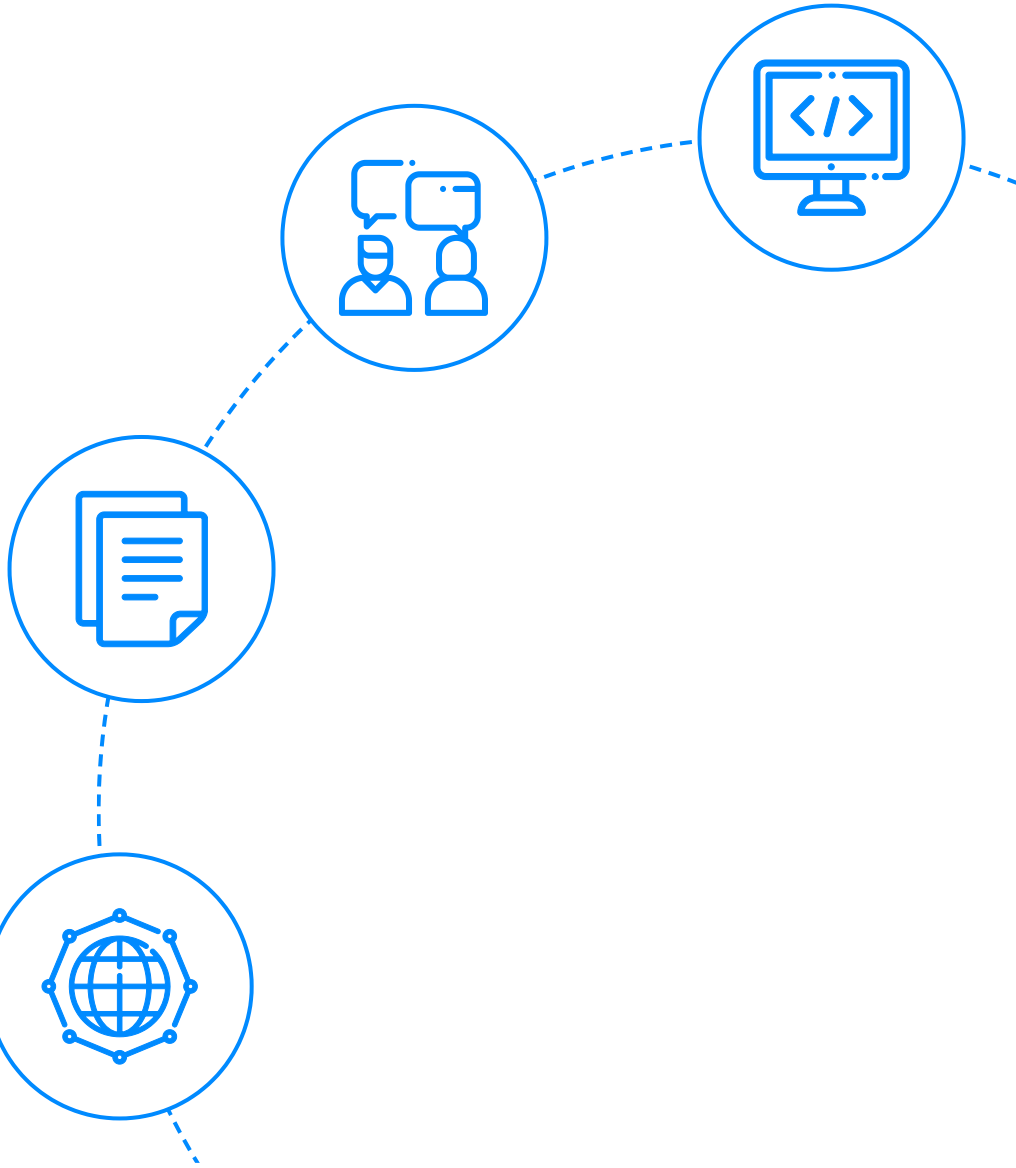
InterviewBit

Java Cheat Sheet



To view the live version of the page, click here.

Contents

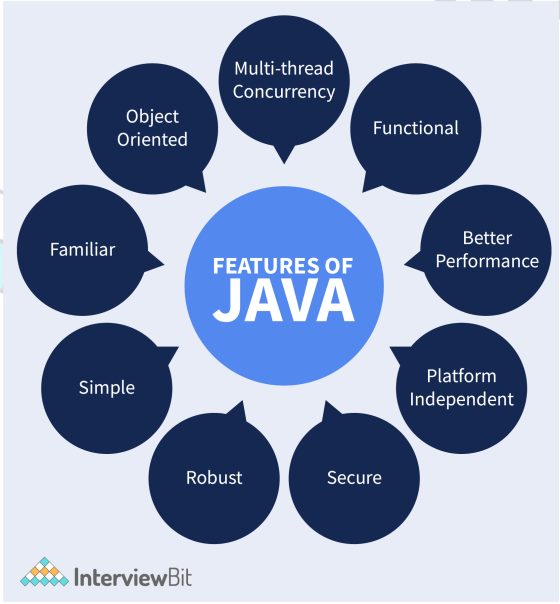
Learn Java: Basics to Advanced Concepts

Learn Java: Basics to Advanced Concepts (.....Continued)

21. Java Commands
22. Java Collections
23. Java Generics
24. Java Multithreading

Let's get Started

Java is a high-level programming language that is famous for its robustness, object-oriented nature, enhanced security, and easy-to-learn features also known as **Java Buzzwords**. Java can be considered both a platform and a programming language. The founder of Java, James Gosling, is recognised as the “Father of Java.” It was known as Oak before Java. Since Oak was already a recognised business, James Gosling and his team changed the programming language’s name to Java. Java allows programmers to write, compile, and debug code easily. It is widely used in developing reusable code and modular programs.



Java is an object-oriented programming language and it focuses on reducing dependencies. A java program can be written once and executed anywhere (WORA). Java programs are first compiled into bytecode and the byte code generated can be run on any Java Virtual Machine. Java is similar to C / C++ in terms of syntax.

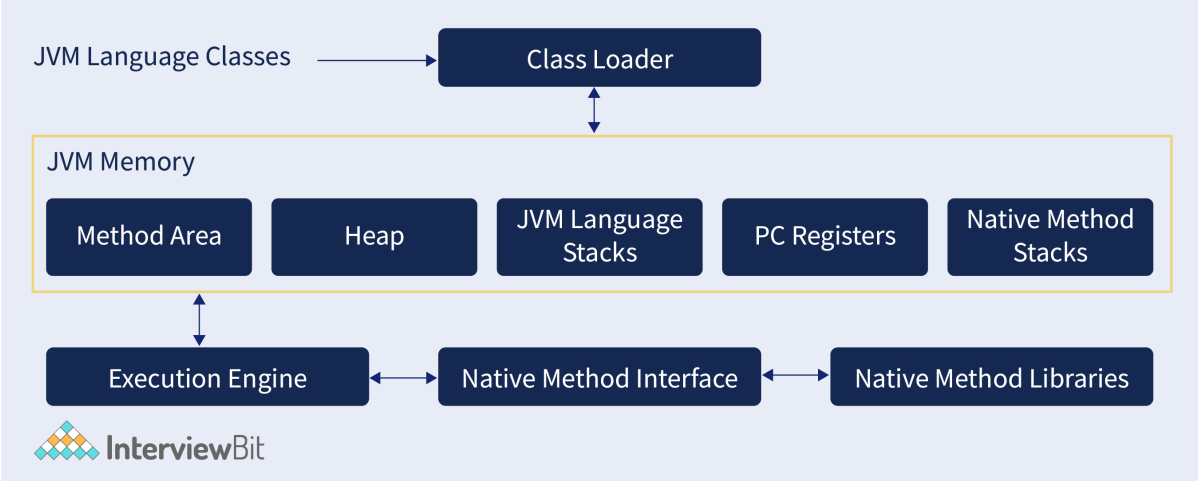
Java is also the name of an Indonesian island. It is here that the first coffee (also known as java coffee) was grown. It is while sipping coffee near his office that James Gosling thought of this name.

Learn Java: Basics to Advanced Concepts

1. Java Terminology

Let us quickly go through some of the most important terms used in the Java programming language.

JVM: The JVM stands for Java Virtual Machine. A program's execution is divided into three stages. A Java program is written, compiled and then run.



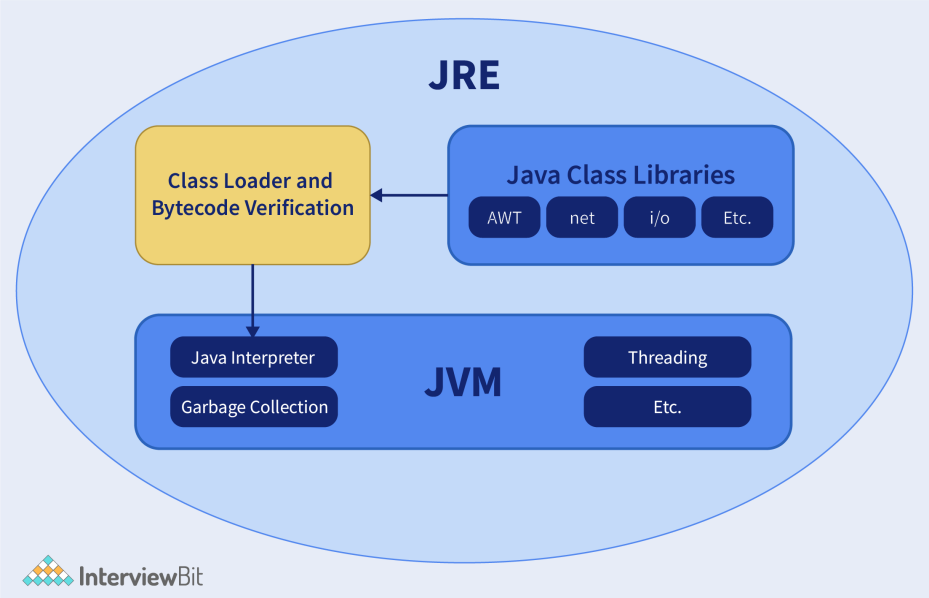
As we can see in the above image, first the JVM language classes are processed by the class loader subsystem which is responsible for loading, linking and initialization of the java classes. After being processed by the class loader, the generated files are stored in the JVM Memory which consists of method area, heap, JVM language stacks, PC registers and native method stacks. The execution engine accesses the files from this JVM memory and makes use of the Native Method Interface and Native Method Libraries.

- A Java programmer creates a program.
- The JAVAC compiler, which is a primary Java compiler provided in the Java development kit (JDK), is used to compile the code. It accepts a Java application as input and outputs bytecode.
- JVM runs the bytecode generated by the compiler during the program's Running phase.

The Java Virtual Machine's job is to run the bytecode generated by the compiler. Although each operating system has its own JVM, the output they provide after bytecode execution is consistent across all of them. Java is known as a platform-independent language for this reason.

Bytecode: Bytecode is a type of intermediate code generated by the compiler after source code has been compiled (JAVA Program). Java is a platform-independent language thanks to this intermediate code.

JRE: The Java Runtime Environment (JRE) is included with the JDK. The JRE installation on our computers allows us to run the Java program, but we cannot compile it. A browser, JVM, applet support, and plugins are all included in JRE. JRE is required for a computer to run a Java program.



In the above image, we can see that JVM together with the Java Class Libraries makes up the JRE.

Java Development Kit (JDK): When we learn about bytecode and JVM, we use the name JDK. As the name implies, it is a complete Java development kit that includes everything from the compiler to the Java Runtime Environment (JRE), debuggers, and documentation. In order to design, compile, and run the java application, we must first install JDK on our computer.

In the above image, we can clearly see that JVM and the Library classes together make up the JRE. JRE when combined with Development Tools makes up JDK.

Garbage Collection: Garbage collection is the technique through which Java programs maintain their memory automatically. Java programs are compiled into bytecode that may be executed by a Java Virtual Machine, or JVM. Objects are produced on the heap, which is a part of memory devoted to the Java application, while it runs on the JVM. Some objects will become obsolete over time. To free up memory, the garbage collector detects these useless objects and deletes them.

finalize method: It's a method that the Garbage Collector calls shortly before deleting or destroying an object that's suitable for Garbage Collection in order to do cleanup. Clean-up activity entails de-allocating or closing the resources associated with that object, such as Database Connections and Network Connections. It's important to remember that it's not a reserved keyword. Garbage Collector destroys the object as soon as the finalise method completes. The finalise method is found in the Object class and has the following syntax: protected void finalize throws Throwable{}

Since the finalize function is contained in the Object class and Object is the superclass of all Java classes, the finalize method is available to all Java classes. As a result, the garbage collector may invoke the finalise function on any java object. We must override the finalize method present in the Object class to specify our own clean-up activities since the finalize function in the Object class has an empty implementation.

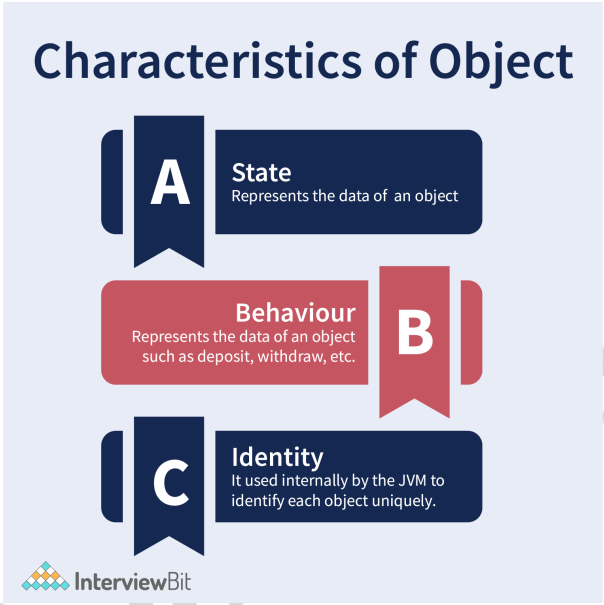
> Check out commonly asked Java Problems in Interviews: **Click**

2. Java Basics

Let us look at some of the basic concepts frequently used in the Java programming language.

object - An object is an entity with state and behaviour, such as a chair, a bike, a marker, a pen, a table, a car, and so on. It could be either physical or logical (tangible or intangible). The financial system is an example of an intangible object.

