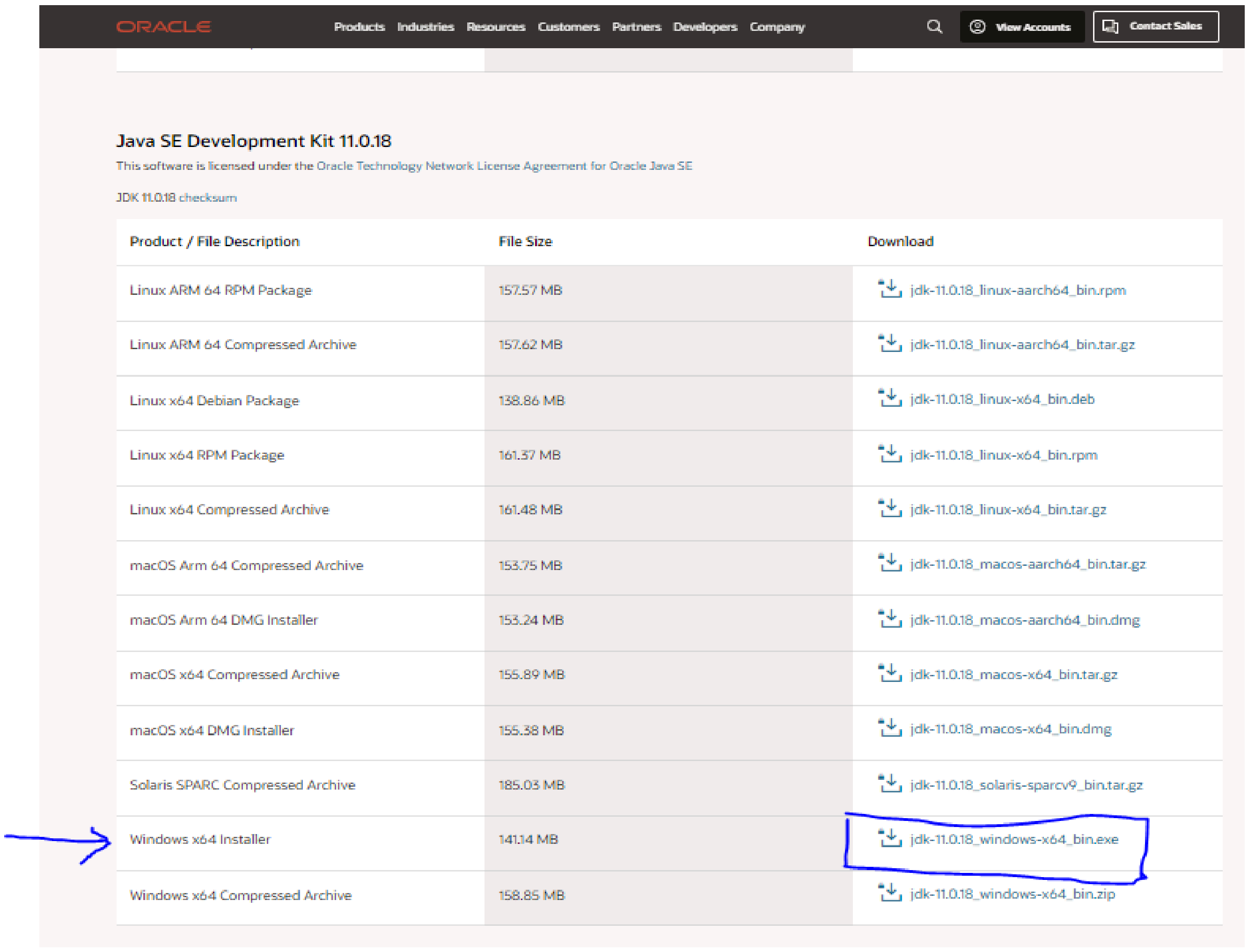
Java Installation :

We will install Java 11 for Windows , officially distributed by Oracle.

Step 1 : Download JDK

Open the browser and search for
https://www.oracle.com/in/java/technologies/javase/jdk11-archive-downloads.html
.
It will show the below page.We will install Windows x64 installer from Java SE Development Kit 11.0.18 .


ORACLE
Products Industries Resources Customers Partners Developers Company
View Accounts
Contact Sales

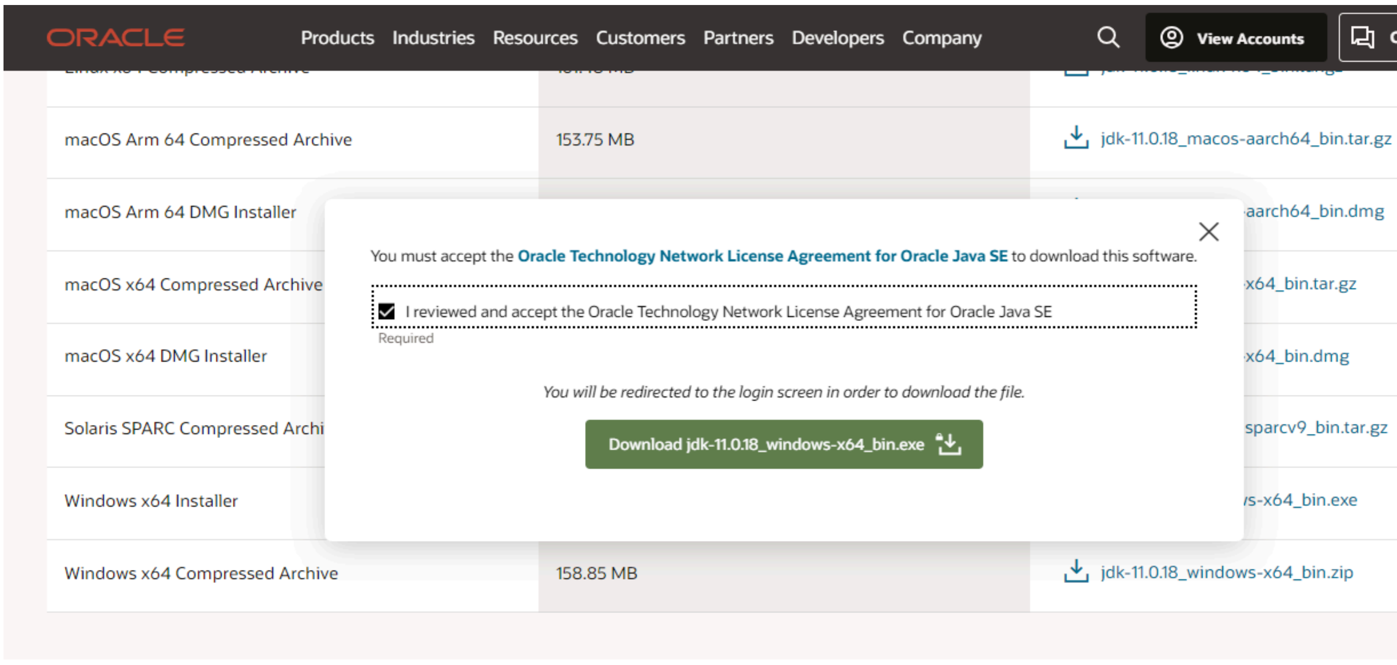
Java SE Development Kit 11.0.18

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE

JDK 11.0.18 checksum

Product / File Description	File Size	Download
Linux ARM 64 RPM Package	157.57 MB	jdk-11.0.18_linux-aarch64_bin.rpm
Linux ARM 64 Compressed Archive	157.62 MB	jdk-11.0.18_linux-aarch64_bin.tar.gz
Linux x64 Debian Package	138.86 MB	jdk-11.0.18_linux-x64_bin.deb
Linux x64 RPM Package	161.37 MB	jdk-11.0.18_linux-x64_bin.rpm
Linux x64 Compressed Archive	161.48 MB	jdk-11.0.18_linux-x64_bin.tar.gz
macOS Arm 64 Compressed Archive	153.75 MB	jdk-11.0.18_macos-aarch64_bin.tar.gz
macOS Arm 64 DMG Installer	153.24 MB	jdk-11.0.18_macos-aarch64_bin.dmg
macOS x64 Compressed Archive	155.89 MB	jdk-11.0.18_macos-x64_bin.tar.gz
macOS x64 DMG Installer	155.38 MB	jdk-11.0.18_macos-x64_bin.dmg
Solaris SPARC Compressed Archive	185.03 MB	jdk-11.0.18_solaris-sparcv9_bin.tar.gz
Windows x64 Installer	141.14 MB	jdk-11.0.18_windows-x64_bin.exe
Windows x64 Compressed Archive	158.85 MB	jdk-11.0.18_windows-x64_bin.zip

Accept the License Agreement and click on the link to download the installer.



macOS Arm 64 Compressed Archive	153.75 MB	jdk-11.0.18_macos-aarch64_bin.tar.gz
macOS Arm 64 DMG Installer		aarch64_bin.dmg
macOS x64 Compressed Archive		x64_bin.tar.gz
macOS x64 DMG Installer		x64_bin.dmg
Solaris SPARC Compressed Archi		sparcv9_bin.tar.gz
Windows x64 Installer		s-x64_bin.exe
Windows x64 Compressed Archive	158.85 MB	jdk-11.0.18_windows-x64_bin.zip

You must accept the **Oracle Technology Network License Agreement for Oracle Java SE** to download this software.

☑ I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE

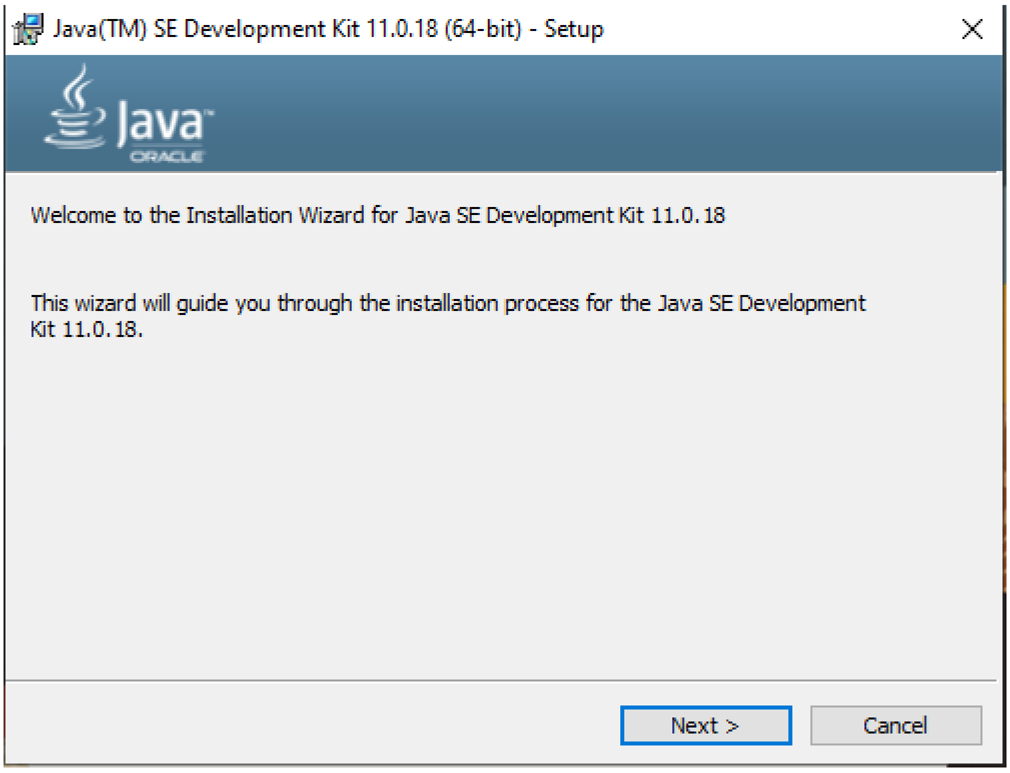
Required

You will be redirected to the login screen in order to download the file.

Download jdk-11.0.18_windows-x64_bin.exe

Step 2 - Install JDK

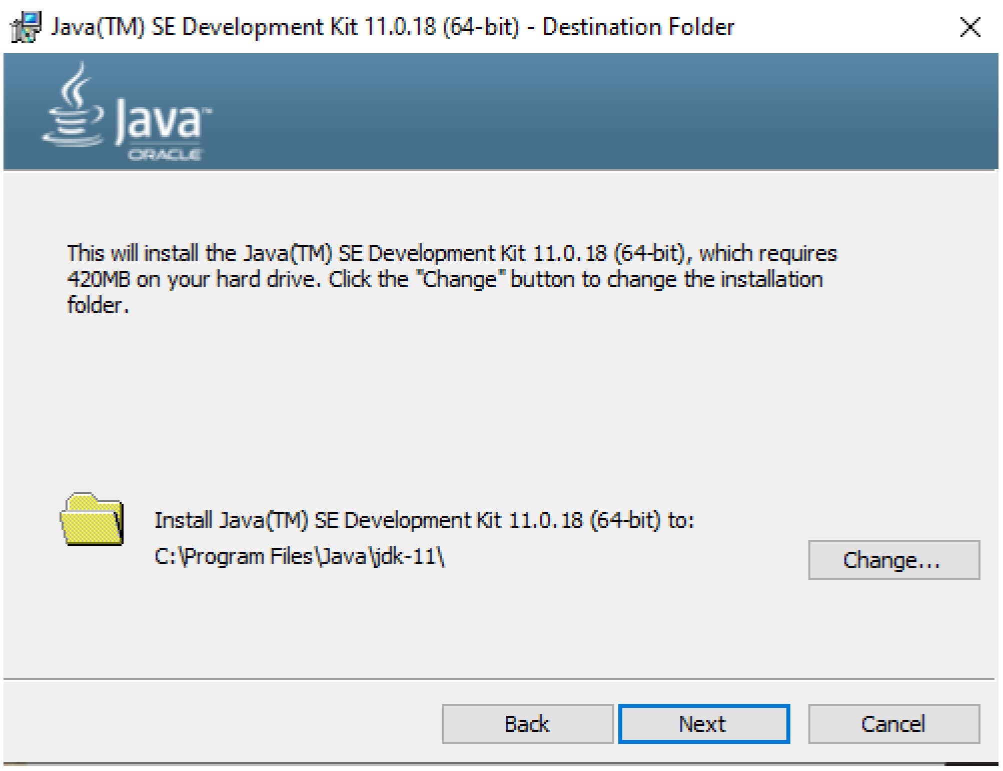
Now execute the downloaded JDK installer by double-clicking it. It might ask system permission before starting the installation. Click on yes to allow the installer to execute itself. It shows the installer welcome screen


Java(TM) SE Development Kit 11.0.18 (64-bit) - Setup

Welcome to the Installation Wizard for Java SE Development Kit 11.0.18

This wizard will guide you through the installation process for the Java SE Development Kit 11.0.18.

