1) **Few Facts of Java-**
   - Java is Object Oriented. However, it is not considered as pure object oriented as it provides support for primitive data types (like int, char, etc)
   - The Java codes are first compiled into byte code (machine independent code). Then the byte code is run on Java Virtual Machine (JVM) regardless of the underlying architecture.
   - Java syntax is similar to C/C++. But Java does not provide low level programming functionalities like pointers. Also, Java codes are always written in the form of classes and objects.

2) JDK, JRE and JVM?

Ans- JVM, JRE and JDK all three are platform dependent because configuration of each Operating System is different. But Java is platform independent.

- **JDK (Java Development Kit)** : The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
  It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

- **JRE (Java Runtime Environment) :** JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.
  It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.
  JRE is part of the Java Development Kit (JDK) but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation

- **JVM**: JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed line by line.
  JVMs are available for many hardware and software platforms.

3) **Important Features of Java**

Ans-

- **Simple**
- **Platform Independent**
- **Architectural Neutral:** A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering their development and compilation.

- **Portable:** If any language supports platform independent and architectural neutral feature known as portable.
- **Multi-Threading:** A flow of control is known as a thread. When any Language executes multiple thread at a time that language is known as multithreaded e. It is multithreaded.
- **Distributed:** Using this language we can create distributed applications. In distributed application multiple client system depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.
- **Networked:** It is mainly designed for web-based applications, J2EE is used for developing network-based applications.
- **Robust:** Simply means of Robust are strong. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection
- **Dynamic:** It supports Dynamic memory allocation due to this memory wastage is reduce and improve performance of the application.
- **Secured:** It is a more secure language compared to other language; In this language, all code is covered in byte code after compilation which is not readable by human.
- **Object Oriented:** It supports OOP's concepts because of this it is most secure language

4) Naming conventions in Java?

Ans- They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

Classes and Interfaces- First letter of each word should be capitalized.

Methods- first letter should be in lowercase and first letter of each word capitalized.

Variables- Should not start with “$” and “_”. One-character variable names should be avoided for temporary variables.

Constant- Should be all uppercase and separated by “_”

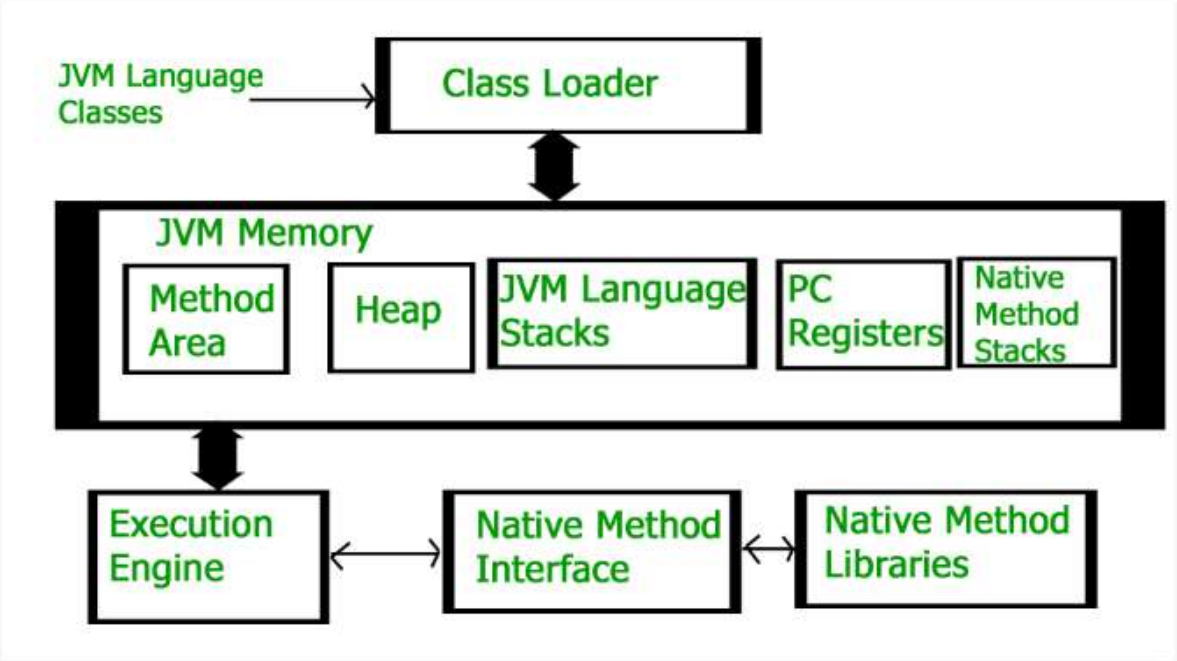
Package- should be in lowercase.

5) How JVM works?

Ans- JVM (Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.



Class Loader Subsystem

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading: The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory.

This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc

```
Student s1 = new Student();

// Getting hold of Class object created by JVM.
Class c1 = s1.getClass();

// Printing type of object using c1.
System.out.println(c1.getName());
```

```
// getting all methods in an array
Method m[] = c1.getDeclaredMethods();

for (Method method : m)
   System.out.println(method.getName());

// getting all fields in an array
Field f[] = c1.getDeclaredFields();

for (Field field : f)
   System.out.println(field.getName());
```

Note: For every loaded *.class* file, only **one** object of Class is created.

```
Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true
```

Linking : Performs verification, preparation, and (optionally) resolution.

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.
- *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization : In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.

In general, there are three class loaders :

- ***Bootstrap class loader*** : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
- ***Extension class loader*** : It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.

- ***System/Application class loader***: It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

Note : JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.
Java ClassLoader is hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way uniqueness is maintained in the runtime environment. If the parent class loader doesn't find the class then the class loader itself tries to load the class.

ClassLoaderTest.java

```
package com.journaldev.classloader;

public class ClassLoaderTest {

        public static void main(String[] args) {

                System.out.println("class loader for HashMap: "
                                + java.util.HashMap.class.getClassLoader());
                System.out.println("class loader for DNSNameService: "
                                + sun.net.spi.nameservice.dns.DNSNameService.class
                                                .getClassLoader());
                System.out.println("class loader for this class: "
                                + ClassLoaderTest.class.getClassLoader());

                System.out.println(com.mysql.jdbc.Blob.class.getClassLoader());

        }

}
```

Output of the above java classloader example program is:

```
class loader for HashMap: null
class loader for DNSNameService: sun.misc.Launcher$ExtClassLoader@7c354093
class loader for this class: sun.misc.Launcher$AppClassLoader@64cbbe37
sun.misc.Launcher$AppClassLoader@64cbbe37
```

As you can see that java.util.HashMap ClassLoader is coming as null that reflects Bootstrap ClassLoader whereas DNSNameService ClassLoader is ExtClassLoader. Since the class itself is in CLASSPATH, System ClassLoader loads it.

When we are trying to load HashMap, our System ClassLoader delegates it to the Extension ClassLoader, which in turns delegates it to Bootstrap ClassLoader that found the class and load it in JVM.

The same process is followed for DNSNameService class but Bootstrap ClassLoader is not able to locate it since it's in $JAVA_HOME/lib/ext/dnsns.jar and hence gets loaded by Extensions Classloader

JVM Memory

Method area: In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap area: Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

Stack area: For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers: Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

Native method stacks : For every thread, separate native stack is created. It stores native method information.

Execution Engine

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

- ***Interpreter*:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- ***Just-In-Time Compiler (JIT)*** : It is used to increase efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
- ***Garbage Collector***: It destroy un-referenced objects.

Java Native Interface (JNI):

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

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

6) JVM Shutdown Hook in Java?

Ans- Shutdown Hooks are a special construct that allows developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.

```
public class ShutDownHook
{
  public static void main(String[] args)
  {
    Runtime.getRuntime().addShutdownHook(new Thread()   {
          public void run()       {
                System.out.println("Shutdown Hook is running !");
          }
    });
    System.out.println("Application Terminating ...");
  }
}
```

Output:

```
Application Terminating ...
Shutdown Hook is running !
```

Note: ***Shutdown hooks are called when the application terminates normally*** *(when all threads finish, or when System.exit(0) is called). Also, when the JVM is shutting down due to external causes such as a user requesting a termination (Ctrl+C), a SIGTERM being issued by O/S (normal kill command, without -9), or when the operating system is shutting down.*

It is not guaranteed that shutdown hooks will always run. *If the JVM crashes due to some internal error, then it might crash down without having a chance to execute a single instruction. Also, if the O/S gives a SIGKILL (http://en.wikipedia.org/wiki/SIGKILL) signal (kill -9 in Unix/Linux) or TerminateProcess (Windows), then the application is required to terminate immediately without doing even waiting for any cleanup activities.*

We can have more than one Shutdown Hooks, *The JVM can execute shutdown hooks in any arbitrary order. Moreover, the JVM might execute all these hooks concurrently.*

Once shutdown sequence starts, it can be stopped by Runtime.halt() only.

7) If a *.java* file has more than one class then each class will compile into a separate class files.

8) Does JVM create an object of class Main?

Ans- The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

Not Main is abstract

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

Output:

```
Hello
```

Since we can’t create object of abstract classes in Java, it is guaranteed that object of class with main() is not created by JVM.

9) What makes it JAVA as platform independent language?

Ans- A compiler is a program that translates the source code for another program from a programming language into executable code.
This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code.**

Step by step Execution of Java Program:

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

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

Java is platform independent but JVM is platform dependent.

10) JDBC?

Ans-It is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational database.
t acts as a middle layer interface between java applications and database.

The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**.
JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver: This driver interacts directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.
-

Which Driver to use When?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

11) Is main method compulsory in Java?

Ans- Prior to JDK 7, the main method was not mandatory in a java program. You could write your full code under static block and it ran normally.

- The static block is first executed as soon as the class is loaded before the main();
- It will run static block first and then it will see no main() is there. Therefore, it will give **"exception"**, as exception comes while execution. However, if we don't want an exception, we can terminate the program by System.exit(0);
- From JDK7 main method is mandatory. The compiler will verify first, whether main() is present or not. If your program doesn't contain the main method, then you will get an **error** "main method not found in the class". It will give an error (byte code verification error because in it's byte code, main is not there) not an exception because the program has not run yet.

12) In java file name and class name should be the same if the class is declared as public.

13) **In Java, Using predefined class name as Class or Variable name is allowed but you cannot use a keyword as name of a class, name of a variable nor the name of a folder used for package**.

```
// Number is predefined class name in java.lang package
// Note : java.lang package is included in every java program by default

public class Number
{
   public static void main (String[] args)    {
      System.out.println("It works");  }}
```

Output: It works

Using String as User Defined Class:

```
// String is predefined class name in java.lang package
// Note : java.lang package is included in every java program by default
public class String
{
   public static void main (String[] args)   {
      System.out.println("I got confused");  } }
```

However, in this case you will get run-time error like this: Main thread is looking for *main* method() with predefined **String class** array argument. But here, it got *main* method() with user defined String class. Whenever Main thread will see a class name, it tries to search that class scope by scope.

First it will see in your program, then in your package.If not found, then JVM follows delegation hierarchy principle to load that class.Hence you will get run-time error. To fix, write this -**public static void main (java.lang.String[] args)**

14) Identifiers are used for identification purpose. In Java, an identifier can be a class name, method name, variable name or a label. It should be alphanumeric character or "$" or "_" and should not start with the digit, are case-sensitive and reserved words can't be used as an identifier.

15) values of type boolean are not converted implicitly or explicitly (with casts) to any other type

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS
boolean	true or false	false	1 bit	true, false
byte	twos complement integer	0	8 bits	(none)
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\','\n',' β'
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

By default, fraction value is double in java and use byte and short if memory is a constraint.

16) Enum in java?

Ans-Enumerations serve the purpose of representing a group of named constants like the planets, days of the week, colors, directions, etc.

- Enums are used when we know all possible values at **compile time**.
- It is not necessary that the set of constants in an enum type stay **fixed** for all time.
- In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).
- Enum declaration can be done outside a Class or inside a Class but not inside a Method.

```
enum Color
{
   RED, GREEN, BLUE;
}
```

```
public class Test {
   public static void main(String[] args) {
      Color c1 = Color.RED;
      System.out.println(c1);
       } }
       Output:RED
```

- First line inside enum should be list of constants and then other things like methods, variables and constructor.

- Every enum internally implemented by using Class.

```
/* internally above enum Color is converted to
class Color
{
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}*/
```

- Every enum constant represents an **object** of type enum.

- Enum type can be passed as an argument to **switch** statement.

- Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can't create child enums.

- We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.

- All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.

- **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name but enum can implement many interfaces.

- **Enum methods**- These methods are present inside **java.lang.Enum**.

- **values() method** can be used to return all values present inside enum.
- Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
- **valueOf() method** returns the enum constant of the specified string value, if exists

```
enum Color
{
   RED, GREEN, BLUE;
}

public class Test
{
   public static void main(String[] args)
   {
      // Calling values()
      Color arr[] = Color.values();

      // enum with loop
      for (Color col : arr)
      {
         // Calling ordinal() to find index
         // of color.
         System.out.println(col + " at index "
                     + col.ordinal());
      }

      // Using valueOf(). Returns an object of Color with given constant.
      // Uncommenting second line causes exception IllegalArgumentException
      System.out.println(Color.valueOf("RED"));
      // System.out.println(Color.valueOf("WHITE"));
   }
}
```

Output: RED at index 0

GREEN at index 1

BLUE at index 2

RED

enum and constructor :

- enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

enum and methods :

- enum can contain **concrete** methods only i.e. no any **abstract** method.

```
// Java program to demonstrate that enums can have constructor
// and concrete methods.

// An enum (Note enum keyword inplace of class keyword)
enum Color
{
   RED, GREEN, BLUE;

   // enum constructor called separately for each constant
   private Color()
   {
      System.out.println("Constructor called for : " + this.toString());
   }

   // Only concrete (not abstract) methods allowed
   public void colorInfo()
   {
      System.out.println("Universal Color");
   }
}

public class Test
{
    public static void main(String[] args)
   {
      Color c1 = Color.RED;
      System.out.println(c1);
      c1.colorInfo();
   }
}
```

Output:

```
Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE
```

RED

Universal Color

17) Enum with customized value in Java?

Ans-By default enums have their own string values, we can also assign some custom values to enums.

```
enum TrafficSignal
{
   // This will call enum constructor with one
   // String argument
   RED("STOP"), GREEN("GO"), ORANGE("SLOW DOWN");

   // declaring private variable for getting values
   private String action;

   // getter method
   public String getAction()
   {
      return this.action;
   }

   // enum constructor - cannot be public or protected
   private TrafficSignal(String action)
   {
      this.action = action;
   }
}
public class EnumConstructorExample
{
   public static void main(String args[])
   {
      // let's print name of each enum and there action
      TrafficSignal[] signals = TrafficSignal.values();

      for (TrafficSignal signal : signals)
      {
         // use getter method to get the value
         System.out.println("name : " + signal.name() +  " action: " + signal.getAction() );
      }
   }
}
```

Output:

```
name : RED action: STOP

name : GREEN action: GO

name : ORANGE action: SLOW DOWN
```

1. We have to create parameterized constructor for this enum class. Why? Because as we know that enum class's object can't be create explicitly so for initializing we use parameterized constructor. And the constructor cannot be the public or protected it must have private or default modifiers. Why? if we create public or protected, it will allow initializing more than one objects. This is totally against enum concept.
2. We have to create one getter method to get the value of enums.

Here enum have three members- RED, GREEN and YELLOW which have their own different custom values- STOP,GO and SLOW DOWN.

18) Scope of a variable is the part of the program where the variable is accessible.

Member variable has class level scope, local variables has method level scope and loop variables has block scope means outside the loop they are not accessible and will get error at compile time

When a method has the same local variable as a member, this keyword can be used to reference the current class variable.

19) Final variables?