There are three characteristics of an object:



- **State:** The data (value) of an object is represented by its state.
- **Behaviour:** The functionality of an object, such as deposit, withdrawal, and so on, is represented by the term behaviour.
- **Identity:** A unique ID is often used to represent an object's identification. The value of the ID is hidden from the outside user. The JVM uses it internally to uniquely identify each object.

class - A class is a collection of objects with similar attributes. It's a blueprint or template from which objects are made. It's a logical thing. It can't be physical. In Java, a class definition can have the following elements:

- **Modifiers:** A class can be private or public, or it can also have a default access level
- **class keyword:** To construct a class, we use the class keyword.
- **class name:** The name of the class should usually start with a capital letter.
- **Superclass (optional):** If the class has any superclass, we use the extends keyword and we mention the name of the superclass after the class name.
- **Interface (optional):** If the class implements an interface, we use the implements keyword followed by the name of the interface after the class name.

Constructors: A constructor in Java is a block of code that is comparable to a method. When a new instance of the class is created, the constructor is invoked. It is only when the constructor is invoked that memory for the object is allocated.

There are two types of constructors in Java. They are as follows:-

Default Constructor - A default constructor is a constructor that has no parameters. If we don't declare a constructor for a class, the compiler constructs a default constructor for the class with no arguments. The compiler does not produce a default constructor if we explicitly write a constructor.

```
import java.io.*;
class Test
{
   int a
   Test()
   {
       System.out.println("Default Constructor called");
   }
}
class Sample
{
   public static void main (String[] args)
   {
       // Creating a new object of Test class
       Test obj = new Test();
       // Default constructor provides the default value to data member 'a'
       System.out.println(obj.a);
   }
}
```

Output :

```
Default Constructor called
0
```

In the above code, the class Test has a default constructor with no arguments. When an object of the class Test is created, the default constructor gets invoked and the statement "Default Constructor called" is printed.

Parameterised Constructor - The term parameterized constructor refers to a constructor that has parameters. If we wish to initialize our own values to the class's fields, we should use a parameterized constructor.

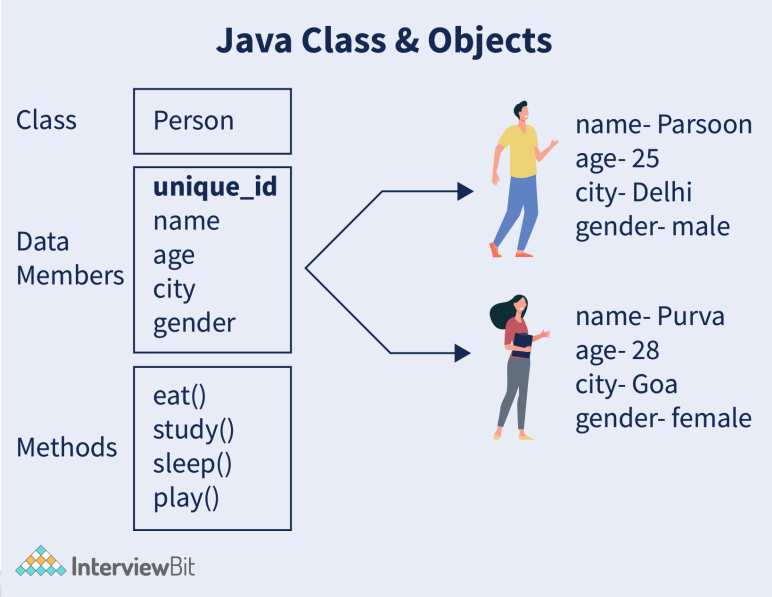
import java.io.*;

```
class Test
{
   int a
   Test(int x)
   {
       a = x;
   }
}
class Sample
{
   public static void main (String[] args)
   {
       // Creating a new object of Test class
       Test obj = new Test(10); // Providing a value for the data member 'a'
       System.out.println(obj.a);
   }
}
```

Output -

```
10
```

In the above code, the class Test has a parameterised constructor with 1 argument. When an object of the class Test is created by providing 1 argument, the parameterised constructor gets invoked and the data member of the object gets initialised to 10.



In the above image, we can see that a class Person has been defined with the data members 'unique_id', 'name', 'age', 'city' and 'gender' and the methods 'eat()', 'study()', 'sleep()' and 'play()'. Two objects of this class have been defined. The first object has 'name' as 'Parsoon', 'age' as '25', 'city' as 'Delhi' and 'gender' as 'male'. The second object of the 'Person' class has 'name' as 'Purva', 'age' as '28', 'city' as 'Goa' and 'gender' as 'female'.

In Java, the above example can be represented as follows:

```
class Person
{
 int unique_id;
 String name;
 int age;
 String city;
 String gender;
 Person()
 {
   unique_id = 0;
   name = "";
   age = 0;
   city = "";
   gender = "";
 }
 Person(int _id, String _name, int _age, String _city, String _gender)
 {
   unique_id = _id;
   name = _name;
   age = _age;
   city = _city;
   gender = _gender;
 }
}
class Test
{
 public static void main(String args[])throws IOException
 {
   Person obj1 = new Person(1, "Parsoon", 25, "Delhi", "male");
   Person obj2 = new Person(2, "Purva", 28, "Goa", "female");
 }
}
```

In the above code, we have created a class ‘Person’ which has the data members ‘unique_id’, ‘name’, ‘age’, ‘city’ and ‘gender’. We have added a default constructor and a parameterised constructor in the ‘Person’ class definition as well. The default constructor initialised the data members to their default value and the parameterised constructor initialises the data members to the values provided as arguments. We create a ‘Test’ class to create 2 objects of the ‘Person’ class as demonstrated in the above example. We pass the values {1, ‘Parsoon’, 25, ‘Delhi’, ‘male’} for the first object and the values {2, ‘Purva’, 28, ‘Goa’, ‘female’} for the second object.

keyword - Reserved words are another name for Java keywords. Keywords are specific terms that have special meanings. Because these are Java's predefined words, they can't be used as variable, object, or class names. Following is the list of keywords used in Java:-

Keyword	Use Case
boolean	The boolean keyword in Java is used to declare a variable to be of the boolean type. It can only store True and False values.
byte	The byte keyword in Java is used to create a variable that can handle 8-bit data values.
break	The break keyword in Java is used to end a loop or switch statement. It interrupts the program's current flow when certain circumstances are met.
abstract	The abstract keyword in Java is used to declare an abstract class.
case	The case keyword in Java is used with switch statements to mark text blocks.
try	The try keyword in Java is used to begin a block of code that will be checked for errors. Either a catch or a finally block must come after the try block.
short	The Java short keyword is used to declare a variable with a 16-bit integer capacity.
void	The void keyword in Java is used to indicate that a method has no return value.
static	The static keyword is used to denote a class method or a class variable in Java.
synchronized	In multithreaded programming, the synchronized keyword is used to designate the critical sections or functions

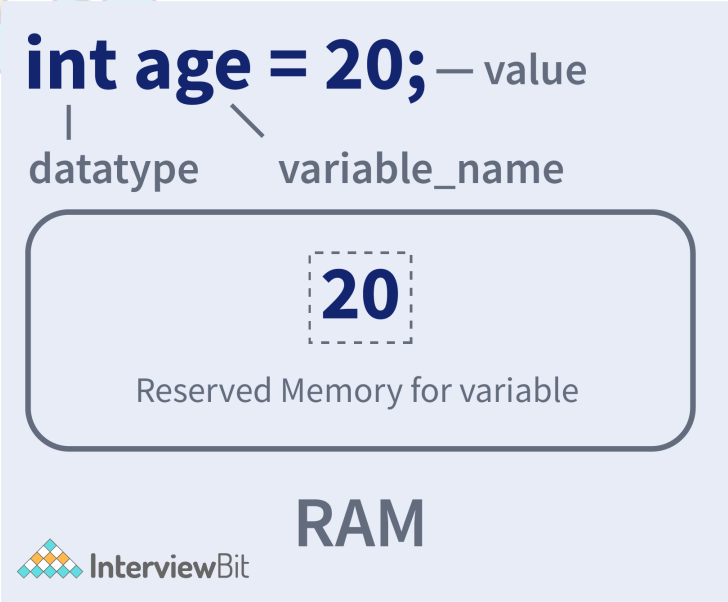
3. Variables in Java

In Java, a variable is a data container that stores data values during the execution of a Java program. A data type is allocated to each variable, indicating the type and quantity of data it may store. It is a program's fundamental storage unit. All variables in Java must be defined before they may be used.

Syntax of declaring a variable:-

```
datatype variable_name;
```

Here, **datatype** refers to the type of data that can be stored in the variable. **variable_name** refers to the name of the variable being declared.



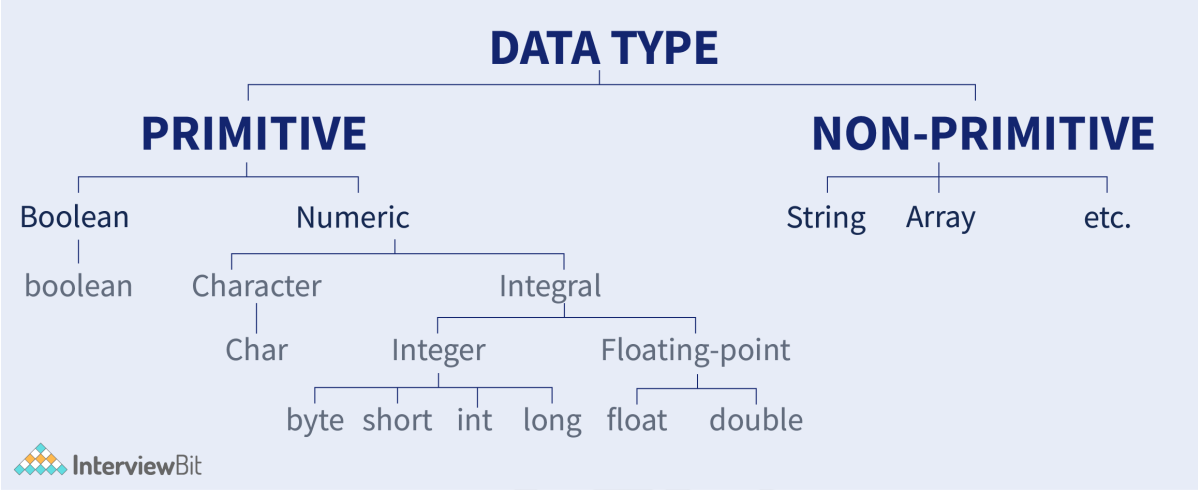
In the above image, we can see that a variable named 'age' has been declared of type 'int' and has been initialised with the value '20'.

Types of variables:-

- **Local variable** - A local variable is a variable defined within a block, method, or constructor. These variables have a scope that is limited to the block in which they are defined. i.e., these variables are only accessible within that block.
- **Instance variable** - Variables that are declared inside a class without the 'static' keyword are referred to as instance variables. Instance variables are replicated for every object of the class being created.
- **Static variable** - Variables that are declared inside a class with the 'static' keyword are referred to as static variables. Static variables are shared by all the objects of the class and are not replicated. Static variables are created at the start of program execution and automatically destroyed when the program is finished. It is not necessary to initialise a static variable. It has a value of 0 by default.

4. Datatypes in Java

The different sizes and values that can be stored in the variable are defined by data types. In Java, there are two types of data types:



Primitive Data Types in Java:-

Primitive data types are the building blocks of data manipulation in the Java programming language. These are the most fundamental data types in the Java programming language. Following are the different primitive data types available in Java:-

Data Type	Default Size	Description
boolean	1 bit	Only two potential values are stored in the Boolean data type: true and false. Simple flags that track true/false circumstances are stored in this data type.
char	2 byte	A single 16-bit Unicode character is represented by the char data type. It has a value range of '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).
byte	1 byte	It's an 8-bit two-s complement signed integer. It has a value range of -128 to 127. (inclusive). It has a value of 0 by default. The byte data type is used to preserve memory in huge arrays where space is at a premium. Because a byte is four times smaller than an integer, it saves space.
short	2 bytes	A 16-bit signed two's complement integer is the short data type. It has a value range of -32,768 to 32,767. (inclusive). It has a value of 0 by default.

Non Primitive data types in Java:-

The non-primitive data types in Java include classes, interfaces and arrays. We will discuss more on them in the upcoming topics.

5. Java Keywords

a) Understanding the this keyword in Java:

The **this** keyword in java can be used in multiple ways. The following are the use cases of the this keyword:

this: to refer to the current instance variable of the class

The this keyword can be used to refer to the current instance variable of a class. If there is any ambiguity between the instance variables and arguments, this keyword clears things out.

Example -

```
class Sample{
   int a;
   Sample(int a){
       this.a=a;
   }
   void display(){
       System.out.println("a = "+a);
   }
}

class Test{
   public static void main(String args[]){
       Sample s1=new Sample(10);
       s1.display();
   }
}
```

Output -

```
a = 10
```

Explanation: In the above code, we can see that both the data member of the class Sample and the parameter of the constructor of the Sample class have the same name 'a'. Here, we use the this keyword to differentiate between the two.

this: to call the method of the current class

The **this** keyword can be used to call a method in the current class. If you don't use the this keyword, the compiler will add it for you when you call the method. Let's look at an example.

```
class Sample{
   void fun()
   {
       System.out.println("hello there");
   }
   void foo(){
       System.out.println("hello foo");
       this.m();
   }
}
class Test{
   public static void main(String args[]){
       Sample s = new Sample();
       s.foo();
   }
}
```

Output -

```
hello foo
hello there
```

Explanation - In the above code, we have 2 methods defined in the Sample class fun and foo. We create an instance of the Sample class and call the method foo(). In the foo() method, we call the fun() method using this keyword.

this() is used to call the constructor of the current class

The current class constructor can be called using the this() call. It allows for constructor chaining.