Click on Next to initiate the installation process. The next screen shows options to select optional features to be installed together. Leave the default options without making any change.

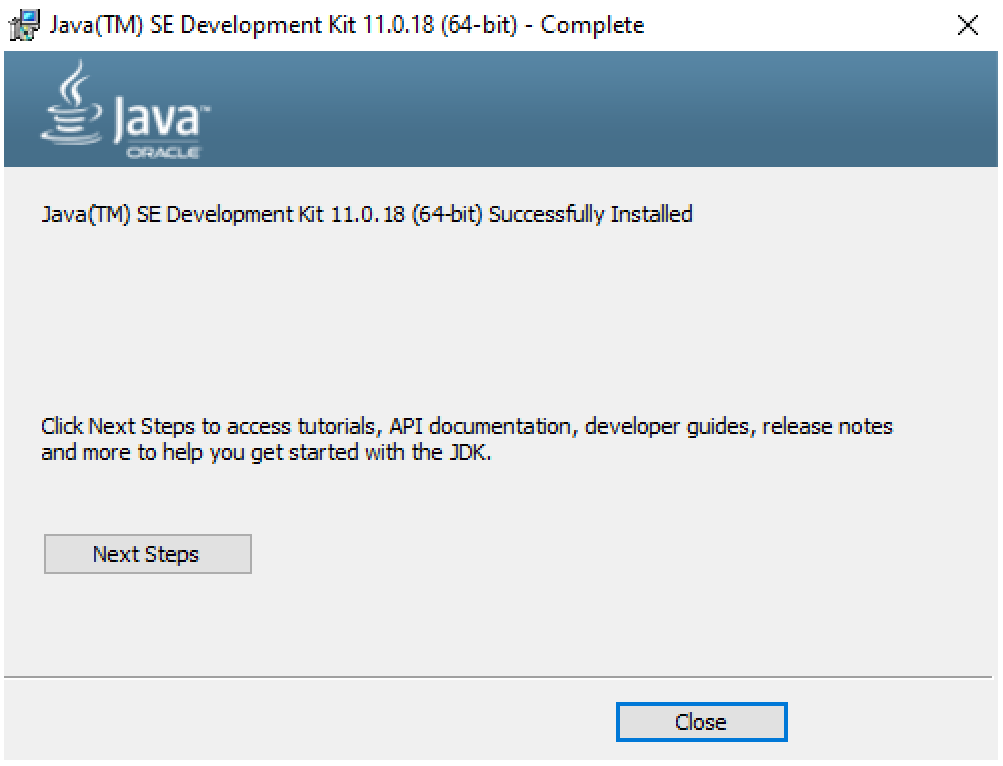

Java(TM) SE Development Kit 11.0.18 (64-bit) - Destination Folder

This will install the Java(TM) SE Development Kit 11.0.18 (64-bit), which requires 420MB on your hard drive. Click the "Change" button to change the installation folder.

Install Java(TM) SE Development Kit 11.0.18 (64-bit) to:
C:\Program Files\Java\jdk-11\

Now click on Next Button to start the installation. It will show the progress.

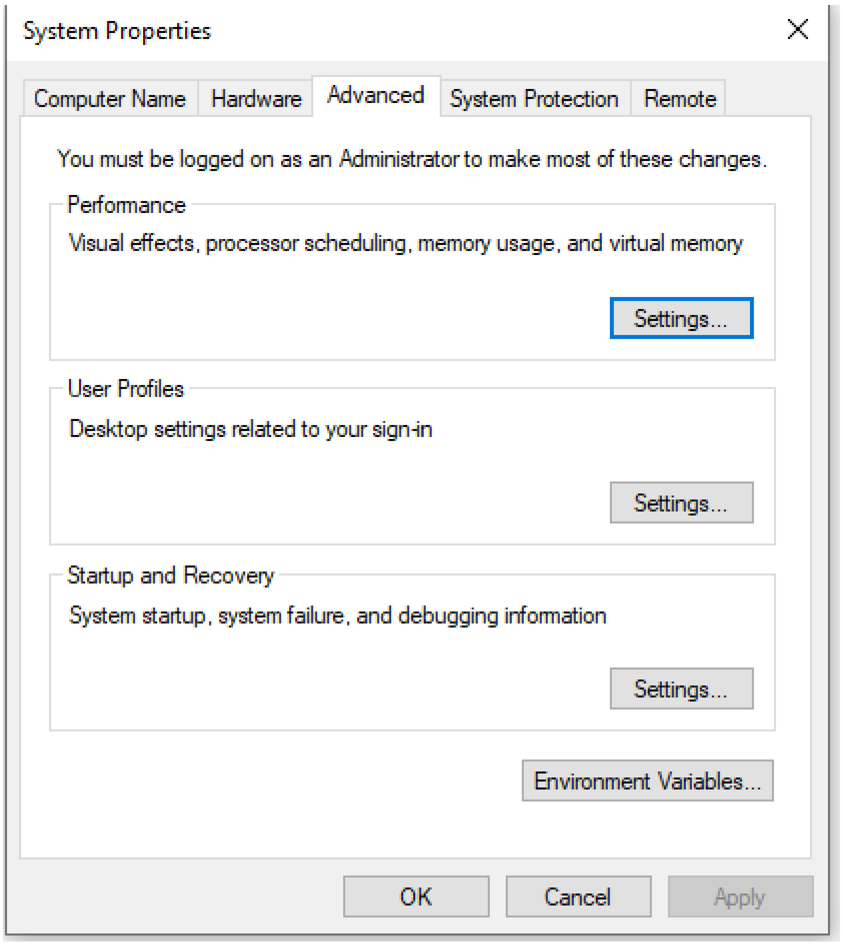
It shows the success screen after completing the installation .


Java(TM) SE Development Kit 11.0.18 (64-bit) - Complete

Java(TM) SE Development Kit 11.0.18 (64-bit) Successfully Installed

Click Next Steps to access tutorials, API documentation, developer guides, release notes and more to help you get started with the JDK.

Step 3 - We will set the environment variables by right clicking on My PC and opening properties.Or, you can search for Environment variables directly.


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

Settings...

Environment Variables...

OK | Cancel | Apply

Click on environment variables.

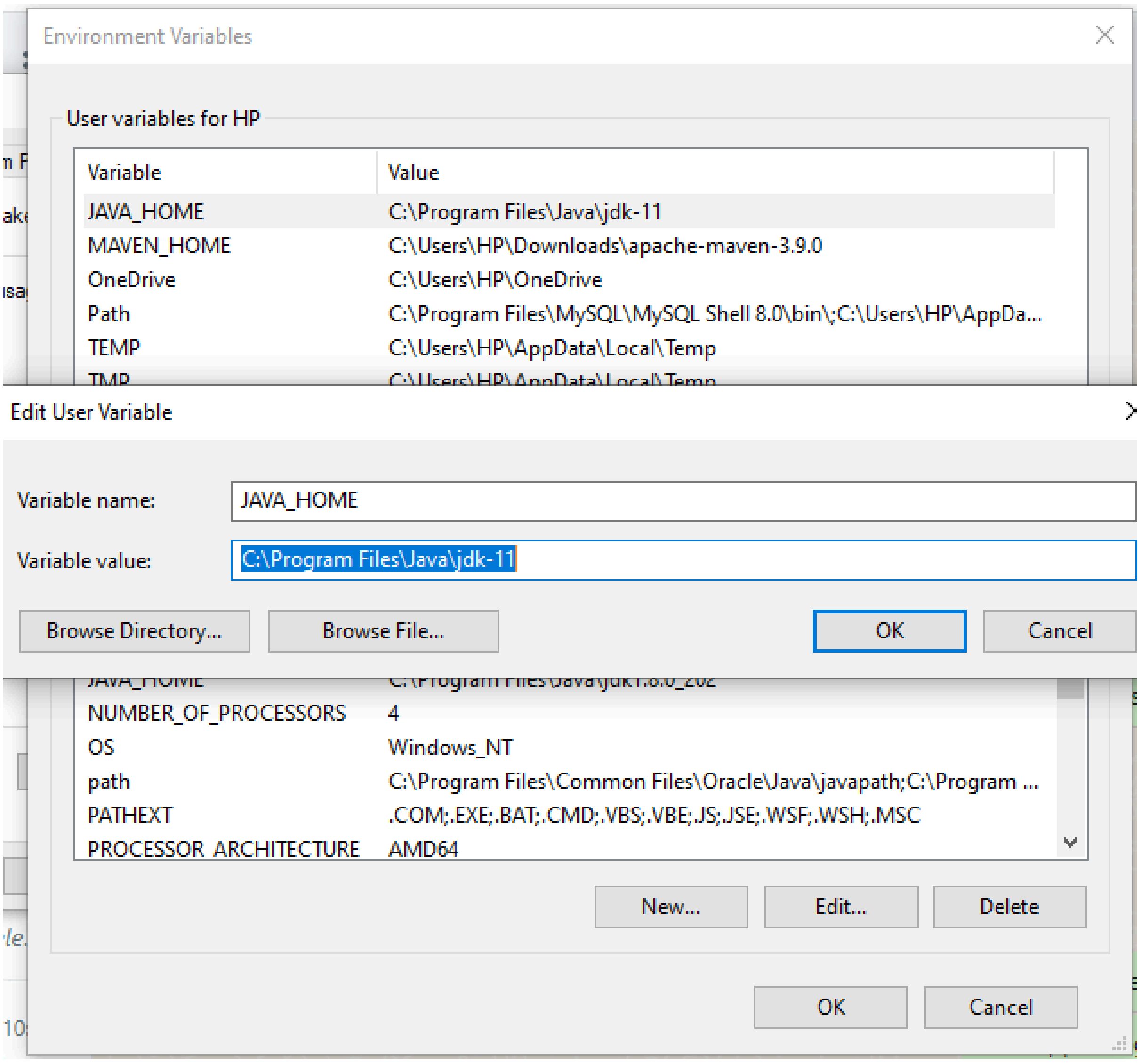
We will set the system and user variable paths.Please do not delete any variable from here.

Click on New under User Variable if no version of java is installed before this.

Set Variable name as JAVA_HOME.

And , set the value to the path where it is installed on the system.

If any other version of java in installed , click on it and than click Edit and change the path.



Environment Variables

User variables for HP

Variable	Value
JAVA_HOME	C:\Program Files\Java\jdk-11
MAVEN_HOME	C:\Users\HP\Downloads\apache-maven-3.9.0
OneDrive	C:\Users\HP\OneDrive
Path	C:\Program Files\MySQL\MySQL Shell 8.0\bin\;C:\Users\HP\AppDa...
TEMP	C:\Users\HP\AppData\Local\Temp

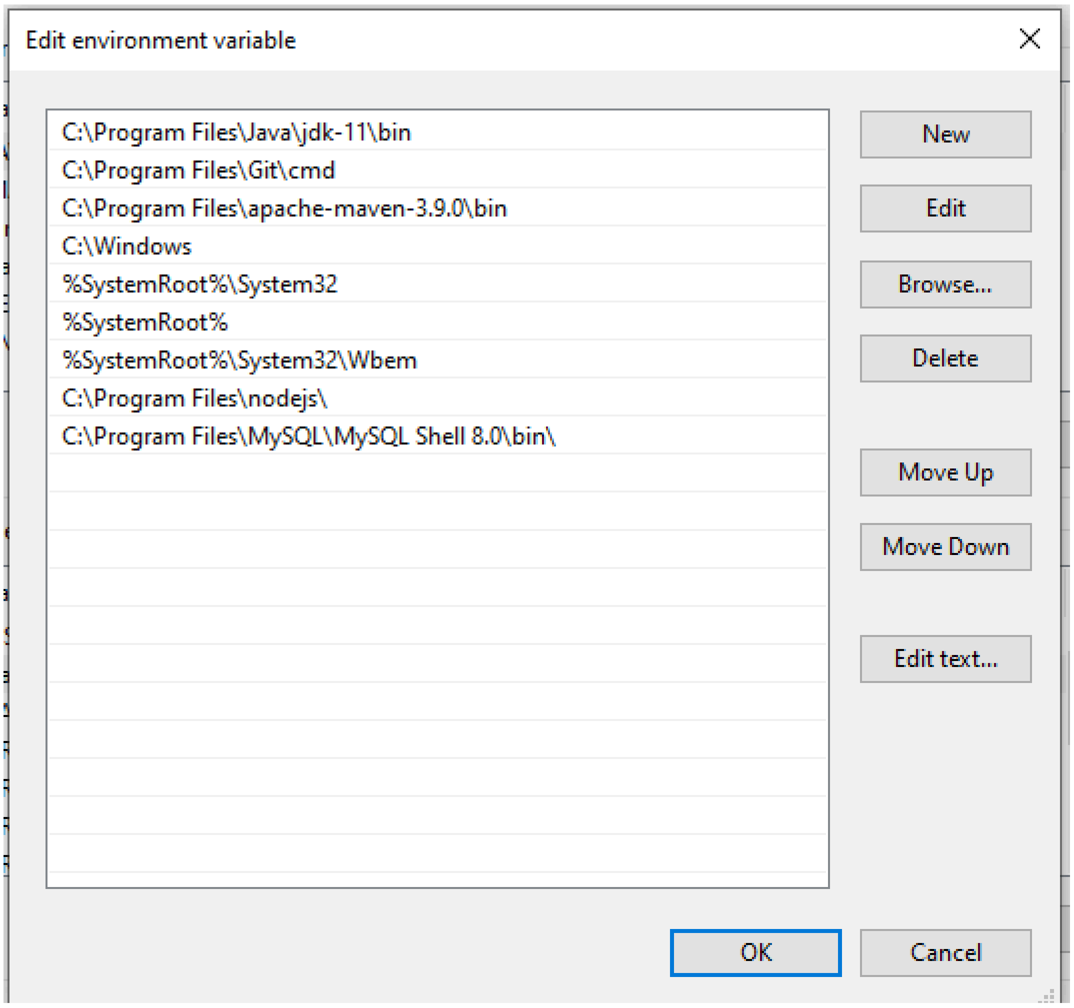
Edit User Variable

Variable name: JAVA_HOME

Variable value: C:\Program Files\Java\jdk-11

Variable	Value
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
path	C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program ...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