Ans- A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later. They are used to create immutable objects (objects whose members can't be changed once initialized).

A **blank final** variable in Java is a final variable that is not initialized during declaration. Values must be assigned in constructor, we do this because but if we initialize here, then all objects get the same value. So, we use blank final.

```
class Test
{
   final public int i;
       Test(int val)
       {  this.i = val;  }

   Test()
   {       // Calling Test(int val)
      this(10);   }
```

```
    public static void main(String[] args)
    {
        Test t1 = new Test();
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

Output:

10

20

If we have more than one constructors or overloaded constructor in class, then blank final variable must be initialized in all of them. However constructor chaining can be used to initialize the blank final variable.

In Java, non-static final variables can be assigned a value either in constructor or with the declaration. But, static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

20) For-each loop?

Ans- For-each is another array traversing technique like other loops. Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

```
for (type var : array) {
   statements using var;
   }
```

Limitations of for-each loop

A) For-each loops are **not appropriate when you want to modify the array**

```
for (int num : marks)
{
   // only changes num, not the array element
   num = num*2;
}
```

B) For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop

```
for (int num : numbers)
{
   if (num == target)
   {
      return ???;   // do not know the index of num
   }
}
```

C) For-each **only iterates forward over the array in single steps**

```
// cannot be converted to a for-each loop
for (int i=numbers.length-1; i>0; i--)
{
    System.out.println(numbers[i]);
}
```

D) For-each **cannot process two decision making statements** at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{
   if (numbers[i] == arr[i])
   { ...
   }
}
```

21) Switch case?

Ans- Expression can be of type byte, short, int, char or an enumeration. Beginning with JDK7, *expression* can also be of type String.

- Duplicate case values are not allowed.
- The value for a case must be a constant or a literal. Variables are not allowed.
- The default statement is optional. Can be placed anywhere inside the switch block.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue into the next case.
- We can use a switch as part of the statement sequence of an outer switch. This is called a nested switch

Break: In Java, break is majorly used for:

- Terminate a sequence in a switch statement (discussed above).
- To exit a loop.

Continue: That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

22) String in switch statements?

Ans- **Important Points:**

- **Expensive operation:** Switching on strings can be more expensive in term of execution than switching on primitive data types. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form.
- **String should not be NULL:** Ensure that the expression in any switch statement is not null while working with strings to prevent a NullPointerException from being thrown at run-time.
- **Case Sensitive Comparison:** The switch statement compares the String object in its expression with the expressions associated with each case label as if it were using the String.equals method; consequently, the comparison of String objects in switch statements is case sensitive.
- **Better than if-else:** The Java compiler generates generally more efficient bytecode from switch statements that use String objects than from chained if-then-else statements.

23) Unlike C++, we don't need forward declarations in Java. Identifiers (class and method names) are recognized automatically from source files.The Java program compiles and runs fine. Note that *Test1* and *fun()* are not declared before their use.

24) The widening primitive conversion happens only when a operator like '+' is present between the operands.

- The result of adding Java chars, shorts or bytes is an int:
- If either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, **both operands are converted to type int**

```
System.out.print("Y" + "O");
System.out.print('L' + 'O');
```

When we use double quotes, the text is treated as a string and "YO" is printed, but when we use single quotes, the characters 'L' and 'O' are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added ('L' is 76 and 'O' is 79) and 155 is printed.

25) Type conversion in Java?

Ans- Widening conversion takes place when two data types are automatically converted else they need to be casted or converted explicitly. F. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type

Widening or Automatic Type Conversion: *Byte -> Short -> Int -> Long -> Float ->Double*

```
int i = 100;

//automatic type conversion
long l = i;

//automatic type conversion
float f = l;
System.out.println("Int value "+i);
System.out.println("Long value "+l);
System.out.println("Float value "+f);
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

Double -> Float -> Long ->Int ->Short ->byte

Ex-1 char and number are not compatible with each other

```
char ch = 'c';
int num = 88;
ch = num;
```

Output: incompatible types: possible lossy conversion from int to char

```
ch = num;
```

Ex-2 `double d = 100.04;`

```
//explicit type casting
long l = (long)d;

//explicit type casting
int i = (int)l;
System.out.println("Double value "+d);

//fractional part lost
System.out.println("Long value "+l);

//fractional part lost
System.out.println("Int value "+i);
```

Output:

```
Double value 100.04

Long value 100

Int value 100
```

Ex-3 While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).

```
byte b;
 int i = 257;
 double d = 323.142;
 System.out.println("Conversion of int to byte.");

 //i%256
 b = (byte) i;
 System.out.println("i = " + i + " b = " + b);
 System.out.println("\nConversion of double to byte.");

 //d%256
 b = (byte) d;
 System.out.println("d = " + d + " b= " + b);
```

Output:

```
Conversion of int to byte.

i = 257 b = 1


Conversion of double to byte.
```

```
d = 323.142 b = 67
```

26) **Explicit type casting in Expressions?**

Ans- While evaluating expressions, the result is automatically updated to larger data type of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.

```
byte b = 50;

//type casting int to byte
b = (byte)(b * 2);
```

27) **Type promotion in Expressions**

Ans- While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion

- Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- if one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

28) Null in Java?

Ans- a) **== and !=** The comparison and not equal to operators are allowed with null in Java.

b) We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won't throw Null pointer Exception.

c) the java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.

d) null is not Object or neither a type. It's just a special value, which can be assigned to any reference type and you can type cast null to any type

e) **null is Case sensitive:** null is literal in Java and because keywords are **case-sensitive** in java, we can't write NULL or 0 as in C language. It will throw compile-time error : can't find symbol 'NULL' *Object obj = NULL;*

29) A new feature was introduced by JDK 7 which allows to **write numeric literals using the underscore character**. Numeric literals are broken to enhance the readability.

```
int inum = 1_00_00_000;
System.out.println("inum:" + inum);

long lnum = 1_00_00_000;
System.out.println("lnum:" + lnum);

float fnum = 2.10_001F;
System.out.println("fnum:" + fnum);

double dnum = 2.10_12_001;
System.out.println("dnum:" + dnum);
```

Output:

```
inum: 10000000

lnum: 10000000

fnum: 2.10001

dnum: 2.1012001
```

30) Binary search in JAVA?

Ans- It can be done in 2 ways:

a) Arrays.binarysearch() works for arrays which can be of primitive data type also

```
int arr[] = { 10, 20, 15, 22, 35 };
Arrays.sort(arr);

int key = 22;
int res = Arrays.binarySearch(arr, key);
if (res >= 0)
    System.out.println(key + " found at index = "
                                          + res);
else
    System.out.println(key + " Not found");

key = 40;
res = Arrays.binarySearch(arr, key);
if (res >= 0)
    System.out.println(key + " found at index = "
```

```
                                                    + res);
    else
        System.out.println(key + " Not found")
```

Output:

```
22 found at index = 3

40 ot found
```

b) Collections.binarysearch() works for objects Collections like ArrayList and LinkedList.

```
List<Integer> al = new ArrayList<Integer>();
al.add(1);
al.add(2);
al.add(3);
al.add(10);
al.add(20);

// 10 is present at index 3.
int key = 10;
int res = Collections.binarySearch(al, key);
if (res >= 0)
    System.out.println(key + " found at index = "
                                            + res);
else
    System.out.println(key + " Not found");

key = 15;
res = Collections.binarySearch(al, key);
if (res >= 0)
    System.out.println(key + " found at index = "
                                             + res);
else
    System.out.println(key + " Not found");
```

Note:

Q- **What if input is not sorted?**
A- If input list is not sorted, the results are undefined.

Q-**What if there are duplicates?**
A- If there are duplicates, there is no guarantee which one will be found.

Q- **What is significant value of negative value returned by both functions?**
A- The function returns an index of the search key, if it is contained in the array; otherwise, (-(insertion point) – 1).

31) Sorting in JAVA?
Ans- There are two in-built methods to sort in Java.

a) Arrays.Sort() works for arrays which can be of primitive data type also
b) Collections.sort() works for objects Collections like ArrayList and LinkedList.

Note:

Q- **Which sorting algorithm does Java use in sort()?**

A- Previously, Java's Arrays.sort method used Quicksort for arrays of primitives and Merge sort for arrays of objects. In the latest versions of Java, Arrays.sort method and Collection.sort() uses Timsort.

Q-Which order of sorting is done by default?

A- It by default sorts in ascending order.

Q-How to sort array or list in descending order?

A-It can be done with the help of Collections.reverseOrder().

```
Arrays.sort(arr, Collections.reverseOrder());
Collections.sort(al, Collections.reverseOrder());
```

Q- **How to sort only a subarray**

A- Sort subarray from index 1 to 4, i.e.,

```
// only sort subarray {7, 6, 45, 21} and keep other elements as it is.
 Arrays.sort(arr, 1, 5);
```

32) Class and objects?

Ans- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

An object represents the real-life entities which consists of –

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

Example of an object: dog

Identity: Name of the dog
State: Breed, age, color
Behaviour: Bark, Sleep and eat

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap.

33) Ways to create object in Java?
Ans-

A) **Using new keyword**

```
// creating object of class Test
Test t = new Test();
```

B) **Using Class.forName(String className) method:** If we know the name of the class & if it has a public default constructor we can create an object. Class.forName actually loads the Class in Java but doesn't create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

```
// creating object of public class Test
// consider class Test present in com.p1 package
Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

C) **Using clone() method**: clone() method is present in Object class. Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.
To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

```
public class Test implements Cloneable
{
   @Override
   protected Object clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
}
// creating object of class Test
Test t1 = new Test();
```

```
// creating clone of above object
Test t2 = (Test)t1.clone();
```

D) **Deserialization**: De-serialization is technique of reading an object from the saved state in a file.
Whenever we serialize and then deserialize an object, JVM creates a separate object.
In **deserialization**, JVM doesn't use any constructor to create the object.
To deserialize an object we need to implement the Serializable interface in the class.

```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

34) Swapping objects in JAVA?

Ans- What if we don't know members of Car or the member list is too big. T*his is a very common situation as a class that uses some other class may not access members of other class*.

Using wrapper class it is possible to do that even if the user class doesn't have access to members of the class whose objects are to be swapped.

If we create a wrapper class that contains references of Car, we can swap cars by swapping references of wrapper class.

```
// A Wrapper over class that is used for swapping
class CarWrapper
{
   Car c;

   // Constructor
   CarWrapper(Car c)   {this.c = c;}
}

// A Class that use Car and swaps objects of Car
// using CarWrapper
class Main
{
    // This method swaps car objects in wrappers
    // cw1 and cw2
    public static void swap(CarWrapper cw1,
                            CarWrapper cw2)
    {
        Car temp = cw1.c;
        cw1.c = cw2.c;
        cw2.c = temp;
    }
}
```

```
public static void main(String[] args)
{
  Car c1 = new Car(101, 1);
  Car c2 = new Car(202, 2);
  CarWrapper cw1 = new CarWrapper(c1);
  CarWrapper cw2 = new CarWrapper(c2);
  swap(cw1, cw2);
  cw1.c.print();
  cw2.c.print();
}
```

35) Inheritance in Java?

Ans- It is the mechanism in java by which one class is allow to inherit the features (fields and methods) of another class. Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.

- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

36) Encapsulation in Java?

Ans- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.

- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program

37) Abstraction in Java?

Ans- Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Abstract classes and Abstract methods :

- An abstract class is a class that is declared with abstract keyword.
- An abstract method is a method that is declared without an implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method defined abstract must always be redefined in the subclass,thus making overriding compulsory OR either make subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

When to use abstract classes and abstract methods with an example

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors.

```
/ Java program to illustrate the concept of Abstraction
abstract class Shape
{
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have constructor
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() {
        return color;
    }
}
class Circle extends Shape
{
    double radius;

    public Circle(String color,double radius) {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override
    public String toString() {
        return "Circle color is " + super.color +
```

```
                        "and area is : " + area();
    }


}
class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(String color,double length,double width) {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override
    double area() {
        return length*width;
    }

    @Override
    public String toString() {
        return "Rectangle color is " + super.color +
                            "and area is : " + area();
    }

}
public class Test
{
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output:

```
Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0
```

Encapsulation vs Data Abstraction

1. Encapsulation is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

Advantages of Abstraction

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.

38) Runtime polymorphism in Java?

Ans- Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.

- In Java, we can override

- methods only, not the variables (data members), so **runtime polymorphism cannot be achieved by data members.**

```
// Java program to illustrate the fact that runtime polymorphism cannot be
achieved by data members

class A

{

    int x = 10;

    void m1()
    {
        System.out.println("Inside A's m1 method");
```

```
    }

}

// class B
class B extends A
{
    int x = 20;
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1 method");
    }

}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
        System.out.println(a.m1());
    }
}
```

Output:

```
10
Inside B's m1 method
```

Explanation: In above program, both the class A(super class) and B(sub class) have a common variable 'x'. Now we make object of class B, referred by 'a' which is of type of class A. Since variables are not overridden, so the statement "a.x" will **always** refer to data member of super class.

Advantages of Dynamic Method Dispatch

1. Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Static vs Dynamic binding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object
- Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e. at run time.

Why binding of static, final and private methods is always a static binding?
Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn't have any difficulty to determine object of class (local class for sure). That's the reason binding for such methods is static.

```
public static class superclass
    {
        static void print()
        {
            System.out.println("print in superclass.");
        }
    }
    public static class subclass extends superclass
    {
        static void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
```

```
        superclass B = new subclass();
        A.print();
        B.print();
    }
```

Output:

```
print in superclass.

print in superclass.
```

As you can see, in both cases print method of superclass is called.
We have created one object of subclass and one object of superclass with the reference of the superclass.
Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity

Dynamic Binding: In Dynamic binding compiler doesn't decide the method to be called. Overriding is a perfect example of dynamic binding.

During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and therefore the corresponding version of print will be called based on type on object

39) Association, aggregation and composition in Java?
Ans- Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.
Composition and **Aggregation** are the two forms of association.

Aggregation
It is a special form of Association where:

- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity

```
/* Department class contains list of student Objects. It is associated with
student class through its Object(s). */
class Department
{

    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {
```

```
        this.name = name;
        this.students = students;

    }

    public List<Student> getStudents()
    {
        return students;
    }
}
```

Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

```
// class book
class Book
{

    public String title;
    public String author;

    Book(String title, String author)
    {

        this.title = title;
        this.author = author;
    }
}

// Libary class contains list of books.
class Library
{

    // reference to refer to list of books.
    private final List<Book> books;

    Library (List<Book> books)
    {
        this.books = books;
    }

    public List<Book> getTotalBooksInLibrary(){

       return books;
    }

}
```

A library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That's why it is composition.

Aggregation vs Composition

a) **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don't exist separate to a Human
b) **Type of Relationship:** Aggregation relation is **"has-a"** and composition is **"part-of"** relation.
c) **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

40) **Access modifiers**- *private, public, protected and package-private*
Non-Access modifiers- *static, final, abstract, synchronized, transient, volatile and native*.

41) 'this' keyword in Java?

Ans- 'this' is a reference variable that refers to the current object.

a) Using 'this' keyword to refer current class instance variables

```
// Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
```

b) **Using this() to invoke current class constructor**

```
//Default constructor
Test()
{
    this(10, 20);
    System.out.println("Inside  default constructor \n");
}

//Parameterized constructor
Test(int a, int b)
{
    this.a = a;
    this.b = b;
    System.out.println("Inside parameterized constructor");
}
```

c) **Using 'this' keyword to return the current class instance**

```
//Method that returns current class instance
    Test get()
    {
        return this;
    }
```

d) Using 'this' keyword as method parameter

```
// Method that send current class instance as a parameter to another
method
    void get()
    {
        display(this);
    }
```

e) **Using 'this' keyword to invoke current class method**

```
void display()
{
     // calling fuction show()
     this.show();

     System.out.println("Inside display function");
}
```

42) Method overloading in JAVA?