Example -

```
class Sample{
   Sample()
   {
       System.out.println("In default constructor");
   }
   Sample(int a)
   {
       this();
       System.out.println("In Parameterised constructor");
   }
}
class Test{
   public static void main(String args[]){
       Sample s = new Sample(10);
   }
}
```

Explanation - In the above code, we have two constructors defined in the Sample class. We call the default constructor from the parameterised constructor using the this() statement.

this: to use as a parameter in a method

The keyword this can also be used as an argument in a method. It is primarily used in event handling. Let's look at an example:

```
class Sample{
   void fun(Sample obj)
   {
       System.out.println("Received the object");
   }
   void foo(){
       fun(this);
   }
}
class Test{
   public static void main(String args[]){
       Sample s = new Sample();
       s.foo();
   }
}
```

In the above code, two functions fun and foo have been defined in the Sample class. The function fun() accepts a parameter of Sample object type. We call the function fun() from the function foo() and pass this as a reference to the current invoking object.

this keyword can be used to get the current instance of a class

This keyword can be returned as a statement from the method. In this situation, the method's return type must be the class type (non-primitive). Let's look at an example:

```
class Sample{
   Sample getObject()
   {
       return this;
   }
   void foo(){
       System.out.println("Inside foo function")
   }
}
class Test{
   public static void main(String args[]){
       Sample s = new Sample();
       s.getObject().foo();
   }
}
```

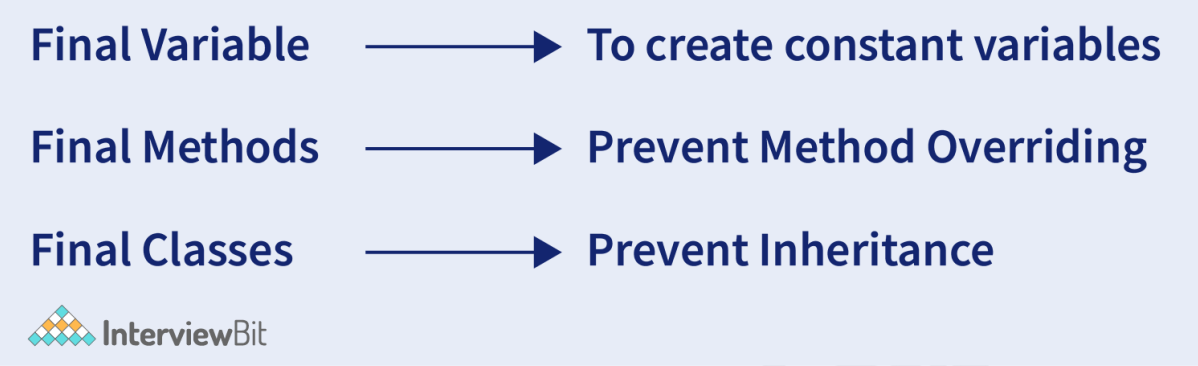
Output -

```
Inside foo function
```

In the above code, we have a function 'getObject' defined in the Sample class which returns a reference to the current invoking object. We use this reference returned to call the foo() method from the Test class.

b) final keyword in Java:

The final keyword is used in a variety of situations. To begin with, final is a non-access modifier that only applies to variables, methods, and classes. The behaviour of final in each scenario is listed below:



Final Variables:
When a variable is declared using the final keyword, the value of the variable cannot be changed, making it a constant. This also necessitates the creation of a final variable. If the final variable is a reference, it cannot be re-bound to reference another object, but the internal state of the object indicated by that reference variable can be altered, allowing you to add or delete elements from the final array or collection.

Example -

```
final int temp = 10;
```

Final Classes:

A final class is one that has been declared with the final keyword. It is not possible to extend a final class. A final class can be used in two ways:

- The first is to avoid inheritance because final classes cannot be extended. All Wrapper Classes, such as Integer, Float, and others, are final classes. We are unable to extend them.
- The final keyword can also be used with classes to construct immutable classes, such as the String class. It is impossible to make a class immutable without making it final.

Example -

```
final class Sample
{
    // code
}
class Test extends Sample // COMPILE-ERROR!
{
   // code
}
```

The above code gives a compile-time error. The reason is that the class Sample is declared as final and since the class Test is trying to extend the Sample class, it gives a compile-time error.

Final Methods:

A method is called a final method when it is declared with the final keyword. It is not possible to override a final method. In the Object class, many methods defined are declared final. This is done so that a user cannot modify the definition of those functions in their java classes. We must use the final keyword to declare methods for which we must use the same implementation throughout all derived classes.

Example -

```
class Sample
{
    final void fun()
    {
       System.out.println("Inside fun function");
    }
}
class Test extends Sample
{
   void fun() // COMPILE-ERROR!
   {
       //code
   }
}
```

The above code on execution gives a compile-time error. The reason is that the fun() function declared in the Sample class is a final method so we cannot override it in the inherited class. In the above code, the Test class tries to override the fun() function defined in the Sample class and this leads to a compilation error.

c) static keyword in Java:

In Java, the static keyword is mostly used to control memory. In Java, the static keyword is used to share a class's variable or method. Static keywords can be used with variables, methods, blocks, and nested classes. The static keyword refers to a class rather than a specific instance of that class.

The static keyword is used in the following ways:

Static Blocks:

You can declare a static block that gets executed exactly once when the class is first loaded if you need to do the computation to initialise your static variables.

Consider the Java program below, which demonstrates the use of static blocks.

```
class Test
{
   static int x = 25;
   static int y;

   // static block
   static {
       System.out.println("Inside Static block.");
       y = x * 4;
   }

   public static void main(String[] args)
   {
      System.out.println("Value of x : " + x);
      System.out.println("Value of y : " + y);
   }
}
```

Output -

```
Inside Static block.
Value of x : 25
Value of y : 100
```

Explanation - In the above code, there are 2 static variables x and y. X is initialised with 25. We can see that first of all, the static block is executed which prints the statement “Inside Static block.” and initialises y to x * 4 (i.e., 100).

Static Variables:

When a variable is marked as static, a single copy of the variable is made at the class level and shared among all objects. Static variables are effectively global variables. The static variable is shared by all instances of the class. In a program, static blocks and static variables are executed in the order in which they appear.

Example -

```
class Test
{
   // static variable
   static int x = fun();

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

   // static method
   static int fun() {
       System.out.println(Inside fun function");
       return 10;
   }

   public static void main(String[] args)
   {
      System.out.println("Value of x : " + x);
      System.out.println("Inside main method");
   }
}
```

Output -

```
Inside fun function
Inside static block
Value of x : 10
Inside main method
```

Explanation - In the above code, first of all, x gets initialised by calling the fun() function. Hence, the statement “Inside fun function” gets printed. Then, the static block gets executed since it is after the initialisation of x statement. At last, the statements inside the main method get printed.

Static Methods:

The static method is defined as a method that is declared using the static keyword. The main() method is the most common example of a static method. Any static member of a class can be accessed before any objects of that class are generated, and without requiring a reference to any object. Static methods are subject to the following constraints:

- They can only call other static methods directly.
- They can only access static data directly.
- They are not allowed to use the words "this" or "super" in any way.

Example -

```
class Test
{

   static int x = 1; // static variable
   int y = 20; // instance variable


   static void fun() // static method
   {
       x = 20;
       System.out.println("Inside fun function");

        y = 10; // compilation error since there is a static reference to the non-stati

        foo();  // compilation error since there is a static reference to the non-stati

        System.out.println(super.a); // compilation error since we cannot use super in
   }

   void foo() // instance method
   {
       System.out.println("Inside foo function");
   }
   public static void main(String[] args)
   {
       // main method
   }
}
```

The above code results in a compilation error. There are 3 reasons for it.

- We are accessing an instance variable inside a static method and changing its value.
- We are invoking an instance method inside a static method.
- We are trying to make use of the super keyword inside a static method.

Static Classes:

When a class is declared with the static keyword, it is said to be a static class. A class can be made static in Java if and only if it is nested inside another class. We can't use the static modifier on top-level classes, but we can use it on nested classes.

Example -

```
public class Test {

   private static String str = "Interview bit";

   // Static class
   static class Sample {

       public void display(){
         System.out.println(str);
       }
   }

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

Output -

```
Interview bit
```

Explanation - In the above code, we have a static nested class named ‘Sample’ inside the ‘Test’ class. We create an object of the Sample class and call the display function.

d) super keyword in Java:

In Java, the super keyword is a reference variable that refers to parent class instances. With the concept of Inheritance, the term "super" entered the picture. It's most commonly used in the following situations:

Using super with variables:

When a derived class and a base class have identical data members, there is uncertainty for the JVM as to which class’s data member is being referred to. In order to resolve this ambiguity, we use the super keyword with the data member’s name. This code snippet will help us comprehend it better:

```
class Sample1
{
   int x = 1;
}

class Sample2 extends Sample1
{
   int x = 2;

   void display()
   {
       System.out.println("The value of x is : " + super.x);
   }
}

class Test
{
   public static void main(String[] args)
   {
       Sample2 s = new Sample2();
       s.display();
   }
}
```

Output -

```
The value of x is : 1
```

Explanation - In the above code both the classes, Sample1 and Sample2 have a data member named 'x'. We use the super keyword inside the display function of the Sample2 class to access the data member x of the parent class Sample1.

Using super with methods:

To resolve ambiguity when a parent and child class have the same-named methods, we employ the super keyword. This code snippet demonstrates how to use the super keyword.

```
class Sample1
{
   void fun()
   {
       System.out.println("Inside Sample1's fun method");
   }
}

class Sample2 extends Sample1
{
   void fun()
   {
       System.out.println("Inside Sample2's fun method");
   }

   void display()
   {
       super.fun();
   }
}

class Test
{
   public static void main(String[] args)
   {
       Sample2 s = new Sample2();
       s.display();
   }
}
```

Output -

```
Inside Sample1's fun method
```

Explanation - In the above code, both the classes Sample1 and Sample2 have a method named fun. However, in the display method of the Sample2 class, the statement “Inside Sample1’s fun method” gets printed because of the super keyword.

Use of the super keyword with constructors:

The super keyword can also be used to access the constructor of the parent class. Another key point to note is that, depending on the situation, 'super' can refer to both parametric and non-parametric constructors. The following is a code snippet to demonstrate the concept:

```
class Sample1
{
   Sample1()
   {
       System.out.println("Inside Sample1's default constructor.")
   }
}

class Sample2 extends Sample1
{
   Sample2()
   {
       super();
       System.out.println("Inside Sample2's default constructor.");
   }
}

class Test
{
   public static void main(String[] args)
   {
       Sample2 s = new Sample2();
   }
}
```

Output -

```
Inside Sample1's default constructor.
Inside Sample2's default constructor.
```

Explanation - In the above code, the class Sample2 extends the class Sample1. We invoke the constructor of the Sample1 class in the constructor of the Sample2 class by using the super keyword. Hence, when we create an instance of the Sample2 class, both the statements get printed.

6. Java Comments

The statements in a program that are not executed by the compiler and interpreter are referred to as Java comments. Comments can be used to explain and improve the readability of Java code. Comments in a program help to make it more human-readable by putting the details of the code involved, and effective use of comments makes maintenance and bug discovery easier. The following are the different types of comments in Java:

- **Single line comments:**

Only one line of code is commented with a single-line comment. It is the most common and straightforward method of commenting on statements.
Two forward slashes (//) begin single-line comments. Any text after the // is ignored by the Java compiler.

Syntax -

```
// This is a sample comment
```

- **Multiline comments:**

Multiple lines of code can be commented with the multi-line comment. Because we must offer '//' at every line to describe a full method in a code or a complex snippet, single-line comments might be tedious to write. To get around this, you can utilise multi-line comments. Between /* and */ are multi-line comments. Java does not execute any text between /* and */.

```
/*
This is an
Example of multi line comment
*/
```

- **JavaDoc:**

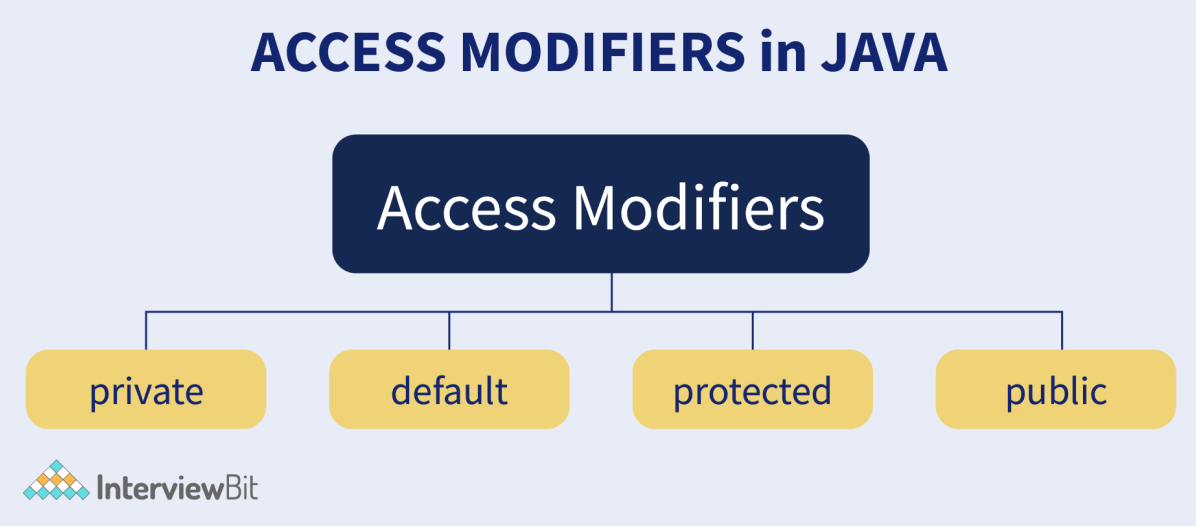
This type of comment is commonly used while writing code for a project/software package because it aids in the generation of a documentation page for reference, which can be used to learn about the methods available, their parameters, and so on.

Syntax -