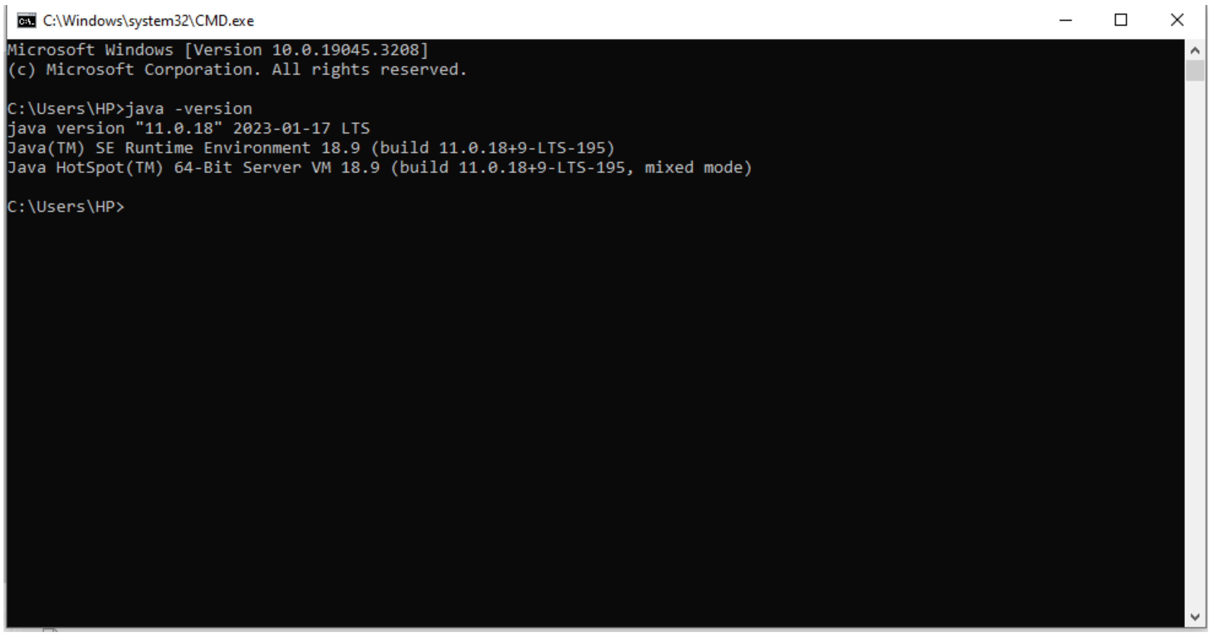
Now set the environment variables.Click on path and than click edit .Click on new and add bin path location where java is installed.



```
C:\Program Files\Java\jdk-11\bin
C:\Program Files\Git\cmd
C:\Program Files\apache-maven-3.9.0\bin
C:\Windows
%SystemRoot%\System32
%SystemRoot%
%SystemRoot%\System32\Wbem
C:\Program Files\nodejs\
C:\Program Files\MySQL\MySQL Shell 8.0\bin\
```

Click ok.

Step 4 - Run the command prompt and execute the command “ java -version” . This will check java version installed.



C:\Windows\system32\CMD.exe

```
Microsoft Windows [Version 10.0.19045.3208]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>java -version
java version "11.0.18" 2023-01-17 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.18+9-LTS-195)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.18+9-LTS-195, mixed mode)

C:\Users\HP>
```

Data Types:

Data types specify the different sizes and values that can be stored in the variable.

Primitive Data Type:

Primitive data types include byte, short, int, long, float, double, boolean, and char.

1. Boolean Data Type

Boolean data type represents only one bit of information either true or false which is intended to represent the two truth values of logic and Boolean algebra.

Syntax: boolean booleanVar;

2. Byte Data Type

The byte data type is an 8-bit signed two’s complement integer. The byte data type is useful for saving memory in large arrays.

Syntax: byte byteVar;

Size: 1 byte (8 bits)

3. Short Data Type

The short data type is a 16-bit signed two’s complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

Syntax: short shortVar;

Size: 2 bytes (16 bits)

4. Integer Data Type

It is a 32-bit signed two’s complement integer.

Syntax: int intVar;

Size: 4 bytes (32 bits)

5. Long Data Type

The range of a long is quite large. The long data type is a 64-bit two’s complement integer and is useful for those occasions where an int type is not large enough to hold the desired value. The size of the Long Datatype is 8 bytes (64 bits).

Syntax: long longVar;

6. Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

Syntax: float floatVar;

7**. Double Data Type**

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

Syntax: double doubleVar;

8. Char Data Type

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

Syntax: char charVar;

Example:

In the below example, we declare variables of different data types and initialize them with corresponding values. Then, we print the values of these variables using System.out.println().

```
public class DataTypesExample {
   public static void main(String[] args) {
      boolean flag = true;
      byte age = 25;
      int count = 100;
      double pi = 3.14159;
      char grade = 'A';
      String name = "Jack Ryan";

      System.out.println ("Flag: " + flag);
      System.out.println ("Age: " + age);
      System.out.println ("Count: " + count);
      System.out.println ("Pi: " + pi);
      System.out.println ("Grade: " + grade);
      System.out.println ("Name: " + name);
   }
}
```

Non-Primitive Data Type :

Non-primitive data types, also known as reference types, are data types in Java that do not hold the actual data directly but rather refer to objects in memory.They are strings, classes , arrays, etc.

1. Strings

Strings are defined as an array of characters. The difference between a character array and a string in Java is, that the string is designed to hold a sequence of characters in a single variable.

Syntax: Declaring a string

String <string_variable> = “<sequence_of_string>”;

Example:

```
// Declare String without using new operator
String value = "OTS Training";
```

2. Arrays

An Array is a group of like-typed variables that are referred to by a common name.

Syntax : dataType[] arrayName = {value1, value2, value3, ...};

Example:

```
// Declare Array
int [] numbers = {1, 2, 3, 4, 5};
```

Variables:

- Variables in Java are named containers used to store data of a specific type. They act as placeholders that hold values that can be accessed, modified, and used within a program. Variables are fundamental to programming and allow you to work with and manipulate data during runtime.

- In Java, variables are declared with a specific data type, which determines the type of values that can be stored in the variable. Each variable also has a unique name that is used to refer to it within the program.

The general syntax for declaring a variable is as follows:

dataType variableName;

Here, dataType represents the type of data that the variable can hold, such as int, double, boolean, String, etc., and variableName is the name given to the variable.

For example, consider the following variable declarations:

```
int age;
double height;
boolean isStudent;
String name;
```

In the above example, four variables are declared. The age variable is of type int and can store integer values, the height variable is of type double and can store decimal values, the isStudent variable is of type boolean and can store either true or false, and the name variable is of type String and can store sequences of characters.

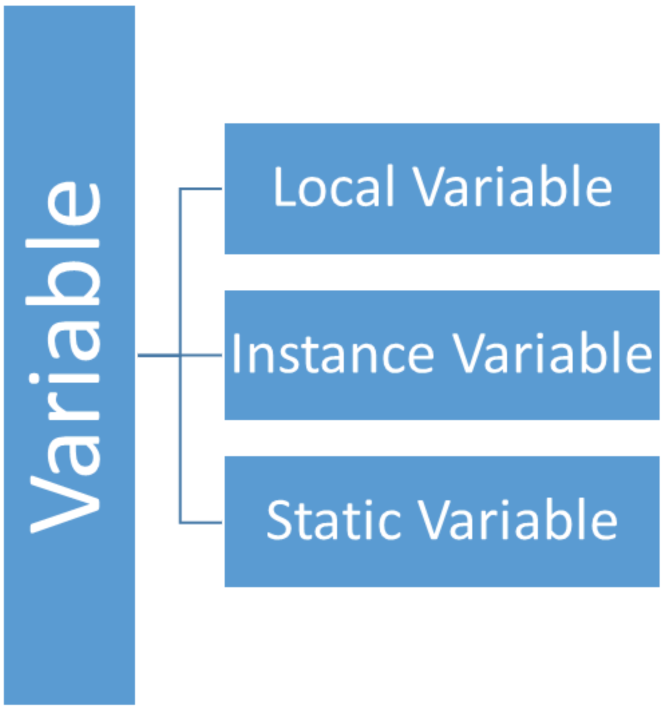
Variables can also be initialized at the time of declaration by assigning an initial value to them.

For example:

```
int count = 0;
double pi = 3.14159;
boolean isActive = true;
String message = "Hello, World!";
```

In this case, the variables count, pi, isActive, and message are declared and assigned initial values.

Once variables are declared and initialized, their values can be accessed, updated, and used in calculations and operations throughout the program.



1. Local Variables

A variable defined within a block or method or constructor is called a local variable.The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
Initialization of the local variable is mandatory before using it in the defined scope.

Example :

```
class Demo {
   public static void main(String[] args)
   {
      // Declared a Local Variable
      int var = 10;

      // This variable is local to this main method only
      System.out.println("Local Variable: " + var);
   }
}
```

2. Instance Variables

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.
As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable.

Example :

```
class Demo {

   // Declared Instance Variable
   public String name;

   public Demo()
   {
      // Default Constructor
      // initializing Instance Variable
      this.name = "OTS Training";
   }

   // Main Method
   public static void main(String[] args)
   {
      // Object Creation
      Demo objectname = new Demo();

      // Displaying O/P
      System.out.println("Name is: " + objectname.name);

}

}
```

3. Static Variables

Static variables are also known as class variables.

These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
Static variables are created at the start of program execution and destroyed automatically when execution ends.

Initialization of a static variable is not mandatory.

Example:

```
class Demo{
   // Declared static variable
   public static String name = "OTS Training";

   public static void main(String[] args)
   {

      // name variable can be accessed without object

      // static variable
      System.out.println("Name is : " + Demo.name);

      // static int c=0;
      // above line,when uncommented,
      // will throw an error as static variables cannot be
      // declared locally.
   }
}
```

Naming Convention :

Identifier Type	**Naming Rules**	**Examples**
Class	It should start with the uppercase letter. It should be a noun such as Color, Button, System, Thread, etc.	public class Employee {

	Use appropriate words, instead of acronyms.	//code snippet }

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		class Employee
		{
		// method
		void draw()
		{
		//code snippet
		}
		}

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		class
		Employee
		{
		// variable
		int id;
		//code snippet
		}

Loops :

Looping is a feature that facilitates the execution of a set of instructions repeatedly until a certain condition holds false. Java provides three types of loops namely the for loop, the for each loop,the while loop, and the do-while loop. Loops are also known as Iterating statements or Looping constructs in Java.

While all three types' basic functionality remains the same, there's a vast difference in the syntax and how they operate.

Loops are fundamental constructs in Java and are widely used in various scenarios, such as iterating over arrays, collections, and performing iterative calculations.

For Loop :

When we know the exact number of times the loop is going to run, we use for loop. It provides a concise way of writing initialization, test condition, and increment/decrement statements in one line.

Syntax:

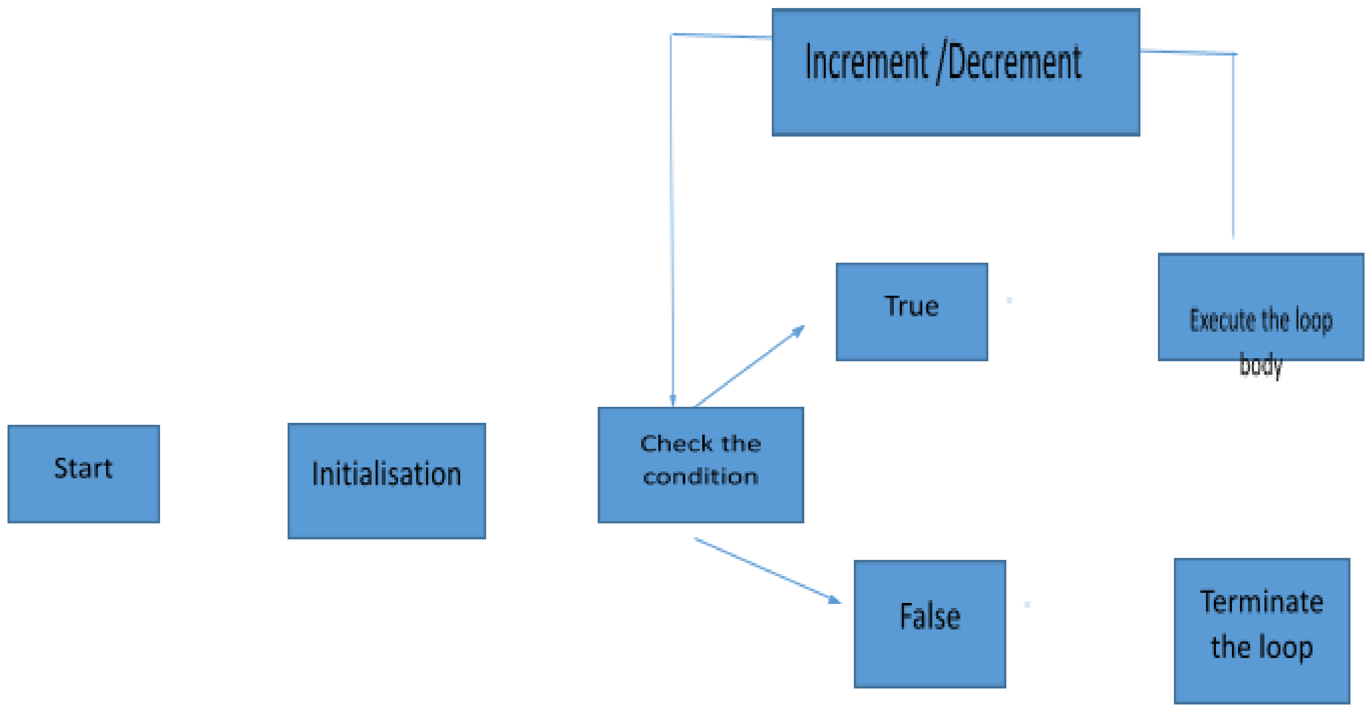
```
for(Initialization ; Test Expression ; Increment/Decrement){
    // Body of the Loop (Statement(s))
}
```

Initialization - expression is used for initializing a variable, and it is executed only once.

