

Unit-1

Programming in Java

Introduction

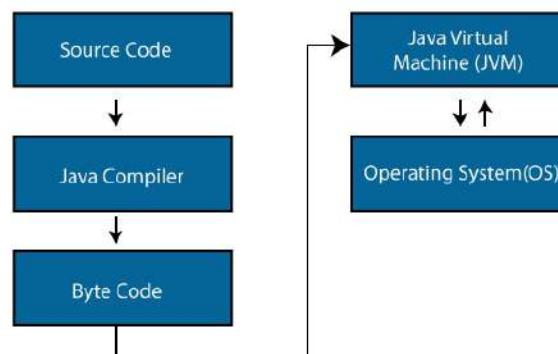
Java is general purpose, object oriented, high-level programming language. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. Java is used to develop Mobile apps, Web apps, Desktop apps, Games and much more.

Java Architecture

Java Architecture is a collection of components, i.e., **JVM**, **JRE**, and **JDK**. It integrates the process of interpretation and compilation. It defines all the processes involved in creating a Java program. Java Architecture explains each and every step of how a program is compiled and executed.

The below-mentioned points and diagram will simply illustrate Java architecture:

1. There is a compilation and interpretation process in Java.
2. After the JAVA code is written, the JAVA compiler comes into the picture that converts this code into byte code that can be understood by the machine.
3. After the creation of bytecode JAVA virtual machine(JVM) converts it to the machine code, i.e. (.class file)
4. And finally, the machine runs that machine code.



Components of Java Architecture

- **JVM (Java Virtual Machine)**: The main feature of Java is **Write Once Run Anywhere**. The feature states that we can write our code once and use it anywhere or on any operating system. Our Java program can run any of the platforms only because of the Java Virtual Machine. It is a Java platform component that gives us an environment to execute java programs. JVM's main task is to convert byte code into machine code.
- **Java Runtime Environment (JRE)**: It provides an environment in which Java programs are executed. JRE takes our Java code, integrates it with the required libraries, and then starts the JVM to execute it. The JRE contains libraries and software needed by your Java programs to run.
- **Java Development Kit (JDK)**: JDK is a software development environment used to develop Java applications and applets. It contains JRE and several development tools, an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) accompanied with another tool.

Java Buzzwords / Features of Java

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- **Simple:** Java programming language is very simple and easy to learn, understand, and code. Most of the syntaxes in java follow basic programming language C and object-oriented programming concepts are similar to C++. In a java programming language, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed.
- **Secure:** Java is said to be more secure programming language because it does not have pointers concept, java provides a feature "applet" which can be embedded into a web application. The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.
- **Portable:** Portability is one of the core features of java which enables the java programs to run on any computer or operating system. For example, an applet developed using java runs on a wide variety of CPUs, operating systems, and browsers connected to the Internet.
- **Object-oriented:** Java is a pure object-oriented language, almost everything in java is an object, all program code and data reside within object and classes that is everything in java is defined within a class.
- **Robust:** Java is more robust because the java code can be executed on a variety of environments, java has a strong memory management mechanism (garbage collector), java is a strictly typed language, it has a strong set of exception handling mechanism, and many more.
- **Architecture Neutral:** Java language and Java Virtual Machine (JVM) helped in achieving the goal of "write once; run anywhere, any time, forever." Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.
- **Multithreaded:** Java supports multi-threading programming that allows to write programs to do several works simultaneously. A thread is an individual process to execute a group of statements. JVM utilizes multiple threads to execute different blocks of code. Creating multiple threads is called 'multithreaded' in Java.
- **Interpreted:** During compilation, Java compiler converts the source code of the program into byte code. This byte code can be executed on any system machine with the help of Java interpreter in JVM.
- **High performance:** Java performance is high because of the use of bytecode. The bytecode was used so that it was easily translated into native machine code.
- **Distributed:** Java programming language supports TCP/IP protocols which enable the java to support the distributed environment of the Internet. Java also supports Remote Method Invocation (RMI), this feature enables a program to invoke methods across a network.
- **Dynamic:** Java programs access various runtime libraries and information inside the compiled code (Bytecode). This dynamic feature allows to update the pieces of libraries without affecting the code using it.

Path and ClassPath Variables

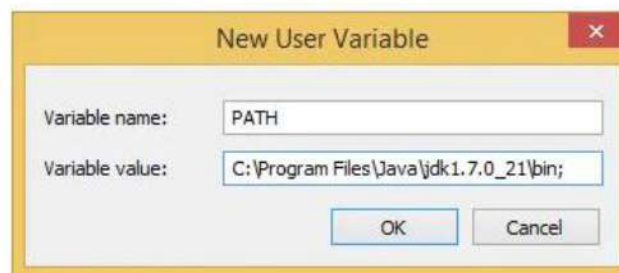
Path

Once we installed Java on our machine, it is required to Set the PATH environment variable to conveniently run the executable (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command, such as:

```
C:\javac TestClass.java
```

Otherwise, you need to specify the full path every time you run it, such as:

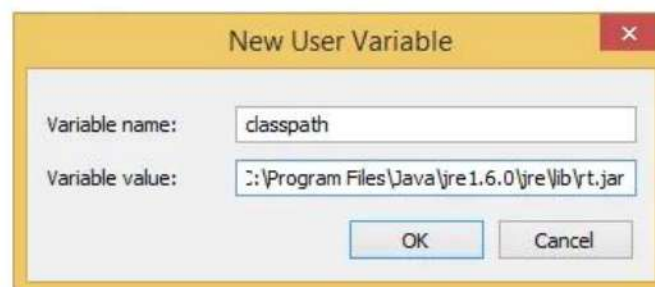
```
C:\Java\jdk1.7.0\bin\javac TestClass.java
```



ClassPath

ClassPath is system environment variable used by the Java compiler and JVM. Java compiler and JVM is used Classpath to determine the location of required class files. It contains a path of the classes provided by JDK.

```
C:\Program Files\Java\jdk1.6.0\bin
```



Sample Java Program

Let's create the Hello World program:

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

- In our Hello World program, we have a class called *HelloWorld*. As a convention, always start the name of your classes with an uppercase letter. To create a class, you use the *class* keyword, followed by the name of the class.

- Every Java program must have a **main** method. This tells the Java compiler that this is the beginning of the Java program. The program then executes all the statements following the main method. Here's what the main method looks like:

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

- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

Compiling and Running Java Programs

Step 1: Write a program on the notepad and save it with **.java** (for example, DemoProg.java) extension.

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

Step 2: Open a command prompt window and go to the directory where you saved the class. Assume it's C:\\demo

Step 3: Type '**javac DemoProg.java**' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line.

Step 4: Now, type '**java DemoProg.java**' to run your program.

Output Methods print() and println()

The **println("...")** method prints the string "... " and moves the cursor to a new line. The **print("...")** method instead prints just the string "...", but does not move the cursor to a new line. Hence, subsequent printing instructions will print on the same line. The **println()** method can also be used without parameters, to position the cursor on the next line.

Example

```
public class JavaExample {  
  public static void main(String[] args) {  
    System.out.println("Welcome to Collegenote!");  
    System.out.println("Welcome to Collegenote!");  
    System.out.print("Collegenote");  
    System.out.print("Collegenote");  
  }  
}
```

Output

```
Welcome to Collegenote!  
Welcome to Collegenote!  
CollegenoteCollegenote
```

Reading Data Input from Users

Java *Scanner* class allows the user to take input from the console. It belongs to *java.util* package.