```
/** Comment starts
* This is a
* Sample comment
* comment ends*/
```

7. Access Modifiers in Java

The accessibility or scope of a field, function, constructor, or class is defined by the access modifiers in Java. The access modifier can be used to adjust the access level of fields, constructors, methods, and classes.
Java access modifiers are divided into four categories as shown in the image below:



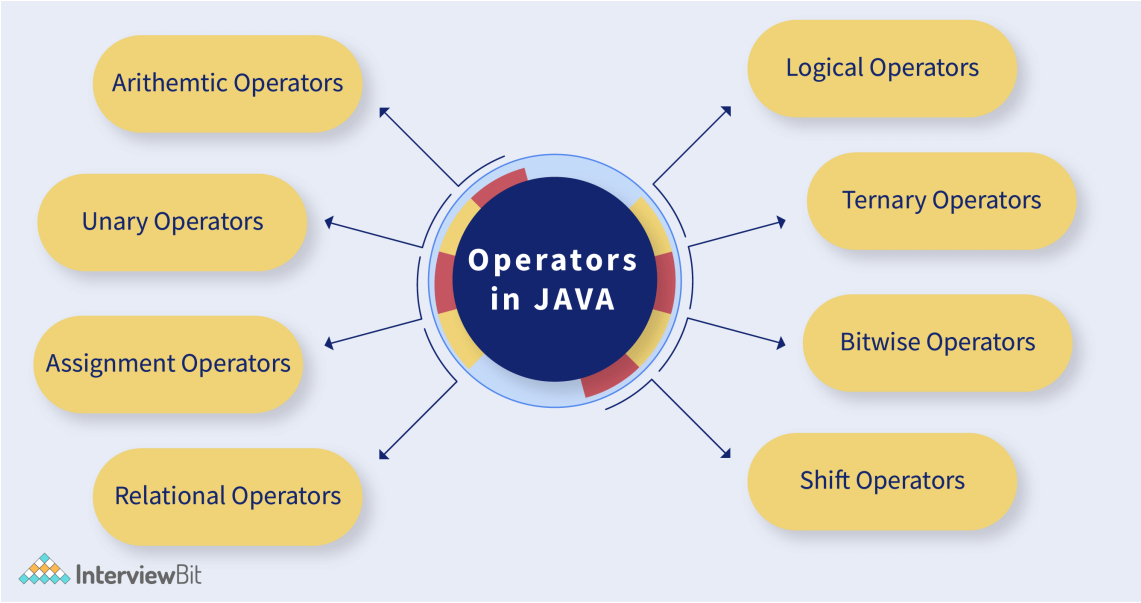
- **private**: A private modifier's access level is restricted to members of the class. It isn't accessible outside of the class.
- default: A default modifier's access level is limited to the package. It's not possible to get to it from outside the package. If you don't indicate an access level, the default will be used.
- **protected:** A protected modifier's access level is both within and outside the package via a child class.
- **public:** A public modifier's access level is universal. It can be accessed from within and outside the class, and from within and outside the package.

The following table depicts the level of accessibility that each access modifier allows in Java.

Access Modifier	within class	within package	outside pa
private	Yes	No	No
default	Yes	Yes	No
protected	Yes	Yes	Yes
public	Yes	Yes	Yes

8. Operators in Java

The below image shows the different types of operators available in Java.



The following table describes each category of operators along with examples of each type.

Operator Type	Description	Operators
Unary Operators	Only one operand is required for unary operators. They are used to increase, decrease, or negate a value.	++, --, +, -, !
Arithmetic Operators	Simple arithmetic operations on primitive data types are performed with them.	+, – , *, / , %
Assignment Operators	The assignment operator is used to give any variable a value. It has right-to-left associativity, which means that the value given on the right-hand side of the operator is assigned to the variable on the left. As a result, the right-hand side value must be declared before use or be a constant.	=, +=, -=, *=, /=, %=, &=, ^=,

Precedence and Associativity of operators in Java:- When dealing with hybrid equations involving more than one type of operator, precedence and associative principles are applied. Because there might be multiple valuations for the same equation, these criteria determine which component of the equation to analyse first. The precedence of operators is shown in the table below in decreasing order of magnitude, with the highest precedence at the top and the lowest precedence at the bottom.

Operators	Associativity	Type
++, –	Right to Left	Unary postfix
++, --, +, -, !	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

9. Identifiers in Java

Identifiers are used to identify items in the Java programming language. A class name, method name, variable name, or label can all be used as identifiers in Java.

Rules for defining Java identifiers:-

- A proper Java identifier must follow certain guidelines. If we don't follow these guidelines, we'll get a compile-time error. These criteria apply to other languages as well, such as C and C++.
- All alphanumeric characters ([A-Z],[a-z],[0-9]), '$' (dollar symbol), and '_' are the only characters that can be used as identifiers (underscore). For example, "bit@" is not an acceptable Java identifier because it contains the special character '@,'.
- Numbers should not be used to begin identifiers ([0-9]). "123geeks," for example, is not a valid Java identifier.
- Case matters when it comes to Java Identifiers. For example 'bit' and 'BIT' would be considered as different identifiers in Java.
- The length of the identifier is not limited, however, it is recommended that it be kept to a maximum of 4–15 letters.
- Reserved Words aren't allowed to be used as identifiers. Because **while** is a reserved term, "int while = 20;" is an incorrect sentence. In Java, there are 53 reserved terms.
- The length of the identifier is not limited, however, it is recommended that it be kept to a maximum of 4–15 letters.

Reserved words:- Some words are reserved in the Java programming language to indicate the language's established features. These are known as reserved words. They can be divided into two categories: keywords(50) and literals (3). Functionalities are defined by keywords, and literals are defined by values. Symbol tables use identifiers in the various analysing phases of a compiler architecture (such as lexical, syntactic, and semantic).

10. Control Flow in Java

The Java compiler runs the code from beginning to end. The statements in the code are executed in the order that they occur in the code. Java, on the other hand, includes statements for controlling the flow of Java code. It is one of Java's most important aspects, as it ensures a smooth program flow.

There are three different types of control flow statements in Java. They are as follows:

Decision-Making Statements:

As the name implies, decision-making assertions determine which and when to execute statements. Decision-making statements analyse the Boolean expression and control the program flow based on the condition's result. In Java, there are two sorts of decision-making statements: If and switch statements.

If Statement: The "if" statement in Java is used to test a condition. Depending on the circumstances, the program's control is diverted. The If statement's condition returns a Boolean value, either true or false. There are four forms of if-statements in Java, as mentioned below.

Simple if: The most basic of all control flow statements in Java is simple if statements. It evaluates a Boolean statement and, if the expression is true, allows the program to begin a block of code. Syntax -

```
if(condition) {
statement; //executes when condition is true
}
```

if-else: The if-else statement is an expansion of the if-statement that employs the else block of code. If the if-condition block is evaluated as false, the else block is executed. Syntax -

```
 if(condition) {
    statement; //executes when condition is true
    }
else{
    statement; //executes when condition is false
    }
```

if-else-if ladder: The if-else-if statement is made up of an if statement and several else-if statements. In other words, a decision tree is created by a sequence of if-else statements in which the computer can enter the block of code where the condition is true. At the end of the chain, we may also define an else statement. Syntax -

```
if(condition 1) {
   statement; //executes when condition 1 is true
   }
else if(condition 2) {
   statement; //executes when condition 2 is true
   }
else {
   statement; //executes when all the conditions are false
}
```

Nested if-statement: The if statement can contain an if or if-else statement inside another if or else-if statement in nested if-statements. Syntax -

```
if(condition 1) {
   statement; //executes when condition 1 is true
   if(condition 2) {
       statement; //executes when condition 2 is true
   }
   else{
       statement; //executes when condition 2 is false
   }
}
```

Switch Statement: Switch statements in Java are similar to if-else if-else statements. A single case is run based on the variable that is being switched in the switch statement, which comprises various blocks of code called cases. Instead of using if-else-if statements, you can use the switch statement. It also improves the program's readability.

1. Integers, shorts, bytes, chars, and enumerations can all be used as case variables. Since Java version 7, the string type is also supported. Cases cannot be duplicated.
2. When any of the cases does not match the value of the expression, the default statement is executed. It's a choice.
3. When the condition is met, the break statement ends the switch block.
4. If it is not utilised, the next case is executed.

We must remember that the case expression will be of the same type as the variable when employing switch statements. It will, however, be a constant value. Syntax -

```
switch (expression){
   case value1:
        statement1;
        break;
   .
   .
   .
   case valueN:
        statementN;
        break;
   default:
        default statement;
}
```

Loop Statements:

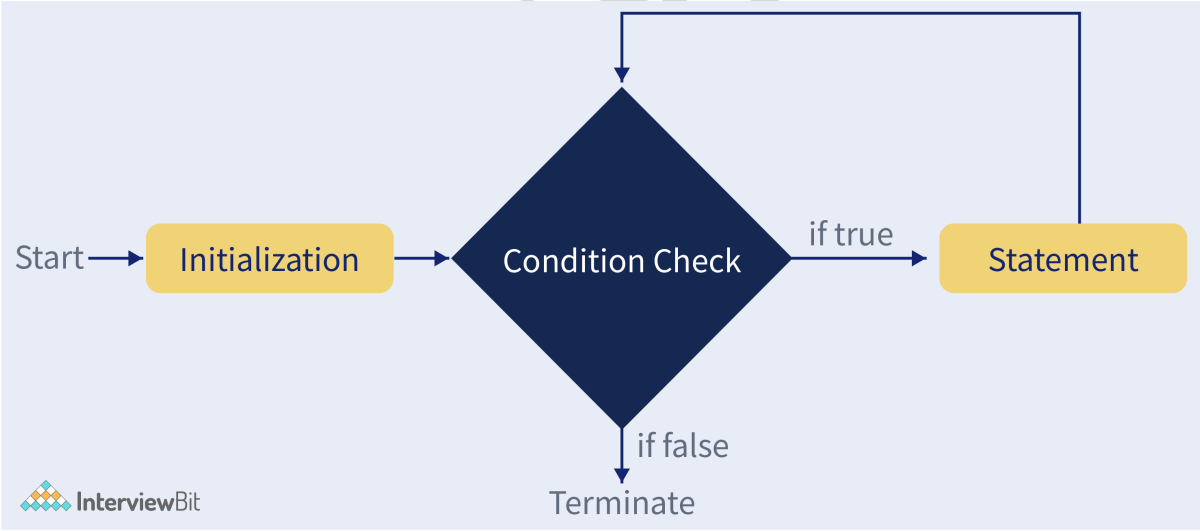
In programming, we may need to run a piece of code multiple times while a condition evaluates to true. Loop statements are used to repeat the set of instructions. The execution of the set of instructions is contingent on a certain circumstance.

In Java, there are three different forms of loops that all work in the same way. However, there are differences in their syntax and the time it takes to check for conditions.

For Loop: The for loop in Java is comparable to the for loop in C and C++. In a single line of code, we may initialise the loop variable, check the condition, and increment/decrement. We only use the for loop when we know exactly how many times we want to run a block of code.

Syntax -

```
for(initialization, condition, increment/decrement)     {
//block of statements
}
```

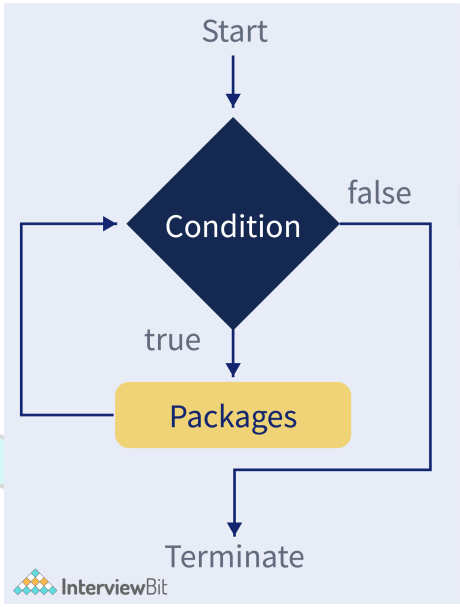


While Loop: The while loop can also be used to repeatedly iterate over a set of statements. If we don't know the number of iterations ahead of time, though, we should use a while loop. In contrast to the for loop, the initialization and increment/decrement do not happen inside the while loop statement.

Because the condition is tested at the start of the loop, it's also known as the entry-controlled loop. The loop body will be executed if the condition is true; else, the statements after the loop will be executed.

The while loop's syntax is seen below.

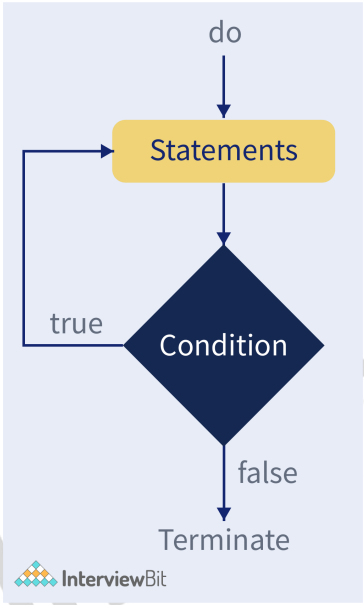
```
while(condition){
//looping statements
}
```



Do While Loop: After running the loop statements, the do-while loop verifies the condition at the end of the loop. We can use a do-while loop when the number of iterations is unknown and we need to run the loop at least once.

Because the condition is not tested in advance, it is also known as the exit-controlled loop. The do-while loop's syntax is seen below.