Test Expression - It executes the condition statement for every iteration. If it evaluates the condition to be true, it executes the body of the loop. The loop will continue to run until the condition becomes false.

Increment/Decrement - It is the increment or decrement statement applied to the variable to update the initial expression.

For loop is an entry-controlled loop as we check the condition first and then evaluate the body of the loop.



Example

```
public class Example {
  public static void main(String[] args) {
    int maxNum = 5;
    /* loop will run 5 times until test condition becomes false */
    for (int i = 1; i <= maxNum; i++) {
      System.out.println(i);
    }
  }
}
```

Output:

```
1
2
3
4
5
```

Explanation - Firstly Initialization expression is executed. Then the test expression is evaluated. If it is true then control goes inside the body of the loop, otherwise, loop is terminated. After execution update expression is carried out. If the test expression evaluates to false the loop is terminated else the body of the loop is again executed. This process continues until the test expression becomes false. If the test expression never becomes false, it is the case of an infinite loop.

For-each Loop

The Java for-each loop is used on an array or a collection type. It works as an iterator and helps traverse through an array or collection elements, and returns them. While declaring a for-each loop, you don't have to provide the increment or decrement statement as the loop will, by default, traverse through each element.

Syntax :

```
for(Type var:array){

//loop body

}
```

Example :

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

      //Array declaration

      int ar[]={1,2,3,5,7,11,13,17,19};

      //Using for-each loop to print the array

      for(int x:ar){

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

Output:

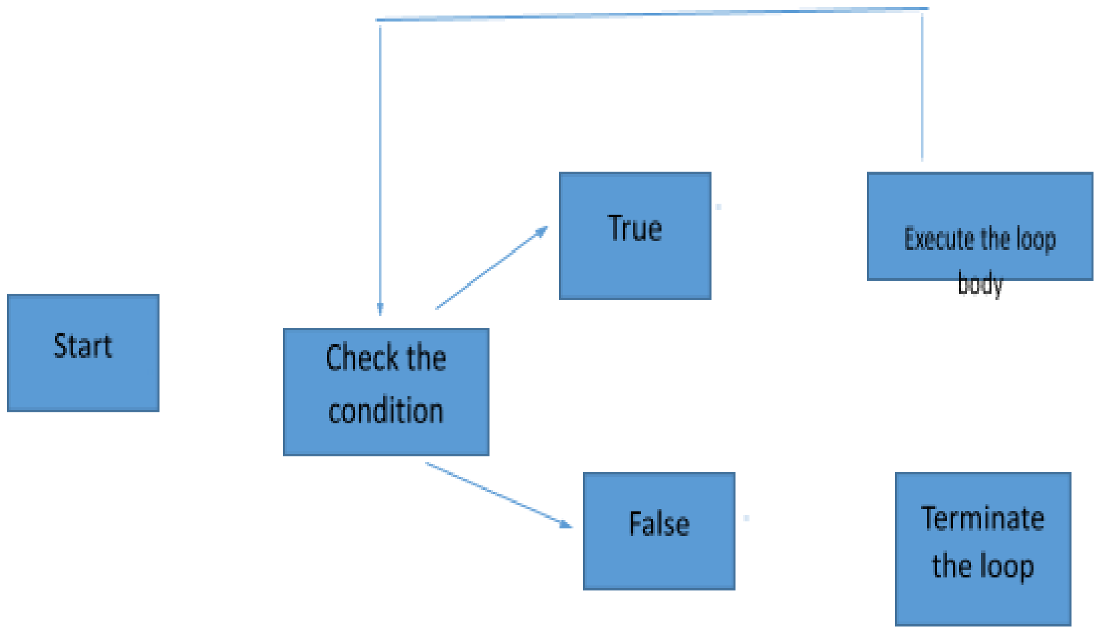
```
1
2
3
5
7
11
13
17
19
```

While Loop :

The while loop is used when the number of iterations is not known but the terminating condition is known. Loop is executed until the given condition evaluates to false. while loop is also an entry-controlled loop as the condition is checked before entering the loop. The test condition is checked first and then the control goes inside the loop.

Syntax:

```
Initialization;
while(Test Expression){
   // Body of the Loop (Statement(s))
   Updation;
}
```



Example :

```
public class Example
{
  public static void main(String args[])
 {
   int factorial = 1, number = 5, tempNum = 0;
   tempNum = number; // Initialization;
   while (tempNum != 0) { // Test Expression
     factorial = factorial * tempNum;
```

```
    --tempNum; //Updation;
  }
  System.out.println("The factorial of " + number + " is: " + factorial);
 }
}
```

Output:

```
The factorial of 5 is: 120
```

Explanation :

Firstly, the test expression is checked. If it evaluates to true, the statement is executed and again condition is checked. If it evaluates to false the loop is terminated.

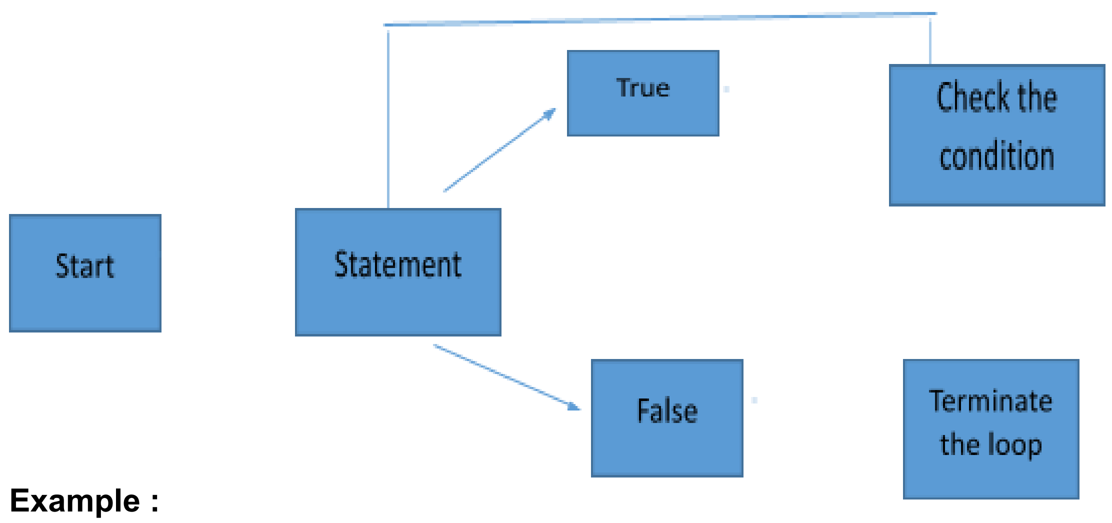
do-while Loop:

The do-while loop is like the while loop except that the condition is checked after evaluation of the body of the loop. Thus, the do-while loop is an example of an exit-controlled loop.

The loop is executed once, and then the condition is checked for further iterations. This loop runs at least once irrespective of the test condition, and at most as many times the test condition evaluates to true. It is the only loop that has a semicolon(;).

Syntax:

```
Initialization;
do {
 // Body of the loop (Statement(s))
 // Updation;
}
while(Test Expression);
```



Example :

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Output:

```
1
2
3
4
5
6
7
8
```

```
9
10
```

Nested Loops in Java :

A nested loop is a loop statement inside another loop statement. In a nested loop, for each iteration of the outer loop, the entire inner loop is executed.

For instance, if the outer loop is to be run m times and the inner loop is to be run n times, the total time complexity of the nested loop is m*n.

Loops can be nested with the same type of loop or another type of loop there is no restriction on the order or type of loop. For example for loop can be inside a while loop and vice versa, or a do-while loop can be under another do-while loop.

Syntax :

```
while (condition) {

  for (initialization; condition; update) {
    do {
      //statement of do-while loop
    } while (condition);

    //statement of for loop
  }

  //statement of while loop
}
```

Conditional Statements :

Decision Making in Java helps to write decision-driven statements and execute a particular set of code based on certain conditions.

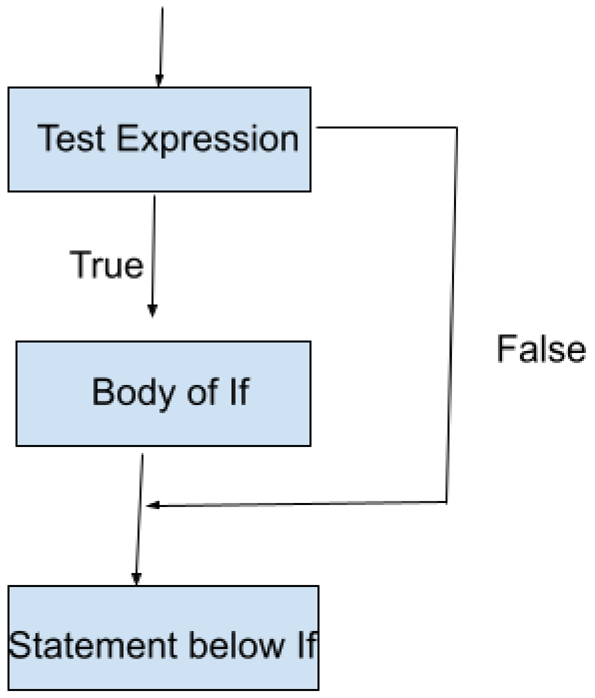
There are different conditional statements ,listed below .

1. If Statement :

The Java if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```



Example 1:

```
// Java program to illustrate If statement

class IfDemo {
    public static void main(String args[])
    {
        int i = 10;
        if (i < 15)
            System.out.println("10 is less than 15");

        System.out.println("Outside if-block");
        // both statements will be printed
    }
}
```

Output:

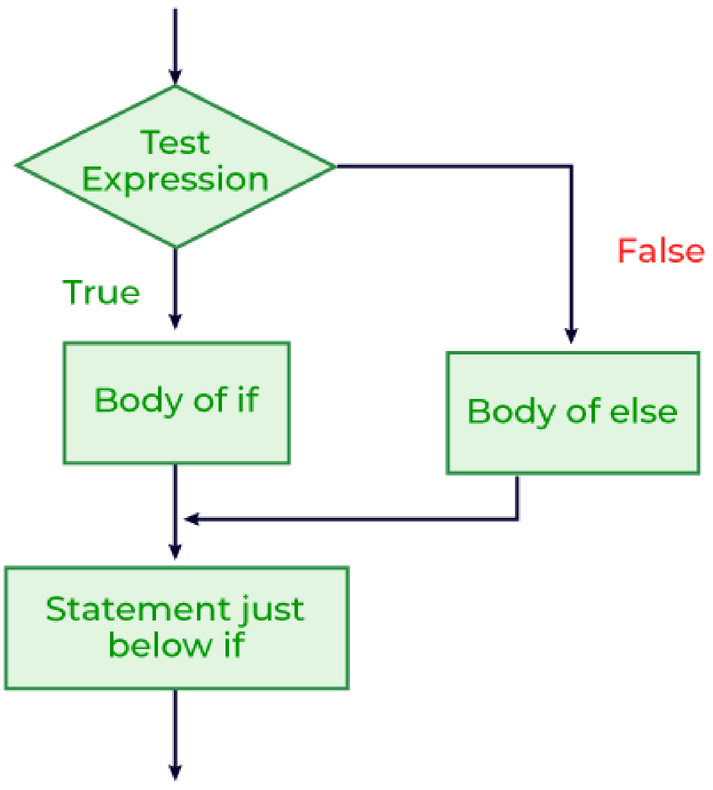
```
10 is less than 15
Outside if-block
```

2. If else :

The if statement tells us that if a condition is true it will execute a block of statements and the else statement with the if statement to execute a block of code when the condition is false.

Syntax :

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```



Example:

```
class IfElseDemo {
    public static void main(String args[])
    {
        int i = 20;

        if (i < 15)
                System.out.println("i is smaller than 15");
        else
            System.out.println("i is greater than 15");



    }
}
```

Output:

```
i is greater than 15
```

3. If-else-If :

Java if-else-if ladder is used to decide among multiple options.