Syntax: *Scanner sc = new Scanner(System.in);*

Example

// *AddTwoNumbers.java*: This program read two numbers from user and finds their sum.

```
import java.util.Scanner;
public class AddTwoNumbers {

    public static void main(String[] args) {

        int num1, num2, sum;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter First Number: ");
        num1 = sc.nextInt();

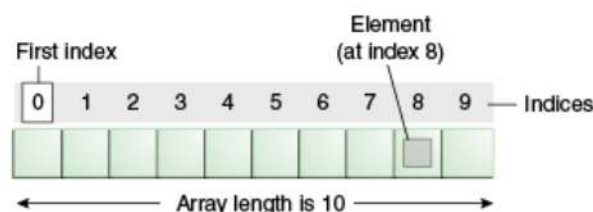
        System.out.println("Enter Second Number: ");
        num2 = sc.nextInt();

        sum = num1 + num2;
        System.out.println("Sum of these numbers: "+sum);
    }
}
```

Arrays in Java

Java array is a collection of similar type of elements that have contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in java is index based, for n –sized array first element of the array is stored at 0 index and last element is stored in index $n - 1$.



There are two types of array: *One-Dimensional Array* & *Multi-Dimensional Array*

One-Dimensional Arrays

In one-dimensional arrays, a list of items can be given one variable name using only one subscript.

Syntax to Declare an Array in Java

data-type arrayName[]; OR *data-type[] arrayName;*

For example: `int[] nums;`

Here, `nums` is an array that can hold values of type `int`.

Creation: When an array is declared, only a reference of an array is created. After declaring an array, we need to create it in the physical memory. Java allows us to create arrays using ***new*** operator only, as shown below:

```
arrayName = new type[size]
```

Example:

```
int[] nums;           // declare an array
nums = new int[10];  // allocate memory
```

OR

```
int[] nums = new int[10]; // combining both statements in one
```

Initialization: In Java, we can initialize arrays during declaration. For example,

```
int[] age = {12, 4, 5, 2, 5}; //declare and initialize an array
```

We can also initialize arrays in Java, using the index number. For example,

// declare an array

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

// initialize array

```
age[0] = 12;
age[1] = 4;
age[2] = 5;
..
```

Accessing Array Elements: We can access the element of an array using the index number. Here is the syntax for accessing elements of an array,

```
array[index]
```

Example:

```
class Main {
    public static void main(String[] args) {
        // create an array
        int[] age = {12, 4, 5, 2, 5};

        // access each array elements
        System.out.println("Accessing Elements of Array:");
        System.out.println("First Element: " + age[0]);
        System.out.println("Second Element: " + age[1]);
        System.out.println("Third Element: " + age[2]);
        System.out.println("Fourth Element: " + age[3]);
        System.out.println("Fifth Element: " + age[4]);
    }
}
```

Q. Write a Java program to find sum and average of all elements in an array.

Solution:

```
//Sum_Average.java
import java.util.Scanner;
public class Sum_Average
{
    public static void main(String[] args)
    {
        int n, sum = 0;
        float average;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array:");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");
        for(int i = 0; i < n ; i++)
        {
            a[i] = s.nextInt();
            sum = sum + a[i];
        }
        System.out.println("Sum:"+sum);
        average = (float)sum / n;
        System.out.println("Average:"+average);
    }
}
```

Multi-Dimensional Arrays

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java:

```
dataType[][] arr;           OR           dataType arr[][];
```

Example to instantiate Multidimensional Array in Java:

```
int[][] ar = new int[3][4]; //3 row and 4 column
```

Here, we have created a multidimensional array named *ar*. It is a 2-dimensional array, that can hold a maximum of 12 elements,

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Example:

```

public class multiDimensional {
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int ar[][] = { { 2, 7, 9, 5 }, { 3, 6, 1, 8 }, { 7, 4, 2, 3 } };
        // printing 2D array
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 4; j++){
                System.out.print(ar[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

Output

```

2 7 9 5
3 6 1 8
7 4 2 3

```

Q. Write a Java program to enter two 3x3 matrices and calculate the sum of given matrices.

Solution:

```

import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner ed = new Scanner(System.in);
        int row, col, i, j;
        System.out.print("Enter number of rows:");
        row = ed.nextInt();
        System.out.print("Enter number of column:");
        col = ed.nextInt();

        int[][] a = new int[row][col];
        int[][] b = new int[row][col];
        int[][] sum = new int[row][col];

        System.out.println("Enter first matrix:");
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                a[i][j] = ed.nextInt();
            }
        }

        System.out.println("Enter Second matrix:");
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                b[i][j] = ed.nextInt();
            }
        }

        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                sum[i][j] = b[i][j] + a[i][j];
            }
        }
    }
}

```

Output

```

Enter number of rows:3
Enter number of column:3
Enter first matrix:
1 6 2
3 7 1
2 3 5
Enter Second matrix:
2 7 1
3 9 5
4 2 8
Sum of two matrices:
3 13 3
6 16 6
6 5 13

```



```

    System.out.println("Sum of two matrices:");
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            System.out.print(sum[i][j] + " ");
        }
        System.out.print("\n");
    }
}
}
}

```

For Each Loop

In Java, the *for-each* loop is used to iterate through elements of arrays and collections (like ArrayList). It is also known as the enhanced for loop.

Advantages:

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Disadvantages:

- It cannot traverse the elements in reverse order.
- We do not have the option to skip any element because it does not work on an index basis.

Syntax

```

for (data_type variableName : array | collection) {
    // code block to be executed
}

```

For each iteration, the *for-each* loop takes each element of the collection and stores it in a loop variable. Thus, it executes the code written in the body of the loop for each element of the array or collection.

Example

```

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

        // create an array
        int[] numbers = {3, 9, 5, -5};

        // for each loop
        for (int number: numbers) {
            System.out.println(number);
        }
    }
}

```

Output

```

3
9
5
-5

```

Here, the *for-each* loop traverses over each element of the array *numbers* one by one until the end.

Class and Object

Class

A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind. It is a template or blueprint from which objects are created.

To define a class in java we use a keyword **class**. The general form to define a direct new class is as follows:

```
<Access><Modifier> class <className> {
    //field, constructor, and method declarations
}
```

The general form to define a new class by extending the class and implementing the interfaces is as follows:

```
<Access> <Modifier> class <className> extends <superClass> implements <YourInterface> {
    //field, constructor, and method declarations
}
```

Example

```
public class Person
{
    //state or field or variable
    String name = "Jayanta";
    int age = 20;

    //creating the methods of the class
    void study()
    {
        //methodBody
    }
    void play()
    {
        //methodBody
    }
    public static void main(String args[])
    {
        System.out.println("Name of the person: " +name);
        System.out.println("Age of the person: " +age);
    }
}
```

Object

An object is an instance of a class. To create an object of a class, first, we need to declare it and then instantiate it with the help of a “**new**” keyword.

Syntax of creating an object of a class: *ClassName objectName = new ClassName();*

Example: *Person object1 = new Person();*

Accessing the members of a Java Class: We can access the data members of a class using the object of the class. We just write the name of the object which is followed by a dot operator then we write the name of the data member (either variables or methods) which we want to access.

Syntax:

```
objectName.variableName;           //accessing the variables
objectName.MethodName();           //accessing the methods
```

Example:

```
object1.age;                         //accessing the variables
object1.play();                       //accessing the methods
```

Q. Write a Java program to create Rectangle class with data member length and breadth. Include methods getData() and displayArea() in the class. Finally create an object of Rectangle class and display its area.

Solution:

```
import java.util.Scanner;
class Rectangle{
    int l, b;
    void getData(){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter length : ");
        l=in.nextInt();

        System.out.print("Enter breadth : ");
        b=in.nextInt();
    }
    void displayArea(){
        int a;
        a = l*b;
        System.out.println("Area = "+a);
    }
    public static void main(String args[]){
        Rectangle obj = new Rectangle();
        obj.getData();
        obj.displayArea();
    }
}
```

Output