```
do
{
//statements
} while (condition);
```



Jump Statements:

Jump statements are used to move the program's control to certain statements. Jump statements, in other words, move the execution control to another area of the program. In Java, there are two types of jump statements: break and continue.

1. **break statement:** The break statement, as its name implies, is used to interrupt the current flow of the program and pass control to the following statement outside of a loop or switch statement. In the event of a nested loop, however, it merely breaks the inner loop. In a Java program, the break statement cannot be used on its own; it must be inserted inside a loop or switch statement.
2. **continue statement:** The continue statement, unlike the break statement, does not break the loop; instead, it skips the specific part of the loop and immediately moves to the next iteration of the loop.

> Check out our Comprehensive Interview Guide on Java: **Click**

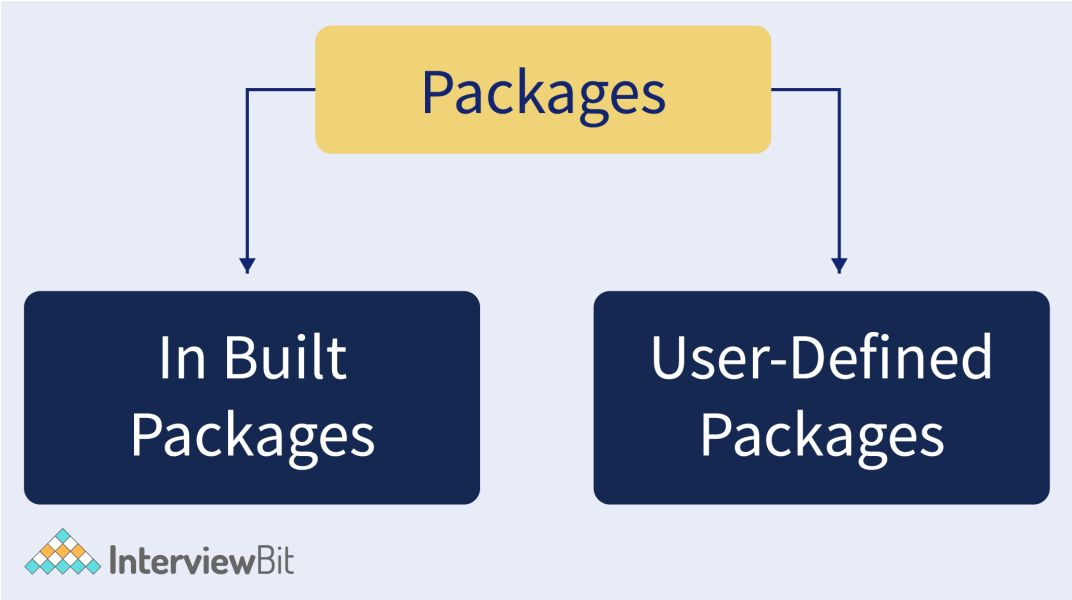
11. Java Packages

In Java, a package is a container for a collection of classes, sub-packages, and interfaces. Packages are used for the following purposes:

- Keeping name problems at bay. For example, in two packages, college.staff.cse.Employee and college.staff.ee.Employee, there could be two classes named Employee.
- Making it easier to search for, locate, and use classes, interfaces, enumerations, and annotations
- Controlling access: package level access control is available in both protected and default modes. Classes in the same package and its subclasses can access a protected member. Only classes in the same package have access to a default member (which has no access specifier).
- Packages help in data encapsulation (or data-hiding).

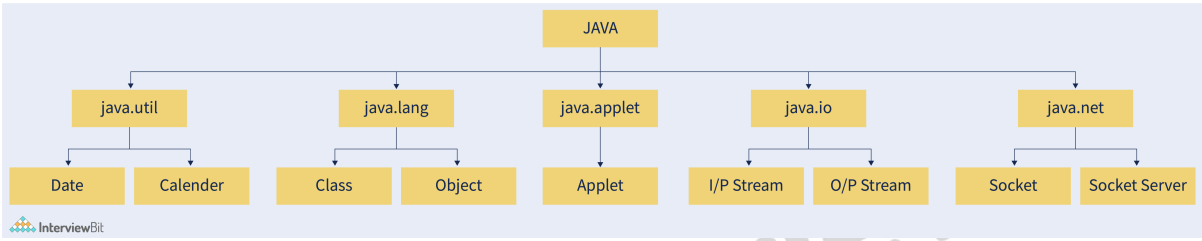
Subpackages - Subpackages are packages that are contained within another package. These are not automatically imported; they must be imported manually. Members of a subpackage also have no access privileges, therefore they are treated differently by protected and default access specifiers.

Types of packages:-



The above image shows that packages can be classified into two broad categories - User Defined Packages and In-Built Packages.

Built-in Packages: These packages contain a huge number of classes that are included in the Java API. The following are some of the most often used built-in packages as shown in the image below:



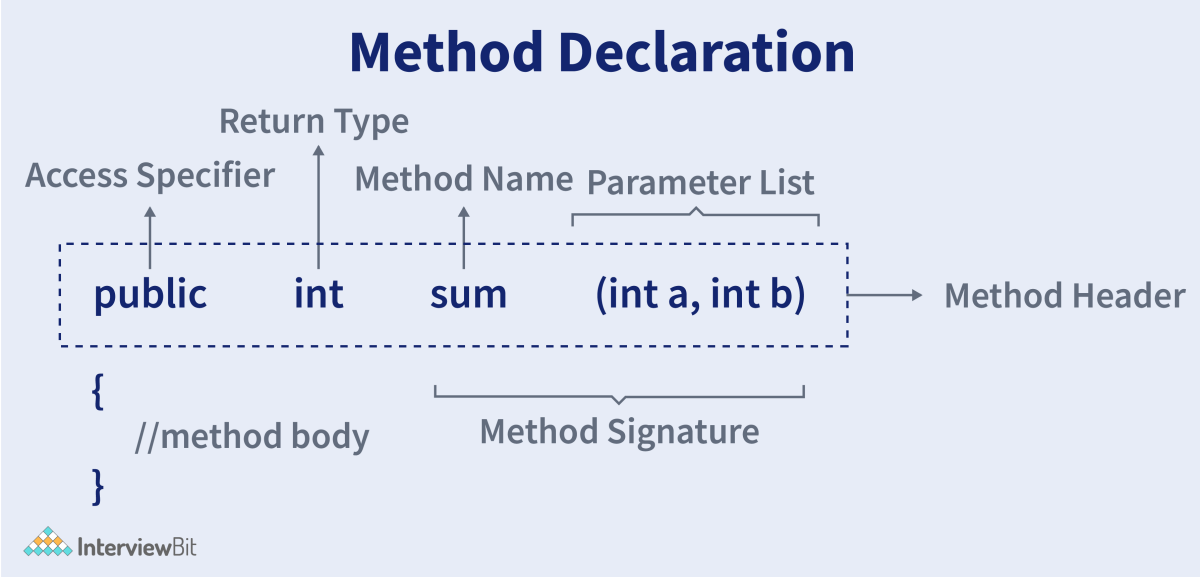
1. **java.lang:** This package contains language-specific classes (e.g classes that define primitive data types, maths operations). This package will be imported automatically.
2. **java.io:** This package contains classes that handle input and output operations.
3. **java.util:** This package contains utility classes that provide data structures such as Linked Lists, Dictionary, and Date/Time operations.
4. **java.applet:** This package contains Applet-related classes.
5. **java.awt:** Contains classes for implementing graphical user interface components (like buttons, menus etc).
6. **java.net:** This package contains classes that help with networking.

User-defined packages: These are the packages that the user has defined.

12. Java Methods

A method is a collection of statements or a series of statements organised together to conduct a specific task or action. It's a technique for making code more reusable. We create a method once and then use it repeatedly. We don't have to write code over and over again. It also allows for easy code modification and readability by simply adding or removing code chunks. Only when we call or invoke the method is it executed.

Method Declaration:- Method properties such as visibility, return type, name, and parameters are all stated in the method declaration. As seen in the following diagram, it consists of six components known as method headers.



Method signature - A method signature is a statement that identifies a method. It's included in the method declaration. It contains the method name as well as a list of parameters.

Access Specifier - The method's access specifier, also known as a modifier, determines the method's access type. It specifies the method's visibility. There are four different types of access specifiers in Java:

- **public:** When we utilise the public specifier in our application, all classes can access the method.
- **private:** The method is only accessible in the classes in which it is declared when we use a private access specifier.
- **protected:** The method is accessible within the same package or subclasses in a different package when we use the protected access specifier.
- **default:** When no access specifier is specified in the method declaration, Java uses the default access specifier. It can only be seen from the same package.

Return Type - The data type that the method returns is known as the return type. It could be a primitive data type, an object, a collection, or void, for example. The void keyword is used when a method does not return anything.

Method name - The name of a method is defined by its method name, which is a unique name. It must be appropriate for the method's functionality. If we're making a method for subtracting two numbers, the name of the method must be subtraction (). The name of a method is used to call it.

Parameter List - The parameter list is a collection of parameters separated by a comma and wrapped in parentheses. It specifies the data type as well as the name of the variable. Leave the parenthesis blank if the method has no parameters.

Method body - The method declaration includes a section called the method body. It contains all of the actions that must be completed. It is protected by a pair of curly braces.

13. Java Polymorphism

Polymorphism is a Java feature that allows us to do a single operation in multiple ways. Polymorphism is made up of two Greek words: poly and morphism. The words "poly" and "morphs" denote "many" and "forms," respectively. As a result, polymorphism denotes the presence of several forms. Polymorphism in Java is divided into two types: compile-time polymorphism and runtime polymorphism.

Compile-time Polymorphism: Compile-time polymorphism is also known as static polymorphism. In Java, this is achieved by function overloading.

Method Overloading: When there are numerous functions with the same name but distinct parameters, this is referred to as overloading. Changes in the number of arguments or the kind of arguments can cause functions to become overloaded.

Example -

```
class Sample {
   // Method with 2 integer parameters
   static int Multiply(int a, int b) // method overloading
   {
       return a * b;
   }

   // Method with same name but with 2 double parameters
   static double Multiply(double a, double b) // method overloading
   {
       return a * b;
   }
}

class Test {

   public static void main(String[] args)
   {

       // Calling method by passing
       // input as in arguments
       System.out.println(Sample.Multiply(1, 4));
       System.out.println(Sample.Multiply(5.5, 4.2));
   }
}
```

Output -

```
4
23.1
```

Explanation: In the above code, the class Sample has two functions with the same name ‘multiply’ but they have different function signatures which implement method overloading. So, we have the same name for a function that returns the multiplication of two integers and a function that returns the multiplication of two doubles.

Runtime Polymorphism:

Runtime polymorphism, often known as Dynamic Method Dispatch, is a technique for resolving calls to overridden methods at runtime rather than at compile time. A superclass's reference variable is used to call an overridden method in this process. The object referred to by the reference variable is used to determine which method should be called.

Example -

```
class Sample{
 void fun(){
  System.out.println("Inside Sample's fun method.");
 }
}
class Test extends Sample{
 void fun(){
  System.out.println("Inside Test's fun method.");
 }
}
class Main{
 public static void main(String args[]){
   Sample s = new Test()
   s.fun();
 }
}
```

Output -

```
Inside Test's fun method.
```

Explanation - In the above code, the Test class inherits from the Sample class and both the Test class and the Sample class have a method named ‘fun’ defined. This leads to method overriding. Now in the main class, we create a ‘Sample’ reference variable and allot an instance of the ‘Test’ class to it. Now, we invoke the fun() method. Since, the reference variable stores an object of the ‘Test’ class, the fun() method of the ‘Test’ class gets invoked.

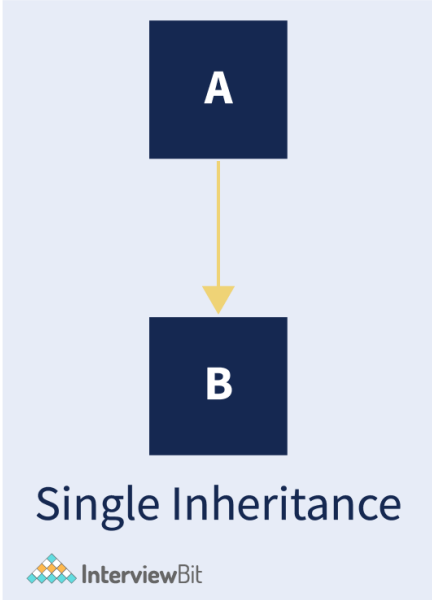
14. Java Inheritance

In Java, Inheritance is a feature that allows one object to inherit all of the characteristics and behaviours of its parent object. It's a crucial component of OOPs (Object Oriented programming systems). Inheritance in Java refers to the ability to build new classes that are based on existing ones. When you inherit from an existing class, you can use the parent class's methods and fields. You can also add additional methods and properties to your current class.

- **Subclasses/Child Classes:** A subclass is a class that inherits from another. A derived class, extended class, or kid class is another name for it.
- **Superclasses**: A superclass (sometimes known as a parent class) is the class from which a subclass derives its features. It's also known as a parent class or a base class.

Types of inheritance in Java:

1. Single Inheritance - Subclasses inherit the features of a single superclass via single inheritance. Class A acts as a base class for the derived class B in the figure below.



Example -

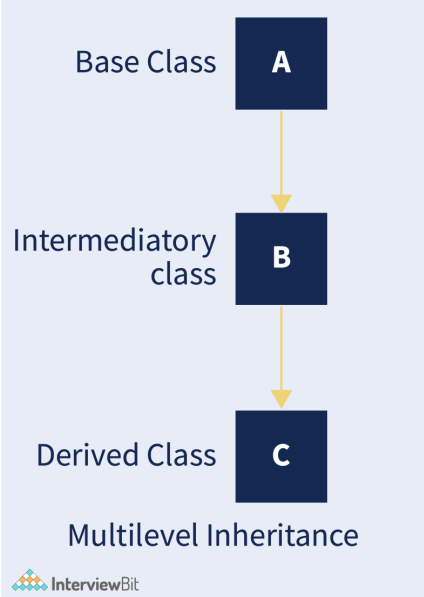
```
class Sample{
 void fun(){
  System.out.println("Inside Sample's fun method.");
 }
}
class Test extends Sample{
 void foo(){
  System.out.println("Inside Test's foo method.");
 }
}
class Main{
 public static void main(String args[]){
   Test t = new Test();
   t.fun();
   t.foo();
 }
}
```

Output -

```
Inside Sample's fun method.
Inside Test's foo method.
```

Explanation - In the above code, the class Test inherits from the class Sample. We create an object of the Test class and call the methods defined in both classes.

2. Multilevel Inheritance - Multilevel Inheritance: In Multilevel Inheritance, a derived class inherits from a base class, and the derived class also serves as the base class for other classes. Class A serves as a base class for derived class B, which in turn serves as a base class for derived class C in the diagram below. In Java, a class cannot directly access the members of a grandparent.

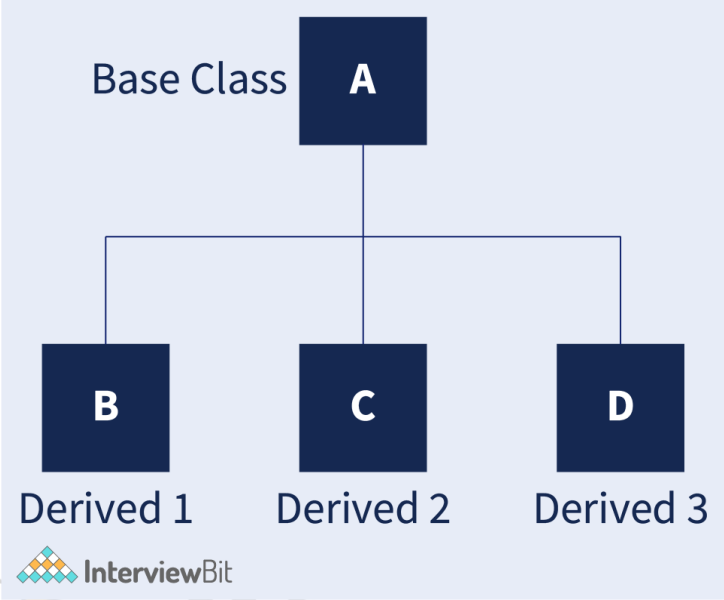


Example -