Syntax:

```
if (condition)
   statement 1;
else if (condition)
   statement 2;
else
   statement;
```

Example:

```
   class GFG {
   public static void main(String[] args)
   {
      // initializing expression
      int i = 20;

      // condition 1
      if (i == 10)
         System.out.println("i is 10\n");

      // condition 2
```

```
        else if (i == 15)
            System.out.println("i is 15\n");

        // condition 3
        else if (i == 20)
            System.out.println("i is 20\n");

        else
            System.out.println("i is not present\n");

        System.out.println("Outside if-else-if");
        }
}
```

Output:

```
i is 20
```

Strings:

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use double quotes to represent a string in Java.

Common string functions and methods that can be very helpful in working with strings in Java.

Method	Description
toCharArray()	Converts the string to a character array.
length()	Returns the length of the string.
charAt(index)	Returns the character at the specified index of the string (0-based index).
indexOf(char)	Returns the index of the first occurrence of the specified character in the string, or -1 if the character is not found
substring(startIndex, endIndex)	Returns a new string that is a substring of the original string, starting from the startIndex (inclusive) to endIndex (exclusive).
toLowerCase()	Returns a new string with all characters converted to lowercase.
toUpperCase()	Returns a new string with all characters converted to uppercase
contains(CharSequence)	Checks if the string contains the specified sequence of characters.
equalsIgnoreCase(String anotherString)	Compares the string to another string, ignoring case.

.

Syntax :

```
String variablename;  //Declaring a String
```

```
String variablename = " Text ";  //Initializing a String:
```

Example:

```
public class StringExample {

   // Main Function
   public static void main(String args[])
   {
      String str = new String("example");



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

Output:

```
example
```

The other way to create string is using new keyboard.

Syntax:

```
String variablename = new String(“Text”);
```

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

Example:

```
String demoString = new String (“Welcome”);
```

Strings Immutable nature -

Strings in java are immutable in nature.This means strings stay the same once created, but you can create new strings by copying and combining parts from the original ones.

Example -

```
String name = "Alice";

// Now, let's try to change the string
name = "Bob"; // Oops! We can't change it. It's immutable.

// But we can create a new string and copy some parts from the old one.
String newName = name + " and Carol"; // We make a new drawing with parts of the old one.

System.out.println(name); // Output: "Bob" - The original name is still the same.
System.out.println(newName); // Output: "Bob and Carol" - The new name with the added part.
```

String Buffer :

In Java, StringBuffer is a class that represents a mutable sequence of characters. It is designed to efficiently handle the manipulation of strings when you need to make frequent modifications, such as appending, inserting, or deleting characters.

Example :

```
public class StringBufferExample {
    public static void main(String[] args) {
        // Create a StringBuffer
        StringBuffer stringBuffer = new StringBuffer();

        // Append words to the StringBuffer
        stringBuffer.append("This");
        stringBuffer.append(" is");
        stringBuffer.append(" a");
        stringBuffer.append(" simple");
        stringBuffer.append(" example");
        stringBuffer.append(" using");
        stringBuffer.append(" StringBuffer.");

        // Get the final sentence
        String sentence = stringBuffer.toString();
        System.out.println(sentence);
    }
}
```

In this example, we create a StringBuffer called stringBuffer, and then we use the append() method multiple times to concatenate words to the string. Finally, we convert the StringBuffer to a regular String using the toString() method and print the final sentence.

String Builder :

In Java, StringBuilder is a class that serves the same purpose as StringBuffer: it represents a mutable sequence of characters. It is designed to efficiently handle string manipulations when you need to make frequent modifications, just like StringBuffer.

The primary difference between StringBuilder and StringBuffer lies in their thread-safety behavior.Unlike StringBuffer, StringBuilder is not designed to be thread-safe. This means that it does not handle multiple threads accessing it simultaneously in a synchronized manner. As a result, it usually performs faster than StringBuffer.

Example:

```
public class SimpleStringBuilderExample {
    public static void main(String[] args) {
        // Create a StringBuilder
        StringBuilder sentenceBuilder = new StringBuilder();

        // Append words to the sentence
        sentenceBuilder.append("This");
        sentenceBuilder.append(" is");
        sentenceBuilder.append(" a");
        sentenceBuilder.append(" simple");
        sentenceBuilder.append(" example");
        sentenceBuilder.append(" using");
        sentenceBuilder.append(" StringBuilder.");
        // Get the final sentence

        String sentence = sentenceBuilder.toString();

        System.out.println(sentence);

    }

}
```

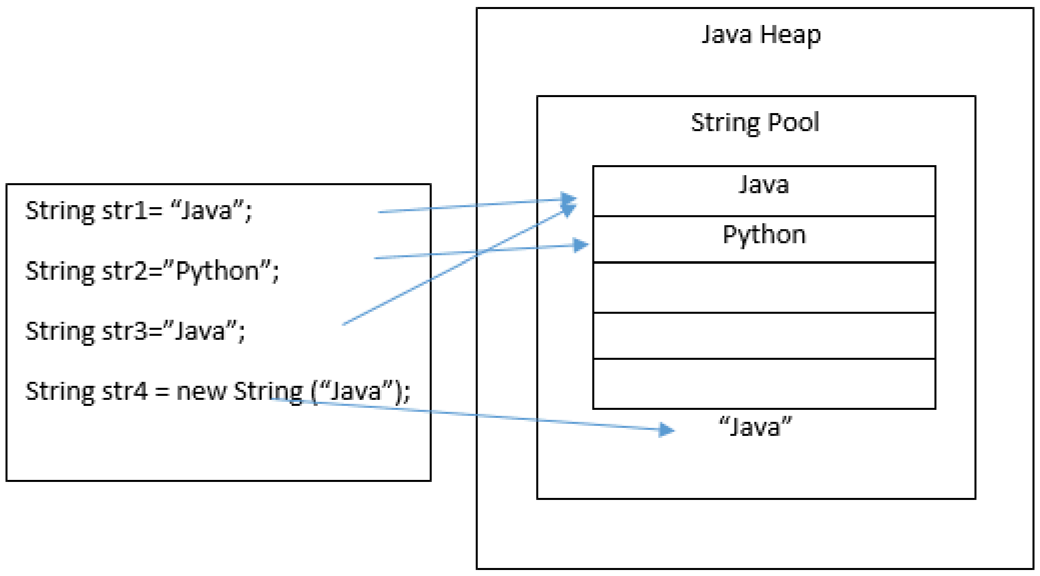
In this example, we create a StringBuilder called sentenceBuilder, and then we use the append() method multiple times to concatenate words to the sentence. Finally, we convert the StringBuilder to a regular String using the toString() method and print the final sentence.

String Pool In Java

String pool is nothing but a storage area in Java heap where string literals stores.By default, it is empty and privately maintained by the Java String class. Whenever we create a string the string object occupies some space in the heap memory. Creating a number of strings may increase the cost and memory too which may reduce the performance also.

When we create a string literal, the JVM first check that literal in the String pool. If the literal is already present in the pool, it returns a reference to the pooled instance. If the literal is not present in the pool, a new String object takes place in the String pool.

String literals created with the new keyword take place in the Java heap, not in the String pool



Example :

```
  public class StringPoolExample
{
public static void main(String[] args)
{
String s1 = "Java";
String s2 = "Java";
String s3 = new String("Java");
String s4 = new String("Java").intern();
System.out.println((s1 == s2)+", String are equal."); // true
System.out.println((s1 == s3)+", String are not equal."); // false
System.out.println((s1 == s4)+", String are equal."); // true
}
}
```

Output:

```
true , String are equal.
false,string are not equal.
true, String are equal.
```

Arrays:

Arrays in Java are non-primitive data types that store elements of a similar data type in the memory. Arrays in Java can store both primitive and non-primitive types of data in it.

There are two types of arrays, single-dimensional arrays have only one dimension, while multi-dimensional have 2D, 3D, and nD dimensions.It is so easy to store and access data from only one place, and also, it is memory efficient.

Syntax:

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

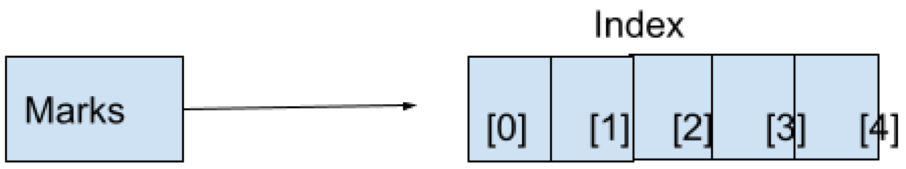
```
arrayRefVar=new datatype[size];
```

Let's use the above example to know how arrays work in java. This is how we can implement a program to store five subject marks of a student:

```
int mark1 = 78;
int mark2 = 84;
int mark3 = 63;
int mark4 = 91;
int mark5 = 75;
```

But what if we store all these five variables in one single variable?

```
marks[0] = 78;
marks[1] = 84;
marks[2] = 63;
marks[3] = 91;
marks[4] = 75;
```



```
int marks = new int[5];
```

Example:

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


      int [] arr=new int [5];
      // 5 is the size of arr
    }
}
```

Types of Array in Java

In Java, there are two types of arrays:

Single-Dimensional Array

Multi-Dimensional Array

1. Single Dimensional Array

An array that has only one subscript or one dimension is known as a single-dimensional array. It is just a list of the same data type variables.

One dimensional array can be of either one row and multiple columns or multiple rows and one column. For instance, a student's marks in five subjects indicate a single-dimensional array.

Example:

```
int marks[] = {56, 98, 77, 89, 99};
```

2. Multi-Dimensional Array

In not all cases, we implement a single-dimensional array. Sometimes, we are required to create an array within an array.Suppose we need to write a program that stores five different subjects and the marks of sixty students. In this case, we just need to implement 60 one-dimensional arrays with a length of five and then implement all these arrays in one array. Hence, we can make the 2D array with five columns and 60 rows.

Example:

```
int marks[][] = {
            {77,85,68,99,87},
            {98,56,79,90,92},
            {78,88,56,70,99}
        };
OR
int marks[][] = new int[3][5];

// both will create a 2D array with 3 rows and 5 columns.
```

Arrays of Objects:

As the name suggests, an array of objects is nothing but a list of objects stored in the array. Notice that it does not store objects in an array but stores the reference variable of that object. The syntax will be the same as above.

Example:

```
Student studentObj[] = new Student[3];
```

After the above statement executes, it will create an array of objects of the Student class with a length of 3 studentObj references. For each object reference, we need to implement an object using new.

Example:

```
class Student {
  Student(int id, String name) {
    System.out.println("Student ID is "+ id + " and name is "+ name );
  }
}

public class Test {
  public static void main (String[] args) {
    // declaring an array of Object
    Student obj[] = new Student[3];

    obj[0] = new Student(1,"Bharat");
    obj[1] = new Student(5,"Vivaan");
    obj[2] = new Student(6,"Smith");

  }
}
```

Output:

```
Student ID is 1 and name is Bharat
Student ID is 5 and name is Vivaan
Student ID is 6 and name is Smith
```

Looping Through Array Elements

For performing any operation or displaying any elements, we used an index. But by using an index, we can only implement one operation at a time. If we are required to modify all elements in an array, we must write an operation for each index.

To overcome this situation, we have a looping feature in Java that follows the principle write once and execute multiple times.

There are two ways to loop through the array elements.

Using for Loop - In this way of looping through the array elements, we need the array's length first. This can be done by java in-built property length. If we start the for loop from 0, then the total iteration will be the array's length - 1, and if we start the for loop from 1, then the total iteration will be the length of an array.

Example:

```
public class Demo {
  public static void main (String[] args) {
    // declaring and initializing an array
    String strArray[] = {"Python", "Java", "C++", "C", "PHP"};

    // Find the length of an array
    int lengthOfArray = strArray.length;
    // using for loop
    for(int i=0;i<lengthOfArray;i++) {
      System.out.println(strArray[i]);
    }
  }
}
```

Output:

```
    Python
    Java
    C++
  C
  PHP
```

In the above code, we have just used a for loop to go through the array element. For the number of iterations, we find the length of the array using the array length property.

Using for-each loop - In this way, we use a for loop, but we do not need the length of the array. The for-each method is much easier to loop through the array element than looping for-loop. While using the for-each loop, we need to define a variable of the same type as the data type of the defined array. If not done, the compiler will throw compile time error saying- incompatible types: String cannot be converted to YourDefinedDataType

For-each Looping

Example:

```
public class Demo {
  public static void main (String[] args) {
    // declaring and initializing an array
    String strArray[] = {"Python", "Java", "C++", "C", "PHP"};

    // using for-each loop
    for(String i : strArray) {
      System.out.println(i);
    }
  }
}
```

Output:

```
Python
```

Java

C++

C

PHP

The above for-each code can be read - for each String element in strArray, print the value of i.

Apart from the above two looping methods, we can also use a while loop for traversing.

OOPS

In Java, Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects." Objects are instances of classes, which are user-defined blueprints or templates that define the properties (attributes) and behaviors (methods) of the objects. OOP is based on following fundamental concepts:

Class

Object

Encapsulation

Inheritance

Polymorphism

Abstraction

Object :

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Example :

Car is an object.The name of the car is Mercedes-Benz S-Class.The attributes are color , make , model , fuel type , number of doors , etc..

The car can start,stop , change gears , honk,get speed , etc are the behaviors /methods of the car object.

Syntax :

```
ClassName objectName = new ClassName();
```

Class:

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. Class is not a real-world entity.Class is a group of variables of different data types and a group of methods.

A Class in Java can contain following components:

Access Modifiers

Class Name

Data member

Method

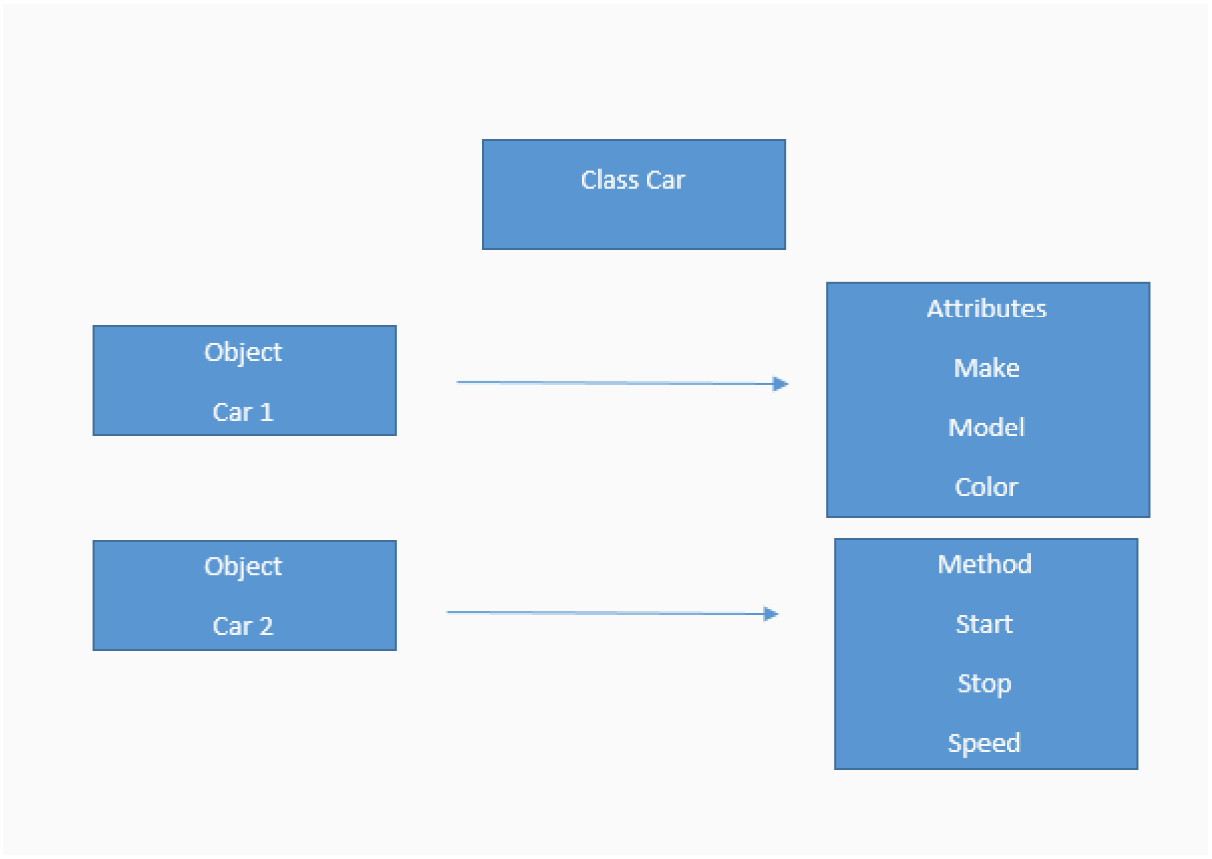
Constructor

Nested Class

Interface

Car is a class.We can create object of this car class.We can say that Mercedes S Class is an object created from Class Car.

The object created from the Car class will have attributes and methods.



Syntax:

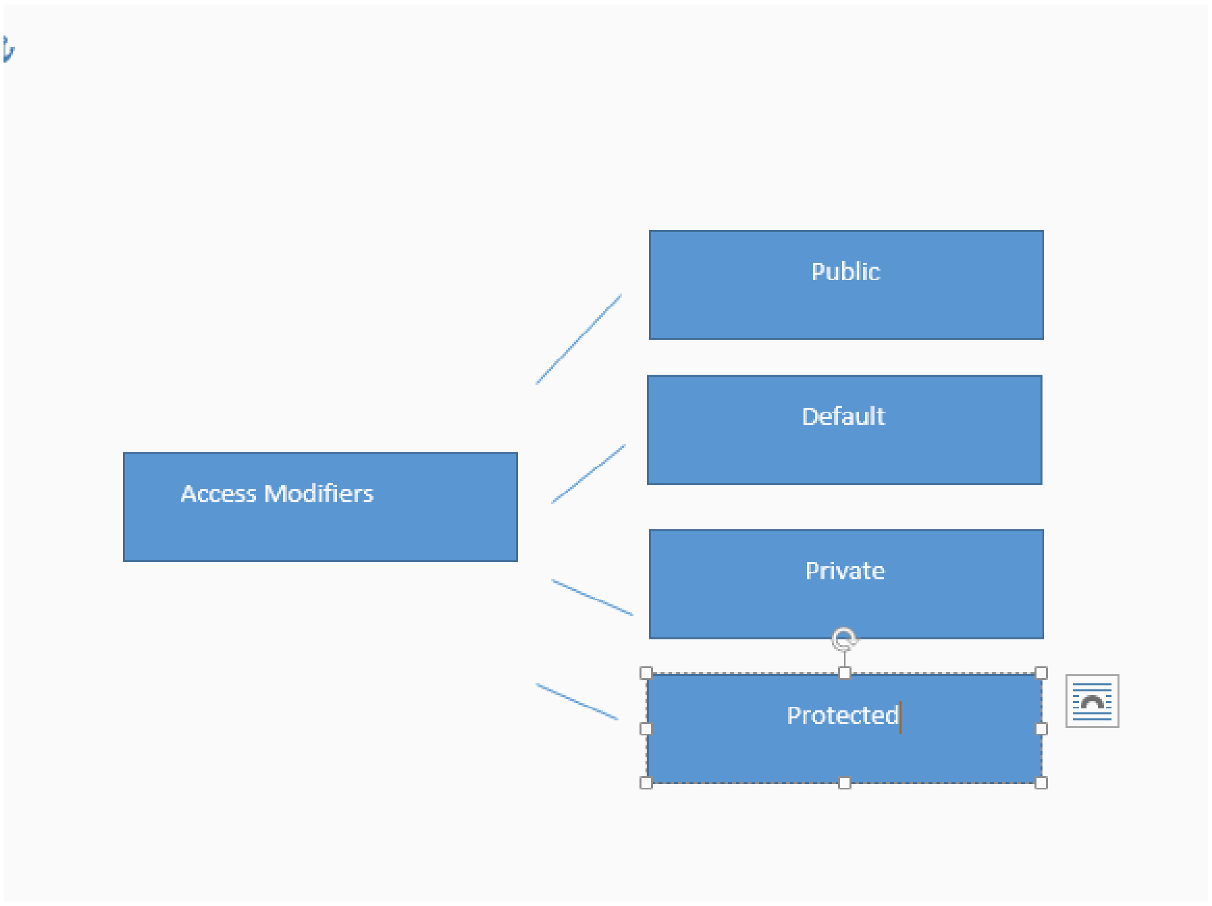
```
class <class_name>{
    field;
    method;
}
```

Example of Class and Object :

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Access Modifier

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods.



Public :

The public access modifier is specified using the keyword public.

The public access modifier has the widest scope among all other access modifiers.

Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Syntax:

```
public class Classname
{
// fields
   public returntype methodname()
   {
   // method body
   }
}
```

Example :

```
package pack;
public class A
{
public void msg()
{
System.out.println("Hello");
}
}
```

Private :
The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

Any other class of the same package will not be able to access these members.

A class cannot be private or protected except nested class.

Syntax:

```
class Classname
{

   private variablename;
   private returntype methodname()
   {

   }
}
```

Example :

```
package pack;
public class A
{
private data ; // variable
private void msg() // private method
{
System.out.println("Hello");
}
}
```

The variable and method are not accessible in other class.

Protected :

The protected access modifier is specified using the keyword protected.

The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

Syntax:

```
class Classname
{
   protected returntype methodname()
   {


   }
}
```

Example :

```
// Java program to illustrate
// protected modifier
package p1;

// Class A
public class A
{
protected void display()
   {
      System.out.println("Hello World");
   }
}
```

The method is accessible in other class / package , if the other class extends this class.

Default :

When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible only within the same package.

Syntax:

```
class Classname
{
returntype methodname()
{
// method body
}
}
```

Example :

```
// Java program to illustrate default modifier
package p1;

// Class  is having Default access modifier
class A
{
   void display()
   {
      System.out.println("Hello World!");
   }
}
```

Class Name :

In Java, a class name is the identifier used to name a class. It is the name by which you refer to the class when creating objects or accessing its members (fields and methods) from other parts of the code.

Java class names must follow certain rules:

The class name should start with a letter (uppercase by convention) or an underscore (_). It cannot start with a digit or any special character.

After the first character, the class name can include letters (both uppercase and lowercase), digits, and underscores.

Java is case-sensitive, so Car, car, and CAR are considered different class names.

The class name should be descriptive and follow standard naming conventions, such as using camelCase (starting with a lowercase letter and using uppercase for subsequent words) and giving meaningful names to represent the class's purpose or the objects it represents.

Data Members:

In Java, data members (also known as instance variables or fields) are variables declared within a class. A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Syntax :

```
accessmodifier class Classname
{
access-modifier variablename; //instance variable
access-modifier return-type methodname()
{    // methods
}
}
```

Methods:

The method in Java or Methods of Java is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Java Methods allow us to reuse the code without retyping the code.

Syntax:

```
<access_modifier> <return_type> <method_name>( list_of_parameters)
{
   //body
}
```

In general, method declarations have 6 components:

Access-Modifier - Optional in syntax

Return Type - Mandatory in syntax

Method Name - Mandatory in syntax

Parameter list- Optional in syntax

Exception list- Optional in syntax

Method body- Optional in syntax

Example :

```
public int add (int num1,int num2)
{
   int sum=num1+num2;
   return sum;
}
```

Constructor:

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method that is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type

Types of Java constructors

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor

Java Default Constructor
A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax:

```
<class_name>(){}
```

Example:

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

```
Bike is created
```

If there is no constructor in a class, compiler automatically creates a default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
   //creating a parameterized constructor
   Student4(int i,String n){
   id = i;
   name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   //creating objects and passing values
   Student4 s1 = new Student4(111,"Aayan");
   Student4 s2 = new Student4(222,"Aryan");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Output:

```
111 Aayan

222 Aryan
```

Inheritance :

Inheritance is one of the critical features of OOPs (Object Oriented Programming System), where a new class is created from an existing class (parent or base class). Inheritance represents the IS-A relationship which is also known as a parent-child relationship.In simple terms, the inheritance would mean passing down the characteristics of parents(habits, looks, height) to their child.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Syntax :

```
class Subclass-name extends Superclass-name

{

  //methods and fields

}
```

The extends keyword indicates that you are making a new class that derives from an existing class.

A class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Example :

```
// Child Class

public class HomeLoan extends Calculator


{
}

public HomeLoan() // Child class constructor
{
   System.out.println("Child class cons");
}
 public static void main(String[] args)
   {
     // creating object of child class and storing in reference of child class.

   HomeLoan obj =new HomeLoan();
  obj.add() ; // accessing method from the parent class
  obj.homeEmi() ; // accessing method of the child class
   }

   public void homeEmi() // Child class method
   {
      System.out.println("home emi in child class");
   }

}

// Parent Class
public class Calculator
{
      public Calculator()

   {
      System.out.println("Parent class cons");
   }

   public void add() // Method in parent class
   {
      System.out.println("Add method in Parent class");
   }
   public void sub()
   {
      System.out.println("Sub method in Parent class");
   }
}
```

In the above example , child class HomeLoan is extending Parent class Calculator.Child class is accessing methods from parent class , thus , exhibiting the property of inheritance.

Types of Inheritance :

Below are the different types of inheritance which are supported by Java.

Single Inheritance

Multilevel Inheritance

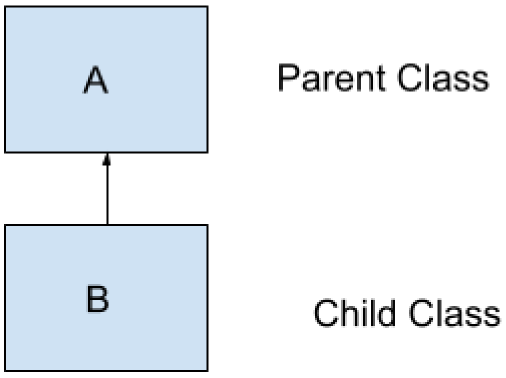
Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance

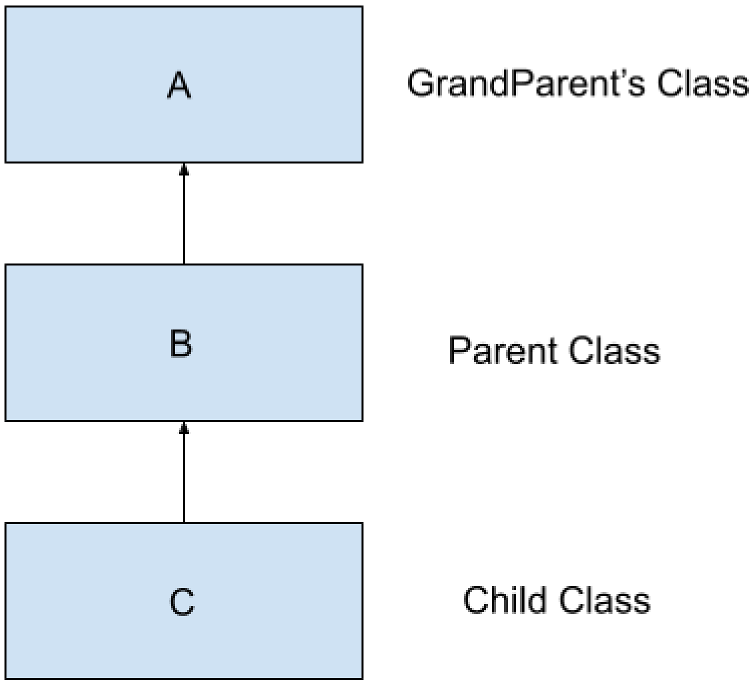
1. **Single Inheritance**

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a parent class for the child class B.



2.Multilevel Inheritance

In Multilevel Inheritance, a child class will be inheriting a parent class , and as well as the child class also acts as the parent class for other classes. In the below image, class A serves as a parent class for the child class B, which in turn serves as a parent class for the child class C. In Java, a class cannot directly access the grandparent's members.



Example of Multilevel Inheritance :

```
// Grand Parent Class
class A {
   public void print_A()
   {
   System.out.println("Class A");
   }
}

class B extends A // Parent class extending Grand Parent Class
{
   public void print_B()
   {
   System.out.println("Class B");
   }
}

// Driver Class
public class Test extends B // Child class extending parent class
 {
   public static void main(String[] args)
   {
      Test obj = new Test();
       obj.print_A();
       obj.print_B();


   }
}
```

Interfaces :

Interface is a group of related methods with empty bodies.An interface in Java is a blueprint of a class. It has static constants and abstract methods.These methods are public and abstract by default(you don’t have to explicitly use the “abstract” keyword), and any class implementing your interface will need to provide implementations of those methods.

Interface achieves 100% abstraction.

Syntax :

```
interface {

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Example:

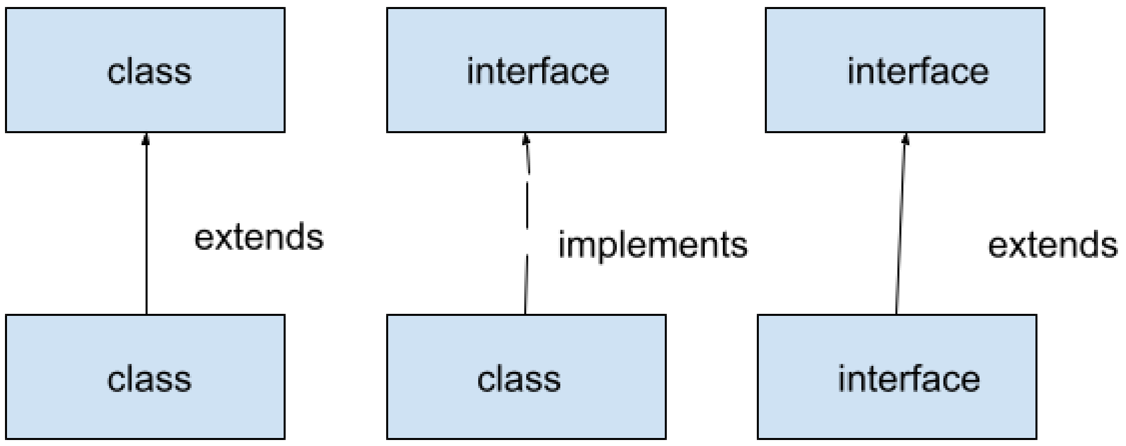
```
interface student
{
public void marks();
}

class School implements student
{
public void marks()
{
System.out.println(“Marks”);
}
public static void main(String args[]){
```

```
School obj = new School();
obj.marks();
 }
}
```

In the above example , student is an interface whose implementation is done in class School.

Relationship between classes and interfaces:



Polymorphism :

The term polymorphism means that an object can exist in different forms. For example, carbon can exist in three common types.

In java , Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.

Types of Java polymorphism

Compile-time Polymorphism

Runtime Polymorphism

Compile-Time Polymorphism :

This type of polymorphism is achieved by method overloading or constructor overloading.

Method Overloading
When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or a change in the type of arguments or sequence in arguements passed.

Example :

```
public class Methodoverloadingdemo {

   public static void main(String[] args) {

      Methodoverloadingdemo obj1 = new Methodoverloadingdemo();
   obj1.add(10.1, 5); // Calling 4th method
   obj1.add(2,3,5); // Calling 2nd method
   }

   public void add(int a,int b) // method add with 1 parameter
   {
      int result = a+b;
      System.out.println("Result is ="+result);
   }

   public void add(int a,int b,int c) // method add with 3 parameter
   {
        int result = a+b+c;

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

    }


    public void add(int a, double x) // method add with 1 integer and 1 double
parameter

    {

        int result = (int) (a+x);

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

    }


/*method add with 1 integer and 1 double parameter.The first passed arguement is
double and than integer */


    public void add(double x,int a)    {

        int result = (int) (a+x);

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

    }


}
```

Output :

```
Result is =15

Result is =10
```

Constructor Overloading:

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Example :

```
public class Constructoroverloading {

        String nameString;
        int age;
        double salary;

        public Constructoroverloading() // 0 parametre constructor
        {
                System.out.println("This is my default cons");
        }

        public Constructoroverloading(String name) // 1 parameter constructor
        {
                this.nameString=name;
        }

        public Constructoroverloading(String name,int age) // 2 parametre constructor
        {
        this.nameString=name;
        this.age=age;
        }
        public Constructoroverloading(String name,int age,double salary)
        {
                System.out.println("Name is "+name);
                System.out.println("Age is "+age);
                System.out.println("Salary is "+salary);
        }
```

```
	public Constructoroverloading(double salary,String name,int age)
	{
		System.out.println("Name is"+name);
		System.out.println("Age is"+age);
		System.out.println("Salary is"+salary);
	}
	public static void main(String[] args) {
		// calling default constructor
		Constructoroverloading obj1 =new Constructoroverloading();


	  // calling 4th constructor
	Constructoroverloading obj2 =new Constructoroverloading("Ritika", 20
,40000);



	}


}
```

Output :

```
This is my default cons
Name is Ritika
Age is 20
Salary is 40000.0
```

Run-Time Polymorphism :

Runtime polymorphism, also known as dynamic polymorphism or late binding, is a key concept in object-oriented programming (OOP). It allows a program to determine the actual method or behavior to be executed at runtime, based on the actual type of the object rather than the reference type.

}

In simple words , when the parent class and the child class have the same methods, ie. method with same name and same number and type of arguements , the method becomes overridden.

In such scenario , java prefers using the child class method.

Example :

```
public class Circle extends Shape // Child Class

{

   public Circle()
   {
      System.out.println("This is child class cons");
   }

   public static void main(String[] args)
 {

      Shape obj =new Circle();
      obj.draw();

   }

   public void draw() // Overridden method
   {
      System.out.println("Draw shape - Child Class");
   }

}
```

```
// Parent Class
public class Shape {
 public Shape()
 {
     System.out.println("Parent cons");
 }



public  void draw() //  overridden method
{
        System.out.println("Draw any shape");
}    }
```

Output :

```
Parent cons
This is child class cons
Draw shape - Child Class
```

In the above example , method draw is overridden.Java prefers using child class method in this case.

Abstraction :

Abstraction in Java refers to hiding the implementation details of a code and exposing only the necessary information to the user. It provides the ability to simplify complex systems by ignoring irrelevant details and reducing complexity.

Abstraction in Java can be achieved using the following tools it provides :

Abstract classes /methods

Interfaces

Java Abstract classes

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

100% abstraction may/may not be achieved in Abstract classes.We cannot create object for abstract class and interface.

An abstract class must be declared with an abstract keyword.

It can have abstract and non-abstract methods.

It cannot be instantiated.

It can have constructors and static methods also.

It can have final methods which will force the subclass not to change the body of the method.

Syntax :

```
abstract class Classname
{
   // body
}
```

Example :

```
abstract class Shape //abstract class
{
abstract void side();  // abstract method

public void draw() // non-abstract method
   {
      System.out.println("Abstract class method - draw circle");
   }
}

public class Circle extends Shape

{

   public static void main(String[] args) {



      Shape obj =new Circle(); // Reference of abstract class
      obj.draw(); //calling non-abstract method
      obj.side(); // calling abstract method
```

```
        Circle obj1=new Circle();
        obj1.color();






    }

    public void side() // implementation of abstract method from abstract class
    {
        System.out.println("Implementation of abstract method from abstract class");
    }

    public void color()
    {
        System.out.println("Circle color");
    }
}
```

In the above example ,implementation of abstract class shape was provided by class Circle.

this Keyword

When the instance variable and the parameter variable have same name (in constructor) , we address the class instance variable with ‘ this ‘ keyword.

Example:

```
class Student
```

```
{
String name ;

public Student (String name)
{
//instance variable and parameter having the same name
// Initialising the instance variable with parameter
//Refering the class instance variable with this keyword

this.name=name;

}

   public static void main(String[] args)
{
Student obj =new Student(“Ritika);
System.out.println(“Name is : “+obj.name);
}
}
```

Output:

```
Name is : Ritika
```

Super keyword

The super keyword in Java is a reference variable that is used to refer to parent class object.

super with Constructor

super is used to call a superclass constructor: When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.

Example

```
public class Square // Parent Class
{
 public Square()  // default constructor
 {
           System.out.println("Parent class constructor");
 }

 public Square(int number)  //parametre constructor
 {
           System.out.println("Parent class constructor"+number);
 }

}

public class Shapesn extends Square // Child Class
{

     public Shapesn() // Child class constructor

   {
             // The control will execute the default constructor from the parent class
           super();
```

```
        }



        public static void main(String[] args)
        {

Shapesn obj = new Shapesn();
        }


}
```

In the above example , when the object is created , control goes to the default constructor.In the default constructor , it will call the parent class constructor .

Output

```
Parent class constructor
```

super with Method

super is used to call a superclass method: A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.

Example:

```
 // superclass Person
class Person {
   void message()
   {
      System.out.println("This is person class\n");
   }
}
```

```
// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is
    // only in Student class
    void display()
    {
        // will invoke or call current
        // class message() method
        message();

        // will invoke or call parent
        // class message() method
        super.message();
    }
}
// Driver Program
class Test {

    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output

```
This is student class

This is person class
```

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of the super keyword, message() of the superclass could also be invoked.

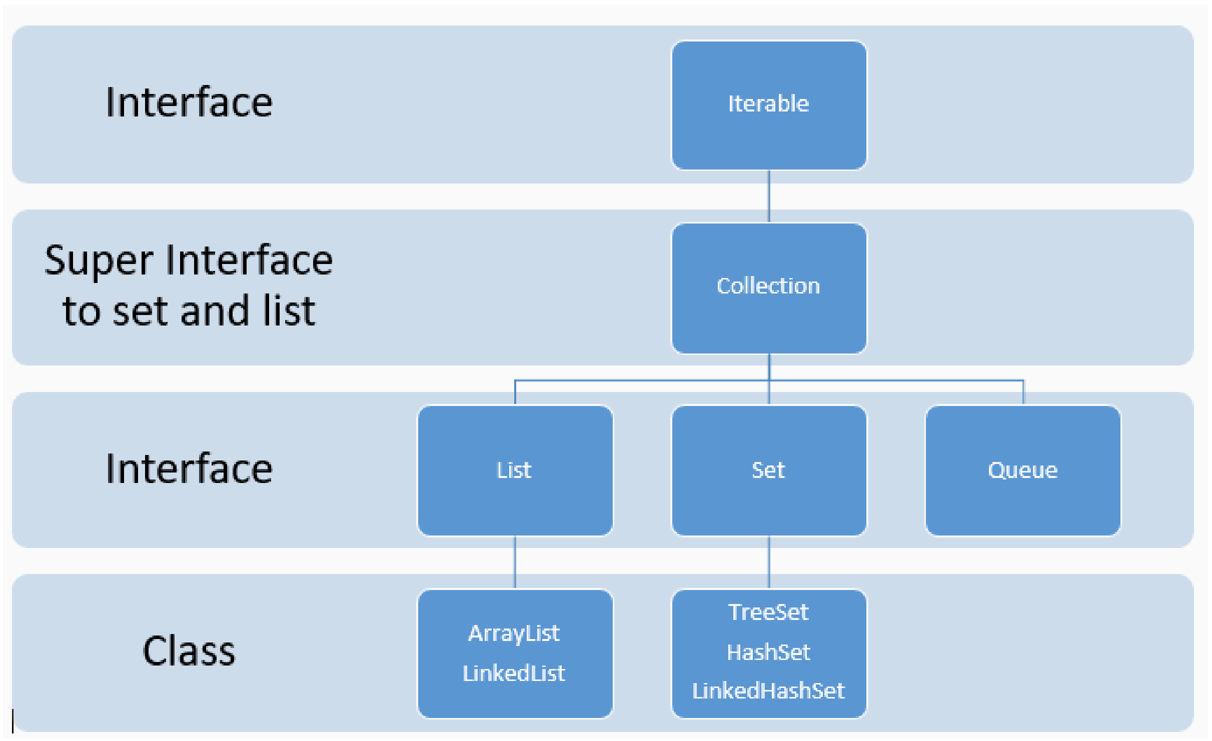
Collections Framework

The Java platform includes a collections framework.It is an architecture to store and manipulate the group of objects.A collection is an object that represents a group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Collections Framework contains all the classes and interfaces that are required under java.util package.

Hierarchy of Collection Framework



List

The List interface in Java provides a way to store the ordered collection. It is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements.

The List interface provides a special iterator, called a ListIterator, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The implementation classes of List interface widely used in Java programming are ArrayList and LinkedList .

ArrayList :

It is a class which implements List interface.

The ArrayList in Java can have the duplicate elements also.

Java ArrayList class maintains insertion order.Java ArrayList allows random access because the array works on an index basis.

In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Syntax:

```
ArrayList<Integer> listname = new ArrayList<Integer>();
```

Example :

```
import java.util.ArrayList;

public class ArrayListDemo {

        public static void main(String[] args) {

                ArrayList list1=new ArrayList();

                // add
                list1.add(10);

                list1.add(20);

                list1.add(300);

                list1.add(500);

                list1.add(true);

                list1.add("Selenium");
```

```
		list1.add(15.5);

		list1.add('c');

		System.out.println("Size before "+list1.size());

		System.out.println("All values" +list1);

		// get method
		System.out.println(list1.get(5));

		list1.remove(0);

		System.out.println("All values after remove" +list1);

		System.out.println("Size after "+list1.size());

	}

}
```

Example

```
import java.util.ArrayList;

public class ArrayListDemo9 {

	public static void main(String[] args) {

		ArrayList<Integer> l1=new ArrayList<Integer>();
		l1.add(100);

		l1.add(200);

		l1.add(300);

		l1.add(500);

		/*
		 *  You can pass another collection while creating list
		 *
		 */
		ArrayList<Integer> l2=new ArrayList<Integer>();

		for(int i=0;i<l1.size();i++)
		{
			l2.add(l1.get(i));
		}

		System.out.println(l1);



	}

}
```

Example of list with for each loop

```
import java.util.ArrayList;

public class ForEachWithList {

	public static void main(String[] args) {


		ArrayList<String> l1=new ArrayList<String>();

		l1.add("Java");

		l1.add("Selenium");

		l1.add("TestNG");

		l1.add("Git");

		l1.add("Github");

		l1.add("Docker");

		for (String str : l1)
		{
			System.out.println(str);
		}


	}
}
```

LinkedList:

Linkedlist is a class implementing List interface.Java LinkedList class can contain duplicate elements.

Java LinkedList class maintains insertion order.

Java LinkedList class is non synchronized.

In Java LinkedList class, manipulation is fast because no shifting needs to occur.

Java LinkedList class can be used as a list, stack or queue.

It provides additional methods like getlast , getfirst,remove first , remove last.

Syntax:

```
LinkedList<Type> linkedList = new LinkedList<>();
```

Example :

```
import java.util.LinkedList;

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

    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();

    // Add elements to LinkedList
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);
  }
}
```

Output

```
LinkedList: [Dog, Cat, Cow]
```

Set

In order to use functionalities of the Set interface, we can use these classes(popular in testing):

HashSet

LinkedHashSet

TreeSet

HashSet

HashSet implements Set Interface.

As it implements the Set Interface, duplicate values are not allowed.

Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.

NULL elements are allowed in HashSet.

Syntax :

```
 HashSet<type> setname = new HashSet<>();
```

Example :

```
// Java program to Demonstrate Working
// of HashSet Class

// Importing required classes
import java.util.*;

// Main class
// HashSetDemo
class Demo{

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty HashSet
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet
        // using add() method
        h.add("India");
        h.add("Australia");
        h.add("South Africa");

        // Adding duplicate elements
        h.add("India");

        // Displaying the HashSet
        System.out.println(h);
        }
        }
```

Output :

```
[South Africa, Australia, India]
```

Example :

```
public class SetDemo4 {

	public static void main(String[] args) {

		/*
		 * iterator
		 * splititerator
		 * listiterator are different
		 *
		 */

		HashSet<String> hs1=new HashSet<String>();

		hs1.add("Pavitra");

		hs1.add("Atul");

		hs1.add("Rajat");

		hs1.add("Riddhi");

		// Iterator generics depends on data available in list or set
		Iterator<String> itr1=hs1.iterator();

		// hasNext will check if any value is available - true/false
		// next method will return the value plus move to next element
		while(itr1.hasNext())
```

```
                {
                        String value=itr1.next();

                        System.out.println(value);
                }


        }

}
```

LinkedHashSet

Java LinkedHashSet class contains unique elements only like HashSet.
Java LinkedHashSet class provides all optional set operations and permits null elements.
Java LinkedHashSet class maintains insertion order.

Syntax :

```
 LinkedHashSet<type> setname = new LinkedHashSet<>();
```

Example :

```
// Java Program to Add Elements to LinkedHashSet

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
// AddingElementsToLinkedHashSet
```

```
class LinkedHashSetdemo {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty LinkedHashSet
        LinkedHashSet<String> hs = new LinkedHashSet<String>();

        // Adding elements to above Set
        // using add() method

        // Note: Insertion order is maintained
        hs.add("OTS");
        hs.add("Training");
        hs.add("Session");

        // Printing elements of Set
        System.out.println("LinkedHashSet : " + hs);
    }
}
```

Output:

```
LinkedHashSet : [OTS,Training,Session]
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage.TreeSet class contains unique elements only like HashSet.

Its access and retrieval times are quiet fast.

TreeSet class doesn't allow null element.It maintains ascending order.

Syntax :

```
 TreeSet<type> setname = new TreeSet<>();
```

Example :

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Java");
  al.add("Selenium");
  al.add("Playwright");
  al.add("Cypress");

   System.out.println(al);

 }
}
```

Output:

```
Java
Selenium
Playwright
Cypress
```

Maps:

Map is a separate interface whic is not a part of collection hierarchy.

Map contains key-value pairs.They have unique keys.Each key and value is called as entry.Searching, updation , deletion are easier in maps.

HashMap :

HashMap implements the map interface.It can take one null key and multiple null values.HashMap does not follows any order.If the key is removed , value will also be removed.

Syntax :

HashMap<K,V> map = new HashMap<K,V>();

Example :

```
// Java Program to illustrate the Hashmap Class

// Importing required classes
import java.util.*;

// Main class
public class Mapdemo {

   // Main driver method
   public static void main(String[] args)
   {

      // Creating an empty HashMap
      Map<String, Integer> map = new HashMap<>();

      // Inserting entries in the Map
      // using put() method
      map.put("Mukesh", 1);
      map.put("Ritika", 2);
      map.put("Komal", 3);
```

```
        // Iterating over Map
        for (Map.Entry<String, Integer> e : map.entrySet())


            // Printing key-value pairs
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }
}
```

Output

```
Mukesh 1
Ritika 2
Komal 3
```

TreeMap

TreeMap implements the map interface.It is a sorted collection in ascending order baked on the key.

No nulls are accepted.

Syntax:

```
TreeMap<Key,Value> map=new TreeMap<Key,Value>();
```

LinkedHashMap

It is the same as HashMap with an additional feature that it maintains insertion order.

Syntax:

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>();
```

Example :

```
import java.util.*;
```

```
class LinkedHashMap1{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();

  hm.put(100,"Mukesh");
  hm.put(101,"Ritika");
  hm.put(102,"Komal");

for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

```
Output:100 Mukesh
       101 Ritika
       102 Komal
```

Iterator

It is an interface.Iterators in Java are used in the Collection framework to retrieve elements one by one.

Syntax :

```
      Iterator itr = c.iterator();
```

Note: Here “c” is any Collection object. itr is of type Iterator interface and refers to “c”.

Methods of Iterator Interface in Java :

1. hasNext(): Returns true if iternation ha smore elements

2. next():Gives current element and returns next element in iterator.

```
// Java program to Demonstrate Iterator

// Importing ArrayList and Iterator classes
// from java.util package
import java.util.ArrayList;
import java.util.Iterator;

// Main class
public class Test {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an ArrayList class object
        // Declaring object of integer type
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Iterating over the List
        for (int i = 0; i < 10; i++)
            al.add(i);

        // Printing the elements in the List
        System.out.println(al);

        // At the beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator<Integer> itr = al.iterator();

        // Checking the next element  where
        // condition holds true till there is single element
        // in the List using hasnext() method
        while (itr.hasNext()) {
            //  Moving cursor to next element
            int i = itr.next();

            // Getting elements one by one
            System.out.print(i + " ");



        }


           }
}
```

Output

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```

Exception Handling

Exceptional handling in Java is a very powerful mechanism as it helps to identify exceptional conditions and maintain the flow of the program as expected, by handling/avoiding the errors if occurred. In some cases, it is used to make the program user-friendly.

An exception is an unwanted or unexpected event that occurs during the execution of the program, that disrupts the flow of the program.

For example, if a user is trying to divide an integer by 0 then it is an exception, as it is not possible mathematically.

There are various types of interruptions while executing any program like errors, exceptions, and bugs. These interruptions can be due to programming mistakes or due to system issues. Depending on the conditions they can be classified as errors and exceptions.

Java has classes that are used to handle built-in exceptions and provision to create user-defined exceptions.

We handle exceptions in java to make sure the program executes properly without any halt, which occurs when an exception is raised.

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Exception

As mentioned earlier exceptions are unwanted conditions that disrupt the flow of the program.

Exceptions usually occur due to the code and can be recovered.

Exceptions can be of both checked(exceptions that are checked by the compiler) and unchecked(exceptions that cannot be checked by the compiler) type.

They can occur at both run time and compile time.

In Java, exceptions belong to java.lang.Exception class.

Exceptions can be categorized in two ways:

Built-in Exceptions

-Checked Exception

-Unchecked Exception

User-Defined Exceptions

Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions:
Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions:
The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.

Error

An error is also an unwanted condition but it is caused due to lack of resources and indicates a serious problem.

Errors are irrecoverable, they cannot be handled by the programmers.

Errors are of unchecked type only.

They can occur only at run time.

In java, errors belong to java.lang.error class.

Eg: OutOfMemmoryError, AssertionsError , VirtualMachineError etc

Exception Keywords

Java uses try-catch blocks and other keywords like finally, throw, and throws to handle exceptions.

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the

	exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
throw	The "throw" keyword is used to throw an exception.The code cannot continue as expected.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It is always used with method signature.
e parameter	It helps in accessing information about the exception.

Try-Catch block

Syntax :

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Example

```
public class TryCatchExample {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

The rest of the code is executed, i.e., the rest of the code statement is printed.

Try with Multi-Catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler.At one time only one exception occurs which means only one catch block will be executed.

Syntax

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref)
{
//catch for exception
}
```

```
catch (Exception_class_Name ref2)
{
// catch for exception
}
```

Example

```
public class MultipleCatchBlock {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }

           System.out.println("rest of the code");
   }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

Nested try block

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

We can write a try block inside the other try block a.It is called as Nested try.

Syntax:

```
....
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
```

```
        }


    }
    catch(Exception e1)
    {
//exception message
    }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

In the below example , there are 2 exceptions handled by inner try-catch block.

Example :

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
    try{
     System.out.println("going to divide by 0");
     int b =39/0;
   }
    //catch block of inner try block 1
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }

    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
       System.out.println(e);
    }
    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
    System.out.println("handled the exception (outer catch)");
  }
     }
}
```

Output :

```
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

In the below example , the inner try block does not handles the exception , hence the parent try block will handle the exception.

```
public class NestedTryBlock2 {

    public static void main(String args[])
    {
        // outer (main) try block
        try {

            //inner try block 1
            try {

                // inner try block 2
                try {
                    int arr[] = { 1, 2, 3, 4 };

                    //printing the array element out of its bounds
                    System.out.println(arr[10]);
                }

                // to handles ArithmeticException
                catch (ArithmeticException e) {
                    System.out.println("Arithmetic exception");
                    System.out.println(" inner try block 2");
                }
            }

            // to handle ArithmeticException
            catch (ArithmeticException e) {
                System.out.println("Arithmetic exception");
                System.out.println("inner try block 1");
            }
        }

        // to handle ArrayIndexOutOfBoundsException
        catch (ArrayIndexOutOfBoundsException e4) {
            System.out.print(e4);
            System.out.println(" outer (main) try block");
        }
        catch (Exception e5) {
            System.out.print("Exception");
            System.out.println(" handled in main try-block");
        }
    }
}
```

Finally

The code within the finally block is always executed even whether the exception is thrown or not.

Syntax :

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref)
{
// catch code block
}
finally{
// final code block
}
```

Example :

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
}
```

Output:

```
5
finally block is always executed
rest of phe code...
```

In the above example , no exception occurred.However , the final block is executed.

Another example is listed below , where exception occurs and than final block is executed.

```
public class TestFinallyBlock2{
      public static void main(String args[]){

      try {

        System.out.println("Inside try block");

        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }

      //handles the Arithmetic Exception / Divide by zero exception
      catch(ArithmeticException e){
        System.out.println("Exception handled");
        System.out.println(e);
      }

      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

      System.out.println("rest of the code...");
      }
    }
```

Output :

```
 Inside try block
Exception handled
```

```
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code…
```

throw Vs throws

Throw	Throws
The throw exception is used inside the method	The throws exception is used with method signature
It only propagates unchecked exception	Using throws keyword, we can declare both checked and unchecked exceptions.
We are allowed to throw only one exception at one time	We can declare multiple exceptions using throws keyword that can be thrown by the method
The throw keyword is followed by an instance of Exception to be thrown	The throws keyword is followed by class names of Exceptions to be thrown.

Syntax:
 	
access-modifier return-type method name()
{ // method body
throw instance
}
	Syntax
	
	access-modifier return-type methodname() throws exceptionslist
	{
	// method body
	}

Example 1

```
// Java program to illustrate throws
class Demo {
    public static void main(String[] args)
```

```
        throws InterruptedException
    {
        Thread.sleep(10000);
        System.out.println("Good Morning");
    }
}
```

Output:

```
Good Morning
```

In the above program, we will get compile time error if exception is not thorwn,InterruptedException. It is because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute the main() method.

Example 2:

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    } else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15);
  }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at
least 18 years old.
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:12)
```

Dead Code

Dead code is code that can never be executed in a running program. The surrounding code makes it impossible for a section of code to ever be executed.

```
    public class DeadCodeExample {
    public static void main(String[] args) {
        int i = 0;

        while (false) {
            // This block of code will never be executed
            System.out.println("This line will never be printed.");
            i++;
        }

        System.out.println("The value of i is: " + i);
    }
}
```

In this example, the while loop condition is false, which means the loop will never execute because the condition is always false. As a result, the code inside the loop will never be executed, making it dead code.

Example with try-catch block

```
public class DeadCodeExample {
    public static void main(String[] args) {
        try {
            throw new IllegalArgumentException("This is an illegal argument.");
        } catch (ArithmeticException ae) {
            // This catch block will never be executed
            System.out.println("Caught an arithmetic exception: " + ae.getMessage());
        }
    }
}
```

In this example, we are throwing an IllegalArgumentException inside the try block. The catch block is expecting an ArithmeticException and will not catch the IllegalArgumentException, making the catch block unreachable and therefore dead code.