```
Enter length : 5
Enter breadth : 3
Area = 15
```

Constructors

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. At the time of calling constructor, memory for the object is allocated in the memory.

Constructor has the same name as its class and is syntactically similar to a method. However, constructors have no return type. All classes have constructors, whether you define one or not, because Java automatically provides a **default constructor** that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Types of Constructor:

<i>No-Argument Constructors</i>	<i>Parameterized Constructors</i>
<p>The no-argument constructor of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.</p> <p>Example:</p> <pre>class Rectangle { double length; double breadth; Rectangle() //No-arg constructor { length = 15.5; breadth = 10.67; } double calculateArea(){ return length*breadth; } } class Rectangle_Main { public static void main(String[] args) { double area; Rectangle myrec = new Rectangle(); area = myrec.calculateArea(); System.out.println("The area of the Rectangle: " +area); } }</pre>	<p>A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructor with parameters).</p> <p>Example:</p> <pre>class Rectangle { double length; double breadth; // Parameterized constructor Rectangle(double l, double b) { length = l; breadth = b; } double calculateArea(){ return length*breadth; } } class Rectangle_Main { public static void main(String[] args) { double area; Rectangle myrec = new Rectangle(41,56); area = myrec.calculateArea(); System.out.println("The area of Rectangle: " +area); } }</pre>

Method Overloading

If a class has multiple methods having same name but different in parameters (different number of parameters, different types of parameters, or both), it is known as **Method Overloading**.

When a method is invoked, Java matches up the method name first and then the number and type of parameters to decide which one of the definition is execute.

Example

```
public class Sum {  
  
    // Overloaded the method sum(). This sum takes two int parameters as input  
    public int sum(int a, int b)  
    {  
        return (a + b);  
    }  
  
    // Overloaded the method sum(). This sum takes three int parameters as input  
    public int sum(int a, int b, int c)  
    {  
        return (a + b + c);  
    }  
  
    // Overloaded the method sum(). This sum takes two double parameters as input  
    public double sum(double a, double b)  
    {  
        return (a + b);  
    }  
  
    // Main code  
    public static void main(String args[])  
    {  
        Sum s = new Sum();  
        System.out.println(s.sum(3, 2));  
        System.out.println(s.sum(2, 2, 4));  
        System.out.println(s.sum(10.5, 20.5));  
    }  
}
```

Output

```
5  
8  
31.0
```

In this example, the number of parameters as well as its data type is changed to overload the method. If the parameters provided are both int then the first sum method is executed. If there are three parameters, all int then the second method is invoked. In case two double data type parameters are given then the third method is invoked.

Static Modifier

The static keyword is a non-access modifier used for methods and variables. Static methods/variables can be accessed without creating an object of a class.

The main purpose of using the static keyword in Java is to save memory. When we create a variable in a class that will be accessed by other classes, we must first create an instance of the class and then assign a new value to each variable instance – even if the value of the new variables are supposed to be the same across all new classes/objects. But when we create a static variables, its value remains constant across all other classes, and we do not have to create an instance to use the variable. This way, we are creating the variable once, so memory is only allocated once.

- ***Static Methods:*** Static methods are also called class methods. It is because a static method belongs to the class rather than the object of a class. And we can invoke static methods directly using the class name. For example,

```
class StaticTest
{
    // non-static method
    int multiply (int a, int b)
    {
        return a * b;
    }

    // static method
    static int add (int a, int b)
    {
        return a + b;
    }
}

public class StaticMethodDemo
{
    public static void main (String[] args)
    {
        // create an instance of the StaticTest class
        StaticTest st = new StaticTest ();

        // call the nonstatic method
        System.out.println (" 5 * 5 = " + st.multiply (5, 5));

        // call the static method
        System.out.println (" 5 + 3 = " + StaticTest.add (5, 3));
    }
}
```

There are two main restrictions for the static method. They are:

1. The static method cannot use non-static data members or call the non-static method directly.
2. this and super cannot be used in a static context.

- **Static Variables:** If we declare a variable static, all objects of the class share the same static variable. It is because like static methods, static variables are also associated with the class. And, we don't need to create objects of the class to access the static variables. For example,

```
class Test {
    // static variable
    static int max = 10;

    // non-static variable
    int min = 5;
}

public class Main {
    public static void main(String[] args) {
        Test obj = new Test();

        // access the non-static variable
        System.out.println("min + 1 = " + (obj.min + 1));

        // access the static variable
        System.out.println("max + 1 = " + (Test.max + 1));
    }
}
```

Access static Variables and Methods within the Class

We are accessing the static variable from another class. Hence, we have used the class name to access it. However, if we want to access the static member from inside the class, it can be accessed directly. For example,

```
public class Main {
    // static variable
    static int age;

    // static method
    static void display() {
        System.out.println("Static Method");
    }

    public static void main(String[] args) {
        // access the static variable
        age = 30;
        System.out.println("Age is " + age);

        // access the static method
        display();
    }
}
```

Inheritance

Inheritance is the mechanism of deriving a new class from existing class. The existing class is known as the superclass or base class or parent class and the new class is called subclass or derived class or child class or extended class. The subclass inherits some of the properties from the superclass and can add its own properties as well.

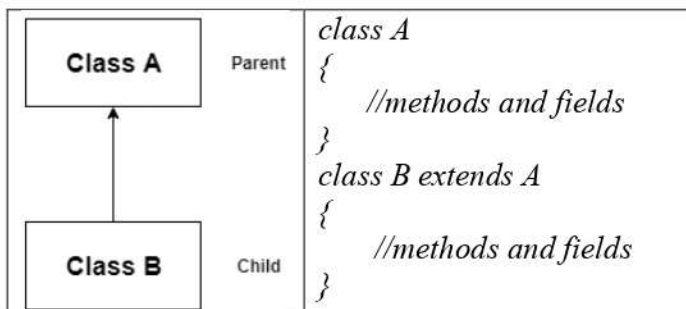
extends is the keyword used to inherit the properties of a class.

Syntax:

```
class BaseClass
{
    //methods and fields
}
class DerivedClass extends BaseClass
{
    //methods and fields
}
```

Types of Inheritance

1. **Single Inheritance:** In single inheritance, a class is derived from only one existing class.

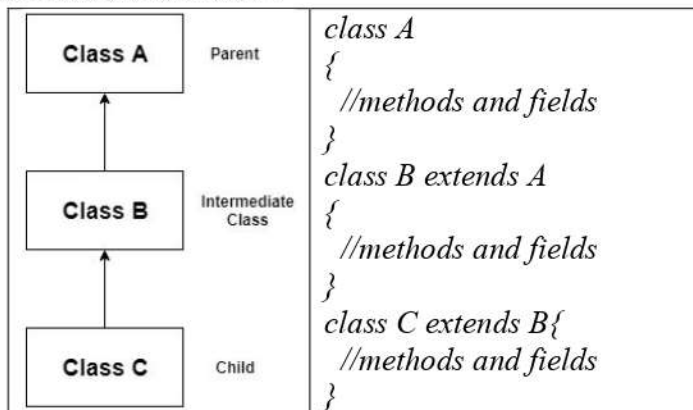


Example:

```
class A{
    void func1 ()
    {
        System.out.println("Method func1 belongs to parent class A");
    }
}
class B extends A{ // Notice the extends keyword ( inheriting B from A)
    void func2 ()
    {
        System.out.println("Method func2 belongs to child class B");
    }
}
class Single{
    public static void main(String[] args)
    {
        B obj = new B();
        obj.func2 ();
        obj.func1 (); // Note that object of class B is used to invoke func1()
    }
}
```

Output: Method func2 belongs to child class B
Method func1 belongs to parent class A

2. **Multilevel Inheritance:** The mechanism of deriving a class from another subclass is known as multilevel inheritance.



Example:

```

class A{
    void func1(){
        System.out.println("Method func1 belongs to parent class A");
    }
}
class B extends A{ // Notice the extends keyword ( inheriting B from A)
    void func2(){
        System.out.println("Method func2 belongs to intermediate class B");
    }
}
class C extends B{
    void func3(){
        System.out.println("Method func3 belongs to child class C");
    }
}
}
class Multilevel{
    public static void main(String[] args){
        C obj = new C();
        obj.func3();
        obj.func2();
        obj.func1();
    }
}

```

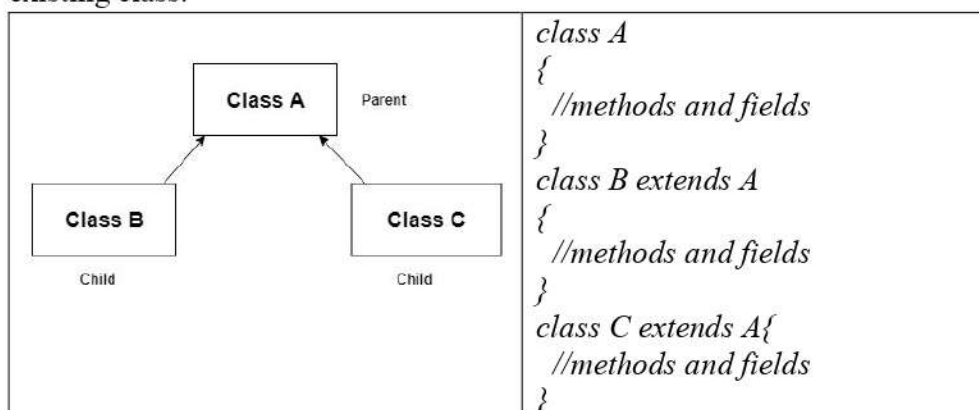
```

Method func3 belongs to child class C
Method func2 belongs to intermediate class B
Method func1 belongs to parent class A

```

Output:

3. **Hierarchical Inheritance:** In this type, two or more classes inherit the properties of one existing class.



Example:

```

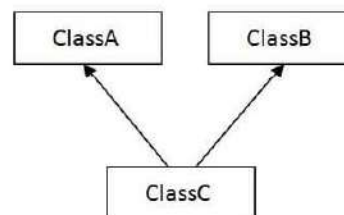
class Values{
    int len,bre;
    void getValue(int l,int b){
        len=l;
        bre=b;
    }
}
class Rect extends Values{
    void recArea(){
        System.out.println("Area of rectangle = "+(len*bre));
    }
}
class Square extends Values{
    void sqArea(){
        System.out.println("Area of Square = "+(len*len));
    }
}

class Heirarchical{
    public static void main(String[] args){
        Rect r = new Rect();
        Square sq = new Square();
        r.getValue(10,20);
        r.recArea();
        sq.getValue(10,10);
        sq.sqArea();
    }
}

```

Output: Area of rectangle = 200
Area of Square = 100

4. **Multiple Inheritance:** When one class inherits multiple classes, it is known as multiple inheritance.



Java doesn't support multiple inheritance. We can achieve multiple inheritances only with the help of **Interfaces**.

Q. Why multiple inheritance is not supported in Java?**Solution:**

In Java multiple inheritance is not supported because it may become ambiguous in case if more than one parent class have same method. Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and we call it from child class object, there will be ambiguity to call method of A or B class. To prevent such situation, multiple inheritances is not allowed in java.

Access Modifiers

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class. E.g.

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello");}
}

public class B{
    public static void main(String args[]){
        A obj = new A();
        System.out.println(obj.data); //Compile Time Error
        obj.msg(); //Compile Time Error
    }
}
```

2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If we do not specify any access level, it will be the default. E.g.

<pre>//save by A.java package pack; class A{ void msg(){System.out.println("Hello");} } The scope of class A and its method msg() is default so it cannot be accessed from outside the package.</pre>	<pre>//save by B.java package mypack; import pack.*; class B{ public static void main(String args[]){ A obj = new A(); //Compile Time Error obj.msg(); //Compile Time Error } }</pre>
--	---

3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If we do not make the child class, it cannot be accessed from outside the package. E.g.

<pre>//save by A.java package pack; public class A{ protected void msg(){ System.out.println("Hello"); } }</pre>	<pre>//save by B.java package mypack; import pack.*; class B extends A{ public static void main(String args[]){ B obj = new B(); obj.msg(); } } Output: Hello</pre>
--	--

4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Method Overriding

Declaring a method in subclass which is already present in parent class is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. For method overriding, the method must have same name and same type signature or parameters in both the superclass and the subclass.

We can use **super** keyword to call overridden member of superclass from subclass.

Example:

```
class Animal {
    public void eat(){
        System.out.println("Eat all eatables");
    }
}
class Dog extends Animal {
    //eat() method overridden by Dog class.
    public void eat(){
        super.eat()           //This will call the eat() method of parent class
        System.out.println("Dog likes eating bones");
    }
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
    }
}
```

Difference Between Method Overloading and Method Overriding

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or cannot be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
Argument list should be different while doing method overloading.	Argument list should be same in method overriding.

Final Modifier

Java final keyword is a non-access modifier that is used to restrict a class, variable, and method.

***final variable** → can never be modified*
***final method** → can never be overridden*
***final class** → can never be inherited*

1. **Final Variable:** Once we declare a variable with the final keyword, we can't change its value again. If we attempt to change the value of the final variable, then we will get a compilation error. Generally, we can consider a final variable as a constant, as the final variable acts like a constant whose values cannot be changed. For example,

```
class Main {
    public static void main(String[] args) {
        // create a final variable
        final int AGE = 20;

        // try to change the final variable
        AGE = 25;
        System.out.println("Age: " + AGE);
    }
}
```

In the above program, we have tried to change the value of the final variable. When we run the program, we will get a compilation error.

2. **Final Method:** The method with final keyword cannot be overridden in the subclasses. The purpose of creating the final methods is to restrict the unwanted and improper use of method definition while overriding the method. For example,

```
public class Parent {
    final void final_method() {
        //definition of the Final Method
    }
}

public class Child extends Parent {
    final void final_method() //overriding the method from the parent class
    {
        // another definition of the final method
    }
}
```

The above example of the final method generates a compile-time error.

3. **Final Class:** When we declare a class as final, we cannot inherit or extend it. If another class attempts to extend the final class, then there will be a compilation error. For example,

```
final class A {
    // methods and variables of the class A
}

class B extends A {
    // COMPILE- TIME error as it extends final class
}
```

Interface

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Interface cannot be instantiated – they can only be implemented by classes or extended by other interfaces. By default all variables defined in interfaces are constants and all the members in an interface are public implicitly. Also, the variables declared in an interface are public, static & final by default.

To use an interface, other classes must implement it. A class that implements an interface must implement all the methods declared in the interface.

We use the ***interface*** keyword to create an interface in Java.

<u>Syntax</u>	<u>Example</u>
<pre>interface InterfaceName{ // constant fields/variable declaration // abstract methods declaration }</pre>	<pre>interface Area{ float PI = 3.1415; float computeArea(float x, float y); }</pre>

The method signature have no braces and are terminated with a semicolon.

Implementing Interfaces

To use an interface, other classes must implement it. We use the ***implements*** keyword to implement an interface. A class can implement any number of interfaces.

Syntax:

```
class ClassName implements InterfaceName1, InterfaceName2,...{
    //define interface methods here
}
```

Example

```
interface Person {
    int PERSON_AGE = 20;
    String PERSON_NAME = "JAYANTA";
    public void displayInfo();
}

public class InterfaceExamples implements Person {
    public void displayInfo() {
        System.out.println("Person Name : " + PERSON_NAME);
        System.out.println("Person Age : " + PERSON_AGE);
    }

    public static void main(String[] args) {
        InterfaceExamples exp = new InterfaceExamples();
        exp.displayInfo();
    }
}
```

The above example shows that the *InterfaceExamples* class implements the *Person* Interface.

Extending Interfaces

Similar to classes, interfaces can extend other interfaces. The **extends** keyword is used for extending interfaces. For example,

```
interface Line {
    // members of Line interface
}
interface Polygon extends Line {
    // members of Polygon interface
    // members of Line interface
}
```

Here, the *Polygon* interface extends the *Line* interface. Now, if any class implements *Polygon*, it should provide implementations for all the abstract methods of both *Line* and *Polygon*.

Extending Multiple Interfaces: An interface can extend multiple interfaces. For example,

```
interface A {
    ... }
interface B {
    ... }
interface C extends A, B {
    ...
}
```

Use of Interface for achieving Multiple Inheritance

We can achieve multiple inheritance, by using the concept of interface in two ways:

1) By implementing more than one interface

```
interface AnimalEat {
    void eat();
}
interface AnimalTravel {
    void travel();
}
class Animal implements AnimalEat, AnimalTravel {
    public void eat() {
        System.out.println("Animal is eating");
    }
    public void travel() {
        System.out.println("Animal is travelling");
    }
}
public class Demo {
    public static void main(String args[]) {
        Animal a = new Animal();
        a.eat();
        a.travel();
    }
}
```

2) By extending one class and implementing another interface

```

class A{
    int a = 10;
    int add (int y)
    {
        int b = y;
        return (a+b);
    }
}

interface B{
    int x = 20;
    void display();
}

class C extends A implements B{
    int mul = a * x;
    int sum = add(20);
    public void display(){
        System.out.println("Sum = "+sum);
        System.out.println("Product = "+mul);
    }
}

class Main{
    public static void main (String[] args) {
        C obj = new C();
        obj.display();
    }
}

```

Output

```

Sum = 30
Product = 200

```

Inner Class

An inner class in Java is defined as a class that is declared inside another class. We use inner classes to logically group classes in one place to be more readable and maintainable. Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner Class:

```

class OuterClass
{
    .....
    class InnerClass
    {
        .....
    }
}

```

Since the inner class exists within the outer class, we must instantiate the outer class first, in order to instantiate the inner class.

Example

```

class CPU {
    double price;

    class Processor{
        double cores;
        String manufacturer;

        double getCache(){
            return 4.3;
        }
    }
}

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

        // create object of Outer class CPU
        CPU cpu = new CPU();

        // create an object of inner class Processor using outer class
        CPU.Processor processor = cpu.new Processor();

        System.out.println("Processor Cache = " + processor.getCache());
    }
}

```

Packages

A *package* is a collection of related classes and interfaces that provides access protection and name space management. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User Defined Packages (create your own packages)

1. Built-in Packages

Built-in packages are existing java packages that come along with the JDK (Java Development Kit). They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the examples of built-in packages are: *Java.lang*, *Java.io*, *Java.util*, *Java.applet*, *Java.awt*, *Java.net* etc.

To use a class or a package from the library, we need to use the ***import*** keyword:

Syntax:

```

import packageName.className;    // Import a single class
import packageName.*;            // Import the whole package

```

Example:

```

import java.util.Scanner;        // Import the class Scanner

```

In the example above, *java.util* is a package, while *Scanner* is a class of the *java.util* package.

This package also contains date and time facilities, random-number generator and other utility classes. To import a whole package, end the sentence with an asterisk sign (*).

```

import java.util.*;

```

2. User Defined Packages

The package which is defined by the user is called a User-defined package. It contains user-defined classes and interfaces. Java supports a keyword called “**package**” which is used to create user-defined packages in java programming. It has the following general form:

```
package packageName;
```

Example:

```
//AddOperation.java  
package myFirstPackage;           //user defined package  
public class AddOperation{  
    public double add(double x, double y){  
        return (x+y);  
    }  
}
```

Using package **myFirstPackage** in program which we define above:

```
//SumTwoNum.java  
import myFirstPackage.AddOperation //importing package  
class SumTwoNum{  
    public static void main (String[] args) {  
        AddOperation obj = new AddOperation();  
        System.out.println("Sum is = "+ obj.add(30, 20));  
    }  
}
```

Exception Handling

An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program. There are three categories of exceptions:

- **Checked Exceptions:** A checked exception is an exception that occurs at the compile time of program and forces programmers to deal with the exception otherwise program will not compile. Normally these are exceptions raised due to user error. They extend *java.lang.Exception* class. E.g. *IOException*, *SQLException*, *FileNotFoundException*, *ClassNotFoundException* etc.
- **Unchecked Exceptions:** An unchecked exception is an exception that occurs at runtime of a program. They are ignored at compile time. They are logical programming errors and extend the *java.lang.RuntimeException* class. E.g. *ArithmeticException*, *NullPointerException*, *ArrayIndexOutOfBoundsException* etc.
- **Errors:** Errors are the problems that arises beyond the control of the user or the programmer. They are also ignored at the time of compilation. E.g. *OutOfMemoryError*, *VirtualMachineError*, *AssertionError* etc.

Exception handling is a mechanism to handle runtime errors so that normal flow of the program can be maintained. It is important to handle exceptions because otherwise program would terminate abnormally whenever an exceptional condition occurs. Exception handling enables a program to deal with exceptional situations and continue its normal execution.

The steps in exception handling mechanism:

- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error information (Catch the exception)
- Take corrective actions (Handle the exception)

In Java, Exception Handling can be done by using five Java keywords: ***try***, ***catch***, ***throw***, ***throws*** and ***finally***.

try and catch keyword

A **try block** is used to enclose the code section which might throw an exception. Every try block is followed by a catch block.

The **catch block** is used to handle the exception. When an exception occurs, it is caught by the **catch block**. The catch block cannot be used without the **try block**.

Syntax:

```
try {
    // code
}
catch(Exception e) {
    // code
}
```

Example:

```

class Main {
    public static void main(String[] args) {
        try {
            // code that generate exception
            int divideByZero = 5 / 0;
            System.out.println("Rest of code in try block");
        }
        catch (ArithmeticException e) {
            System.out.println("ArithmeticException => " + e.getMessage());
        }
    }
}

```

finally keyword

The *finally* block follows a try block or a catch block. In Java, the *finally* block is always executed no matter whether there is an exception or not. The *finally* block is optional. And, for each *try* block, there can be only one *finally* block.

The basic syntax of *finally* block is:

```

try {
    //code
}
catch (ExceptionType1 e1) {
    // catch block
}
finally {
    //finally block always executes
}

```

Example:

```

class Main {
    public static void main(String[] args) {
        try {
            int divideByZero = 5 / 0;
        }
        catch (ArithmeticException e) {
            System.out.println("ArithmeticException => " + e.getMessage());
        }
        finally {
            System.out.println("This is the finally block");
        }
    }
}

```

Note: It is a good practice to use the *finally* block. It is because it can include important cleanup codes like,

- code that might be accidentally skipped by return, continue or break
- closing a file or connection

throw keyword

The *throw* keyword is used to throw an exception explicitly whenever an exceptional condition occurs. It is followed by only one single instance of the exception class we want to throw. The general form of throw is: `throw new exception_class("error message");`

Example:

```
public class ExceptionDemo {
    static void canVote(int age){
        try{
            if(age<18)
                throw new ArithmeticException("You are not an adult!");
            else
                System.out.println("You can vote!");
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
    public static void main (String[] args) {
        canVote(20);
        canVote(10);
    }
}
```

throws keyword

The *throws* keyword indicates what exception type may be thrown by a method. It is used to handle checked Exceptions as the compiler will not allow code to compile until they are handled. We use the *throws* keyword in the method declaration to declare the type of exceptions that might occur within it.

Syntax:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 ... {
    // method code
}
```

Example:

```
public class ExceptionDemo {
    static void func(int a) throws ArithmeticException{
        System.out.println(10/a);
    }
    public static void main (String[] args) {
        try{
            func(10);
            func(0);
        }catch(Exception e){
            System.out.println("can't divide by zero");
        }
    }
}
```

Multiple Catch Blocks

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Example:

```
class Main {
    public static void main(String[] args) {
        try {
            int array[] = new int[10];
            array[10] = 30 / 0;
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Creating Exception Class

Steps to create a custom exception:

- Create a new class whose name should end with *Exception* like *ClassNameException*. This is a convention to differentiate an exception class from regular ones.
- Make the class extends one of the exceptions which are subtypes of the *java.lang.Exception* class. Generally, a custom exception class always extends directly from the *Exception* class.
- Create a constructor with a *String* parameter which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

Example:

The following is a custom exception class which is created by following the above steps:

```
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str)
    {
        super(str);    // calling the constructor of parent Exception
    }
}
```

And the following example shows the way a custom exception is used is nothing different than built-in exception:

```
public class TestCustomException
{
    static void validate (int age) throws InvalidAgeException{
        if(age < 18){
            throw new InvalidAgeException("age is not valid to vote");
        }
    }
}
```

```

else {
    System.out.println("welcome to vote");
}
}

// main method
public static void main(String args[])
{
    try
    {
        // calling the method
        validate(15);
    }
    catch (InvalidAgeException ex)
    {
        System.out.println("Exception occurred: " + ex);
    }
}
}
}

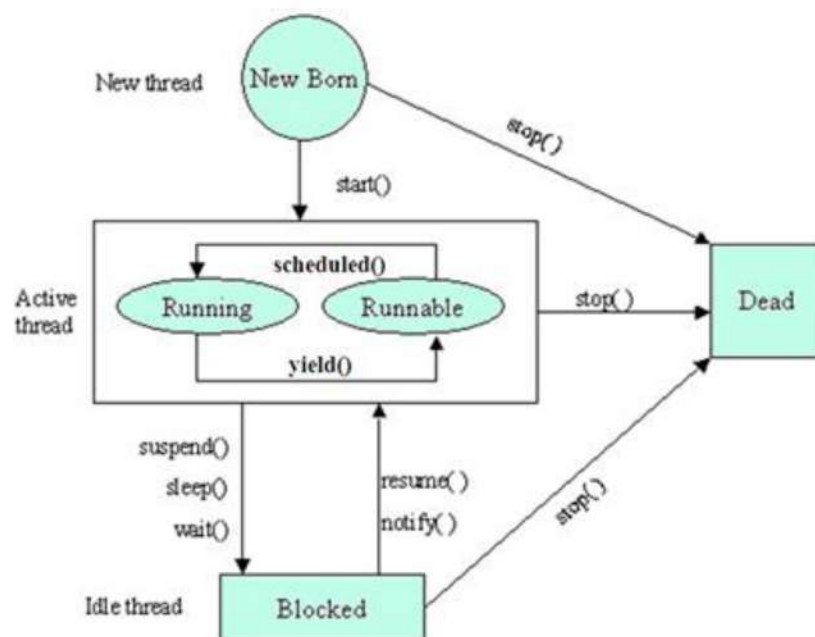
```

Multithreading

The process of executing multiple threads simultaneously is known as multithreading. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.

Life Cycle of a Thread (Thread States)

A thread goes through various stages in its life cycle. A thread can be in one of the five stages: **New**, **Runnable**, **Running**, **Blocked** and **Terminated/Dead**.



1. **Newborn State:** When a new thread is created, it is in the new state. It remains in this state until the program starts the thread. A thread is started by calling its *start()* method.
2. **Runnable State:** If the thread is ready for execution but waiting for the CPU the thread is said to be in runnable state. All the events that are waiting for the processor are queued up in the runnable state and are served in FIFO manner or priority scheduling. From this state the thread can go to running state if the processor is available using the *scheduled()* method.
3. **Running State:** If the thread is in execution then it is said to be in running state. The thread can finish its work and end normally. The thread can also be forced to give up the control when one of the following conditions arise:
 - A thread can be suspended by *suspend()* method. A suspended thread can be revived by using the *resume()* method.
 - A thread can be made to sleep for a particular time by using the *sleep(milliseconds)* method. The sleeping method re-enters runnable state when the time elapses.
 - A thread can be made to wait until a particular event occur using the *wait()* method, which can be run again using the *notify()* method.
4. **Blocked State:** A thread is said to be in blocked state if it is prevented from entering into the runnable state and so the running state. The thread enters the blocked state when it is suspended, made to sleep or wait. A blocked thread can enter into runnable state at any time and can resume execution.
5. **Dead State:** The running thread ends its life when it has completed executing the *run()* method which is called natural dead. The thread can also be killed at any stage by using the *stop()* method.

Writing Multithreaded Programs

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface

1. Thread creation by extending the Thread class

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

Example:

```
class MultithreadingDemo extends Thread{
    public void run(){
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[]){
        MultithreadingDemo obj=new MultithreadingDemo();
        obj.start();
    }
}
```


2. Thread creation by implementing the Runnable Interface

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

Example:

```
class MultithreadingDemo implements Runnable{
    public void run(){
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[]){
        MultithreadingDemo obj=new MultithreadingDemo();
        Thread tobj =new Thread(obj);
        tobj.start();
    }
}
```

Q. Write a program to create two threads. The first thread should print numbers from 1 to 10 at intervals of 0.5 second and the second thread should print numbers from 11 to 20 at the interval of 1 second.

Solution:

```
class First extends Thread
{
    public void run()
    {
        for (int i=1; i<=10; i++)
        {
            try
            {
                System.out.println(i);
                Thread.sleep(500);
            }
            catch (InterruptedException e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}
class Second extends Thread
{
    public void run()
    {
        for (int i=11; i<=20; i++)
        {
            try{
                System.out.println(i);
                Thread.sleep(1000);
            }
        }
    }
}
```

```

    }
    catch (InterruptedException e)
    {
        System.out.println(e.getMessage());
    }
}
}
}
public class ThreadInterval
{
    public static void main(String[] args)
    {
        Thread first = new First();
        Thread second = new Second();
        first.start();
        second.start();
    }
}

```

- **start()**: This method starts the execution of the thread by calling its *run()* method.
- **run()**: This method is used to do an action for a thread.
- **sleep()**: It sleeps a thread for a specified amount of time.
Thread.sleep() statement must be enclosed within try-catch block.

Thread Priorities

Thread Priorities determines how a thread should be treated with respect to others. Several threads executes concurrently. Every thread has some priority. Which thread will get a chance first to execute it is decided by thread scheduler based on thread priority. High priority threads can get more chances of execution. The valid range of thread priority is 1 to 10 (i.e. 1,2,3,4.....10.) and 1 is the min priority and 10 is the max priority. We can also represent thread priority in terms of constants. Basically, we have three types of constants like `MIN_PRIORITY`, `MAX_PRIORITY`, `NORM_PRIORITY`. By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).

To set thread's priority, we use *setPriority()* method, which is a member of Thread class. The general form is:

```
final void setPriority(int level)
```

To get the current priority of thread, we use *getPriority()* method. The general form is:

```
final int getPriority()
```

Q. Write a program to execute multiple threads in priority base.

Solution:

```
class First extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }
    }
}

class Second extends Thread
{
    public void run()
    {
        for (int i = 11; i <= 20; i++)
        {
            System.out.println(i);
        }
    }
}

class Third extends Thread
{
    public void run()
    {
        for (int i = 21; i <= 30; i++)
        {
            System.out.println(i);
        }
    }
}

public class ThreadPriority
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread t1 = new First();
        Thread t2 = new Second();
        Thread t3 = new Third();
        t1.setPriority(Thread.MAX_PRIORITY);
        t2.setPriority(Thread.MIN_PRIORITY);
        t3.setPriority(Thread.NORM_PRIORITY);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Synchronization

When two or more threads need to access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this achieved is called synchronization.

If we do not use synchronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results. Consider an example, suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file *temporary.txt* (*temporary.txt* is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be *locked*, and no other thread will be able to access or modify it until T1 returns.

We can add the *synchronized* keyword in the method declaration to make the method synchronized. The synchronized method is accessible for only one thread at a time.

Example:

```
class PrintTable{
    public synchronized void printTable(int n){
        System.out.println("Table of " + n);
        for(int i=1;i<=10;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(500);
            }catch(Exception e){
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread{
    PrintTable pt;
    MyThread1(PrintTable pt){
        this.pt=pt;
    }
    public void run(){
        pt.printTable(2);
    }
}

class MyThread2 extends Thread{
    PrintTable pt;
    MyThread2(PrintTable pt){
        this.pt=pt;
    }
    public void run(){
        pt.printTable(5);
    }
}
```

```

public class MultiThreadExample{
    public static void main(String args[]){
        //creating PrintTable object.
        PrintTable obj = new PrintTable();

        //creating threads.
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);

        //start threads.
        t1.start();
        t2.start();
    }
}

```

File Handling

File handling implies performing I/O on files. In Java, file I/O is performed through streams. Streams are the sequence of bits (data). There are two types of streams: *Input Streams* and *Output streams*

- **Input Streams:** Input streams are used to read data from various input devices like keyboard, file, network etc.
- **Output Streams:** Output streams are used to write the data to various output devices like monitor, file, network etc.

Streams Based on Data

There are two types of streams based on data:

- **Byte Stream:** Java Byte streams are used to perform input and output of 8-bit bytes. We do this with the help of different Byte stream classes. Two most commonly used Byte stream classes are *FileInputStream* and *FileOutputStream*.
- **Character Stream:** Java Character streams are used to perform input and output of characters. For input and output of characters, we have Character stream classes. Two most commonly used Character stream classes are *FileReader* and *FileWriter*.

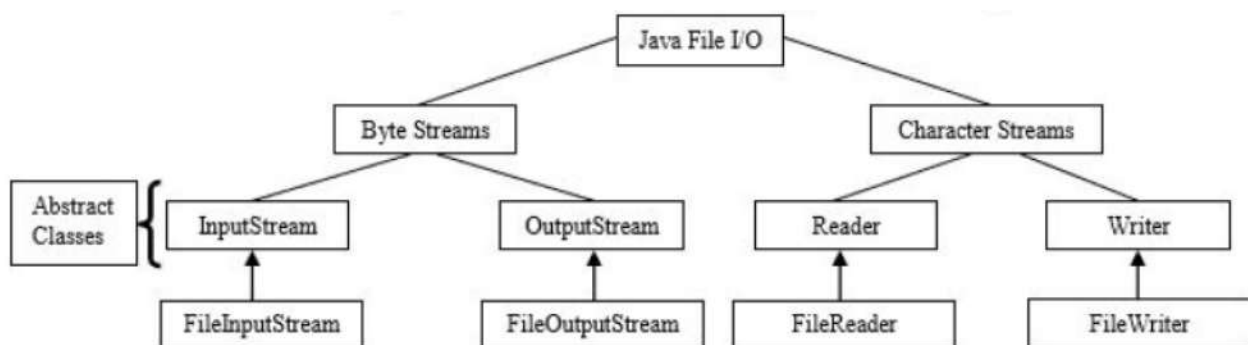


Fig: Classes used for file handling in Java

Methods for reading bytes:

<code>int read()</code>	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
<code>int read(byte buffer[])</code>	Attempts to read up to <code>buffer.length</code> bytes into <code>buffer</code> and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
<code>int read(byte buffer[], int offset, int numBytes)</code>	Attempts to read up to <code>numBytes</code> bytes into <code>buffer</code> starting at <code>buffer[offset]</code> , returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.

Methods for writing bytes:

<code>void write(int b)</code>	Writes a single byte to an output stream. Note that the parameter is an int , which allows you to call <code>write()</code> with an expression without having to cast it back to byte .
<code>void write(byte buffer[])</code>	Writes a complete array of bytes to an output stream.
<code>void write(byte buffer[], int offset, int numBytes)</code>	Writes a subrange of <code>numBytes</code> bytes from the array <code>buffer</code> , beginning at <code>buffer[offset]</code> .

FileInputStream

The `FileInputStream` class creates an `InputStream` that we can use to read bytes from a file. Its two most common constructors are shown here:

```
FileInputStream(String filepath)
FileInputStream(File fileObj)
```

They can throw a `FileNotFoundException`. Here, **filepath** is the full path name of a file, and **fileObj** is a `File` object that describes the file.

E.g.

```
FileInputStream f0 = new FileInputStream("D:\abc.txt")
File f = new File("D:\abc.txt");
FileInputStream f1 = new FileInputStream(f);
```

Example:

```
import java.io.*;
class ReadHello{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("hello.txt");
            int i=0;
            while((i=fin.read())!=-1){
                System.out.println((char)i);
            }
            fin.close();
        } catch(Exception e){
            system.out.println(e);}
    }
}
```

FileOutputStream

FileOutputStream creates an *OutputStream* that we can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(*String filePath*), *FileOutputStream*(*File fileObj*),
FileOutputStream(*String filePath*, *boolean append*)

Here, *filePath* is the full path name of a file, and *fileObj* is a *File* object that describes the file. If *append* is true, the file is opened in append mode.

Example:

```
import java.io.*;
public class WriteHello{
    public static void main(String args[]){
        try{
            FileOutputStream fo=new FileOutputStream("hello.txt");
            String i= "Hello World!!!";
            byte b[]=i.getBytes();           //converting string into byte array
            fo.write(i);
            fo.close();
        }
        catch(Exception e){
            system.out.println(e);
        }
    }
}
```

FileReader

The *FileReader* class creates a *Reader* that we can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(*String filePath*), *FileReader*(*File fileObj*)

Either can throw a *FileNotFoundException*. Here, *filePath* is the full path name of a file, and *fileObj* is a *File* object that describes the file.

Example:

```
import java.io.FileReader;
import java.io.IOException;
public class FileReaderDemo {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("test.txt");
            int c;
            while ((c = fr.read()) != -1) {
                System.out.print((char) c);
            }
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

FileWriter

FileWriter creates a Writer that we can use to write to a file. Its most commonly used constructors are shown here:

```
FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
FileWriter(File fileObj, boolean append)
```

They can throw an IOException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, then output is appended to the end of the file.

Example:

```
import java.io.FileWriter;
import java.io.IOException;
class WriteTest
{
    public static void main(String args[]){
        {
            try
            {
                FileWriter fw = new FileWriter("D:\\myfile.txt");
                String str="Write this string to my file";
                fw.write(str);
                fw.close();
            }
            catch (IOException e){
                e.printStackTrace();
            }
        }
    }
}
```

Random Access File

In java, the *java.io* package has a built-in class *RandomAccessFile* that enables a file to be accessed randomly. The *RandomAccessFile* class has several methods used to move the cursor position in a file.

- Java allows a program to perform *random file access*.
- In random file access, a program may immediately jump to any location in the file.
- To create and work with random access files in Java, you use the *RandomAccessFile* class.

```
RandomAccessFile(String filename, String mode)
```

- *filename*: the name of the file.
- *mode*: a string indicating the mode in which you wish to use the file.
 - "r" = reading
 - "rw" = for reading and writing.

- The *RandomAccessFile* class lets you move the file pointer.
- This allows data to be read and written at any byte location in the file.
- The seek method is used to move the file pointer.


```
rndFile.seek(long position);
```
- The argument is the number of the byte that you want to move the file pointer to.

Example:

```

import java.io.*;
public class RandomAccessFileDemo {
    public static void main(String [] args) {
        int a=123;
        long b=435525;
        String s="Here is some text";
        try{
            RandomAccessFile raf=RandomAccessFile("Demo.txt","rw");
            raf.writeInt(a);           //writing integer value in file
            raf.writeLong(b);         //writing long value in the file
            raf.writeUTF(s)           //writing string in the file
            raf.seek(0);              //set pointer beginning of the file
            System.out.println(raf.read());
            raf.close();
        }
        catch(IOException e){
            e.printStackTrace();
        }
    }
}

```

Q. Write a simple Java program to read from and write to files.**Solution:**

Let us suppose we have a file named "test.txt" in D-drive. Now we first read from this file character-by-character and later we write the contents of this file to another file say "testwrite.txt" in E-drive. For these tasks, we use the character stream classes namely *FileReader* and *FileWriter*. The code is given below:

```

import java.io.*;
public class FileReadWrite {
    public static void main(String [] args) {
        try{
            FileReader fr = new FileReader("D:\\test.txt");
            FileWriter fw = new FileWriter("E:\\testwrite.txt");
            int c;
            while((c=fr.read())!=-1) {
                fw.write(c);
                System.out.print((char)c);
            }
            fr.close();
            fw.close();
        }
        catch (IOException ex){
            ex.printStackTrace();
        }
    }
}

```

Q. Write a program to duplicate each character in a text file and write the output in a separate file using character stream.

e.g. source.txt: csit, destination.txt: ccssiitt

Solution:

```
import java.io.FileReader;
import java.io.FileWriter;

public class CharacterStreamDemo
{
    public static void main(String[] args) throws Exception
    {
        FileReader in = new FileReader("source.txt");
        FileWriter out = new FileWriter("destination.txt");
        int charData;
        while(true)
        {
            charData = in.read();
            if (charData == -1)
                break;
            else
            {
                out.write(charData);
                out.write(charData);
            }
        }
        in.close();
        out.close();
    }
}
```

Q. Write a program to read an input string from the user and write the vowels of that string in VOWEL.TXT and consonants in CONSONANT.TXT.

Solution:

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String input = sc.nextLine();

        FileWriter vowels = new FileWriter("VOWEL.TXT");
        FileWriter consonants = new FileWriter("CONSONANT.TXT");

        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);
```

```

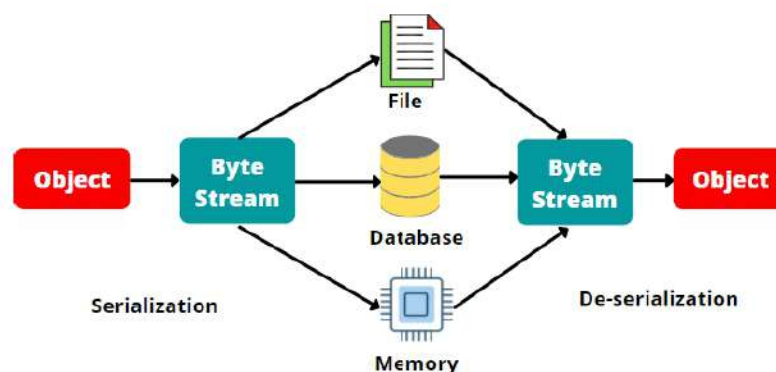
    if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
        c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
        vowels.write(c + "\n");
    } else {
        consonants.write(c + "\n");
    }
}
vowels.close();
consonants.close();
sc.close();
}
}

```

Serialization

Serialization can be defined as a process by which we convert the object state into its equivalent byte stream to store the object into the memory in a file or persist the object.

When we want to retrieve the object from its saved state and access its contents, we will have to convert the byte stream back to the actual Java object and this process is called **deserialization**.



The class whose objects are serialized must implement *java.io.Serializable* interface. Then the object of this class (implementing *Serializable* interface) will use *writeObject ()* and *readObject ()* methods respectively for serializing and deserializing the class object.

- When we serialize an object in Java we use *ObjectOutputStream*'s *writeObject* method to write the object to a file.
- For deserializing the object in Java we use the *ObjectInputStream*'s *readObject ()* method to read the contents of the file and read them into an object.

Steps to store (write) objects in a file

1. First, connect *objfile.txt* file to *FileOutputStream*. It will write data into *objfile.txt* file.
`FileOutputStream fos = new FileOutputStream("objfile.txt");`
2. Then, connect *FileOutputStream* to *ObjectOutputStream* by code below:
`ObjectOutputStream oos = new ObjectOutputStream(fos);`
3. Now, call *writeObject()* method of *ObjectOutputStream* to write objects to *FileOutputStream*, which stores them into *objfile.txt* file.

Steps to read objects from file

1. Connect *objfile.txt* file to *FileInputStream*. It will read objects from *objfile.txt* file.
`FileInputStream fis = new FileInputStream("objfile.txt");`
2. Connect *FileInputStream* to *ObjectInputStream* to retrieve objects from *FileInputStream*.
`ObjectInputStream ois = new ObjectInputStream(fis);`
3. Now, call *readObject()* method of *ObjectInputStream* class to read objects by code below:
`Employee e = (Employee) ois.readObject(); // Here, Employee is class that implements Serializable interface.`

Example

Q. Write a java program that writes objects of Employee class in the file named emp.doc. Create Employee class as of your interest.

Solution:

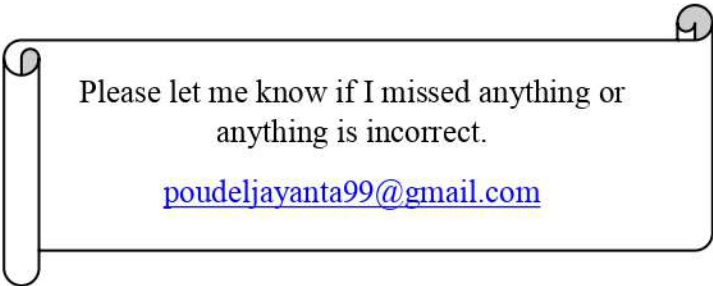
```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Employee implements Serializable {
    String name;
    int age;
    String department;

    public Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
}

class EmployeeSerialize{
    public static void main(String[] args) {
        Employee emp1 = new Employee("Jayanta", 20, "IT");
        Employee emp2 = new Employee("Manoj", 25, "HR");

        try {
            FileOutputStream fileOut = new FileOutputStream("emp.doc");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(emp1);
            out.writeObject(emp2);
            out.close();
            fileOut.close();
            System.out.println("Employee objects written to emp.doc");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Please let me know if I missed anything or
anything is incorrect.

poudeljayanta99@gmail.com