```
class Sample{
 void fun(){
  System.out.println("Inside Sample's fun method.");
 }
}
class Test extends Sample{
 void foo(){
  System.out.println("Inside Test's foo method.");
 }
}
class Result extends Test{
void display(){
 System.out.println("Inside display method of Result class.");
}
}
```

In the above code snippet, the class Test inherits from the class Sample and the class Result inherits from the class Test.

3. Hierarchical Inheritance - One class serves as a superclass (base class) for several subclasses in Hierarchical Inheritance. Class A acts as a base class for the derived classes B, C, and D in the diagram below.

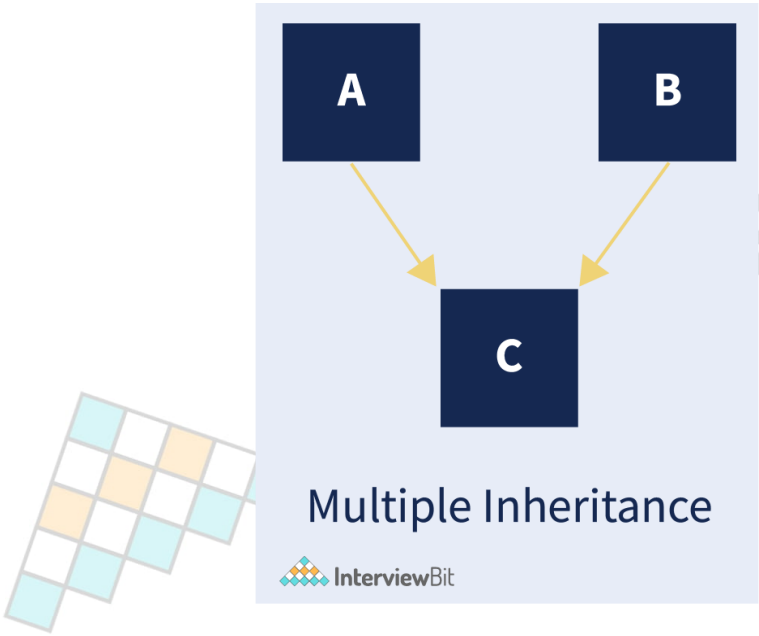


Example -

```
class Sample{
 void fun(){
  System.out.println("Inside Sample's fun method.");
 }
}
class Test extends Sample{
 void foo(){
  System.out.println("Inside Test's foo method.");
 }
}
class Result extends Sample{
void display(){
 System.out.println("Inside display method of Result class.");
}
}
class Example extends Sample{
void display(){
 System.out.println("Inside display method of Example class");
}
}
```

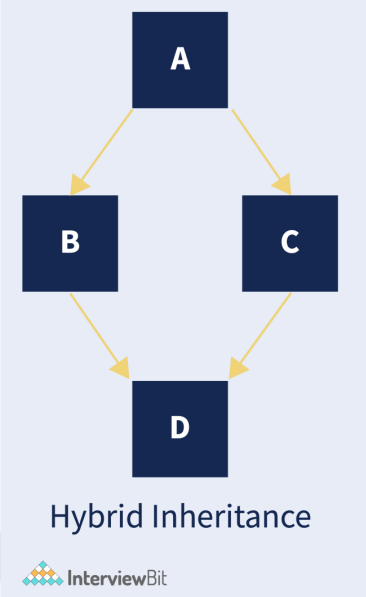
In the above code snippet, the classes Test, Result and Example inherit from the Sample class.

4. Multiple Inheritance - When a class inherits from more than one class, it is referred to as multiple inheritances. The below image shows class C inheriting from classes A and B.



Java does not support multiple inheritances as it can lead to ambiguity. We can implement the concept of multiple inheritances through the use of interfaces which we will discuss later in this article.

Hybrid Inheritance - Hybrid Inheritance is a blend of two or more of the inheritance kinds listed above. The below figure shows two classes B and C inheriting from class A and a class D inheriting from both classes B and C.



Thus hybrid inheritance requires multiple inheritances and since Java does not support multiple inheritances, hybrid inheritance is not supported by Java.

Diamond Problem in Java:

Let us consider the previous example of hybrid inheritance to understand the diamond problem. Let us also consider the following code snippet:

```
class Sample{
 void fun(){
  System.out.println("Inside Sample's fun method.");
 }
}
class Test extends Sample{
 void foo(){
  System.out.println("Inside Test's foo method.");
 }
}
class Result extends Sample{
void foo(){
 System.out.println("Inside display method of Result class.");
}
}
class Example extends Test and Result{
void display(){
 System.out.println("Inside display method of Example class");
}
}
class Main{
 public static void main(String args[]){
   Example e = new Example();
   e.foo(); // Ambiguity
 }
}
```

In the above code, the classes Test and Result inherit from the class Sample and the class Example inherits from the classes Test and Result. Here, we have assumed for now that Java supports multiple inheritances.
The above code gives a compilation error. The reason is that on calling the foo() method, there is an ambiguity of which foo() method is being referred to. There are 2 foo() method definitions available: one in the Test class and the other in the Result class. This is known as the diamond problem.

It is because of this problem that Java does not support multiple inheritances of classes and hence all of this is avoided.

15. Java Math Class

16. Abstract class and Interfaces

Abstract Class:-

In C++, a class becomes abstract if it has at least one pure virtual function. In Java, unlike C++, an abstract class is created using a separate keyword abstract. The following are some key points to remember about Java abstract classes.

- An abstract class instance cannot be created.
- The use of constructors is permitted.
- There is no need for an abstract method in an abstract class.
- Final methods aren't allowed in abstract classes since they can't be overridden, but abstract methods are designed to be overridden.
- We are prohibited from creating objects for any abstract class.
- In an abstract class, we can define static methods.

Example -

```
abstract class Sample1 {
   abstract void fun();
}

class Sample2 extends Sample2 {
   void fun()
   {
       System.out.println("Inside Sample2's fun function.");
   }
}

class Test {

   public static void main(String args[])
   {

       Base b = new Derived(); // We can have references of Base type.
       b.fun();
   }
}
```

Output -

```
Inside Sample2's fun function.
```

Explanation - In the above code, we have defined an abstract class Sample1. We declare the function definition in this class. We define another class Sample2 and extend it from Sample1. Here, we provide the implementation for the method of the abstract class inherited. Now we create an instance of the Sample2 class and invoke the fun() function.

Java Interfaces:-

- An interface, like a class, can include methods and variables, but the methods declared in an interface are abstract by default (only method signature, no body).
- Interfaces define what a class must do, not how it must do it.
- An interface is about capabilities; for example, a Player may be an interface, and any class that implements it must be able to (or must implement) movement (). As a result, it provides a set of methods that must be implemented by the class.
- If a class implements an interface but does not offer method bodies for all of the interface's functionalities, the class must be abstracted.

Syntax -

```
interface INTERFACE_NAME {
// constant fields
// methods
}
```

The interface keyword is used to declare an interface. Its purpose is to provide complete abstraction. All methods in an interface are declared with an empty body and are public by default, and all fields are public, static, and final. A class that implements an interface is required to implement all of the interface's functions. The keyword implements is used to implement an interface.

Reasons for introducing interfaces in Java:

- It's a technique for achieving complete abstraction.
- Because java does not provide multiple inheritances in the case of classes, multiple inheritances can be achieved by using interfaces.
- It can also be used for loose coupling.
- Abstraction is implemented through interfaces. So one question that comes to mind is why we should utilise interfaces when we have abstract classes. The reason for this is because abstract classes can have non-final variables, whereas interface variables are final, public, and static.

Example -

```
interface Sample
{

   final int a = 1; // public, static and final
   void display(); // public and abstract
}

// A class that implements the interface.
class Test implements Sample
{
   // Implementing the capabilities of the interface.
   public void display()
   {
       System.out.println("InterviewBit");
   }

   public static void main (String[] args)
   {
       Test t = new Test();
       t.display();
       System.out.println(a);
   }
}
```

Output -

```
InterviewBit
1
```

Explanation - In the above code, we created an interface Sample having a public, static and final data member ‘a’ and a function signature with name ‘display’. We create a class Test that implements this interface and provides the definition for the display function. We also access the data member ‘a’ defined in the interface.

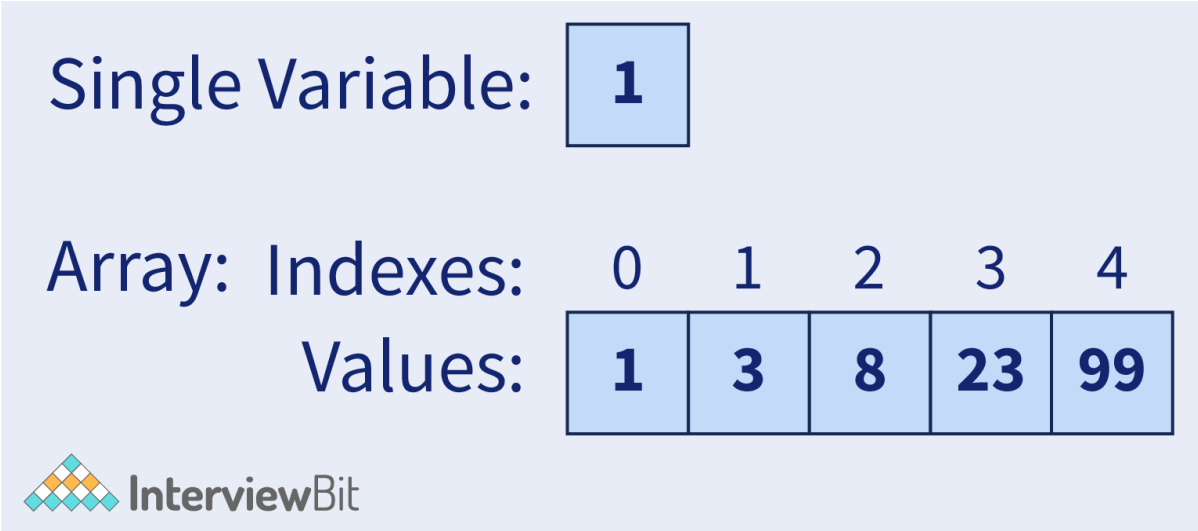
Abstract Class vs Interface:

The following table depicts the differences between abstract classes and interfaces in Java:

Abstract Class	Interface
Abstract and non-abstract methods can both be found in an abstract class. It can also contain default and static methods starting with Java 8.	An interface can only contain abstract methods.
Non-final variables can be present in abstract classes.	Variables declared in a Java interface are by default final variables.
Variables in an abstract class can be final, non-final, static, or non-static.	Only static and final variables are used in the interface.
The interface can be implemented using an abstract class.	An abstract class cannot be implemented using an interface.
The keyword "extends" can be used to extend an abstract class.	The keyword "implements" can be used to implement a Java interface.

17. Arrays in Java

In Java, an array is a collection of like-typed variables with a common name. Arrays in Java are not the same as those in C/C++. The following are some key points to remember regarding Java arrays.



- All arrays in Java are allocated dynamically. (explained further down)
- Because arrays are objects in Java, we may use the object attribute length to determine their length. This differs from C/C++, where we use sizeof to find the length.
- With [] following the data type, a Java array variable can be declared just like any other variable.
- The array's variables are sorted, and each has an index starting at 0.
- An array's size must be given using an int or short integer rather than a long number.
- Object is the array type's direct superclass.
- Every array type implements the Cloneable and java.io.Serializable interfaces.

Syntax for declaring an array:-

```
datatype variable_name[];
OR
datatype[] variable_name;
```

Despite the fact that the above declaration declares variable_name to be an array variable, no actual array exists. It just informs the compiler that this variable (variable_name) will contain an array. You must allocate memory space to it using the new operator.

Syntax of instantiating an array in java:-

```
variable_name = new datatype [size];
```

For example,

```
int sample_array[];    //declaring array
sample_array = new int[20];  // allocating memory for 20 integers to array
```

Array Literal:-

Array literals can be utilised in situations where the size of the array and its variables are already known.

For example,

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

The produced array's length is determined by the length of this sequence. In the most recent versions of Java, there is no need to write the new int[] portion.

18. Strings in Java

Strings are Objects in Java that are internally supported by a char array. Strings are immutable (that is, their content cannot be changed once initialised) because arrays are immutable. Every time you make a change to a String, a new String is produced.

String syntax in java:-

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

In Java, there are two ways to make a string:

Using String literal:

```
String str = "Interview Bit";
```

Using new keyword:

```
String str = new String ("Interview Bit");
```

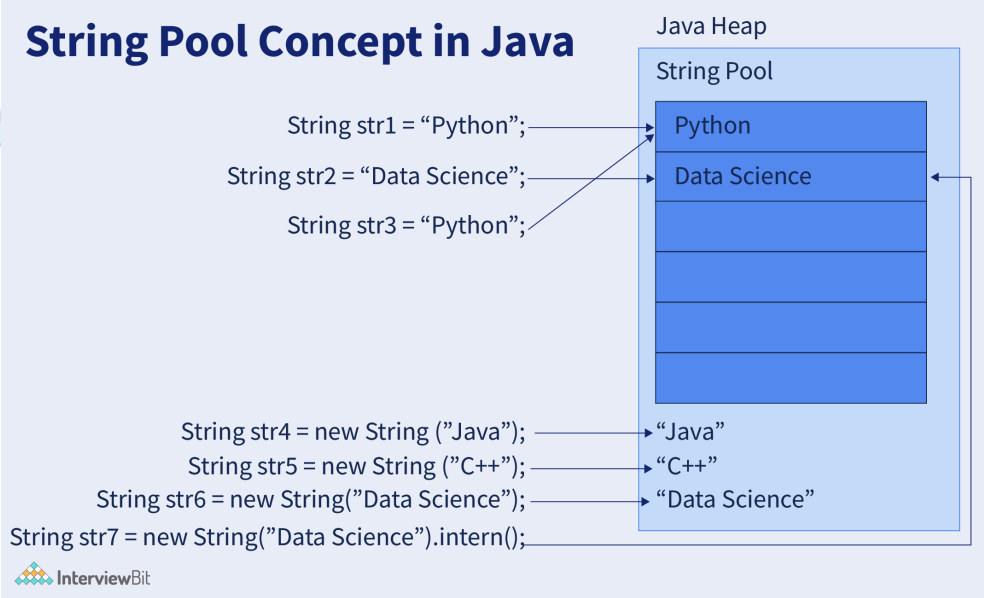
Difference between String literal and String object:

The following table lists the differences between String literal and String object:

String literal	String Object
In Java, a string literal is a collection of characters formed by enclosing them in a pair of double-quotes.	String Object is a Java object characters produced with the
If the String already exists in a literal, the new reference variable will point to the currently existing literal.	A new String object will be pr String currently exists or not.
The syntax for creating a String literal is as follows: `String string_name = “CONTENT_OF_STRING”;`	The syntax for creating a Strin `String object_name = new Str`

String Pool: String pool is a Java heap storage area where string literals are stored. String Intern Pool or String Constant Pool are other names for it. It's the same as object allocation. It is empty by default and is maintained privately by the Java String class. When we create a string, the string object takes up some memory in the heap. Creating a large number of strings may raise the cost and memory requirements, lowering performance.

During the initialization of string literals, the JVM takes several efforts to improve efficiency and reduce memory usage. The String class keeps a pool of strings to reduce the number of String objects created in the JVM.
When we construct a string literal, the JVM looks it up in the String pool first. It returns a reference to the pooled instance if the literal is already existing in the pool. If the literal isn't found in the pool, the String pool is filled with a new String object.



In the above image, we can see that the String pool is a portion of the heap memory maintained by Java to store String literals. We can see that the literals 'str1' and 'str2' point to the same memory area. However, we can see that a new String object is always created whether or not the String exists already. However, we can make the String object to check if the String already exists by using the intern() method.

Built-in String Methods:

Method	Return type	Use Case
charAt()	char	The character at the provided index is returned by charAt().
codePointBefore()	int	Returns the Unicode of the character before the specified index.
codePointAt()	int	codePointAt() returns the Unicode of the character at the specified index.
compareTo()	int	It compares two strings lexicographically
compareToIgnoreCase()	int	It compares two strings lexicographically, ignoring case differences.
concat()	String	A string is appended to the end of another string by this function.

- **StringBuffer:** A StringBuffer is a peer class that provides a lot of the same functionality as a String. StringBuffer represents growable and writable character sequences, whereas string represents fixed-length, immutable character sequences.
    - Syntax:

```
StringBuffer str = new StringBuffer("Interview Bit");
```

- **StringBuilder:** A mutable series of characters is represented by the StringBuilder in Java. Because Java's String Class creates an immutable sequence of characters, the StringBuilder class provides an alternative by creating a mutable sequence of characters.
    - Syntax:

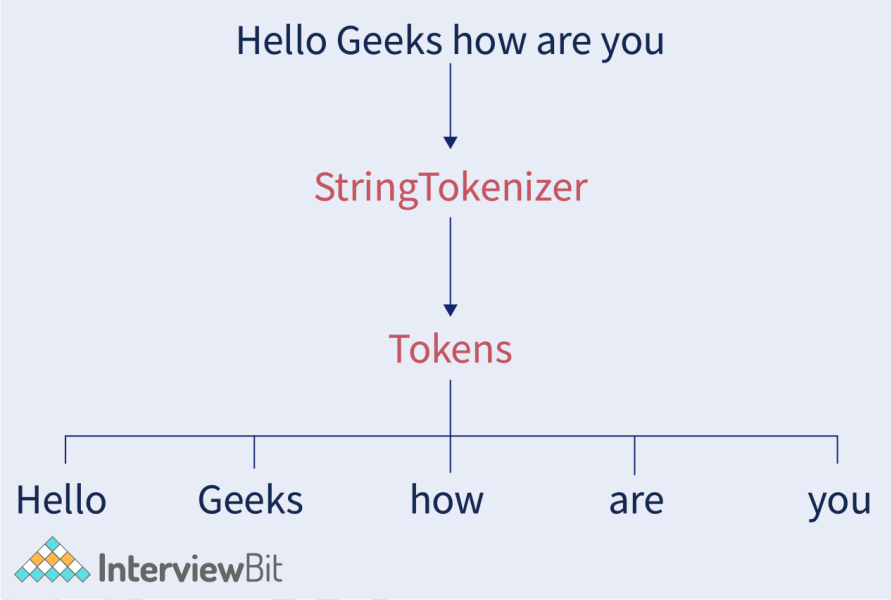
```
StringBuilder str = new StringBuilder();
str.append("Interview Bit");
```

String Buffer vs String Builder:

The following table lists the differences between String Buffer and String Builder in Java:

String Buffer	String Builder
StringBuffer is thread-safe since it is synchronised. It means that two threads can't execute the StringBuffer functions at the same time. If they call the StringBuffer function at the same time, only one of the two threads acquires the lock and executes the method. The second thread has to wait until the execution of the first thread has been completed.	StringBuilder is not thread-safe because it is not synchronised. It indicates that two threads can use StringBuilder's methods at the same time. In this case, no thread has to wait for the execution of the other thread.
String Buffer is considered to be less efficient than String Builder.	String Builder is considered to be more efficient than String Buffer.
String Buffer was introduced in Java 1.0.	String Builder was introduced in Java 1.5.

StringTokenizer: StringTokenizer is a Java class that is used to split a string into tokens.



In the above image, we can see that StringTokenizer divides the inputted string into tokens based on the default delimiter which is space.
Internally, a StringTokenizer object keeps track of where it is in the string being tokenized. Some procedures move this current location beyond the characters that have been processed. By taking a substring of the string that was used to generate the StringTokenizer object, a token is returned.

19. Java Regex

Regex (short for Regular Expressions) is a Java API for defining String patterns that may be used to search, manipulate, and modify strings. Regex is frequently used to specify the limits in a number of areas of strings, including email validation and passwords. The java.util.regex package contains regular expressions. This is made up of three classes and one interface. The java.util.regex package consists of the following three classes, which are listed below in the tabular format:

Class	Description
util.regex.Pattern	It is used to define patterns.
util.regex.Matcher	It is used to conduct match operations on text using patterns.
PatternSyntaxException	In a regular expression pattern, it's used to indicate a syntax problem.

Pattern class: There are no public constructors in this class. It is a collection of regular expressions that can be used to define various types of patterns. This can be done by executing the compile() method, which takes a regular expression as its first input and returns a pattern after it has been executed.

The following table lists the methods present in this class and their description:

Method	Description
compile(String regex)	Its purpose is to compile a regular expression into a pattern.
compile(String regex, int flags)	It's used to turn a regular expression into a pattern using the flags provided.
flags()	It's used to get the match flags for this pattern.
matcher(CharSequence input)	It's used to build a matcher that compares the given input to the pattern.
matches(String regex, CharSequence input)	matches(String regex, CharSequence input) is a function that compiles a regular expression and tries to match it against the given input.
pattern()	It's used to get the regular expression that was used to create this pattern.
quote(String s)	It's used to generate a literal pattern String from the given String.
split(CharSequence input)	This splits the given input sequence around patterns that match this pattern.

Matcher Class: This object is used to evaluate the previously described patterns by performing match operations on an input string in Java. There are no public constructors defined here either. Invoking a matcher() on any pattern object can be used to accomplish this.

The following table lists the methods present in this class and their description:

Method	Description
find()	find() is mostly used to look for multiple occurrences of regular expressions in a text.
find(int start)	It is used to find occurrences of regular expressions in the text starting from the provided index.
start()	start() is used to retrieve the start index of a match found with the find() method.
end()	It's used to acquire the end index of a match discovered with the find() method.
groupCount()	groupCount() is a function that returns the total number of matched subsequences.
matches()	It's used to see if the pattern matches the regular expression.

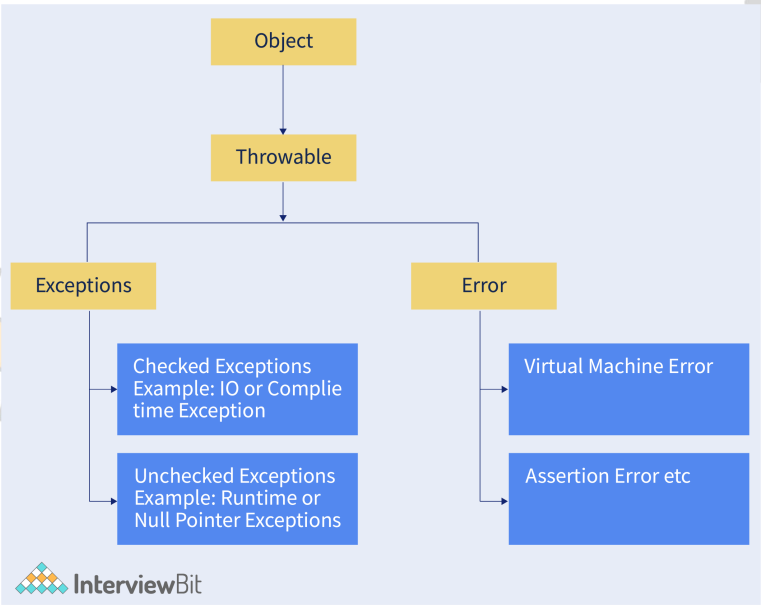
20. Java Exception Handling

Meaning of Exception: An exception is an unwelcome or unexpected occurrence that occurs during the execution of a program, i.e. at run time, and disturbs the program's usual flow of instructions.

Error vs Exception: What is the Difference?

An error implies that there is a major problem that a reasonable programme should not attempt to solve whereas an exception denotes a set of circumstances that a reasonable programme would attempt to catch.

Hierarchy of Exceptions:



As we can see in the above image, all exception and error kinds are subclasses of the hierarchy's root class, Throwable. Exceptions can be divided into two categories:

- **Checked Exceptions:** This includes IO Exceptions and Compile time Exceptions.
- **Unchecked Exceptions:** This includes Runtime Exceptions and Null Pointer Exceptions.

Built-in Exceptions in Java: Exceptions that are present in Java libraries are known as built-in exceptions. The following table lists the built-in exceptions in Java:

Exception	Description
ArithmeticException	When an unusual condition in an arithmetic operation occurs, it is thrown.
ArrayIndexOutOfBoundsException	It's thrown when an array has been accessed with an incorrect index. The index, in this case, is either negative, or greater than or equal to the array's size.
ClassNotFoundException	When we try to access a class whose definition is missing, this Exception is thrown.
FileNotFoundException	When a file is not accessible or does not open, this Exception is thrown.
IOException	When an input-output operation fails or is interrupted, this exception is issued.
InterruptedException	When a thread is interrupted while waiting, sleeping, or processing, this exception is issued

Java Try-Catch:-

- **try block**: The try block comprises a set of statements that may throw an exception.

```
Syntax -
try
{
// code to be checked
}
```

- **catch block:** The catch block is used to manage the try block's unclear condition. A try block is always followed by a catch block, which deals with any exceptions thrown by the try block.

```
Syntax -
catch
{
// code to handle exception
}
```

finally keyword:-

In Java, the finally block represents code that is to be executed whether or not an exception is caught. In case a catch block is present after the try block, it is run after the catch block has been executed. However, in case there is no catch block present after the try block, the finally block is executed after the try block has been executed.

Syntax -

```
finally
{
// code to be executed
}
```

Difference between final, finally and finalize:

The following table lists the differences between final, finally and finalize:

final	finally	finalize
final is a keyword and access modifier for restricting access to a class, method, or variable	finally represents a block of code to be executed whether or not an exception has been caught.	finalize is a Java method that performs cleanup operations immediately before an object is garbage collected.
The final keyword is used with the classes, methods and variables.	The finally block is always linked to the try and catch block in exception handling.	The finalize() method is used with the objects.
A final method is executed only when we call it	As soon as the try-catch block is finished, the finally block is started.	It doesn't rely on exceptions for execution.

throw keyword:- In Java, the throw keyword is used to throw an exception from a method or any block of code. We can either throw a checked or an unchecked exception. Throwing custom exceptions is the most common use of the throw keyword.

throws keyword:- The throws keyword is used to handle exceptions in the absence of a try/catch block. It specifies the exceptions that a method should throw in case an exception occurs.

Example -

```
class Test {

    // This method throws an exception
    static void foo() throws IllegalAccessException
    {
        System.out.println("Inside the foo() method");
        throw new IllegalAccessException("demo");
    }

    public static void main(String args[])
    {
        try {
            foo();
        }
        catch (IllegalAccessException e) {
            System.out.println("Exception caught");
        }
    }
}
```

Output -

```
Inside the foo() method
Exception caught
```

Explanation - In the above code, the function foo() specifies that an IllegalAccessException should be thrown in case an exception occurs using the throws keyword. In the foo() function, we explicitly throw an exception. This is caught by the catch block present in the main method of the class.

21. Java Commands

Following are the most widely used java commands:-

1. java –version: This is one of the most fundamental Java commands for checking the Java version installed on your machine. This is also used to confirm that the installation and PATH variable settings are correct.

```
java version "1.8.0_181"
java<TM> SE Runtime Environment <build 1.8.0_181-b13>
Java HotSpot<TM> 64-Bit Server VM <build 25.181-b13,mixed mode>
```

InterviewBit

The above image is a snapshot of the output we get on running the java -version command.

2. javac –version: This command displays the version of the compiler that is in charge of compiling the source code. This is also a component of the Java Development Kit, or JDK for short.

```
C:\Program Files\Java\jdk1.8.0_181\bin\javac -version
javac 1.8.0_181
```

InterviewBit

The above image is a snapshot of the output we get on running the javac -version command.

3. whereis: This Java command searches the directory for a given component. In the example below, we've taken into account javac.

```
where is javac
javac: /usr/bin/javac /usr/bin/X11/javac
```

InterviewBit

The above image is a snapshot of the output we get on running the command whereis javac.

4. echo: The echo command is a must-know command in Java since it allows you to display the contents of a specific file. In most cases, this is used to verify the PATH variables.

