean operation which takes two arguments.

271. What is the difference between Function and BiFunction?

Function is a functional interface which represents an operation which takes one argument of type T and returns result of type R.

BiFunction is also functional interface which represents an operation which takes two arguments of type T and U and returns a result of type R.

272. Which functional interface do you use if you want to perform some operations on an object and returns nothing?

Consumer

273. Which functional interface is the best suitable for an operation which creates new objects?

Supplier

274. When you use UnaryOperator and BinaryOperator interfaces?

UnaryOperator performs same operation as Function but it is used when type of the argument and result should be of same type.

BinaryOperator performs same operation as BiFunction but it is used when type of the arguments and result should be of same type.

275. Along with functional interfaces which support object types, Java 8 has introduced functional interfaces which support primitive types. For example,

Consumer for object types and intConsumer, LongConsumer, DoubleConsumer for primitive types. What do you think, is it necessary to introduce separate interfaces for primitive types and object types?

Yes. If an input or output to an functional interface is a primitive type then using functional interfaces which support primitive types improves performance rather than using functional interfaces which support object types. Because it removes unnecessary boxing and unboxing of data.

276. How functional interfaces and lambda expressions are inter related?

Lambda expressions are introduced to implement functional interfaces in a simplest way and new functional interfaces are introduced to support lambda expressions in Java 8. Both together have given a new dimension to Java programming where you can write more complex data processing queries in a few lines of code.

277. What are the method references? What is the use of them?

Java 8 method references can be defined as shortened versions of lambda expressions calling a specific method. Method references are the easiest way to refer a method than the lambdas calling a specific method. Method references will enhance the readability of your code.

278. What are the different syntax of Java 8 method references?

Method Type	**Syntax**
Static Method	ClassName::MethodName
Instance method of an existing object	ReferenceVariable::MethodName

Instance method of non-existing object	ClassName::MethodName
Constructor Reference	ClassName::new

279. What are the major changes made to interfaces from Java 8?

From Java 8, interfaces can also have concrete methods i.e methods with body along with abstract methods. This is the major change made to interfaces from Java 8 to help Java API developers to update and maintain the interfaces. The interfaces can have concrete methods either in the form of default methods or static methods.

280. What are default methods of an interface? Why they are introduced?

Default methods of an interface are the concrete methods for which implementing classes need not to give implementation. They inherit default implementation.

Default methods are introduced to add extra features to current interfaces without disrupting their existing implementations. For example, stream() is a default method which is added to Collection interface in Java 8. If stream() would have been added as abstract method, then all classes implementing Collection interface must have implemented stream() method which may have irritated existing users. To overcome such issues, default methods are introduced to interfaces from Java 8.

281. As interfaces can also have concrete methods from Java 8, how do you solve diamond problem i.e conflict of classes inhering multiple methods with same signature?

To solve the diamond problem, Java 8 proposes 3 rules to follow. They are,

Rule 1 : Select classes over interfaces

If your class inherit multiple methods with same signature then a method from super class is selected (Remember a class can inherit only one class).

Rule 2 : Select most specific interfaces than general interfaces.

If your class doesn't extend any class and inherit multiple methods with same signature from multiple interfaces which belong to same hierarchy, then a method from most specific interface is selected.

Rule 3 : InterfaceName.super.methodName()

If your class doesn't extend any class and inherit multiple methods with same signature from multiple interfaces which doesn't belong to same hierarchy, then override that method and from within body explicitly call desired method as InterfaceName.super.methodName().

282. Why static methods are introduced to interfaces from Java 8?

Java API developers have followed the pattern of supplying an utility class along with an interface to perform basic operations on such objects.

For example, Collection and Collections. Collection is an interface and Collections is an utility class containing only static methods which operate on Collection objects.

But from Java 8, they have break this pattern by introducing static methods to interfaces. With the introduction of static methods to interface, such utility classes will disappear gradually and methods to perform basic operations will be kept as static methods in interface itself.

283. What are streams? Why they are introduced?

streams can be defined as operations on data. They are the sequence of elements from a source which support data processing operations. Using Java 8 Streams, you can write most complex data processing queries without much difficulties.

Almost every Java application use Collections API to store and process the data. Despite being the most used Java API, it is not easy to write the code for even some common data processing operations like filtering, finding, matching, sorting, mapping etc using Collections API . So, there needed Next-Gen API to process the data. So Java API designers have come with Java 8 Streams API to write more complex data processing operations with much of ease.

284. Can we consider streams as another type of data structure in Java? Justify your answer?

You can't consider streams as data structure. Because they don't store the data. You can't add or remove elements from the streams. They are the just operations on data. Stream consumes a data source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. They don't modify the source.

285. What are intermediate and terminal operations?

The operations which return stream themselves are called intermediate operations. For example – filter(), distinct(), sorted() etc.

The operations which return other than stream are called terminal operations. count(). min(), max() are some terminal operations.

286. What do you mean by pipeline of operations? What is the use of it?

A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. Using this, you can write more complex data processing queries with much of ease.

287. "Stream operations do the iteration implicitly" what does it mean?

Collections need to be iterated explicitly. i.e you have to write the code to iterate over collections. But, all stream operations do the iteration internally behind the scene for you. You need not to worry about iteration at all while writing the code using Java 8 Streams API.

288. Which type of resource loading do Java 8 streams support? Lazy Loading OR Eager Loading?

Lazy Loading.

289. What are short circuiting operations?

Short circuiting operations are the operations which don't need the whole stream to be processed to produce a result. For example – findFirst(), findAny(), limit() etc.

290. What are selection operations available in Java 8 Stream API?

Operation	Description
filter()	Selects the elements which satisfy the given predicate.
distinct()	Selects only unique elements
limit()	Selects first *n* elements
skip()	Selects the elements after skipping first *n* elements

291. What are sorting operations available in Java 8 streams?

There is only one sorting operation available in Java 8 streams which is sorted(). It has two versions. One which takes no argument sorts the elements in natural order and another one which takes Comparator as an argument sorts the elements according to supplied Comparator.

292. What are reducing operations? Name the reducing operations available in Java 8 streams?

Reducing operations are the operations which combine all the elements of a stream repeatedly to produce a single value. For example, counting number of elements, calculating average of elements, finding maximum or minimum of elements etc.

Reducing operations available in Java 8 streams are,

Operation	Description
min()	Returns minimum element
max()	Returns maximum element
count()	Returns the number of elements
collect()	Returns mutable result container

293. What are the matching operations available in Java 8 streams?

Operation	Description

anyMatch()	Returns true if any one element of a stream matches with given predicate
allMatch()	Returns true if all the elements of a stream matches with given predicate
noneMatch()	Returns true only if all the elements of a stream doesn't match with given predicate.

294. What are searching / finding operations available in Java 8 streams?

Operation	**Description**
findFirst()	Returns first element of a stream
findAny()	Randomly returns any one element in a stream

295. Name the mapping operations available in Java 8 streams?

Operation	**Description**

map()	Returns a stream consisting of results after applying given function to elements of the stream.
flatMap()	

296. What is the difference between map() and flatMap()?

Java 8 map() and flatMap() are two important methods of java.util.stream.Stream interface used for transformation or mapping operations. Both are intermediate operations. The only difference is that map() takes Stream<T> as input and return Stream<R> where as flatMap() takes Stream<Stream<T> as input and return Stream<R> i.e flatmap() removes extra layer of nesting around input values.

297. What is the difference between limit() and skip()?

limit() is an intermediate operation in Java 8 streams which returns a stream containing first *n* elements of the input stream.

skip() is also an intermediate operation in Java 8 streams which returns a stream containing the remaining elements of the input stream after skipping first n elements.

NEXT IS NOW

298. What is the difference between findFirst() and findAny()?

findFirst() is a terminal operation in Java 8 streams which returns first element of the input stream. The result of this operation is predictable.

findAny() is also terminal operation in Java 8 streams which randomly returns any one element of the input stream. The result of this operation is unpredictable. It may select any element in a stream.

299. Do you know Stream.collect() method, Collector interface and Collectors class? What is the relation between them?

collect() method is a terminal operation in Stream interface. It is a special case of reduction operation which returns mutable result container such as List, Set or Map.

Collector is an interface in java.util.stream package.

Collectors class, also a member of java.util.stream package, is an utility class containing many static methods which perform some common reduction operations.

All the methods of Collectors class return Collector type which will be supplied to collect() method as an argument.

300. Name any 5 methods of Collectors class and their usage?

Method	**Description**
joining()	Concatenates input elements separated by the specified delimiter.
counting()	Counts number of input elements
groupingBy()	Groups the input elements according supplied classifier and returns the results in a *Map*.
partitioningBy()	Partitions the input elements according to supplied *Predicate* and returns a *Map<Boolean, List<T>>*

toList()	Collects all input elements into a new *List*

301. What are the differences between collections and streams?

Collections	Streams
Collections are mainly used to store and group the data.	Streams are mainly used to perform operations on data.
You can add or remove elements from collections.	You can't add or remove elements from streams.
Collections have to be iterated externally.	Streams are internally iterated.
Collections can be traversed multiple times.	Streams are traversable only once.
Collections are eagerly constructed.	Streams are lazily constructed.
Ex : List, Set, Map…	Ex : filtering, mapping, matching…

302. What is the purpose of Java 8 Optional class?

Java 8 Optional class is used represent an absence of a value i.e null. Before Java 8, if-constructs are used to check for null value. But, Optional class gives better mechanism to handle null vale or absence of a value.

303. What is the difference between Java 8 Spliterator and the iterators available before Java 8?

Iterator	**Spliterator**
It performs only iteration.	It performs splitting as well as iteration.
Iterates elements one by one.	Iterates elements one by one or in bulk.
Most suitable for serial processing.	Most suitable for parallel processing.
Iterates only collection types.	Iterates collections, arrays and streams.
Size is unknown.	You can get exact size or estimate of the size.
Introduced in JDK 1.2.	Introduced in JDK 1.8.
You can't extract properties of the iterating elements.	You can extract some properties of the iterating elements.
External iteration.	Internal iteration.

304. What is the difference between Java 8 StringJoiner, String.join() and Collectors.joining()?

StringJoiner is a class in java.util package which internally uses StringBuilder class to join the strings. Using StringJoiner, you can join only the strings, but not the array of strings or list of strings.

String.join() method internally uses StringJoiner class. This method can be used to join strings or array of strings or list of strings, but only with delimiter not with prefix and suffix.

Collectors.joining() method can also be used to join strings or array of strings or list of strings with delimiter and it also supports prefix and suffix.

305. Name three important classes of Java 8 Date and Time API?

java.time.LocalDate, java.time.LocalTime and java.time.LocalDateTime

306. How do you get current date and time using Java 8 features?

```
LocalDateTime currentDateTime = LocalDateTime.now();
```

Questions from 47 to 53 depends on the following Student class.

PARVAM
NEXT IS NOW

```
class Student
{
    String name;
    int id;
    String subject;
    double percentage;
    public Student(String name, int id, String subject, double percentage)
    {
        this.name = name;
        this.id = id;
        this.subject = subject;
        this.percentage = percentage;
    }
```

```
public String getName()

{

    return name;

}

public int getId()

{

    return id;

}

public String getSubject()

{

    return subject;

}

public double getPercentage()
```

```
    {
        return percentage;
    }
    @Override
    public String toString()
    {
        return name+"-"+id+"-"+subject+"-"+percentage;
    }
}
```

307. Given a list of students, write a Java 8 code to partition the students who got above 60% from those who didn't?

```
Map<Boolean, List<Student>> studentspartionedByPercentage =
studentList.stream().collect(Collectors.partitioningBy(student ->
student.getPercentage() > 60.0));
```

308. Given a list of students, write a Java 8 code to get the names of top 3 performing students?

```
List<Student> top3Students =
studentList.stream().sorted(Comparator.comparingDouble(Student::getPercentage).reversed()).limit(3).collect(Collectors.toList());
```

309. Given a list of students, how do you get the name and percentage of each student?

```
Map<String, Double> namePercentageMap =
studentList.stream().collect(Collectors.toMap(Student::getName, Student::getPercentage));
```

310. Given a list of students, how do you get the subjects offered in the college?

```
Set<String> subjects =
studentList.stream().map(Student::getSubject).collect(Collectors.toSet());
```

311. Given a list of students, write a Java 8 code to get highest, lowest and average percentage of students?

```
DoubleSummaryStatistics studentStats =
studentList.stream().collect(Collectors.summarizingDouble(Student::getPercentage)
);

System.out.println("Highest Percentage : "+studentStats.getMax());

System.out.println("Lowest Percentage : "+studentStats.getMin());
```

```
System.out.println("Average Percentage : "+studentStats.getAverage());
```

312. How do you get total number of students from the given list of students?

```
Long studentCount = studentList.stream().collect(Collectors.counting());
```

313. How do you get the students grouped by subject from the given list of students?

```
Map<String, List<Student>> studentsGroupedBySubject =
studentList.stream().collect(Collectors.groupingBy(Student::getSubject));
```

NEXT IS NOW