Ans- Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or orders of parameters or combination of them. Overloading is related to compile time (or static) polymorphism.

```
// Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }
```

43) **What if the exact prototype does not match with arguments.**

Ans. Priority wise, compiler take these steps:

1. Type Conversion but to higher type (in terms of range) in same family.

2. Type conversion to next higher family (suppose if there is no long data type available for an int data type, then it will search for float data type).

But if not found by compiler than there will be an error. Like for float datatype if double is not available then there will be an error.

44) **What is the advantage?**

We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

45) We **cannot** overload by return type. It will throw a compile time error because the compiler will give error as the return value alone is not sufficient for the compiler to figure out which function it has to call. Only if both methods have different parameter types (so, they have different signature), then Method overloading is possible.

46) **Can we overload static methods?**

Ans- The answer is '**Yes**'. We can have two or more static methods with same name, but differences in input parameters. Also we can overload main() in Java like other static methods.

```
// Normal main()
    public static void main(String[] args)
    {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }

    // Overloaded main methods
    public static void main(String arg1)
    {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek", "My Geek");
    }
    public static void main(String arg1, String arg2)
    {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
```

Output :

```
Hi Geek (from main)
Hi, Geek
Hi, Dear Geek, My Geek
```

47) **Can we overload methods that differ only by static keyword?**

Ans- We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same).

48) **Does Java support Operator Overloading?**
Ans- Unlike C++, Java doesn't allow user-defined overloaded operators. Internally Java overloads operators, for example, + is overloaded for concatenation.

49) **What is the difference between Overloading and Overriding?**

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.

- Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

50) Method overloading with Autoboxing and unboxing.
Ans- Conversion of primitive type to its corresponding wrapper Object is called Autoboxing and Wrapper Object to its corresponding primitive type is known as Unboxing.

```
public void method(Integer i)
{

    System.out.println("Reference type Integer formal argument :" + i);
}

public void method(long i)
{

    System.out.println("Primitive type long formal argument :" + i);
}

Now invoking them via-
c.method(new Integer(15));
c.method(new Long(100));
```

Output:

Reference type Integer formal argument :15

Primitive type long formal argument :100

If you are using wrapper class Object as an actual argument and compiler does not find the method with parameter(s) of the same reference type (i.e. class or interface type), then it starts searching a method with parameter(s) having the corresponding primitive data type.

Method Overloading with Widening

If compiler fails to find any method corresponding to autoboxing, then it starts searching a method parameter(s) of the widened primitive data type.

We are invoking another method with argument of Long wrapper Object. Compiler starts searching for the method having the same reference type (Long wrapper class). Since there is no method having with parameter of **Long wrapper class**. So, It searches for method which can accept the parameter bigger than long primitive data type as an argument

```
// invoking the method with signature has widened data type
     c.method(10);
     c.method(new Long(100));


 public void method(int i)
 {
     System.out.println("Primitive type int formal argument :" + i);
 }

 public void method(float i)
 {

     System.out.println("Primitive type float formal argument :" + i);
 }
```

Output:

```
Primitive type int formal argument :10
Primitive type float formal argument :100.0
```

51) **What happens if widening and boxing happen together? What method invocation will compiler be able to do?**

Ans- Widening of primitive type has taken priority over boxing and var-args. But widening and boxing of primitive type can not work together.

```
//overloaded method primitive(byte) var-args formal argument
    public void method(byte... a)
    {
        System.out.println("Primitive type byte formal argument :" + a);
    }

    // overloaded method primitive(int) formal arguments
    public void method(long a, long b)
    {
        System.out.println("Widening type long formal argument :" + a);
    }
```

```
// invokes the method having widening primitive type parameters.
        byte val = 5;
        c.method(val,val);
```

Output:

```
Widening type long formal argument :5
```

52) Shadowing of static members in JAVA?

Ans- if name of a derived class static function is same as base class static function then the base class static function shadows (or conceals) the derived class static function.

```
class A {
   static void fun() {
      System.out.println("A.fun()");
   }
}

class B extends A {
   static void fun() {
      System.out.println("B.fun()");
   }
}

public class Main {
   public static void main(String args[]) {
      A a = new B();
      a.fun();  // prints A.fun()
   }
}
```

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”.

53) Instance and static methods in JAVA?

Ans-

Instance method

- They are the methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in within which it defined.
- They can be overridden since they are resolved using **dynamic binding** at run time.
- **Memory allocation:** These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack.

Static Method

- Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the **class name itself** or reference to the Object of that class. They are designed with aim to be shared among all Objects created from the same class.
- Their memory allocation is same as of instance methods
- Static methods cannot be overridden. But can be overloaded since they are resolved using **static binding** by compiler at compile time.

Instance method vs Static method

- Instance method can access the instance methods and instance variables directly.
- Instance method can access static variables and static methods directly.
- Static methods can access the static variables and static methods directly.
- Static methods can't access instance methods and instance variables directly. They must use reference to object. And static method can't use this keyword as there is no instance for 'this' to refer to.

54) Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be **sub-type** of parent's return type. Overriding method becomes **variant** with respect to return type.

```
// Two classes used for return types.
class A {}
class B extends A {}

class Base
{
    A fun()
    {
        System.out.println("Base fun()");
        return new A();
    }
}

class Derived extends Base
{
    B fun()
    {
        System.out.println("Derived fun()");
        return new B();
    }
}
```

55) Object class in JAVA?

Ans-**Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child

class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes

Its methods-

a) **toString()** :

```
// Default behavior of toString() is to print class name, then  @, then unsigned hexadecimal representation of the hash code of the object

public String toString()

{

   return getClass().getName() + "@" + Integer.toHexString(hashCode());

}
```

It is always recommended to override **toString()** method to get our own String representation of Object.

```
Student s = new Student();

// Below two statements are equivalent

System.out.println(s);

System.out.println(s.toString());
```

b) **hashCode()-** It convert the internal address of object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.

Use of hashCode() method JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.

Note: Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example, for a Student class we can return roll no. of student from hashCode() method as it is unique.

```
// Overriding hashCode()
@Override
public int hashCode()
{
  return roll_no; }
```

c) **equals(Object obj)-** Compares the given object to “this” object (the object on which the method is called).

It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

d) **finalize()** method : This method is called just before an object is garbage collected. It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object.

We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.

Note: finalize method is called just **once** on an object even though that object is eligible for garbage collection multiple times.

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

e) **clone()** : It returns a new object that is exactly the same as this object

f) The remaining three methods **wait()**, **notify() notifyAll()** are related to Concurrency.

56) **Can a class be static in Java ?**
Ans- The answer is YES, we can have static class in java. In java, we have static instance variables as well as static methods and also static block.

Only nested classes can be static.

a) Nested static class doesn't need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference. An instance of Inner class cannot be created without an instance of outer class

b) Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

57) Overriding equal methods?

Ans-

```
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1 == c2) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output: `Not Equal - >` Because when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not. But c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true. `Complex c3 = c1;  // (c3 == c1) will be true`

how do we check for equality of values inside the objects? We can override the equals method in our class to check whether two objects have same data or not.

```
// Overriding equals() to compare two Complex objects
    @Override
    public boolean equals(Object o) {

        // If the object is compared with itself then return true
        if (o == this) {
            return true;
        }
        /* Check if o is an instance of Complex or not
          "null instanceof [type]" also returns false */
        if (!(o instanceof Complex)) {
            return false; }

        // typecast o to Complex so that we can compare data members
        Complex c = (Complex) o;

        // Compare the data members and return accordingly
        return Double.compare(re, c.re) == 0
                && Double.compare(im, c.im) == 0;
    }
```

When we override equals(), it is recommended to also override the hashCode() method. If we don't do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly

58) Overriding equal methods in java?

Ans- The default toString() method in Object prints "class name @ hash code". We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print "Real + i Imag" form.

```
    /* Returns the string representation of this Complex number.
       The format of string is "Re + iIm" where Re is real part
       and Im is imagenary part.*/

    @Override
    public String toString() {
        return String.format(re + " + i" + im);
    }
```

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

59) Static blocks in JAVA?

Ans- static block (also called static clause) which can be used for static initializations of a class.

This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class.

Also static blocks are executed before constructors.

```
 class Test {
    static int i;
    int j;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    Test(){
        System.out.println("Constructor called");
    }
}

Test t1 = new Test();
Test t2 = new Test(); // Although we have two objects, static block is
executed only once.
```

Output:
static block called
Constructor called
Constructor called

60) Initializer block in java?

Ans- Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initialize the common part of various constructors of a class.

```
// Initializer block starts..
    {
        // This code is executed before every constructor.
        System.out.println("Common part of constructors invoked !!");
    }
    // Initializer block ends

    public GFG()
    {
        System.out.println("Default Constructor invoked");
    }
    public GFG(int x)
    {
        System.out.println("Parametrized constructor invoked");
    }

GFG obj1, obj2;
obj1 = new GFG();
obj2 = new GFG(0);
```

Output:

```
Common part of constructors invoked!!

Default Constructor invoked

Common part of constructors invoked!!

Parametrized constructor invoke
```

- The contents of initializer block are executed whenever any constructor is invoked (before the constructor's contents).

- It is not at all necessary to include them in your classes.

- We can also have multiple IIBs in a single class. If compiler finds multiple IIBs, then they all are executed from top to bottom i.e. the IIB which is written at top will be executed first.

- You can have IIBs in parent class also. Instance initialization block code runs immediately after the call to super() in a constructor. Order of execution will be
  I. Instance Initialization Block of super class
  II. Constructors of super class
  III. Instance Initialization Blocks of the class
  IV. Constructors of the class

61) Why JAVA is not purely Object-oriented language?

Ans- Pure Object-Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language which supports or have features which treats everything inside program as objects.

There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

Example: Smalltalk

Java supports property 1, 2, 3, 4 and 6 but fails to support property 5 and 7 given above

Hence it is not pure object oriented

Primitive Data Type ex. int, long, bool, float, char, etc as Objects: Smalltalk is a "pure" object-oriented programming language unlike Java and C++ as there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects.

The static keyword: When we declares a class as static then it can be used without the use of an object in Java.

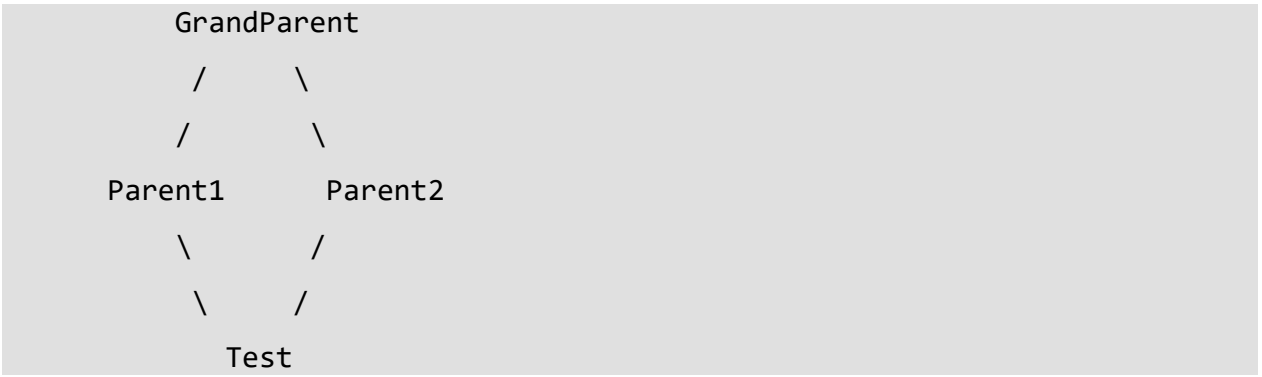
Wrapper Class: Wrapper class provides the mechanism to convert primitive into object and object into primitive. Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. And if you do any mathematical operation on it, under the hoods Java is going to use primitive type int only.

62) Why Java doesn't support multiple inheritance?

Ans- Multiple Inheritance is a feature of object-oriented concept, where a class can inherit properties of more than one parent class.

The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

The Diamond Problem:




```
//First Parent class
class Parent1
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2
{
    void fun()
    {
        System.out.println("Parent2");
    }
}

// Error : Test is inheriting from multiple classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

Output :

```
Compiler Error
```

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.

Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

63) **How are above problems handled for Default Methods and Interfaces ?**

Ans- Java 8 supports default methods where interfaces can provide default implementation of methods.

And a class can implement two or more interfaces.

In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

```
//A simple Java program to demonstrate multiple inheritance through default methods.
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
```

```
}
```

Output:

```
Default PI1

Default PI2
```

64) In inheritance, subclass acquires super class properties. An important point to note is, when subclass object is created, a separate object of super class object will not be created. Only a subclass object is created that has super class variables. So we can't blindly say that whenever a class constructor is executed, object of that class is created or not.

// A Java program to demonstrate that both super class and subclass constructors refer to same object

```
// super class
class Fruit
{
    public Fruit()
    {
        System.out.println("Super class constructor");
        System.out.println("Super class object hashcode :" +
                           this.hashCode());
        System.out.println(this.getClass().getName());
    }
}

// sub class
class Apple extends Fruit
{
    public Apple()
    {
        System.out.println("Subclass constructor invoked");
        System.out.println("Sub class object hashcode :" +
                           this.hashCode());
        System.out.println(this.hashCode() + "   " +
                           super.hashCode());
        System.out.println(this.getClass().getName() + "  " +
                           super.getClass().getName());
    }
}

// driver class
public class Test
{
    public static void main(String[] args)
    {
        Apple myApple = new Apple();
    }
}
```

Output:

```
super class constructor
super class object hashcode :366712642
Apple
sub class constructor
sub class object hashcode :366712642
366712642   366712642
Apple  Apple
```

As we can see that both super class(Fruit) object hashcode and subclass(Apple) object hashcode are same, so only one object is created.

65) In Java, constructor of base class with no argument gets automatically called in derived class constructor. if we want to call parameterized constructor of base class, then we can call it using super().

66) **An interface can also extend multiple interfaces.**

```
// Java program to demonstrate multiple inheritance in interfaces

interface intfA
{
    void geekName();
}

interface intfB
{
    void geekInstitute();
}

interface intfC extends intfA, intfB
{
    void geekBranch();
}

// class implements both interfaces and provides implementation to the method.
class sample implements intfC
{
    public void geekName()
    {
        System.out.println("Rohit");
    }

    public void geekInstitute()
    {
        System.out.println("JIIT");
    }

    public void geekBranch()
    {
```

```
        System.out.println("CSE");
    }

    public static void main (String[] args)
    {
        sample ob1 = new sample();
        // calling the method implemented within the class.
        ob1.geekName();
        ob1.geekInstitute();
        ob1.geekBranch();
    }
}
```

Output:

```
Rohit

JIIT

CSE
```

67) Final with inheritance?

Ans- **final** is a keyword in java used for restricting some functionalities. We can declare variables, methods and classes with final keyword. During inheritance, we must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes.

- When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when creating an immutable class like the predefined String class
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations
- When a method is declared as final then it cannot be overridden by subclasses and since final methods cannot be overridden, a call to one can be resolved at compile time. This is called **early or static binding**.

68) Accessing grandparent member in Java?

Ans- **In Java, we can access grandparent's members only through the parent class. *Below program will give compile error***

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}
```

```
class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print()
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output: Compiler Error

There is error in line “super.super.print();”. In Java, a class cannot directly access the grandparent’s members. It is allowed in C++ though.

69) Can we override private members?

Ans-

```
class Base {
  private void fun() {
     System.out.println("Base fun");
  }
}
 class Derived extends Base {
  private void fun() {
     System.out.println("Derived fun");
  }
  public static void main(String[] args) {
      Base obj = new Derived();
      obj.fun();
  }
}
```

We get compiler error “fun() has private access in Base”.. So the compiler tries to call base class function, not derived class, means fun() is not overridden.

An inner class can access private members of its outer class. What if we extend an inner class and create fun() in the inner class?

```
class Outer {
     private String msg = "GeeksforGeeks";
     private void fun() {
          System.out.println("Outer fun()");
     }
     class Inner extends Outer {
         private void fun()  {
               System.out.println("Accessing Private Member of Outer: " + msg);
         }
     }

     public static void main(String args[])  {

          // In order to create instance of Inner class, we need an Outer
          // class instance..
          Outer o = new Outer();
          Inner  i   = o.new Inner();
          // This will call Inner's fun, the purpose of this call is to
          // show that private members of Outer can be accessed in Inner.
          i.fun();

          // o.fun() calls Outer's fun (No run-time polymorphism).
          o = i;
          o.fun();
     }
}
```

Output:

```
Accessing Private Member of Outer: GeeksforGeeks
Outer fun()
```

In the above program, we created an outer class and an inner class. We extended Inner from Outer and created a method fun() in both Outer and Inner.

If we observe our output, then it is clear that the method fun() has not been overriden. It is so because ***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***.

Comparison With C++
1) In Java, inner Class is allowed to access private data members of outer class. This behavior is same as C++ (See this).
2) In Java, methods declared as private can never be overridden, they are in-fact bounded during compile time.

70) In Java, it is compiler error to give more restrictive access to a derived class function which overrides a base class function.

For example, if there is a function *public void foo()* in base class and if it is overridden in derived class, then access specifier for *foo()* cannot be anything other than public in derived class.

But if *foo()* is private function in base class, then access specifier for it can be anything in derived class.

71) Are data members get overridden in JAVA?
Ans- A Java program to demonstrate that non-method members are accessed according to reference type (Unlike methods which are accessed according to the referred object)

```
class Parent
{
    int value = 1000;
    Parent()
    {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent
{
    int value = 10;
    Child()
    {
        System.out.println("Child Constructor");
    }
}

// Driver class
class Test
{
```

```
    public static void main(String[] args)
    {
        Child obj=new Child();
        System.out.println("Reference of Child Type :" + obj.value);

        // Note that doing "Parent par = new Child()" would produce same result

        Parent par = obj;

        // Par holding obj will access the value variable of parent class
        System.out.println("Reference of Parent Type : "+ par.value);
    }
}
```

Output:

```
Parent Constructor

Child Constructor

Reference of Child Type : 10

Reference of Parent Type : 1000
```

If a parent reference variable is holding the reference of the child class and we have the “value” variable in both the parent and child class, it will refer to the parent class “value” variable, whether it is holding child class object reference.

The reference holding the child class object reference will not be able to access the members (functions or variables) of the child class. It is because compiler uses special run-time polymorphism mechanism only for methods.

72) Object serialization and inheritance?

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

Case 1: If superclass is serializable then every subclass is automatically serializable : Hence, even though subclass doesn’t implement Serializable interface(and if it’s superclass implements Serializable), then we can serialize subclass object.

Class A implements Serializable{}

Class B extends A{}

```
    B b1 = new B(10,20);

    System.out.println("i = " + b1.i);
    System.out.println("j = " + b1.j);
```

```
        /* Serializing B's(subclass) object */

        //Saving of object in a file
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        // Method for serialization of B's class object
        oos.writeObject(b1);

        // closing streams
        oos.close();
        fos.close();

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

        /* De-Serializing B's(subclass) object */

        // Reading the object from a file
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        // Method for de-serialization of B's class object
        B b2 = (B)ois.readObject();

        // closing streams
        ois.close();
        fis.close();

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

        System.out.println("i = " + b2.i);
        System.out.println("j = " + b2.j);
```

Output:

```
i = 10

j = 20

Object has been serialized

Object has been deserialized

i = 10

j = 20
```

73) **When to go for this approach to refer a subclass object- Referencing using superclass reference**

Ans-If we don't know exact runtime type of an object, then we should use this approach. Else if we know the exact runtime type of an object, then this approach is better- *A subclass reference can be used to refer its object.*

For example, consider an ArrayList containing different objects at different indices. Now when we try to get elements of arraylist using *ArrayList.get(int index)* method then we must use Object reference, as in this case, we don't know exact runtime type of an object

By using superclass reference, we will have access **only** to those parts(methods and variables) of the object which are defined by the superclass. We are unable to call subclass specific methods or access subclass fields. This problem can be solved using type casting in java.

```
// declaring MountainBike reference
MountainBike mb3;
//Bicycle is a Superclass of MountainBike
Bicycle mb2 = new MountainBike(4, 200, 20);
// assigning mb3 to mb2 using typecasting.
 mb3 = (MountainBike)mb2;

//now can access subclass specific methods and field
mb3.seatHeight);
mb3.setHeight(21);
```

74) Does overloading works with inheritance?

Ans-

```
class Base {

    public int f(int i)
    {
        System.out.print("f (int): ");
        return i+3;
    }
}
class Derived extends Base {
    public double f(double i)
    {
        System.out.print("f (double) : ");
        return i + 3.3;
    }
}
```

```
Derived obj = new Derived();
System.out.println(obj.f(3));
System.out.println(obj.f(3.3));
```

```
f (int): 6
f (double): 6.6
```

So in Java overloading works across scopes. Java compiler determines correct version of the overloaded method to be executed at compile time based upon the type of argument

75) Shift operators?

Ans-

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

- **>>, Signed Right shift operator:** shifts the bits of the number to the right If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler.

  The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.

- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right It always fills 0 irrespective of the sign of the number. The leftmost bit is set to 0.

```
int a = 0x0005;
int b = -10;
```

```
a<<2 = 20 0000 0101<<2 =0001 0100(20)
a>>2 = 1 0000 0101 >> 2 =0000 0001(1)
b>>>2 = 1073741821  0000 1010


a = 5 = 0000 0101
b = -10 = 1111 0110


a << 1 = 0000 1010 = 10


b << 1 = 0000 1010 = -20
b << 2 = 0001 0100 = -40
```

76) Preceedance and associativity in JAVA?

Ans-

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Output-1

```
// a=b+++c is compiled as
// b++ +c
// a=b+c then b=b+1
a = b++ + c;
```

```
Value of a(b+c), b(b+1), c = 10, 11, 0
```

Output-2

```
System.out.println(2+0+1+6+"GeeksforGeeks");
System.out.println("GeeksforGeeks"+2+0+1+6);
System.out.println(2+0+1+5+"GeeksforGeeks"+2+0+1+6);
System.out.println(2+0+1+5+"GeeksforGeeks"+(2+0+1+6));
```

The output is

9GeeksforGeeks
GeeksforGeeks2016
8GeeksforGeeks2016
8GeeksforGeeks9

77) Character and Byte Streams in Java?

Ans- **Stream** – A sequence of data.
Input Stream: reads data from source.
Output Stream: writes data to destination

Character Stream:
In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character.
For example: FileReader and FileWriter are character streams used to read from source and write to destination.

```
FileReader sourceStream = null;
try
{
      sourceStream = new FileReader("test.txt");

      // Reading sourcefile and writing content to
      // target file character by character.
      int temp;
      while ((temp = sourceStream.read()) != -1)
      System.out.println((char)temp);
}
finally
{
      // Closing stream as no longer in use
      if (sourceStream != null)
      sourceStream.close();
}
```

Byte Stream
Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

```
FileInputStream sourceStream = null,targetStream = null;
try{
      sourceStream = new FileInputStream("sorcefile.txt");
      targetStream = new FileOutputStream ("targetfile.txt");

      // Reading source file and writing content to target
      // file byte by byte
      int temp;
      while ((temp = sourceStream.read()) != -1)
      targetStream.write((byte)temp);
}
Finally {
      if (sourceStream != null)
      sourceStream.close();
      if (targetStream != null)
      targetStream.close();
}
```

When to use Character Stream over Byte Stream?

- In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream?

- Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

Notes:

- Names of character streams typically end with Reader/Writer and names of byte streams end with InputStream/OutputStream

- The streams used in example codes are unbuffered streams and less efficient. We typically use them with buffered readers/writers for efficiency. We will soon be discussing use BufferedReader/BufferedWriter (for character stream) and BufferedInputStream/BufferedOutputStream (for byte stream) classes.

- It is always recommended to close the stream if it is no longer in use. This ensures that the streams won't be affected if any error occurs.

78) Passing value at runtime from console?

Ans-

a) **Command line argument**: Java <classname> <arg1> <arg2> <arg3> . To read them, used following code-

```
if (args.length > 0)
{
   System.out.println("The command line"+" arguments are:");

   // iterating the args array and printing the command line
   arguments
   for (String val:args)
         System.out.println(val);
         }
```

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

- Does not work in non-interactive environment (such as in an IDE).

b) Using scanner class: Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program, though not very efficient.

To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.

```
Scanner sc = new Scanner(System.in);

// String input
String name = sc.nextLine();

// Character input
char gender = sc.next().charAt(0);

// Numerical data input- byte, short and float can be read
//using similar-named functions.

int age = sc.nextInt();
long mobileNo = sc.nextLong();
double cgpa = sc.nextDouble();
```

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- Regular expressions can be used to find tokens.

Drawback:

- The reading methods are not synchronized

Note 1: But there is no nextChar() in Scanner class. To read a char, we use **next().charAt(0)**. next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string

```
// Character input
  char c = sc.next().charAt(0);
```

Note 2: Sometimes, we have to check if the next value we read is of a certain type or if the input has ended (EOF marker encountered). We check if the scanner's input is of the type we want with the help of hasNextXYZ() functions where XYZ is the type we are interested in.

```
// Initialize sum and count of input elements
  int sum = 0, count = 0;

  // Check if an int value is available
  while (sc.hasNextInt())
  {
      // Read an int value
      int num = sc.nextInt();
```

```
        sum += num;
        count++;
    }
```

c) Using BufferedReader Class:

```
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

System.out.println("Enter an integer");
int a = Integer.parseInt(br.readLine());

System.out.println("Enter a String");
String b = br.readLine();
System.out.printf("You have entered:- " + a +
" and name as " + b);
```

Advantages

- The input is buffered for efficient reading.

Drawback:

- The wrapping code is hard to remember.

79) Difference between Scanner and BufferedReader class.

Ans- java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters.

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.

- BufferedReader has significantly larger buffer memory than Scanner. The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.

- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of character

- In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() doesn't not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().

- In BufferReader class there is no such type of problem.This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads till first newline character.

80) FAST I/O in JAVA?

Ans-

a) **Scanner Class** – (easy, less typing, but not recommended very slow, In most of the cases we get TLE (time limit exceeded) while using scanner class.

b) **BufferedReader –** (fast, but not recommended as it requires lot of typing): The Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. With this method we will have to parse the value every time for desired type.

Reading multiple words from single line adds to its complexity because of the use of Stringtokenizer and hence this is not recommended

```
BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in));

  StringTokenizer st = new StringTokenizer(br.readLine());
  int n = Integer.parseInt(st.nextToken());
  int k = Integer.parseInt(st.nextToken());
```

c) **Userdefined FastReader Class-** (which uses bufferedReader and StringTokenizer): This method uses the time advantage of BufferedReader and StringTokenizer and the advantage of user defined methods for less typing and therefore a faster input altogether.

```
static class FastReader
    {
        BufferedReader br;
        StringTokenizer st;

        public FastReader()
        {
            br = new BufferedReader(new
                     InputStreamReader(System.in));
        }

        String next()
        {
            while (st == null || !st.hasMoreElements())
            {
                try
                {
                    st = new StringTokenizer(br.readLine());
                }
                catch (IOException  e)
                {
                    e.printStackTrace();
```

```
            }
        }
        return st.nextToken();
    }

    int nextInt()
    {
        return Integer.parseInt(next());
    }

    long nextLong()
    {
        return Long.parseLong(next());
    }

    double nextDouble()
    {
        return Double.parseDouble(next());
    }

    String nextLine()
    {
        String str = "";
        try
        {
            str = br.readLine();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return str;
    }
}
```

81) String class in JAVA?

Ans- In Java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

- ***String literal***

  String s = “GeeksforGeeks”;

- **Using *new* keyword**

  String s = new String (“GeeksforGeeks”);

Constructors

a) **String(byte[] byte_arr)** – Construct a new String by decoding the *byte array*. It

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s_byte =new String(b_arr); //Geeks
```

b) **String(char[] char_arr)** – Allocates a new String from the given *Character array*
Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr); //Geeks
```

c) **String(char[] char_array, int start_index, int count)** – Allocates a String from a given *character array* but choose *count* characters from the *start_index*.
Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr , 1, 3); //eek
```

d) **String(StringBuffer s_buffer)** – Allocates a new string from the string *s_buffer*
Example:

```
StringBuffer s_buffer = new StringBuffer("Geeks");

String s = new String(s_buffer); //Geeks
```

e) **String(StringBuilder s_builder)** – Allocates a new string from the string in *s_builder*
Example:

```
StringBuilder s_builder = new StringBuilder("Geeks");

String s = new String(s_builder); //Geeks
```

Method:

1. **int length():** Returns the number of characters in the String.

```
"GeeksforGeeks".length();  // returns 13
```

2. **Char charAt(int i):** Returns the character at i^{th} index.

```
"GeeksforGeeks".charAt(3); // returns  'k'
```

3. **String substring (int i):** Return the substring from the i^{th} index character to end.

```
"GeeksforGeeks".substring(3); // returns "ksforGeeks"
```

4. **String substring (int i, int j):** Returns the substring from i to j-1 index.

```
"GeeksforGeeks".substring(2, 5); // returns "eks"
```

5. **String concat(String str):** Concatenates specified string to the end of this string.

```
String s1 = "Geeks";

String s2 = "forGeeks";

String output = s1.concat(s2); // returns "GeeksforGeeks"
```

6. **boolean equals(Object otherObj):** Compares this string to the specified object.

```
Boolean out = "Geeks".equals("Geeks"); // returns true

Boolean out = "Geeks".equals("geeks"); // returns false
```

7. **boolean equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.

```
Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true

Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true
```

8. **int compareTo(String anotherString):** Compares two string lexicographically.

```
int out = s1.compareTo(s2);  // where s1 ans s2 are

                    // strings to be compared

This returns difference s1-s2. If :

out < 0  // s1 comes before s2

out = 0  // s1 and s2 are equal.

out > 0   // s1 comes after s2.
```

9. **String toLowerCase():** Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;

String word3 = word1.toLowerCase(); // returns “hello"
```

10. **String toUpperCase():** Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;

String word2 = word1.toUpperCase(); // returns “HELLO”
```

11. **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;

String word2 = word1.trim(); // returns “Learn Share Learn”
```

12. **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*

```
String s1 = “feeksforfeeks“;

String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”
```

82) StringBuffer class in JAVA?

Ans- String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer( ): It reserves room for 16 characters without reallocation.

```
StringBuffer s=new StringBuffer();
```

StringBuffer(int size) It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer s=new StringBuffer(20);
```

StringBuffer(String str): It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s=new StringBuffer("GeeksforGeeks");
```

Methods:

A) **length( ) and capacity( ):** The length of a StringBuffer can be found by the length( ) method, while the total allocated capacity can be found by the capacity( ) method.

B) **append( ):** It is used to add text at the end of the existence text. Here are a few of its forms:

```
StringBuffer append(String str)

StringBuffer append(int num)
```

C) **insert( ):** It is used to insert text at the specified index position. These are a few of its forms:

```
StringBuffer insert(int index, String str)

StringBuffer insert(int index, char ch)
```

D) **reverse( ):** It can reverse the characters within a StringBuffer object using **reverse( )**

E) **delete( ) and deleteCharAt( ):** It can delete characters within a StringBuffer by using the methods **delete( )** and **deleteCharAt( )**

```
StringBuffer delete(int startIndex, int endIndex)

StringBuffer deleteCharAt(int loc)
```

F) **replace( ):** The substring being replaced is specified by the indexes start Index and endIndex

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

G) **char charAt(int index)**: This method returns the char value in this sequence at the specified index.
Syntax:

```
public char charAt(int index)
```

H) **String substring(int start)**: This method returns a new String that contains a subsequence of characters currently contained in this character sequence.

```
public String substring(int start)

public String substring(int start,int end)
```

I) **CharSequence subSequence(int start, int end)**: This method returns a new character sequence that is a subsequence of this sequence.
Syntax:

```
public CharSequence subSequence(int start, int end)
```

Some Interesting facts:

1. java.lang.StringBuffer extends (or inherits from) Object class.
2. All Implemented Interfaces of StringBuffer class:Serializable, Appendable, CharSequence.
3. String buffers are safe for use by multiple threads.

83) StringBuilder class in JAVA?
Ans-The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters.

However the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

StringBuilder will be faster under most implementations

Constructors in Java StringBuilder:

- **StringBuilder():** Constructs a string builder with no characters in it and an initial capacity of 16 characters.
- **StringBuilder(int capacity):** Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
- **StringBuilder(CharSequence seq):** Constructs a string builder that contains the same characters as the specified CharSequence.
- **StringBuilder(String str):** Constructs a string builder initialized to the contents of the specified string.

Note: Methods of StringBuffer and StringBuilder are same.

84) StringTokenizer class in Java?

Ans- It is used to break a string into tokens.

Constructor

StringTokenizer(String str) :
str is string to be tokenized.
Considers default delimiters like new line, space, tab, carriage return and form feed.

StringTokenizer(String str, String delim) :
delim is set of delimiters that are used to tokenize the given string.

StringTokenizer(String str, String delim, boolean flag):
The first two parameters have same meaning. The flag serves following purpose.

If the **flag** is **false**, delimiter characters serve to separate tokens. For example, if string is "hello geeks" and delimiter is " ", then tokens are "hello" and "geeks".

If the **flag** is **true**, delimiter characters are considered to be tokens. For example, if string is "hello geeks" and delimiter is " ", then tokens are "hello", " " and "geeks".

```
System.out.println("Using Constructor 1 - ");
StringTokenizer st1 = new StringTokenizer("Hello Geeks How are you", " ");
while (st1.hasMoreTokens())
     System.out.println(st1.nextToken());


System.out.println("Using Constructor 2 - ");
StringTokenizer st2 = new StringTokenizer("JAVA : Code : String", " :");
while (st2.hasMoreTokens())
     System.out.println(st2.nextToken());


System.out.println("Using Constructor 3 - ");
StringTokenizer st3 = new StringTokenizer("JAVA : Code : String", " :",  true);
while (st3.hasMoreTokens())
     System.out.println(st3.nextToken());
```

Output :

```
Using Constructor 1 -
```

```
Hello
Geeks
How
are
you
Using Constructor 2 -
JAVA
Code
String
Using Constructor 3 -
JAVA

:

Code

:

String
```

Method of this class-

```
public boolean hasMoreTokens()
Returns: True if and only if next token to the current
         position in the string exists, else false.
```

```
public String nextToken()
Return: the next token from the given StringTokenizer
         if present.
Throws: NoSuchElementException - if no more token are left.
```

```
public int countTokens()
```

```
Return : the number of tokens remaining in the
         string using the current delimiter set.
```

```
public Object nextElement() -> Work similar to nextToken() except it
returns Object rather than String.
```

```
public Object hasMoreElement() -> Work similar to hasMoreToken() except it
returns Object rather than String.
```

85) How to initialize and compare the strings?

Ans-

a) **Direct Initialization (String Constant)** : In this method, a String constant object will be created in String pooled area which is inside heap area in memory. As it is a **constant**, we can't modify it, i.e. String class is **immutable**.

Examples:

```
String str = "GeeksForGeeks";


str = "geeks"; // This statement will make str
        // point to new String constant("geeks")
        // rather than modifying the previous
        // String constant.
```

Note: If we again write **str = "GeeksForGeeks"** as next line, then it first check that if given String constant is present in String pooled area or not. If it present then str will point to it, otherwise creates a new String constant.

b) **Object Initialization (Dynamic)**: In this method, a String object will be created in heap area (not inside String pooled area as in upper case).

Note: If we again write **str = new String("very")**, then it will create a new object with value "very", rather than pointing to the available objects in heap area with same value

Comparing Strings and their References

A. **equals() method:** It compares **values** of string for equality. Return type is boolean. In almost all the situation you can use useObjects.equals().

B. **== operator:** It compares **references not values**. Return type is boolean. == is used in rare situations where you know you're dealing with interned strings.

C. **compareTo() method:** It compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.For example, if str1 and str2 are two string variables then if
str1 == str2 : return 0
str1 > str2 : return a positive value
str1 < str2 : return a negative value

Example:

```
String s1 = "Ram";
String s2 = "Ram";
String s3 = new String("Ram");
String s4 = new String("Ram");
String s5 = "Shyam";

System.out.println(s1.equals(s2)); -> True
System.out.println(s1.equals(s3)); -> True
System.out.println(s1.equals(s5)); -> False
// System.out.println(nulls1.equals(nulls2)); -> Null Pointer Exception

System.out.println(s1 == s2); -> true
System.out.println(s1 == s3); -> False
System.out.println(s3 == s4); -> False
System.out.println(nulls1 == nulls2); ->True

 System.out.println(s1.compareTo(s3)); -> 0
System.out.println(s1.compareTo(s5)); -> -1
// System.out.println(nulls1.compareTo(nulls2)); ->NPE
```

86) String, StringBuilder and StringBuffer?

Ans-

- If a string is going to remain constant throughout the program, then use String class object because a String object is immutable.
- If a string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a StringBuilder is good enough.
- If a string can change, and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous so you have thread-safety.

Conversion between types of strings in Java

a) **From String to StringBuffer and StringBuilder :** Just pass the string object to either one of them and it became editable.

```
String str = "Geeks";
StringBuffer sbr = new StringBuffer(str); // From string to stringBuffer
sbr.reverse();
System.out.println(sbr);
```

```
// conversion from String object to StringBuilder
StringBuilder sbl = new StringBuilder(str);
sbl.append("ForGeeks");
System.out.println(sbl);
```

b) **From StringBuffer and StringBuilder to String :** This conversions can be perform using *toString()* method which is overridden in both StringBuffer and StringBuilder classes.

```
StringBuffer sbr = new StringBuffer("Geeks");
StringBuilder sbdr = new StringBuilder("Hello");

// conversion from StringBuffer object to String
String str = sbr.toString();
System.out.println("StringBuffer object to String : ");
System.out.println(str);

// conversion from StringBuilder object to String
String str1 = sbdr.toString();
System.out.println("StringBuilder object to String : ");
System.out.println(str1);
```

c) **From StringBuffer to StringBuilder or vice-versa :** This conversion is tricky.There is no direct way to convert the same. In this case, We can use a String class object. We first convert StringBuffer/StringBuilder object to String using *toString()* method and then from String to StringBuilder/StringBuffer using constructors.

87) Integer to String conversion and vice-versa

Ans- Many ways for converting Integer to String-

a) **Integer.toString(int)**
b) **String.valueOf(int)**
c) Integer obj = new Integer(d);
String str4 = obj.toString();

From String to Integer

public static int **parseInt(String s)** throws NumberFormatException

- This function parses the string argument as a signed decimal integer.

public static int **parseInt(String s, int radix)** throws NumberFormatException

- This function parses the string argument as a signed integer in the radix specified by the second argument.

Exceptions:

NumberFormatException is thrown by this method if any of the following situations occurs:

For both the variants:

- String is null or of zero length
- The value represented by the string is not a value of type int
- Specifically for the parseInt(String s, int radix) variant of the function:
  - The second argument radix is either smaller than Character.MIN_RADIX or larger than Character.MAX_RADIX
  - Any character of the string is not a digit of the specified radix, except that the first character may be a minus sign '-' ('\u002D') or plus sign '+' ('\u002B') provided that the string is longer than length 1

```
parseInt("20") returns 20
parseInt("+20") returns 20
parseInt("-20") returns -20
parseInt("20", 16) returns 16, (2)*16^1 + (0)*16^0 = 32
parseInt("2147483648", 10) throws a NumberFormatException
parseInt("99", 8) throws a NumberFormatException
                  as 9 is not an accepted digit of octal number system
parseInt("geeks", 28) throws a NumberFormatException
parseInt("geeks", 29) returns 11670324,
                  Number system with base 29 can have digits 0-9
                  followed by characters a,b,c... upto s
parseInt("geeksforgeeks", 29) throws a NumberFormatException as the
                              result is not an integer.
```

88) Swap two string without using third variables.

Ans-

```
// append 2nd string to 1st
a = a + b;

// store intial string a in string b
b = a.substring(0,a.length()-b.length());

// store initial string b in string a
a = a.substring(b.length());
```

89) Searching the character and substring in String?

Ans- there are following methods for searching character in String

a) It starts searching from beginning to the end of the string (from left to right) and returns the corresponding index if found otherwise returns -1.

```
int indexOf(char c)
```

Note: If given string contains multiple occurrence of specified character then it returns index of only first occurrence of specified character.

b) It starts searching backward from end of the string

```
public int lastIndexOf(char c)

// Accepts character as argument, Returns an

// index of the last occurrence specified

// characte
```

c) Returns the character existing at the specified index, indexNumber in the given string. If the specified index number does not exist in the string, the method throws an unchecked exception, StringIndexOutOfBoundsException.
Syntax:

```
char charAt(int indexNumber)
```

Searching Substring in the String

- The methods used for searching a character in the string which are mentioned above can also be used for searching the substring in the string.

    - int firstIndex = str.indexOf("Geeks");
    - System.out.println("First occurrence of char Geeks"+
        - " is found at : " + firstIndex);

- **contains(CharSequence seq):** It returns true if the String contains the specified sequence of char values otherwise returns false.

88) Compare two strings in Java?

Ans-

a) **Using String.equals() :**In Java, string equals() method compares the two given strings based on the data/content of the string

```
str1.equals(str2);
```

b) **String.equalsIgnoreCase() :** The String.equalsIgnoreCase() method compares two strings irrespective of the case (lower or upper) of the string

c) **Using String.compareTo() :**

Syntax:

```
int str1.compareTo(String str2)
```

Working:

It compares and returns the following values as follows:

i. if (string1 > string2) it returns a **positive value**.
ii. if both the strings are equal lexicographically i.e.(string1 == string2) it returns **0**.
iii. if (string1 < string2) it returns a **negative value**.

Note: One can use == operators for reference comparison **(address comparison)** and .equals() method for **content comparison**.

90) Reverse a string?

Ans-

A) Converting String into Bytes: getBytes() method is used to convert the input string into bytes[].

Method:

```
// getBytes() method to convert string into bytes[].
byte [] strAsByteArray = input.getBytes();

//temp array to hold the result.
byte [] result = new byte [strAsByteArray.length];

// Store result in reverse order into the result byte[]
for (int i = 0; i<strAsByteArray.length; i++)
      result[i] = strAsByteArray[strAsByteArray.length-i-1];
```

B) **Using built in reverse() method of the StringBuilder class-**

```
StringBuilder input1 = new StringBuilder();

// append a string into StringBuilder input1
input1.append(input);

// reverse StringBuilder input1
input1 = input1.reverse();
```

C) **Converting String to character array:** using the built in Java String class method toCharArray().

```
// convert String to character array by using toCharArray
char[] try1 = input.toCharArray();

for (int i = try1.length-1; i>=0; i--)
      System.out.print(try1[i]);
```

91) Removing leading zeros from String?

Ans- We use StringBuffer class as Strings are immutable.

1) Count leading zeros.
2) Use StringBuffer replace function to remove characters equal to above count.

```
// Count leading zeros

int i = 0;
while (i < str.length() && str.charAt(i) == '0')
    i++;

// Convert str into StringBuffer as Strings are immutable.
StringBuffer sb = new StringBuffer(str);

// The  StringBuffer replace function removes
// i characters from given index (0 here)
sb.replace(0, i, "");
```

92) Remove the leading and trailing spaces?

Ans- Using trim() method which is defined under the String class of java.lang package.

- It does not eliminated the middle spaces of the string.
- By calling the trim() method, a new String object is returned.

93) Count no of words, lines and characters in text file

Ans-

```
File file = new File("C:\\Users\\Mayank\\Desktop\\1.txt");
FileInputStream fileStream = new FileInputStream(file);
InputStreamReader input = new InputStreamReader(fileStream);
BufferedReader reader = new BufferedReader(input);
String line;

// Initializing counters
int countWord = 0;
int sentenceCount = 0;
int characterCount = 0;
int paragraphCount = 1;
int whitespaceCount = 0;

// Reading line by line from the file until a null is returned
while((line = reader.readLine()) != null)
{
if(line.equals(""))
{
      paragraphCount++;
}
if(!(line.equals("")))
{
```

```
            characterCount += line.length();

            // \\s+ is the space delimiter in java
            String[] wordList = line.split("\\s+");

            countWord += wordList.length;
            whitespaceCount += countWord -1;

            // [!?.:]+ is the sentence delimiter in java
            String[] sentenceList = line.split("[!?.:]+");

            sentenceCount += sentenceList.length;
        }
        }

        System.out.println("Total word count = " + countWord);
        System.out.println("Total number of sentences = " + sentenceCount);
        System.out.println("Total number of characters = " + characterCount);
        System.out.println("Number of paragraphs = " + paragraphCount);
        System.out.println("Total number of whitespaces = " + whitespaceCount);
```

94) Check if string contains only alphabets?

Ans- Match the string with the Regex using matches(). And the regex is ***^[a-zA-Z]*$***

```
        public static boolean isStringOnlyAlphabet(String str)

        {
            return ((!str.equals(""))
                    && (str != null)
                    && (str.matches("^[a-zA-Z]*$")));
        }
```

95) a string contains only alphabets or not using ASCII values?

Ans-

```
        if (str == null || str.equals("")) {
              return false;
        }
        for (int i = 0; i < str.length(); i++) {
              char ch = str.charAt(i);
              if ((!(ch >= 'A' && ch <= 'Z')) && (!(ch >= 'a' && ch <= 'z'))) {
                    return false;
              }
        }
        return true;
```

96) Arrays in JAVA?

Ans- In Java all arrays are dynamically allocated. So we can find their length using member length.

The **size** of an array must be specified by an int value and not long or short.

In case of primitives data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.

```
type var-name[];
OR
type[] var-name;
```

When an array is declared, only a reference of array is created. To actually create or give memory to array, do this:

```
var-name = new type [size];
or in one step, do this
type[] var-name= new type [size];
```

Note: The elements in the array allocated by *new* will automatically be initialized to **zero** (for numeric types), **false** (for boolean), or **null** (for reference types).

Array literals
When the size of the array and variables of array are already known, array literals can be used.

```
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
 or
int[] intArray = { 1,2,3,4,5,6,7,8,9,10 };
```

Multidimensional Arrays

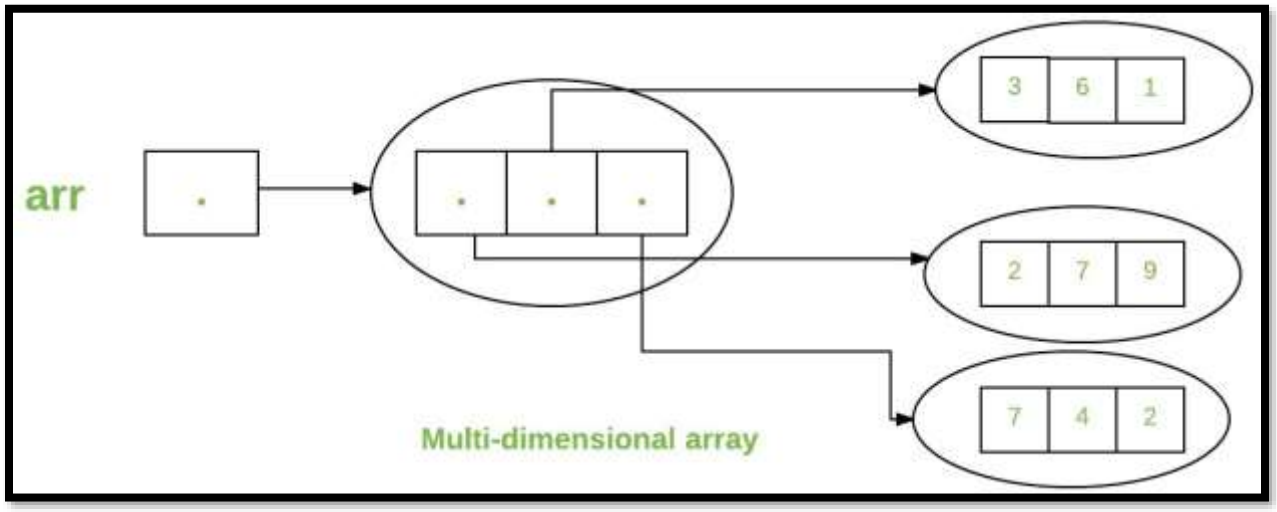
Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array. These are also known as Jagged Arrays.

```
int[][] intArray = new int[10][20]; //a 2D array or matrix
int[][][] intArray = new int[10][20][10]; //a 3D array
```

```
// declaring and initializing 2D array
int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

// printing 2D array
for (int i=0; i< 3 ; i++)
{
for (int j=0; j < 3 ; j++)
System.out.print(arr[i][j] + " ");

System.out.println();
}
```

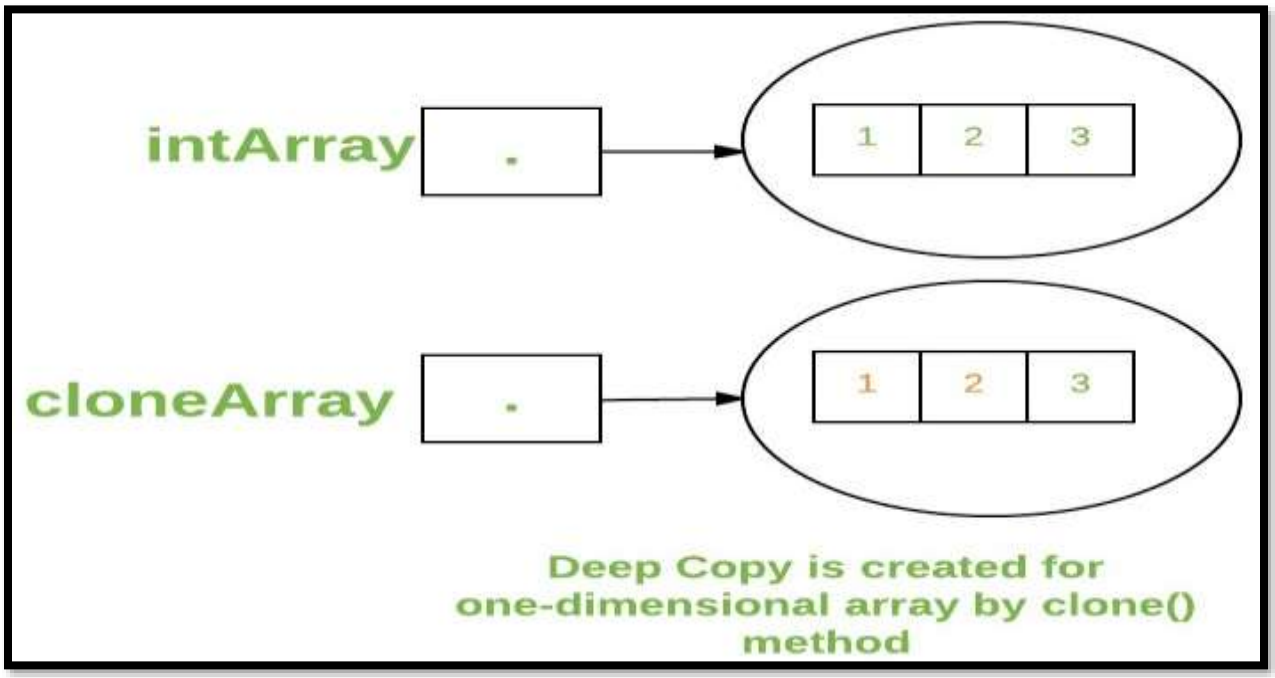


Note: We can pass array as a parameter to method and also return it.

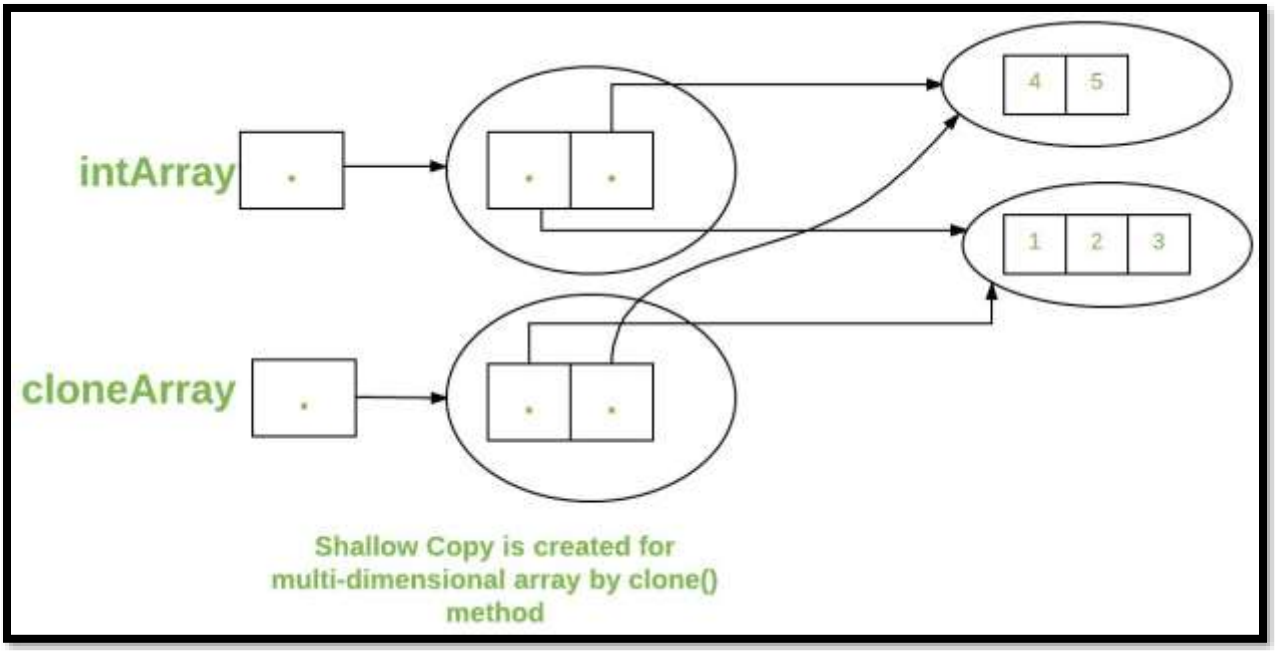
Cloning of arrays

- When you clone a single dimensional array, such as Object[], a “deep copy” is performed with the new array containing copies of the original array’s elements as opposed to references.

```
//Will printfalse as deep copy is created for one-dimensional array
System.out.println(intArray == cloneArray);
```



- clone of a multidimensional array (like Object[][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array but subarrays are shared.



97) Array class in JAVA?

Ans- The **Arrays** class in **java.util package** is a part of the **Java Collection Framework**. This class provides static methods to dynamically create and access **Java arrays.**

Methods in Java Array:

a) **static <T> List<T> asList(T... a)**: This method returns a fixed-size list backed by the specified Arrays.

```
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };

        // To convert the elements as List
        System.out.println("Integer Array as List: "
                           + Arrays.asList(intArr));
    }
```

b) **static int binarySearch(elementToBeSearched):**

```
// Get the Array
  int intArr[] = { 10, 20, 15, 22, 35 };

  Arrays.sort(intArr);

  int intKey = 22;
  System.out.println(intKey+ " found at index = "+ Arrays
                              .binarySearch(intArr, intKey)); }
```

98) A final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of array can be modified.

```
int p = 20;
public static void main(String args[])
{
   final Test t1 = new Test();
   Test t2 = new Test();
   t1 = t2;
   System.out.println(t1.p);
}
```

Output: Compiler Error: cannot assign a value to final variable t1

```
int p = 20;
    public static void main(String args[])
    {
       final Test t = new Test();
       t.p = 30;
       System.out.println(t.p);
    }
}
```

Output: 30

99) Interesting facts about array assignment

a) For primitive data types : In case of primitive type arrays, as array elements we can provide any type which can be **implicitly promoted to the declared type array**. Apart from that, if we are trying to use any other data-types then we will get compile-time error saying possible loss of precision.

```
int[] arr = new int[3];
arr[0] = 1;
arr[1] = 'a'; //will store 97 here
byte b = 10;
arr[2] = b; //will store 10 as it is

// Assigning long value to int type. ->Possible loss of precision.
arr[0] = 10l;
```

b) **Object type Arrays :** If we are creating object type arrays then the elements of that arrays can be either declared type objects or it can be child class object.

```
Number[] num = new Number[2];
num[0] = new Integer(10);
num[1] = new Double(20.5);

// Here String is not the child class of Number class.
num[2] = new String("GFG");  //Compile time error- incompatible type
```

100) Jagged Arrays?

Ans- It is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row

```
// Declaring 2-D array with 2 rows
        int arr[][] = new int[2][];

        // Making the above array Jagged

        // First row has 3 columns
        arr[0] = new int[3];

        // Second row has 2 columns
        arr[1] = new int[2];

        // Initializing array
        int count = 0;
        for (int i=0; i<arr.length; i++) {
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;
    }
```

101) Array vs ArrayList?

Ans-

a) Array: Simple fixed sized arrays that we create in Java, like below

```
int arr[] = new int[10]
```

ArrayList : Dynamic sized arrays in Java that implement List interface. One need not to mention the size of Arraylist while creating its object. Even if we specify some initial capacity, we can add more elements.

```
ArrayList<Type> arrL = new ArrayList<Type>();

Here Type is the type of elements in ArrayList to be created
```

b) Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.
Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object.

c) In array, it depends whether the arrays is of primitive type or object type. In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

Since ArrayList can't be created for primitive data types, members of ArrayList are always references to objects at different memory locations

102) Java provides a direct method *Arrays.equals()* to compare two arrays. the Arrays.equals() works fine and compares arrays contents. Now the questions, what if the arrays contain arrays inside them or some other references which refer to different object but have same values.

So *Arrays.equals()* is not able to do deep comparison. Java provides another method for this Arrays.deepEquals() which does deep comparison.

103) ArrayList to array conversion in JAVA?

Ans- public Object[] toArray() -It returns an array containing all of the elements in this list in the correct order. We need to typecast it to Integer before using as Integer objects.

104) **Returning multiple values in JAVA?**

Ans- Java doesn't support multi-value returns. We can use following solutions to return multiple values.

a) If all returned element are of same type, then use array.
b) If returned elements are of different type and are only 2 returned values then use Pair class which comes in JAVA 8-

```
// Returning a pair of values from a function
public static Pair<Integer, String> getTwo()
 {
     return new Pair<Integer, String>(10, "GeeksforGeeks");
 }

 // Return multiple values from a method in Java 8
 public static void main(String[] args)
 {
     Pair<Integer, String> p = getTwo();
     System.out.println(p.getKey() + " " + p.getValue());
 }
```

Methods provided by the javafx.util.Pair class

- **Pair (K key, V value) :** Creates a new pair
- **boolean equals() :** It is used to compare two pair objects. It does a deep comparison, i.e., it compares on the basic of the values (<Key, Value>) which are stored in the pair objects.
- **String toString() :** This method will return the String representation of the Pair.
- **K getKey() :** It returns key for the pair.
- **V getValue() :** It returns value for the pair.
- **int hashCode() :** Generate a hash code for the Pair.

c) **If there are more than two returned values**
We can encapsulate all returned types into a class and then return an object of that class.

d) **Returning list of Object class**

```
public static List<Object> getDetails()
{
	String name = "Geek";
	int age = 35;
	char gender = 'M';

	return Arrays.asList(name, age, gender);
}
```

104) Throwable fillInStackTrace()?

Ans- The **fillInStackTrace()** method, of **java.lang.Throwable** class, records within this Throwable object information about the current state of the stack frames for the current thread. It means using this method one can see the Exception messages of the current method of a class, in which fillInStackTrace() method is called.

Using fillInStackTrace() only returns information of the active state of frames for the current thread. So when fillInStackTrace() is called, then the method returns details up to main method in which fillInStackTrace() method was called.

105) Main() method variant?

Ans-

public: JVM can execute the method from anywhere.
static: Main method can be called without object.
void: The main method doesn't return anything.
main(): Name configured in the JVM.
String[]: Accepts the command line arguments.

a) **Order of Modifiers:** We can swap positions of static and public in main().

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

b) We can place square brackets at different positions and we can use varargs (…) for string parameter.

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

c) We can make String args[] as final or We can make main() as final

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

d) We can make main() synchronized.

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

e) strictfp can be used to restrict floating point calculations.

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

f) **Combinations of all above keyword to static main method:**

```
final static synchronized strictfp static void main(String[] args)
```

g) **Overloading Main method:** We can overload main() with different types of parameters

```
public static void main(String[] args)
{
      System.out.println("Main Method String Array");
}
public static void main(int[] args)
{
      System.out.println("Main Method int Array");
}
```

h) **Method Hiding of main(), but not Overriding:** Since main() is static, derived class main() hides the base class main

```
class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Parent");
    }
}
class B extends A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Child");
    }
}
```

Note: Two classes, A.class and B.class are generated by Java Compiler javac. When we execute both the .class, JVM executes with no error.

106) Variable arguments in Java?

Ans- Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. Prior to JDK 5, variable length arguments could be handled into two ways- One was using overloading, other was using array argument.

```
    static void fun(int ...a)
        {
            System.out.println("Number of arguments: " + a.length);

            // using for each loop to display contents of a
            for (int i: a)
                System.out.print(i + " ");
            System.out.println();
        }

// Calling the varargs method with different number of parameters
       fun(100);         // one parameter
       fun(1, 2, 3, 4);  // four parameters
       fun();            // no parameter
```

The … syntax tells the compiler that varargs has been used and these arguments should be stored in the **array referred to by a**.

Note: A method can have variable length parameters with other parameters too, but one should ensure that there exists only one varargs parameter that should be written last in the parameter list of the method declaration.

```
int nums(int a, float b, double … c)
```

Error varags Example:

a) Specifying two varargs in a single method:

```
void method(String... gfg, int... q)
{
// Compile time error as there are two varargs
}
```

b) Specifying varargs as the first parameter of method instead of last one:

```
void method(int... gfg, String q){
// Compile time error as vararg appear before normal argument
}
```

107) Overloading main() in Java?

Ans- The main method in Java is no extra-terrestrial method. Apart from the fact that main() is just like any other method & can be overloaded in a similar manner, JVM always looks for the method signature to launch the program.

- The normal main method acts as an entry point for the JVM to start the execution of program.
- We can overload the main method in Java. But the program doesn't execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

```
// Normal main()
   public static void main(String[] args) {
       System.out.println("Hi Geek (from main)");
       Test.main("Geek");
   }

   // Overloaded main methods
   public static void main(String arg1) {
       System.out.println("Hi, " + arg1);
       Test.main("Dear Geek","My Geek");
   }
   public static void main(String arg1, String arg2) {
       System.out.println("Hi, " + arg1 + ", " + arg2);
   }
```

108) Overriding the toString()?

Ans-

```
class Complex {

  private double re, im;

  public Complex(double re, double im) {
  this.re = re;
  this.im = im;
}
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

Output: Complex@19821f

The default toString() method in Object prints "class name @ hash code". We can override toString() method in our class to print proper output.

```
/* Returns the string representation of this Complex number. The format of
string is "Re + iIm" where Re is real part and Im is imagenary part.*/
    @Override
    public String toString() {
        return String.format(re + " + i" + im);
    }
```

109) Since private methods are inaccessible, they are implicitly final in Java. So adding *final* specifier to a private method doesn't add any value. It may in-fact cause unnecessary confusion.

110) **Although JAVA strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed. Java is kind of hybrid between pass-by-value and pass-by-reference**.

Ans- Java is strictly pass by value because We pass an int to the function "change()" and as a result the change in the value of that integer is not reflected in the main method. Like C/C++, Java creates a copy of the variable being passed in the method and then do the manipulations. Hence the change is not reflected in the main method. This happens in the case with primitives.

While in case of non-primitive, If we do not change the reference to refer some other object (or memory location), we can make changes to the members and these changes are reflected back as it creates a copy of references and pass it to method, but they still point to same memory reference.

When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

111) **java.lang.System.exit()** ?

Ans- This method exits current program by terminating running Java virtual machine. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination.

public static void exit(int status)

exit(0) : Generally used to indicate successful termination.
exit(1) or exit(-1) or any other non-zero value – Generally indicates unsuccessful termination.
Note : This method does not return any value.

112) Constructor in JAVA?

Ans-

- A constructor in Java cannot be abstract, final, static and Synchronized.
- There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.
- Constructor(s) must have the same name
- Constructor is called only once at the time of Object creation and are used to initialize the object's state
- Java automatically creates default constructor if there is no default or parameterized constructor written by user,
- the default constructor automatically calls parent default constructor.
- default constructor in Java initializes member data variable to default values (numeric values are initialized as 0, booleans are initialized as *false* and references are initialized as *null*

113) In Java, non-static final variables can be assigned a value either in constructor or with the declaration.

But, static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

```
// Since i is static final, it must be assigned value here or inside
static block.

 static final int i;
 static
 {
     i = 10;
 }
```

114) Copy Constructor?

Ans- Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

```
// copy constructor
Complex(Complex c) {
    System.out.println("Copy constructor called");
    re = c.re;
    im = c.im;
}
```

```
Complex c1 = new Complex(10, 15);

// Following involves a copy constructor call
Complex c2 = new Complex(c1);

// Note that following doesn't involve a copy constructor call as
// non-primitive variables are just references.
Complex c3 = c2;
```

115) Constructor chaining in JAVA?

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

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

Constructor chaining occurs through **inheritance**. A sub class constructor's task is to call super class's constructor first. This ensures that creation of sub class's object starts with the initialization of the data members of the super class.

Why do we need constructor chaining?

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Rules of constructor chaining:

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.
4. Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

Alternative method : using Init block :

When we want certain common resources to be executed with every constructor we can put the code in the **init block**. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

116) Private constructor and Singleton classes?

Ans-

Do we need such 'private constructors ' ?

There are various scenarios where we can use private constructors. The major ones are

1. Internal Constructor chaining
2. Singleton class design pattern

What is a Singleton class?

As the name implies, a class is said to be singleton if it limits the number of objects of that class to one.

We can't have more than a single object for such classes.

Singleton classes are employed extensively in concepts like Networking and Database Connectivity.

```
class MySingleton
{
    static MySingleton instance = null;
    public int x = 10;

    // private constructor can't be accessed outside the class
    private MySingleton() {  }

    // Factory method to provide the users with instances
    static public MySingleton getInstance()
    {
        if (instance == null)
             instance = new MySingleton();

        return instance;
    }
}
```

117) Interview question on Constructor?

Ans-

a) **Do we have destructors in Java?**
No, Because Java is a garbage collected language you cannot predict when (or even if) an object will be destroyed. Hence there is no direct equivalent of a destructor.

b) **Can we call sub class constructor from super class constructor?**
No. There is no way in java to call sub class constructor from a super class constructor.

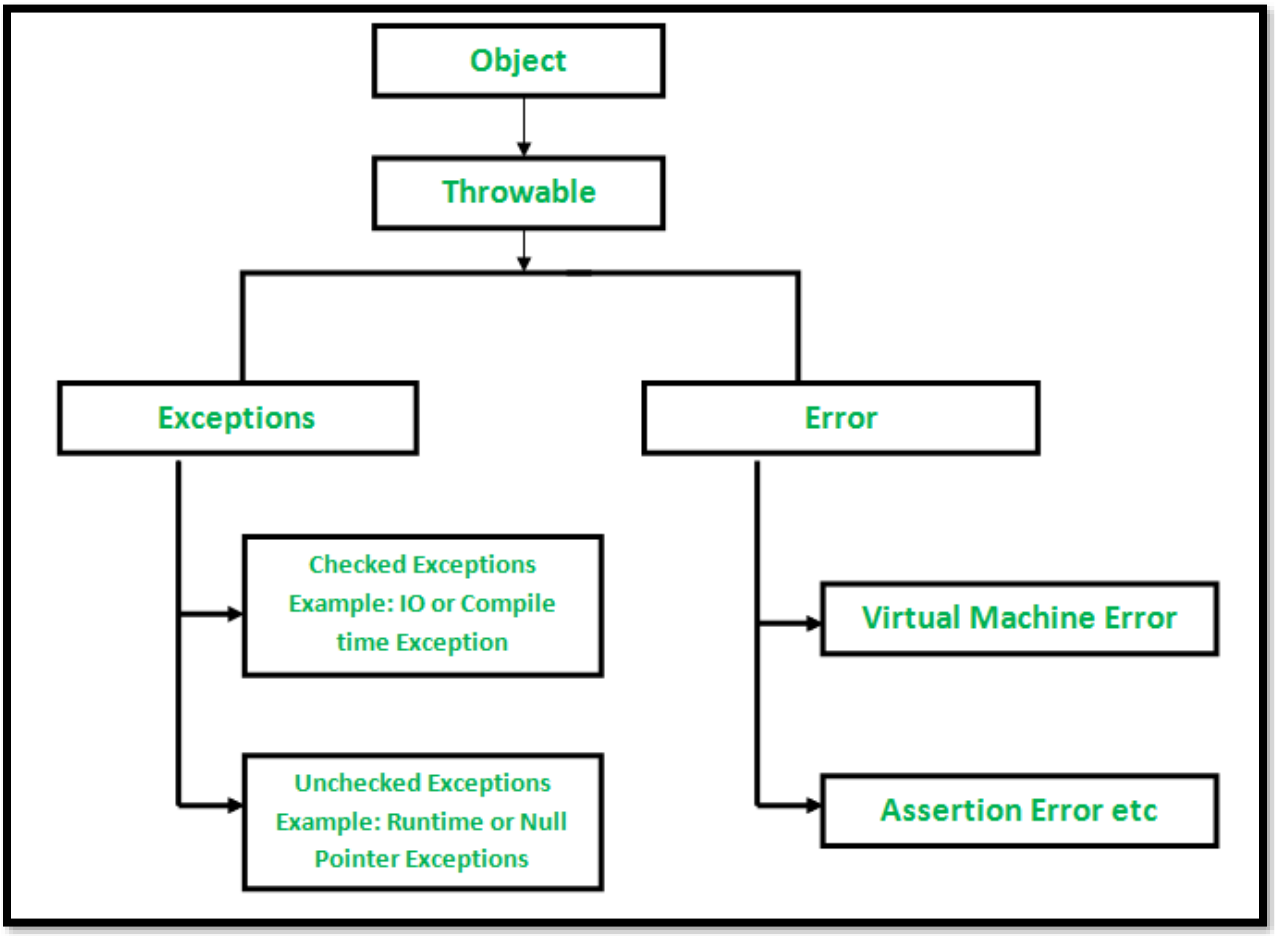
c) **What happens if you keep a return type for a constructor?**
Ideally, Constructor must not have a return type. By definition, if a method has a return type, it's not a constructor.(JLS8.8 Declaration) It will be treated as a normal method. But compiler gives a warning saying that method has a constructor name. Example:

```
class GfG
{
    int GfG()
    {
        return 0;    // Warning for the return type
    }
```

d) **When do we need Constructor Overloading?**
Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. For example, Thread class has 8 types of constructors. If we do not want to specify anything about a thread then we can simply use default constructor of Thread class, however if we need to specify thread name, then we may call the parameterized constructor of Thread class with a String args like this:

118) What is exception?

Ans- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.



How JVM handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack.** Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.

- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.

- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.

- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler** , which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

```
Exception in thread "xxx" Name of Exception : Description
... ..... ..  // Call Stack
```

How Programmer handles an exception?

Customized Exception Handling : Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner.

System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw.

Any exception that is thrown out of a method must be specified as such by a throws clause.

Any code that absolutely must be executed after a try block completes is put in a finally block.

- For each try block there can be zero or more catch blocks, but **only one** finally block.

- The finally block is optional. It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

119) Error vs Exception?

Ans- **Error:** An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

120) OutOfMemoryError Exception?

Ans-This error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

OutOfMemoryError usually means that you're doing something wrong, either holding onto objects too long, or trying to process too much data at a time. Sometimes, it indicates a problem that's out of your control, such as a third-party library that caches strings, or an application server that doesn't clean up after deploys. And sometimes, it has nothing to do with objects on the heap.

The **java.lang.OutOfMemoryError exception** can also be thrown by native library code when a native allocation cannot be satisfied (for example, if swap space is low).

121) Three different ways to print exception?

Ans-

a) **java.lang.Throwable.printStackTrace() method :** By using this method, we will get name(e.g. java.lang.ArithmeticException) and description(e.g. / by zero) of an exception separated by colon, and stack trace (where in the code, that exception has occurred) in the next line.

Syntax:

```
public void printStackTrace()
```

Runtime Exception:

```
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:9)
```

b) **toString() method :** By using this method, we will only get name and description of an exception. Note that this method is overridden in Throwable class.

Output:

```
java.lang.ArithmeticException: / by zero
```

c) **java.lang.Throwable.getMessage() method :** By using this method, we will only get description of an exception.

Syntax:

```
public String getMessage()
```

Output:

```
/ by zero
```

122) Try catch finally block?

Ans-

Flow-1

```
try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement

            System.out.println("Inside try block");
        }

        catch(ArrayIndexOutOfBoundsException ex)
        {
            System.out.println("Exception caught in catch block");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will be executed
        System.out.println("Outside try-catch-finally clause");
    }
}
```

Output:

```
Exception caught in catch block

finally block executed

Outside try-catch-finally clause
```

Flow-2

```
 try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        // not a appropriate handler
        catch(NullPointerException ex)
        {
            System.out.println("Exception has been caught");
        }

        finally
```

```
        {
            System.out.println("finally block executed");
        }

        // rest program will not execute
        System.out.println("Outside try-catch-finally clause");
}
```

Output :

```
finally block executed
```

Run Time error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:12)
```

123) Exception type?

Ans- Java exception are of 2 type- Build-in exception and user defined exception.

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

I. **ArithmeticException**

It is thrown when an exceptional condition has occurred in an arithmetic operation.

Example:

```
int a = 30, b = 0;
int c = a/b;  // cannot divide by zero
```

II. **ArrayIndexOutOfBoundsException**

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Example:

```
int a[] = new int[5];
a[6] = 9; // accessing 7th element in an array of size 5
```

III. **ClassNotFoundException**

This Exception is raised when we try to access a class whose definition is not found

IV. **FileNotFoundException**

This Exception is raised when a file is not accessible or does not open.

V. **IOException**

It is thrown when an input-output operation failed or interrupted

VI. **InterruptedException**

It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.

VII. **NoSuchFieldException**

It is thrown when a class does not contain the field (or variable) specified

VIII. **NoSuchMethodException**

It is thrown when accessing a method which is not found.

IX. **NullPointerException**

This exception is raised when referring to the members of a null object. Null represents nothing

X. **NumberFormatException**

This exception is raised when a method could not convert a string into a numeric format.

Example:

```
// "akki" is not a number
int num = Integer.parseInt ("akki") ;
```

XI. **RuntimeException**

This represents any exception which occurs during runtime.

XII. **StringIndexOutOfBoundsException**

It is thrown by String class methods to indicate that an index is either negative than the size of the string

124) Catching base and derived class exception?

Ans- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

In Java, catching a base class exception before derived is not allowed by the compiler itself.

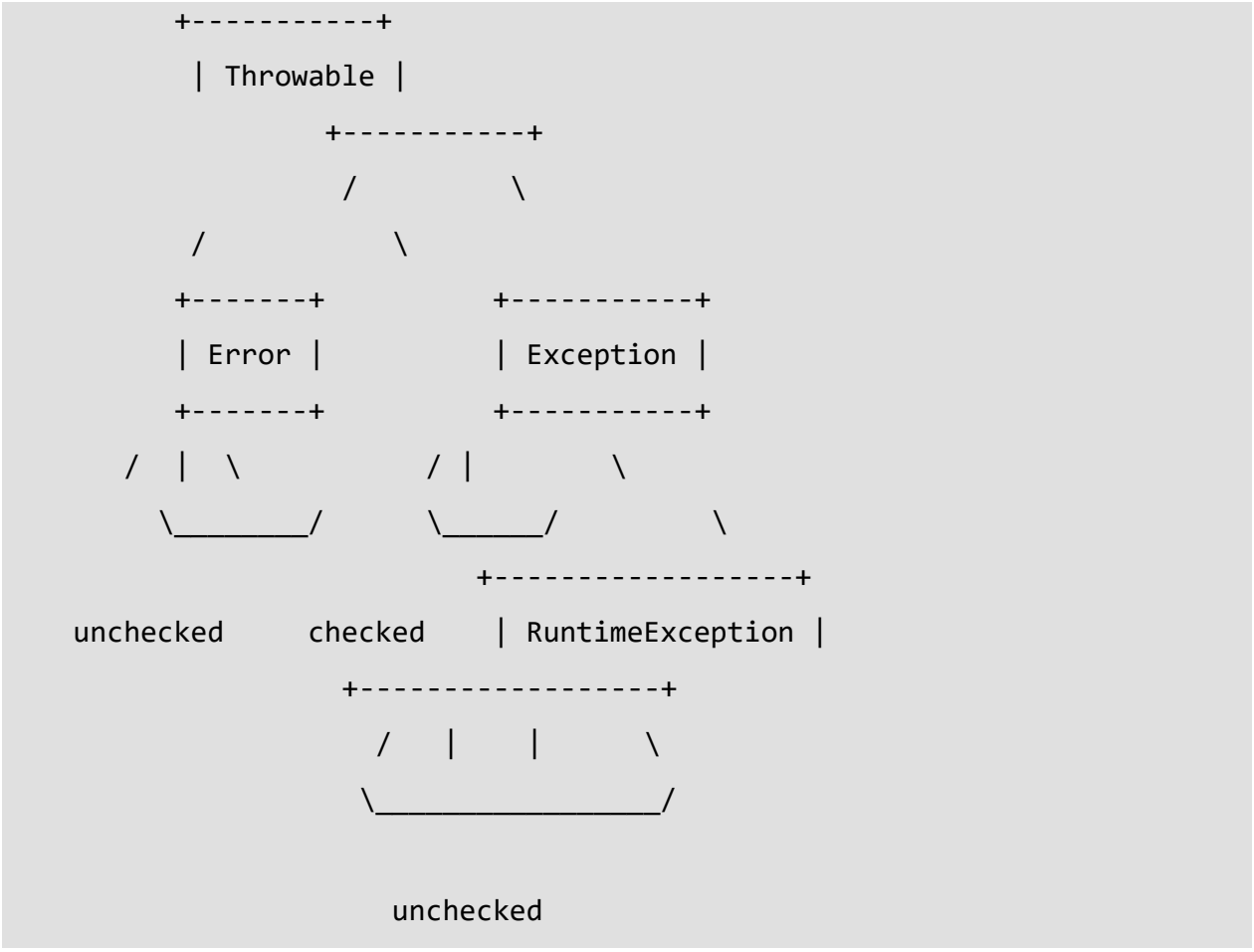
```
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
       throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
} -> fails in compilation with error message "exception Derived has already been caught"
```

125) Checked vs unchecked exception?

Ans- **Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

For example, consider the following Java program that opens file at location "C:\test\a.txt" and prints the first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException.* To fix this, we either need to specify list of exceptions using throws, or we need to use try-catch block.

Unchecked are the exceptions that are not checked at compiled time. In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.



If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

126) Throw and Throws in Java?

Ans- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

```
throw Instance
Example:
throw new ArithmeticException("/ by zero");
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

```
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");

            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Output:

```
Caught inside fun().

Caught in main.
```

throws

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

Important points to remember about throws keyword:

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program while try-catch prevents it.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

127) User-defined exception?

Ans- Sometimes, the built-in exceptions in Java are not able to describe a certain situation like if balance in the account goes below 100rs, the program should throw the exception. In such cases, user can also create exceptions which are called 'user-defined Exceptions'.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
```

```
}
```

Output:

```
Caught
GeeksGeeks
```

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

128) Infinity or exception by 0?

Ans-

```
double p = 1;
System.out.println(p/0);        -> Infinity

int p = 1;
System.out.println(p/0);        -> Exception in thread "main":arithmetic
```

- In case of double/float division, the output is **Infinity**, the basic reason behind that it implements the floating point arithmetic algorithm which specifies a special values like "Not a number" OR "infinity" for "divided by zero cases" as per IEEE 754 standards.
- In case of integer division, it throws ArithmeticException.

129) Multicatach in java?

Ans-

Prior to Java 7, we had to catch only one exception type in each catch block. So whenever we needed to handle more than one specific exception, but take same action for all exceptions, then we had to have more than one catch block containing the same code

In <= Java 6.0

```
try
  {
      int n = Integer.parseInt(scn.nextLine());
      if (99%n == 0)
          System.out.println(n + " is a factor of 99");
  }
  catch (ArithmeticException ex)
  {
      System.out.println("Arithmetic " + ex);
  }
  catch (NumberFormatException ex)
  {
      System.out.println("Number Format Exception " + ex);
  }
```

In >= Java 7.0 it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in catch block.

```
try
  {
      int n = Integer.parseInt(scn.nextLine());
      if (99%n == 0)
          System.out.println(n + " is a factor of 99");
  }
  catch (NumberFormatException | ArithmeticException ex)
  {
      System.out.println("Exception encountered " + ex);
  }
```

Important points:

- A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler, that is, the bytecode has no replication of exception handlers.

- Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions cannot be caught in one statement. For Example

```
// Not Valid as Exception is an ancestor of
// NumberFormatException
catch(NumberFormatException | Exception ex)
```

130) Chained exception in JAVA?

Ans- Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception.

For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception.

Constructors Of Throwable class Which support chained exceptions in java :

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. getCause() method :- This method returns actual cause of an exception.

2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

```
try
        {
            // Creating an exception
            NumberFormatException ex =
                        new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                       "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
```

Output:

```
java.lang.NumberFormatException: Exception
java.lang.NullPointerException: This is actual cause of the exception
```

131) Interface in JAVA?

Ans- Interfaces specify what a class must do and not how. It is the blueprint of the class.

It is used to provide total abstraction. That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.

New features added in interfaces in JDK 8

I. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.
Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of

default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
/ An example to show that interfaces can
// have methods from JDK 1.8 onwards
interface in1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
    }
}

// A class that implements interface.
class testClass implements in1
{
    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
    }
}
```

Output :

```
hello
```

II. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```
/ An example to show that interfaces can
// have methods from JDK 1.8 onwards
interface in1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}

// A class that implements interface.
class testClass implements in1
{
    // Driver Code
    public static void main (String[] args)
    {
        in1.display();
    }
}
```

Output:hello

Important points about interface or summary of article:

- We can't create instance (interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final and they must be initialized.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.

New features added in interfaces in JDK 9

From Java 9 onwards, interfaces can contain following also

1. Static methods
2. Private methods
3. Private Static methods

132) the classes and interfaces themselves can have only two access specifiers when declared outside any other class.

1) public
2) default (when no access specifier is specified).

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
//filename: Main.java
protected class Test {}

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

  }
}
```

Note : Nested interfaces and classes can have all access specifiers.

133) Abstract class?

Ans-

a) an instance of an abstract class cannot be created, we can have references of abstract class type though.

b) an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

Output:

```
Base Constructor Called

Derived Constructor Called
```

c) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited

d) Abstract classes can also have final methods (methods that cannot be overridden).

Interview question on abstract class

a) Is it possible to create abstract methods in final class?

Ans- it's not possible to have an abstract method in a final class in Java. because as soon as you declare an abstract method in a Java class, as per Java specification, the class automatically becomes an abstract class and it's not possible to make an abstract class final in Java. It will throw a compile time error.

b) **can an abstract class have static methods in Java?**

Ans- The answer is yes, there is no problem with declaring a static method inside an abstract class in Java because you don't need to instantiate a class to use the static method, you can just call them by using the class name.

134) Difference between abstract class and interface?

Ans-

a) abstract keyword is used to create an abstract class and it can be used with methods also whereas interface keyword is used to create interface and it can't be used with methods.

b) Abstract classes can have methods with implementation whereas interface provides absolute abstraction and can't have any method implementations. Note that from Java

8 onwards, we can create default and static methods in interface that contains the method implementations.

c) Abstract classes can have constructors but interfaces can't have constructors.
d) Abstract classes methods can have access modifiers as public, private, protected, static but interface methods are implicitly public and abstract, we can't use any other access modifiers with interface methods.
e) A subclass can extend only one abstract class but it can implement multiple interfaces.
f) Abstract classes can extend other class and implement interfaces but interface can only extend other interfaces, it cannot implement other interfaces.
g) We can run an abstract class if it has main() method but we can't run an interface because they can't have main method implementation.

135) when Interfaces are the best choice and when can we use abstract classes?

Ans-

a) Java doesn't support multiple class level inheritance, so every class can extend only one superclass. But a class can implement multiple interfaces. So most of the times Interfaces are a good choice for providing the base for class hierarchy and contract. Also coding in terms of interfaces is one of the best practices for coding in java.
b) If there are a lot of methods in the contract, then abstract class is more useful because we can provide a default implementation for some of the methods that are common for all the subclasses. Also if subclasses don't need to implement a particular method, they can avoid providing the implementation but in case of interface, the subclass will have to provide the implementation for all the methods even though it's of no use and implementation is just empty block.
c) If our base contract keeps on changing then interfaces can cause issues because we can't declare additional methods to the interface without changing all the implementation classes, with the abstract class we can provide the default implementation and only change the implementation classes that are actually going to use the new methods.

Note: From Java 8 onwards, we can have method implementations in the interfaces. We can create default as well as static methods in the interfaces and provide an implementation for them. This has bridged the gap between abstract classes and interfaces and now interfaces are the way to go because we can extend it further by providing default implementations for new methods.

136) Comparator interface in JAVA?

Ans- Comparator interface is used to order the objects of user-defined classes.

A comparator object is capable of comparing two objects of two different classes.

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an array/arraylist of our own class type, containing fields like rollno, name, address, DOB etc and we need to sort the array based on Roll no or name?

Method 1: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criterion like Roll No. and Name.
Method 2: Using comparator interface- Comparator interface is used to order the objects of user-defined class. This interface is present in java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using comparator, we can sort the elements based on data members. For instance it may be on rollno, name, age or anything else.

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

```
// To sort a given list. ComparatorClass must implement
// Comparator interface.
public void sort(List list, ComparatorClass c)
```

How does Collections.Sort() work?
Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0 or 1 to say if it is less than, equal, or greater to the other.

```
// A class to represent a student.
class Student
{
    int rollno;
    String name, address;
    public Student(int rollno, String name, Sring address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }
      // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +  " " + this.address;
    }
}
```

```
class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

class Sortbyname implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i))

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

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

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

        System.out.println("\nSorted by name");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Output:

```
Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur
```

```
Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc

Sorted by name
131 aaaa nyc
111 bbbb london
121 cccc jaipu
```

Note: for descending order just change the positions of a and b in above compare method.

Sort collection by more than one field:

If we have a requirement to sort ArrayList objects in accordance with more than one fields like firstly sort, according to student name and secondly sort according to student age.

```
static class CustomerSortingComparator implements Comparator<Student> {

        @Override
        public int compare(Student customer1, Student customer2) {

            // for comparison
            int NameCompare =
customer1.getName().compareTo(customer2.getName());

            int AgeCompare =
customer1.getAge().compareTo(customer2.getAge());

            // 2-level comparison using if-else block
            if (NameCompare == 0) {
                return ((AgeCompare == 0) ? NameCompare : AgeCompare);
            } else {
                return NameCompare;
            }
        }
    }
```

137) **There is a rule that every member of interface is only and only public whether you define or not.** So when we define the method of the interface in a class implementing the interface, we have to give it public access as child class can't assign the weaker access to the methods. If we change fun() to anything other than public in class B, we get compiler error "attempting to assign weaker access privileges; was public"

138) We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.
The access specifier in above example is default. We can assign public, protected or private also.

Interface in class:

```
class Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}
```

Interface in interface:

```
interface Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}
```

Note: In the above example, access specifier is public even if we have not written public. If we try to change access specifier of interface to anything other than public, we get compiler error.

139) Inner class in Java?

Ans- Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) **Nested Inner class** can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.

   we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself.

2) **Method Local inner classes**: Inner class can be declared within a method of an outer class.

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");

        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
```

   Note : Local inner class cannot access non-final local variable till JDK 1.7*. It generates compiler error (Note that x is not final in outerMethod() and innerMethod() tries to access it)*

   Since JDK 1.8, it is possible to access the non-final local variable in method local inner class.

   Method local inner class can't be marked as private, protected, static and transient but can be marked as abstract and final, but not both at the same time.

3) **Anonymous inner classes:** Anonymous inner classes are declared without any name at all. They are created in two ways.

   a) *As subclass of specified type*

```
class Demo {
   void show() {
      System.out.println("i am in show method of super class");
   }
}
```

```
class Flavor1Demo {
   //  An anonymous class with Demo as base class
   static Demo d = new Demo() {
       void show() {
           super.show();
           System.out.println("i am in Flavor1Demo class");
       }
   };
   public static void main(String[] args){
       d.show();
   }
}
```

Output

```
i am in show method of super class
i am in Flavor1Demo class
```

b) *As implementer of the specified interface*

```
class Flavor2Demo {

    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };

    public static void main(String[] args) {
        h.show();
    }
}

interface Hello {
    void show();
}
```

Output:

```
i am in anonymous class
```

4) **Static nested class:** discussed in next question

140) Nested class in Java?

Ans- In java, it is possible to define a class within another class, such classes are known as *nested* classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and create more readable and maintainable code. A nested class has access to the members, including private members, of the class in which it is nested. However, reverse is not true i.e. the enclosing class does not have access to the members of the nested class.

- a nested class can be declared *private*, *public*, *protected*, or *package private*(default).
- Nested classes are divided into two categories:
    1. **static nested class :** Nested classes that are declared *static* are called static nested classes.
    2. **inner class :** An inner class is a non-static nested class

Static nested classes

- As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.
  They are accessed using the enclosing class name.

```
OuterClass.StaticNestedClass
```

  For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
    new OuterClass.StaticNestedClass();
```

Inner classes:

There are two special kinds of inner classes :

1. Local inner classes
2. Anonymous inner classes

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Difference between static and inner(non-static nested) classes

- Static nested classes do not directly have access to other members(non-static variables and methods) of the enclosing class because as it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

- Non-static nested classes (inner classes) has access to all members(static and non-static variables and methods, including private) of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

141) Functional interface in JAVA?

Ans- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods.

Runnable, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces. Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
// Java program to demonstrate functional interface

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

Java 8 onwards, we can assign lambda expression to its functional interface object like this:

```
class Test

{

  public static void main(String args[])

  {

    // lambda expression to create the object

    new Thread(()->

       {System.out.println("New thread created");}).start();

  }

}
```

@FunctionalInterface Annotation
@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method.
In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message.

However, it is not mandatory to use this annotation.

```
// Java program to demonstrate lamda expressions to implement
// a user defined functional interface.

@FunctionalInterface
interface Square
{
    int calculate(int x);
}
```

Note: The java.util.function package contains many built-in functional interfaces in Java 8.

142) Marker interface in JAVA?
Ans- It is an empty interface (no field or methods). Examples of marker interface are Serializable, Clonnable and Remote interface. All these interfaces are empty interfaces.

```
public interface Serializable
{
  // nothing here
}
```

Marker interface is used as a tag to inform a message to the Java compiler so that it can add special behavior to the class implementing it.

When a Java class is to be serialized, you should intimate the Java compiler in some way that there is a possibility of serializing this java class. In this scenario, marker interfaces are used.

We cannot create marker interfaces, as you cannot instruct JVM to add special behavior to all classes implementing (directly) that special interface.

Example of it are-

a) **Cloneable interface** : Cloneable interface is present in java.lang package. There is a method clone() in Object class. A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown. By convention, classes that implement this interface should override Object.clone() method.

```
class A implements Cloneable
{
    int i;
    String s;

    // A class constructor
    public A(int i,String s)
    {
        this.i = i;
        this.s = s;
    }

    // Overriding clone() method by simply calling Object class clone()

    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

A a = new A(20, "GeeksForGeeks");

  // cloning 'a' and holding new cloned object reference in b

  // down-casting as clone() return type is Object
  A b = (A)a.clone();
```

b) Serializable interface : Serializable interface is present in java.io package. It is used to make an object eligible for saving its state into a file. Classes that do not implement this interface will not have any of their state serialized or deserialized.

```
// By implementing Serializable interface we make sure that state of instances of
class A can be saved in a file.
class A implements Serializable
{
    int i;
    String s;

    // A class constructor
    public A(int i,String s)
    {
        this.i = i;
        this.s = s;
    }
}
```

```
public class Test
{
    public static void main(String[] args)
      throws IOException, ClassNotFoundException
    {
        A a = new A(20,"GeeksForGeeks");

        // Serializing 'a'
        FileOutputStream fos = new FileOutputStream("xyz.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a);

        // De-serializing 'a'
        FileInputStream fis = new FileInputStream("xyz.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        A b = (A)ois.readObject();//down-casting object

        System.out.println(b.i+" "+b.s);

        // closing streams
        oos.close();
        ois.close();
    }
}
```

Output:

```
20 GeeksForGeeks
```

c) **Remote interface** : Remote interface is present in java.rmi package. A remote object is an object which is stored at one machine and accessed from another machine. So, to make an object a remote object, we need to flag it with Remote interface.

Here, Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.
Any object that is a remote object must directly or indirectly implement this interface. RMI (Remote Method Invocation) provides some convenience classes that remote object implementations can extend which facilitate remote object creation.

143) Static methods in Interface?

Ans- Static Methods in **Interface** are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function.

And since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

```
// Java program to demonstrate scope of static method in Interface.

interface PrintDemo {

    // Static Method
    static void hello()
    {
        System.out.println("Called from Interface PrintDemo");
    }
}

public class InterfaceDemo implements PrintDemo {

    public static void main(String[] args)
    {

        // Call Interface method as Interface name is preceeding with method
        PrintDemo.hello();

        // Call Class static method
        hello();
    }

    // Class Static method is defined
    static void hello()
    {
        System.out.println("Called from Class");
    }
}
```

Output:

```
Called from Interface PrintDemo

Called from Class
```

If same name method is implemented in the implementation class then that method becomes a static member of that respective class. Static method don't get overridden.