```
C:\Program Files\Java\jdk1.8.0_181>echo %PATH%
C:\Program Files <x86>\Common Files\Oracle\Java\javapath;C:\Windows\system32;C:\
Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\
MinGW\msys\1.0\bin;C:\Java\bin;C:\Program Files\PuTTY\;C:\Program Files <x86>\Go
w\bin;C:\Program Files <x86>\Common Files\Oracle\Java\javapath;C:\Windows\system
32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.
0\;C:\MinGW\ms ys\1.0\bin;C:\Java\bin;C:\Program Files\PuTTY\;C:\Program Files <X
86>\Gow\bin;C:\opt\spark\spark-2.3.0-bin-hadoop2.7\bin;C:\ProgramData\Anaconda3;
C:\ProgramData\Anaconda3\Scripts
```

InterviewBit

The above image is a snapshot of the output we get on running the command echo %PATH%

5. javap: The javap command is used to disassemble one or more class files. The outcome is determined by the choices made. When no options are specified, the javap command prints both protected and public fields, as well as all of the methods of the classes supplied to it.

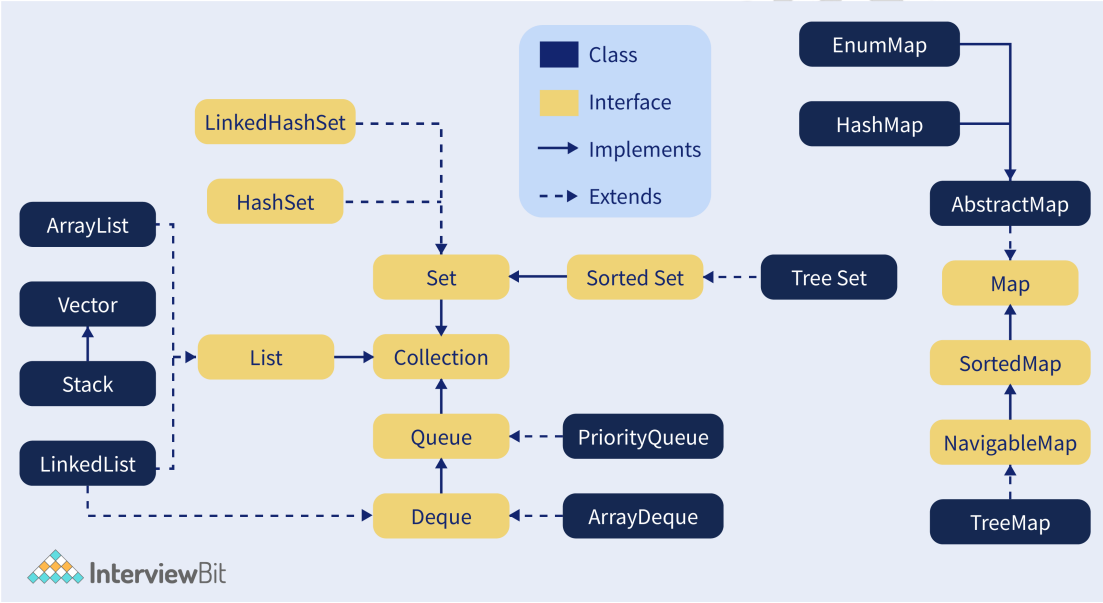
6. javah: Using this tool, you can produce c header and source files from a java class automatically. The generated c headers and source files are used to implement native methods and to refer to an object's instance variable in the native source code.

7. Javadoc: The Javadoc command and its arguments are used to generate HTML pages or API documentation from a group of Java source files in a seamless manner. This tool parses the declarations and documentation comments in an array of java source files and generates HTML pages that describe the public and protected classes, interfaces, fields, and nested classes, according to the default parameters.

22. Java Collections

The collection of objects refers to any group of individual objects that are represented as a single unit. In JDK 1.2, a new framework called “Collection Framework” was established, which contains all of the collection classes and interfaces.

The Collection interface (present in the java.util.Collection package) and the Map interface (present in the java.util.Map package) are the two basic root interfaces of Java collection classes.



The above image shows various interfaces and the classes present in those interfaces in the **Java Collections** framework. The ones in yellow colour represent interfaces while the ones in blue colour represent classes that implement those interfaces.

The following table lists the methods present in the collection interface:

Method	Description
add(Object)	It is a method for adding an object to a collection.
addAll(Collection c)	This function populates this collection with all of the pieces from the provided collection.
clear()	This method clears the collection of all its items.
contains(Object o)	This method clears the collection of all its items.
containsAll(Collection c)	If the collection contains all of the elements in the given collection, this method returns true.
equals(Object o)	This method checks for equality between the specified object and this collection.
hashCode()	It is a method that returns the hash code value for a collection.
isEmpty()	It returns true if there are no elements in this collection.
iterator()	It returns an iterator that traverses the elements of this collection.
max()	The maximum value in the collection is returned by this method.
size()	The number of elements in the

Interfaces that extends the Collections interface:-

The following are the interfaces present in the Collection framework of Java:

1. **Iterable Interface:** This is the collection framework's root interface. The iterable interface is extended by the collection interface. As a result, all interfaces and classes implement this interface by default. This interface's main purpose is to provide an iterator for the collections. As a result, this interface only has one abstract method, the iterator.
2. **Collection Interface:** The Collection Interface extends the iterable interface and is implemented by all of the collection framework's classes. This interface covers all of the basic methods that every collection has, such as adding data to the collection, removing data from the collection, clearing data, and so on. All of these methods are implemented in this interface since they are used by all classes, regardless of their implementation style. Furthermore, including these methods in this interface guarantees that the method names are consistent across all collections. In summary, we may conclude that this interface lays the groundwork for the implementation of collection classes.
3. **List Interface:** The collection interface has a child interface called the list interface. This interface is dedicated to list data, in which we can store all of the objects in an ordered collection. This also allows for the presence of redundant data. Various classes, such as ArrayList, Vector, Stack, and others, implement this list interface. We can create a list object with any of these classes because they all implement the list.
    1. **Java ArrayList:** In Java, ArrayList allows us to create dynamic arrays. Though it may be slower than normal arrays, it might be useful in programs that require a lot of array manipulation. If the collection increases, the size of an ArrayList is automatically increased, and if the collection shrinks, the size of an ArrayList is automatically decreased. The Java ArrayList allows us to access the list at random. Primitive types, such as int, char, and so on, cannot be utilised with ArrayList. In such circumstances, we'll require a wrapper class.
    Syntax -
    ```
    ArrayList<ObjectType>name=new ArrayList<ObjectType>(size_of_collection);
    ```
    2. **Java LinkedList:** The LinkedList class implements the LinkedList data structure, which is a linear data structure with items not stored in contiguous locations with each element being a separate object having a data and address part. Pointers and addresses are used to connect the elements. Every element is referred to as a node.
    Syntax -
    ```
    LinkedList<ObjectType> name = new LinkedList<ObjectType>
    (size_of_collection);
    ```

23. Java Generics

Generics refer to types that have been parameterized. The goal is to make type (Integer, String, etc., as well as user-defined types) a parameter for methods, classes, and interfaces. Generics can be used to design classes that function with a variety of data types. A generic entity is a type that works on a parameterized type, such as a class, interface, or method. In Java, generics are equivalent to templates in C++. Generics are used extensively in classes such as HashSet, ArrayList, HashMap, and others.

Generic class:

In the same way that C++ specifies parameter types, we use <> to define parameter types in generic class formation. The following syntax is used to construct generic class objects.

```
ClassName <Type> obj = new ClassName <Type>()
```

Here, 'ClassName' denotes the name of the class whose instance is to be created. <Type> specifies the data type to be used while instantiating the object. 'obj' is the name of the object to be created.

Example -

```
class Sample<T>
{
   // Declaring an object of type T
   T obj;
   Test(T obj) // constructor
   {
       this.obj = obj;
   }
   public T getObject()
   {
       return this.obj;
   }
}
class Test
{
   public static void main (String[] args)
   {
       // instance of Integer type
       Sample <Integer> obj1 = new Sample<Integer>(10);
       System.out.println(obj1.getObject());
       // instance of String type
       Test <String> obj2 = new Test<String>("Interview Bit");
       System.out.println(obj2.getObject());
   }
}
```

Output -

```
10
Interview Bit
```

In the above example, a generic class Sample has been created which accepts a parameter T to determine the type of the class. We create 2 instances of the Sample class. One with an Integer type and the other with a String type.

The following are the advantages of using generics in Java:

Type-safety: In generics, we can only hold one type of object. It does not allow for the storage of objects of different types. Example -

```
List list = new ArrayList();
list.add(1);
list.add("Interview Bit");
```

The above code snippet runs fine. This is because we have not specified any type while creating an instance of the List class. However, if we run the below code snippet, we get a compile-time error.

```
List<Integer> list = new ArrayList<Integer>();
list.add(1);
list.add("Interview Bit");// compile-time error
```

This is because we have specified the instance of the List class to be of type Integer.

Typecasting isn't necessary: When we use generics, we do not need to typecast the object when we access it.

Example -

```
List list = new ArrayList();
list.add("Interview Bit");
String s = (String) list.get(0);//typecasting
```

In the above code, we can see that before accessing the element of the list, we need to typecast it to String type explicitly. This is because we have not declared the list instance to be of a specific type. However, if we run the below code snippet, we do not need to typecast it explicitly.

```
List<String> list = new ArrayList<String>();
list.add("Interview bit");
String s = list.get(0);
```

This is because we have specified the instance of the List class to be of String type.

Checking at Compile Time: It is checked at compile time to ensure that an issue does not arise at runtime. It is considerably better to handle the problem at compile time than at runtime, according to an excellent programming approach.

Example -

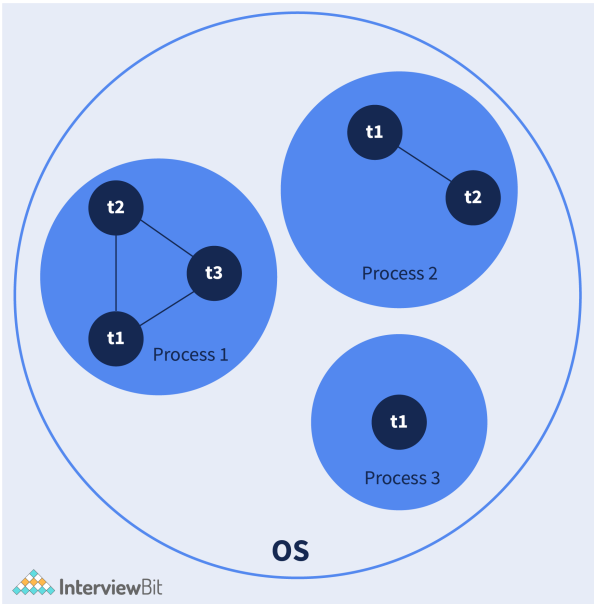
```
List<String> list = new ArrayList<String>();
list.add("Interview Bit");
list.add(100);//Compile Time Error
```

The above code snippet gives a compile-time error. This verifies that the code is checked at compile-time.

24. Java Multithreading

What is Multithreading?

Multithreading is a Java feature that permits the execution of two or more portions of a program at the same time to maximise CPU efficiency. Each such portion of the program is referred to as a thread.
Threads are lightweight processes within processes. Multitasking is accomplished through the use of multiprocessing and multithreading. Because threads share a memory, we employ multithreading rather than multiprocessing. They conserve memory by not allocating separate memory space, and context-switching between threads takes less time than processing.



The above image shows 3 processes wherein process 1 consists of 3 threads, process 2 consists of 2 threads and process 3 consists of 1 thread.

Java Runnable Interface: In Java, java.lang.Runnable is an interface that a class must implement if its instances are to be executed by a thread.

Java Thread Class: Thread programming is possible with Java's Thread class. The Thread class contains constructors and methods for creating and operating on threads. Thread is a subclass of Object that implements the Runnable interface.

Methods of the Java Thread Class:

The following table shows the methods of the Java Thread class and its use cases.

Method	Modifier and return type	Use Case
start()	void	start() is used the thread's e
run()	void	It specifies the be executed b thread.
sleep()	static void	This function thread for the time provided
currentThread()	static Thread	It returns a ref the thread obj presently runn
join()	void	This function thread to term
getPriority()	int	This function the thread's p
getName()	String	getName() ret thread's name
setName()	void	setName() mo thread's name
isAlive()	boolean	isAlive() is a bo function that determines w not the thread
getId()	long	This function the thread's id

Java Multithreading has the following **advantages**:

1. It does not impede the user because threads are independent and can conduct many operations at the same time.
2. It saves time by allowing you to conduct multiple procedures at once.
3. Because threads are self-contained, an exception in one thread has no impact on other threads.

How to implement multithreading in Java?

Multithreading can be performed in Java using two different mechanisms:

- By Extending the Thread class.
- By implementing the Runnable Interface.

By extending the Thread class:

We'll make a class that extends the java.lang.Thread class. The run() method of the Thread class is overridden by this class. The run() procedure is where a thread starts its life. To begin thread execution, we construct an object of our new class and use the start() method. Start() calls the Thread object's run() function.

Example -

```
class Sample extends Thread {
   public void run()
   {
       try {
           // Displaying the thread that is running
           System.out.println(
               "Thread " + Thread.currentThread().getId()
               + " is running");
       }
       catch (Exception e) {
           // Throwing an exception
           System.out.println("An exception is caught");
       }
   }
}

class Multithread {
   public static void main(String[] args)
   {
       int n = 5; // Number of threads
       for (int i = 0; i < n; i++) {
           Sample obj
               = new Sample();
           obj.start();
       }
   }
}
```

Output -

```
Thread 20 is running
Thread 18 is running
Thread 21 is running
Thread 19 is running
Thread 17 is running
```

Explanation - In the above code, the Sample class extends the Thread class present in java.lang package. In the Multithread class, we create 5 threads by creating an instance of the Sample class. We then invoke the run() method of the instance created by calling the start() method of the instance.

By implementing the Runnable Interface:

We make a new class that implements the java.lang.Runnable interface and overrides the run() method. After that, we create a Thread object and call its start() method.

Example -

```
class Sample implements Runnable {
   public void run()
   {
       try {
           // Displaying the thread that is running
           System.out.println(
               "Thread " + Thread.currentThread().getId()
               + " is running");
       }
       catch (Exception e) {
           // Throwing an exception
           System.out.println("An exception is caught");
       }
   }
}

class Multithread {
   public static void main(String[] args)
   {
       int n = 5; // Number of threads
       for (int i = 0; i < n; i++) {
           Thread obj
               = new Thread(new Sample());
           obj.start();
       }
   }
}
```

Output -

```
Thread 20 is running
Thread 18 is running
Thread 21 is running
Thread 19 is running
Thread 17 is running
```

Explanation: In the above code, the class Sample implements the Runnable interface. In the sample class, we override the run() method. In the Multithread class, we create 5 threads by creating an instance of the Thread class.

Runnable Interface vs. Thread Class:

The following are the key differences between using the Runnable interface and the Thread class:

1. Because Java doesn't support multiple inheritances, if we extend the Thread class, we won't be able to extend any other classes. Our class, however, can still extend other base classes if we implement the Runnable interface.
2. We can obtain rudimentary thread functionality by extending the Thread class, which has several built-in methods like yield() and interrupt() that aren't available in the Runnable interface.
3. When you use runnable, you'll get an object that can be shared by numerous threads.

Additional Resources:

1. https://www.interviewbit.com/blog/java-developer-skills/
2. https://www.interviewbit.com/blog/java-projects/
3. https://www.interviewbit.com/blog/java-frameworks/
4. https://www.interviewbit.com/java-mcq/

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions