Flavours of Java:

- J2SE (Standard Edition)
- J2EE (Enterprise Edition)
- J2ME (Micro Edition)

Java Program Syntax

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

class -> Reserved keyword(we can't change like Class, cLaSS…etc)

FirstJavaProgram -> Class Name (we can named anything but meaningful like FirstProgram, MyFirstProgram, HelloWorldProgram … etc)

{ ….. } -> Represents block (It contains n number of statements)

public static void main(String[] args){..} -> Main method of java program. It is the starting point of your program.

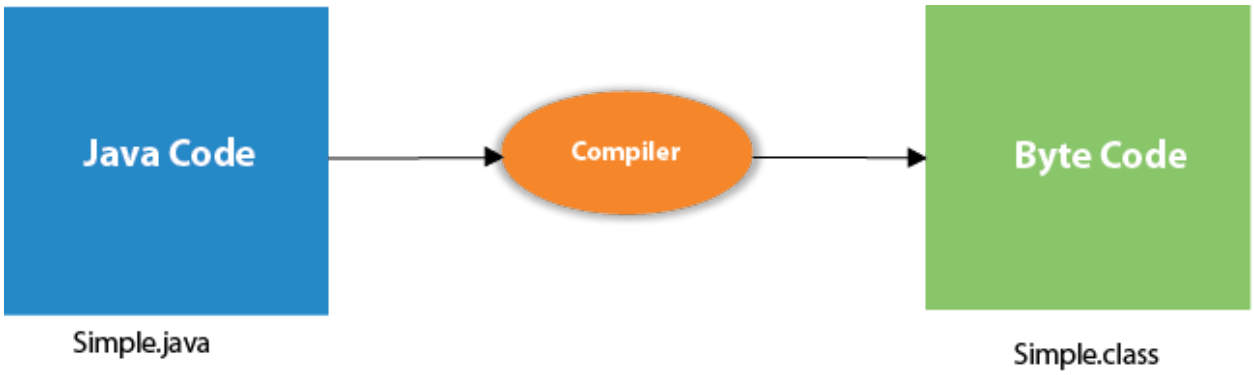
System.out.println(“hello”) -> It is a method used to print messages.

"Hello" -> This text will display on command prompt. Text must be within double quotes (**" "**)

Compilation & Execution

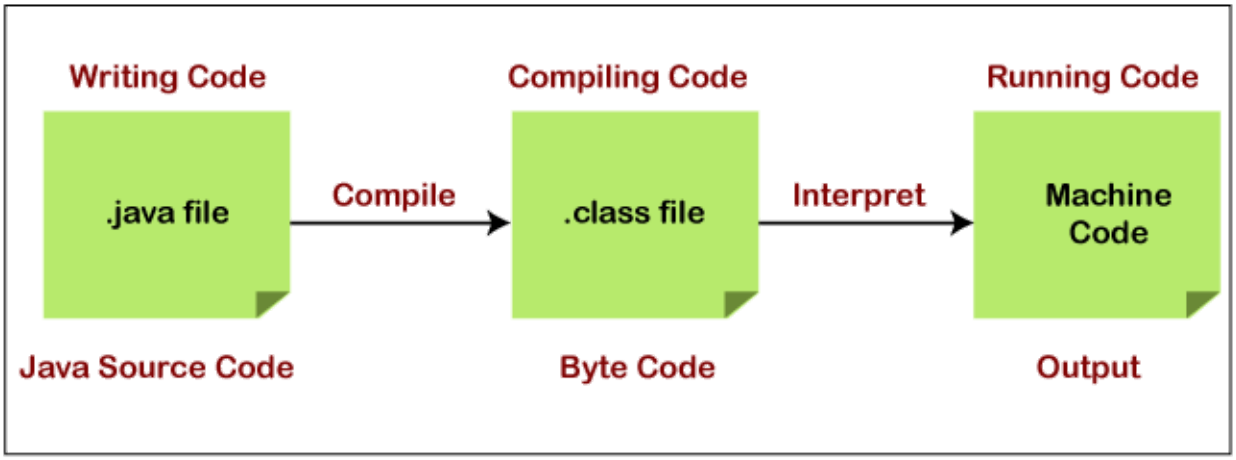
Compilation

In Java, programs are not compiled into executable files; they are compiled into bytecode. Java source code is compiled into bytecode when we use the javac compiler.



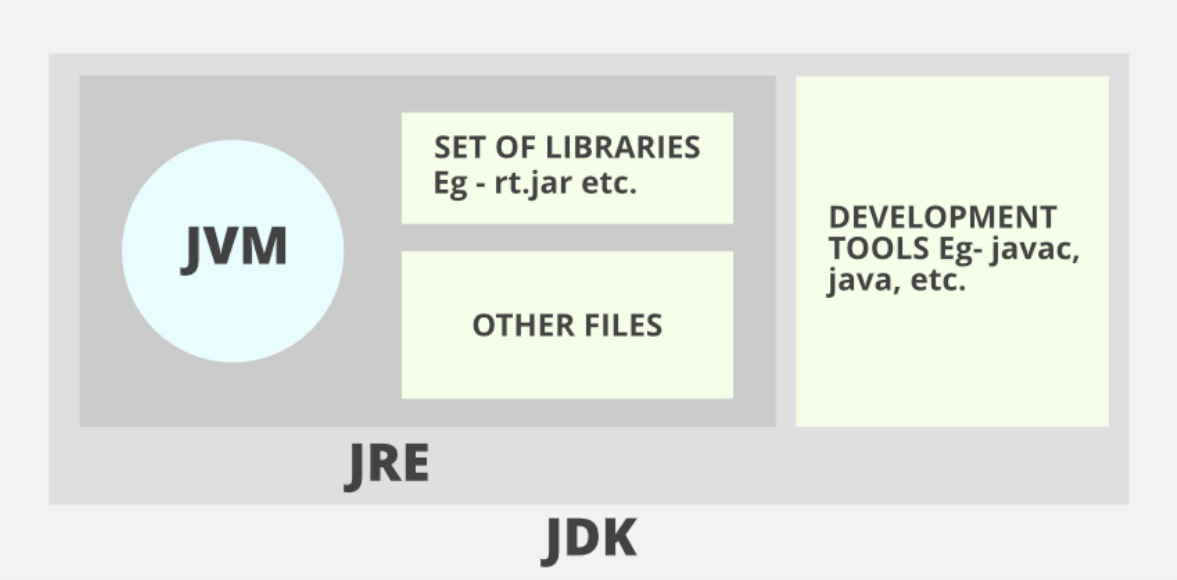
Execution

Interpreter in Java is a computer program that converts high-level program statements into Assembly Level Language.



JDK

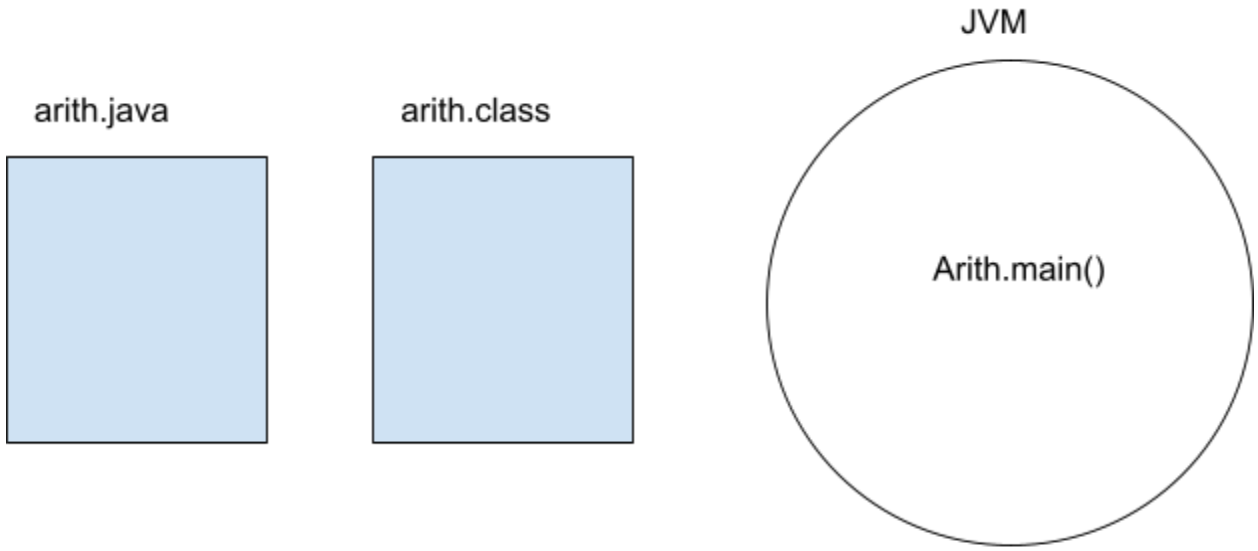
Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java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



JRE (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

JVM (**Java Virtual Machine)** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an **i*nterpreter*.**

JVM



High level language→ Java,

Low level language → Machine code(microprocessor 8085)

Features of Java

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

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 said to be a pure object-oriented programming language. In java, everything is an object. It supports all the features of the object-oriented programming paradigm. The primitive data types java also implemented as objects using wrapper classes, but still, it allows primitive data types to archive high-performance.

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 (or) Platform Independent

Java has invented to archive "write once; run anywhere, any time, forever". The java provides JVM (Java Virtual Machine) to to archive architectural-neutral or platform-independent. The JVM allows the java program created using one

operating system can be executed on any other operating system.

Multi-threaded

Java supports multi-threading programming, which allows us to write programs that do multiple operations simultaneously.

Interpreted

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. The byte code is interpreted to any machine code so that it runs on the native machine.

High performance

Java provides high performance with the help of features like JVM, interpretation, and its simplicity.

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 is said to be dynamic because the java byte code may be dynamically updated on a running system and it has a dynamic memory allocation and deallocation (objects and garbage collector).

Lexical tokens

A token is the smallest element of a program that is meaningful to the compiler.

Keywords:

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

```
Abstract continue for new switch assert default
goto package synchronized boolean do if private
this break double implements protected
throw byte else import public Throws
case enum instanceof return transient
catch extends int short try char final
interface static void class finally long
strictfp volatile const float Native super while
```

Identifiers

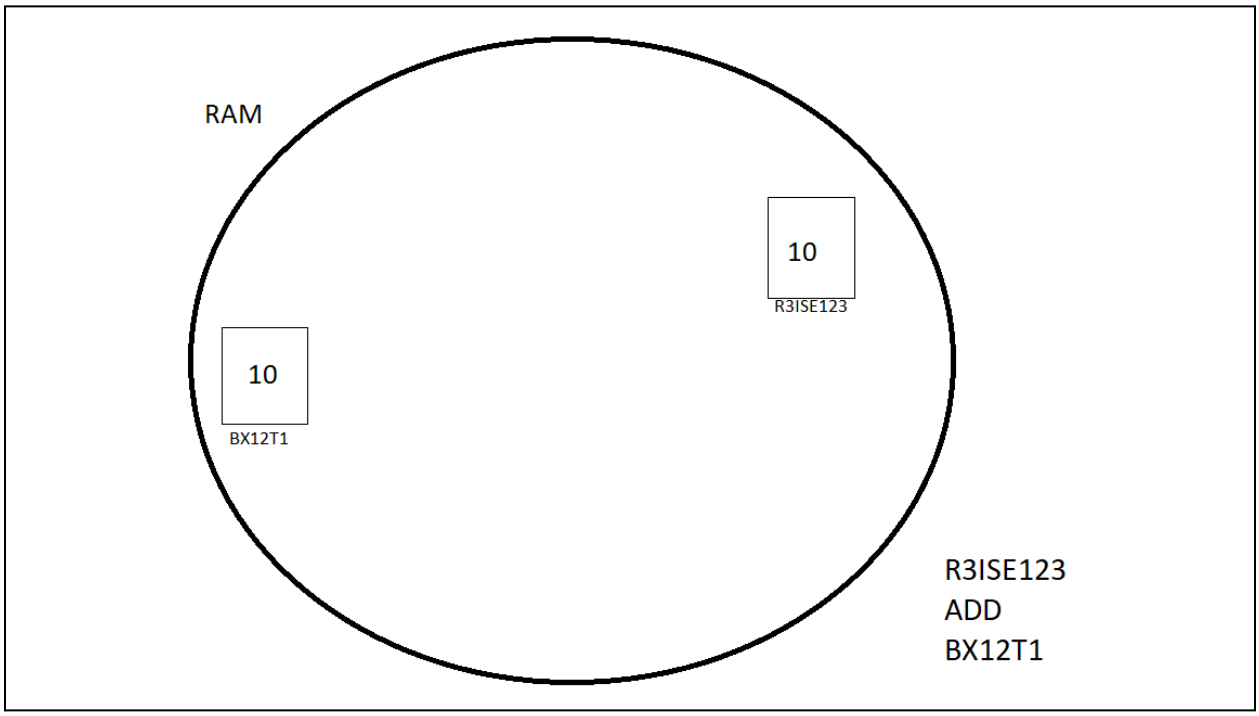
Identifiers are used as the general terminology for naming of variables, functions and arrays…etc

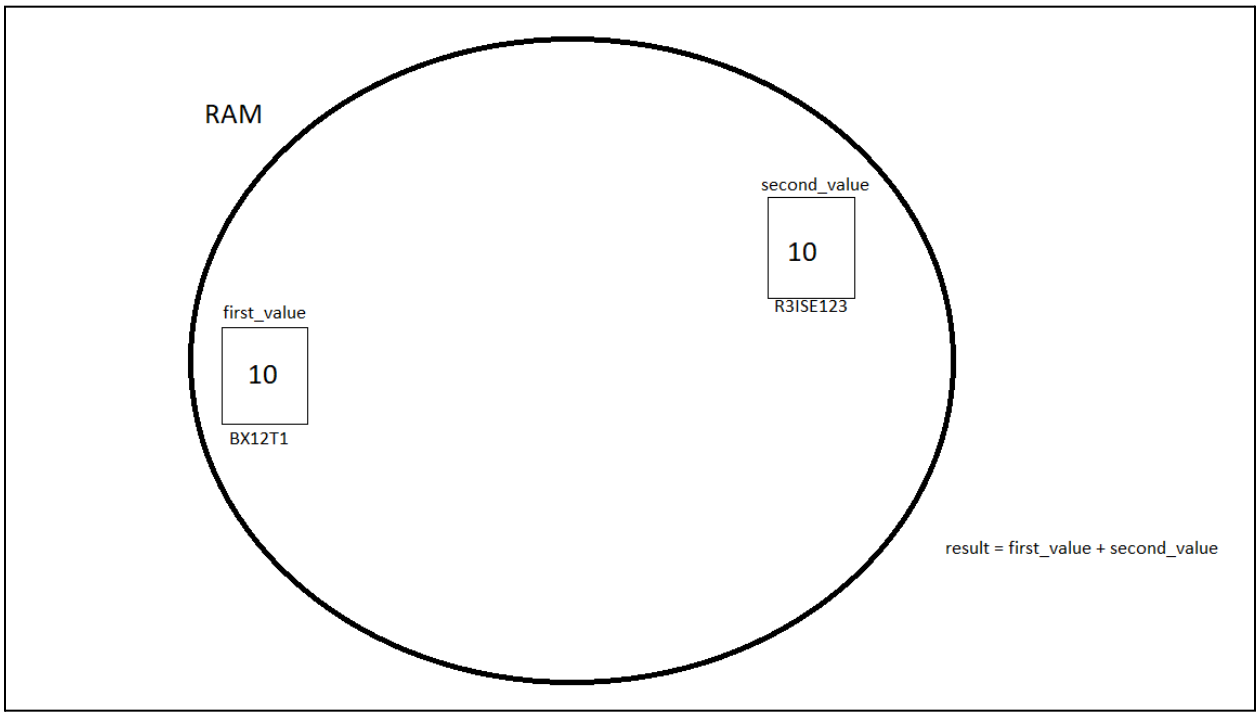
Rules for defining Java Identifiers:

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '**_**' (underscore).For example "value@" is not a valid java identifier as it contain '@' special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123value" is a not a valid java identifier.
- **Reserved Words** can't be used as an identifier. For example "int class= 20;" is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Variable:

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location.





Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

Data types

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Primitive data types:

boolean - True or False

char - ('\u0000' (or 0) to '\uffff')

byte - (-128 to 127)

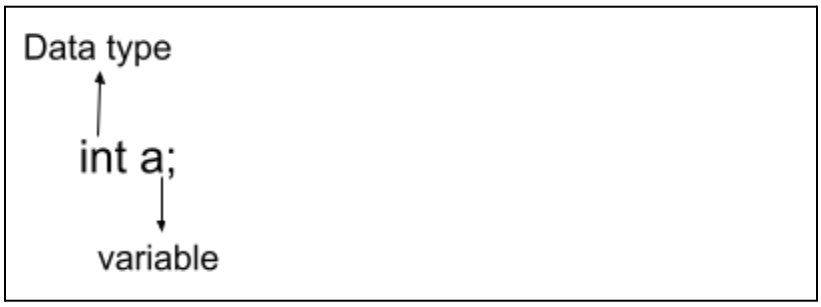
short - (-32,768 to 32,767)

int - (- 2,147,483,648 to 2,147,483,647)

long - (- 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

float - 32-bit IEEE 754 floating point

double - 64-bit IEEE 754 floating point



Variable Declaration & Initialization

```
int a; //varible declaration

int a1 = 10; //variable declaration with initialization

float f; //varible declaration

float f1 = 3.5f; //variable declaration with initialization
```

Literal: Any constant value which can be assigned to the variable is called literal/constant. In simple words, Literals in Java is a synthetic representation of boolean, numeric, character, or string data.

```
// Here 100 is a constant/literal.
int x = 100;
```

Comments: The Java comments are the statements in a program that are not executed by the compiler and interpreter.

Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements. Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there). Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Type casting:

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

int x=10;
 	
float y=x; // implicit conversion
 	
System.out.println(y);
	int x=10;
	
	float y=(float)x; // explicit conversion
	
	System.out.println(y);

Operators

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

1. **Arithmetic Operators**

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

2. **Assignment Operators**

Operator	Example	Equivalent to
`=`	`a = b;`	`a = b;`
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a - b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

3. **Relational Operators**

Operator	Description	Example
`==`	Is Equal To	`3 == 5` returns **false**
`!=`	Not Equal To	`3 != 5` returns **true**
`>`	Greater Than	`3 > 5` returns **false**
`<`	Less Than	`3 < 5` returns **true**
`>=`	Greater Than or Equal To	`3 >= 5` returns **false**
`<=`	Less Than or Equal To	`3 <= 5` returns **true**

4. Logical Operators

Operator	Example	Meaning
`&&` (Logical AND)	expression1 **&&** expression2	`true` only if both `expression1` and `expression2` are `true`
`||` (Logical OR)	expression1 **||** expression2	`true` if either `expression1` or `expression2` is `true`
`!` (Logical NOT)	**!**expression	`true` if `expression` is `false` and vice versa

5. Unary Operators

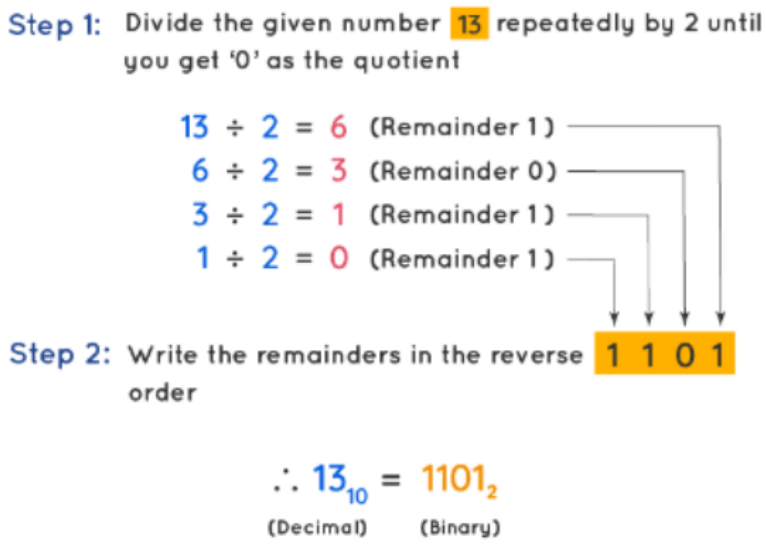
Operator	Meaning
`+`	**Unary plus**: not necessary to use since numbers are positive without using it
`-`	**Unary minus**: inverts the sign of an expression
`++`	**Increment operator**: increments value by 1
`--`	**Decrement operator**: decrements value by 1
`!`	**Logical complement operator**: inverts the value of a boolean

```
int i = 3;
int a = i++; // a = 3, i = 4
int b = ++a; // b = 4, a = 4
```

6. Bitwise Operators

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

Decimal to Binary



Signed Left Shift Operator

int a = 8;

int b = a << 2;

a = 1 0 0 0

b = 1 0 0 0 0 0   (decimal -> 32)

<< Left Shift Operator appends two more zero. Increasing element counts.

Signed Right Shift Operator

int a = 8;

int b = a >> 2;

a = 1 0 0 0

b = 1 0 (decimal -> 2)

>> Right Shift Operator skip two bits. Increasing element counts.

Unsigned Right Shift Operator

int a = 240;

int b = a >>> 2;

a = 1 1 1 1 0 0 0 0

b = 0 0 1 1 1 1 0 0 (decimal -> 60)

>>> Right Shift Operator moved the right side number of bits. Shifted values are filled with 0. Excess bits are discarded

Looping and Control Statements (if - else)

if(condition)
{
//code if condition is true
}
else
{
//code if condition is false
}
	if(condition1){
	//code to be executed if condition1 is true
	}**else if**(condition2){
	//code to be executed if condition2 is true
	}
	else if(condition3){
	//code to be executed if condition3 is true
	}
	...
	else{
	//code to be executed if all the conditions are false
	}

for loop

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

while loop

```
while (condition){
//code to be executed
Increment / decrement statement
}
```

do while loop

```
do{
//code to be executed / loop body
}while (condition);
```

break, continue

```
jump-statement;
break;
```

```
jump-statement;
continue;
```

switch case

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
.....
default:
 code to be executed if all cases are not matched;
}
```

Ternary Operator(? :)

variable = (condition) ? expression1 : expression2

Methods

A method is a block of code or collection of statements to perform a certain task or operation.

It is used to achieve the **reusability of code**. We write a method once and use it many times.

It also provides the easy modification and readability of code.

The method is executed only when we **call** or invoke it.

```
<access_specifier>  <return_type>  method_name(parameters)

{

// method definition

}
```

---

1. Java Program to Swap Two Numbers
2. Java Program to Check Whether a Number is Even or Odd
3. Java Program to Check Whether a Number is Perfect or Not Perfect
4. Java Program to Calculate the Sum of Natural Numbers
5. Java Program to Generate Multiplication Table
6. Java Program to Reverse a Number
7. Java Program to Display Prime Numbers Between Two Intervals
8. Java Program to Display ODD numbers Between Two Intervals

9. Java Program to Display EVEN numbers Between Two Intervals

10. Java Program to Find Factorial of a Number Using Recursion

Object Oriented Programming(OOP)

Class

A class is a group of objects which have common properties. It is a **template or blueprint** from which objects are created. It is a logical entity. Class does not occupy memory. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Interface

Class Declaration

```
class <class name>
{
        // Code
}
```

Object

- An object is an instance of a class.
- An object is a real-world entity.
- The object is an entity which has state and behavior.

Object Creation:

<Class_name> object = new <Class_name>();

Encapsulation


Capsule

Encapsulation refers to the **bundling of fields and methods inside a single class.**

It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve **data hiding.**

In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.

```
class Employee
{
   private int empId;
   private String empName;
   private int empSalary;

   public int getEmpId(){
      return empId;
   }

   public void setEmpId(int Id)
   {
      this.empId = Id;
   }

   public String getEmpName()
   {
      return empName;
   }
   public void setEmpName(String name){
      this.empName = name;
   }

    public int getEmpSalary()
   {
      return empSalary;
   }
   public void setEmpSalary(int salary){
      this.empSalary = salary;
```

```
    }


}


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

        Employee obj = new Employee();

        obj.setEmpId(1000);
        obj.setEmpName("Arun");
        obj.setEmpSalary(30000);

        System.out.println("name:" + obj.getEmpId());
        System.out.println("Salary:" + obj.getEmpSalary());

    }
}
```

Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.

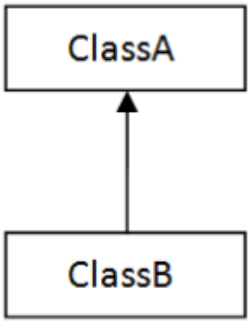
The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.When you inherit from an existing class, you can reuse methods and fields of the parent class.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

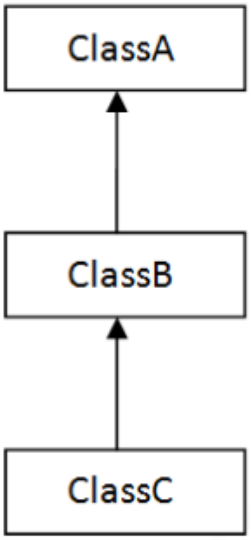
- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Multiple Inheritance

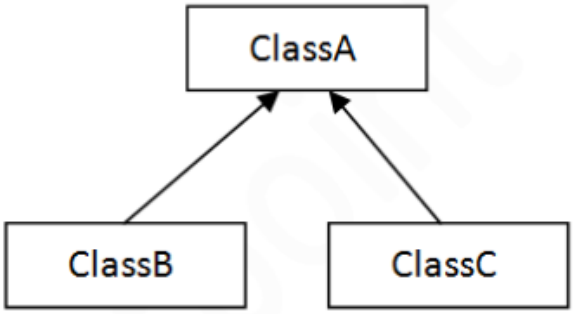
Hybrid Inheritance



1) Single

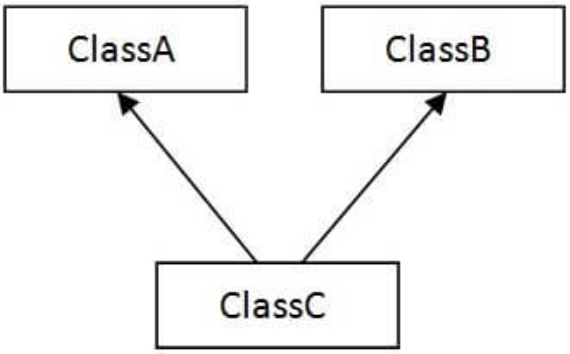


2) Multilevel

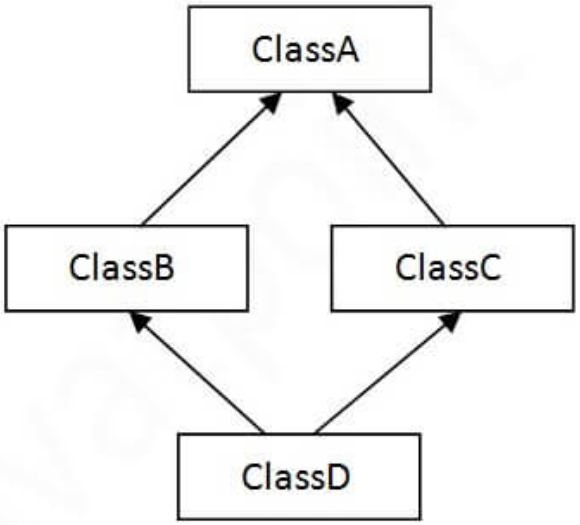


3) Hierarchical

NOTE: Object creation for least child class



4) Multiple



5) Hybrid

Parent class
Base class - - > Base class
Super class

Derived class
Child class - - > Child class
Sub class

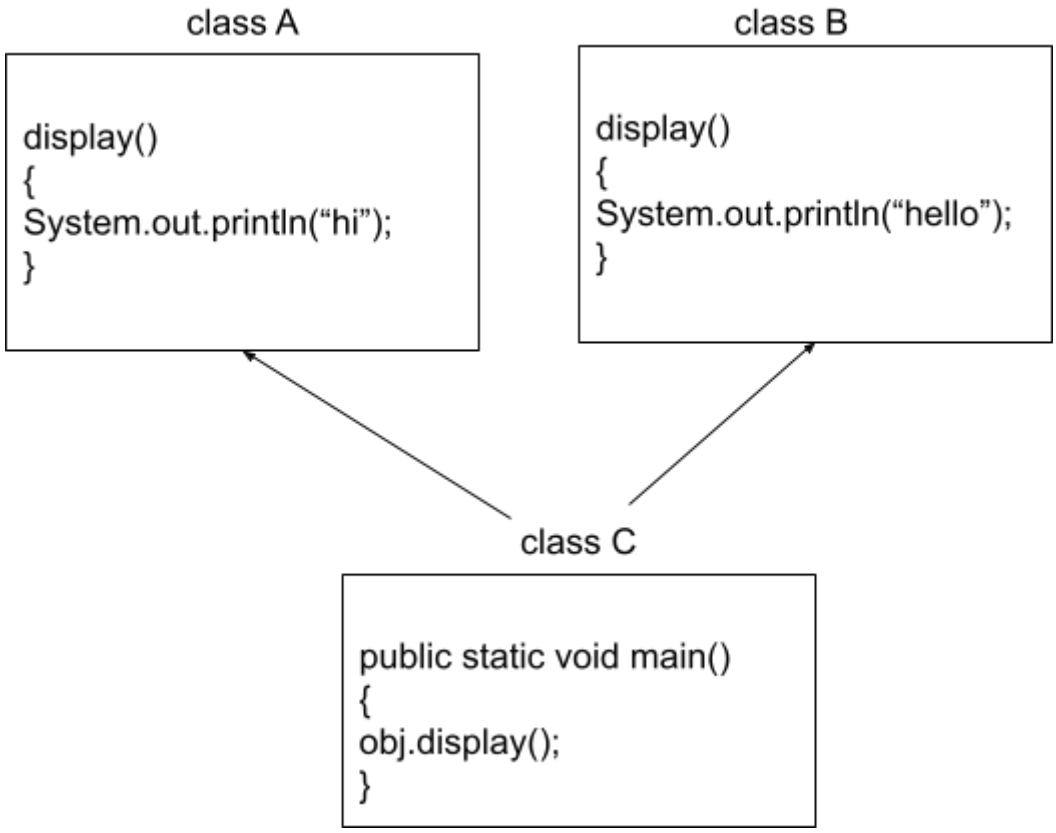
```
import java.util.*;
class StudentDetails
{
int stuId;
String name;
Scanner scan=new Scanner(System.in);

void getDetails(){
System.out.println("------------------- Student Mark Calculation System -------------------");
System.out.println("Type Student ID");
stuId=scan.nextInt();

System.out.println("Type Student Name");
name=scan.nextLine();
}

}

class StudentMarks extends StudentDetails{
int m1, m2, m3, total;

void calculateResult(){
System.out.println("Type Mark1");
m1=scan.nextInt();
System.out.println("Type Mark2");
m2=scan.nextInt();
System.out.println("Type Mark3");
m3=scan.nextInt();
total=m1+m2+m3;
System.out.println("********* MarkReport *********");
System.out.println("Id:"+stuId+"\nName:"+name+"\nTotal:"+total+"\n**************************");
}

}

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

StudentMarks obj = new StudentMarks();
obj.getDetails();
obj.calculateResult();

}}
```

Why multiple inheritance is not supported in Java?

The reason behind this is to prevent ambiguity.

Consider a case where class C extends class A and Class B and both class A and B have the same method display().

Now the Java compiler cannot decide which display method it should inherit. To prevent such a situation, multiple inheritance is not allowed in java.



Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*.

Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

- Compile-time polymorphism
- Runtime polymorphism.

Compile Time polymorphism is implemented through **Method overloading**.
Run time polymorphism is implemented through **Method overriding.**

Compile-time Polymorphism

Method Overloading occurs when a class has many methods with the same name but different parameters. Two or more methods may have the same name if they have other **numbers of parameters, different data types, or different numbers of parameters and different data types.**

```
public class MethodOverloading {

	void show(int num1)
	{
		System.out.println("number 1 : " + num1);
	}

	void show(int num1, int num2)
	{
		System.out.println("number 1 : " + num1
					+ " number 2 : " + num2);
	}

	public static void main(String[] args)
	{
		MethodOverloading obj = new MethodOverloading();
		obj.show(3);
		obj.show(4, 5);
	}
}
```

Runtime Polymorphism

In this process, an **overridden method is called through the reference variable of a superclass**. The determination of the method to be called is based on the object being referred to by the reference variable.

```
class Bike{
  void run(){
     System.out.println("running");
  }
}
class Splendor extends Bike{
  void run(){
      System.out.println("running safely with 60km");
}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Abstraction

Abstraction is a process of **hiding the implementation details** and showing only functionality to the user.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class

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.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.

```
abstract class Bike{

  abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

 Bike obj = new Honda4();

 obj.run();
```

}

}

Interface

An interface in Java is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to **achieve abstraction***. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

```
interface <interface_name>{
        // declare constant fields
        // declare methods that abstract
}
class class_name implements interface_name{


}
```

```
interface Drawable{
void draw();
}

class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
```

```
}


class InterfaceExample{
public static void main(String args[]){
Drawable d=new Circle();
d.draw();
}}
```

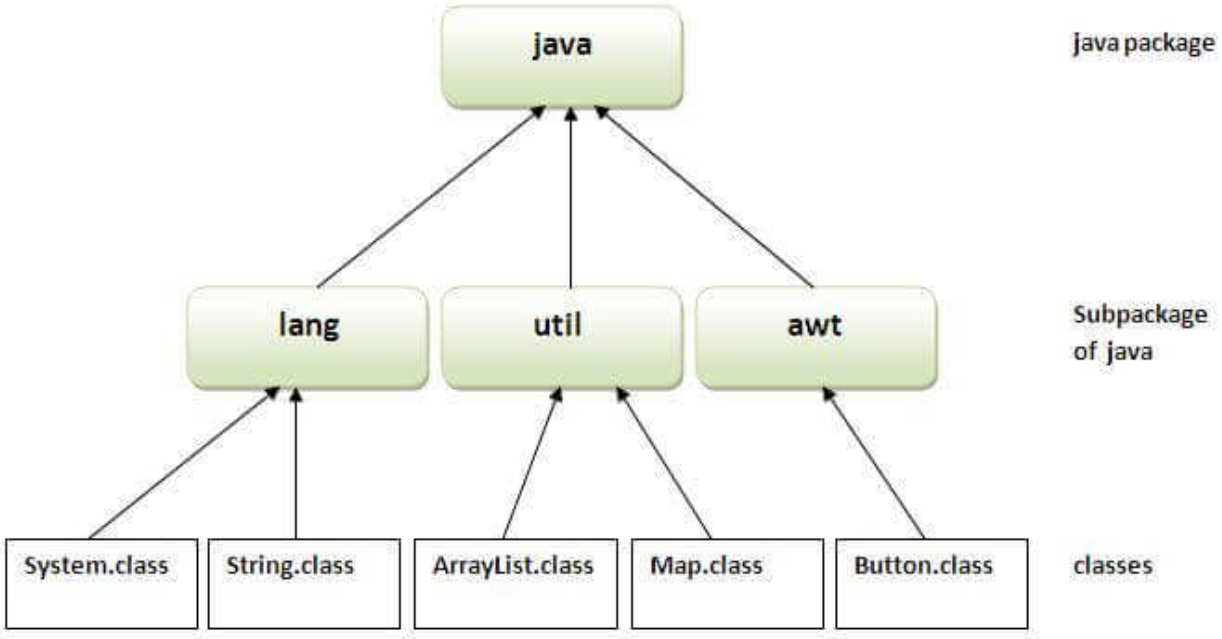
Packages

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



compile java package

javac -d directory javafilename

1. We can include interface into Package
2. We can include inheritance into Package

```
package calculator;
public class Arith{
public int addition(int a, int b){
return a+b;
}
public int subtraction(int a, int b){
return a-b;
}
public int multiplication(int a, int b){
return a*b;
```

```
}

public int division(int a, int b){
return a/b;
}
public static void display()
{
System.out.println(“this is static method”);
}
}
```

—----------

```
import calculator.Arith;
class PackageAccess{
public static void main(String[] args){
int a=10, b=20;
Arith obj = new Arith();
System.out.println(obj.addition(a,b));
System.out.println(obj.subtraction(a,b));
System.out.println(obj.multiplication(a,b));
System.out.println(obj.division(a,b));
}
}
```

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.

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

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

There are two types of constructors in Java:

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

Default constructor

```
class Bike{
Bike(){
    System.out.println("Bike is created");
}
public static void main(String args[]){
Bike b=new Bike();
}
}
```

Parameterized constructor

```
class Student{
    int id;
    String name;
    Student(int i,String n){
    this.id = i;
    this.name = n;
    }
    void display(){
```

```
    System.out.println(id+" "+name);
}

    public static void main(String args[]){

    Student s1 = new Student(101,"Arjun");
    Student s2 = new Student(102,"Bala");

    s1.display();
    s2.display();
   }
}
```

Aggregation(HAS-A relationship)

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

```
class Employee{

int id;

String name;

Address address;//Address is a class

...

}
```

Address.java

```
public class Address {

String city,state,country;

public Address(String city, String state, String country) {

    this.city = city;

    this.state = state;

    this.country = country;

}

}
```

Emp.java

```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
    this.id = id;
    this.name = name;
    this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("abc","TN","india");
Address address2=new Address("xyz","Ban","india");

Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}
}
```

Array

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a 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, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

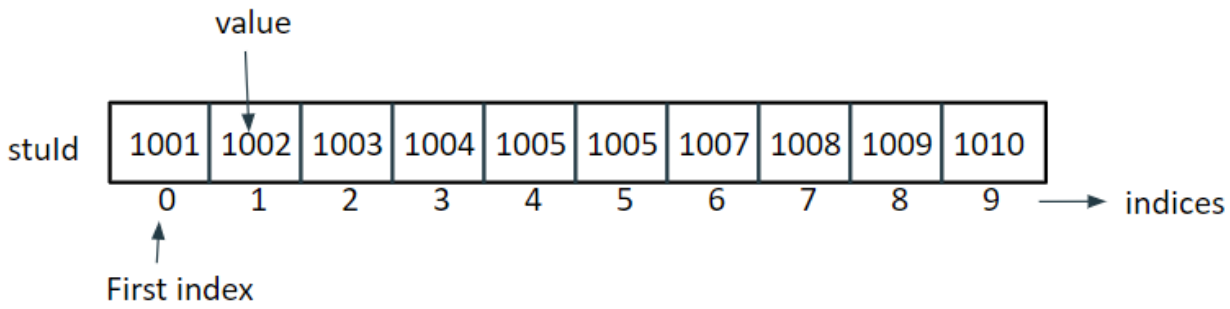
There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

Array Declaration [ ]

1. dataType[] stuId;
2. dataType []stuId;
3. dataType stuId[];
4. stuId = new datatype[size];




```
int stuId[] =  new int[10];

stuId[0] = 1001;

stuId[1] = 1002;

stuId[2] = 1003;

stuId[3] = 1004;

…..

stuId[9] = 1010;
```

Two Dimensional Array (2D array)

Array Declaration

1. dataType[][] variableName;
2. dataType   [][]variableName;
3. dataType variableName[][];
4. dataType []variableName[];
5. int[][] arr=new int[3][3];

A

12	5	89	0
A[0][0]	A[0][1]	A[0][2]	A[0][3]
15	6	27	-8
A[1][0]	A[1][1]	A[1][2]	A[1][3]
4	2	90	2
A[2][0]	A[2][1]	A[2][2]	A[2][3]

int[][] A = new int[3][4];

```
import java.util.*;
class TwoDimensionalArray{
 public static void main(String[] args){
Scanner scan=new Scanner(System.in);
int[][] A=new int[3][3];

  int i,j;
  for(i=0;i<3;i++){
          for( j=0; j<3; j++){
          A[i][j] = scan.nextInt();
          }
     }
System.out.println("Matrix A is:");
    for(i=0;i<3;i++){
         for(j=0;j<3;j++){
              System.out.print(A[i][j]+" ");
```

```
        }

          System.out.print("\n");
        }

    }
}
```

Three Dimensional Array (3D array)

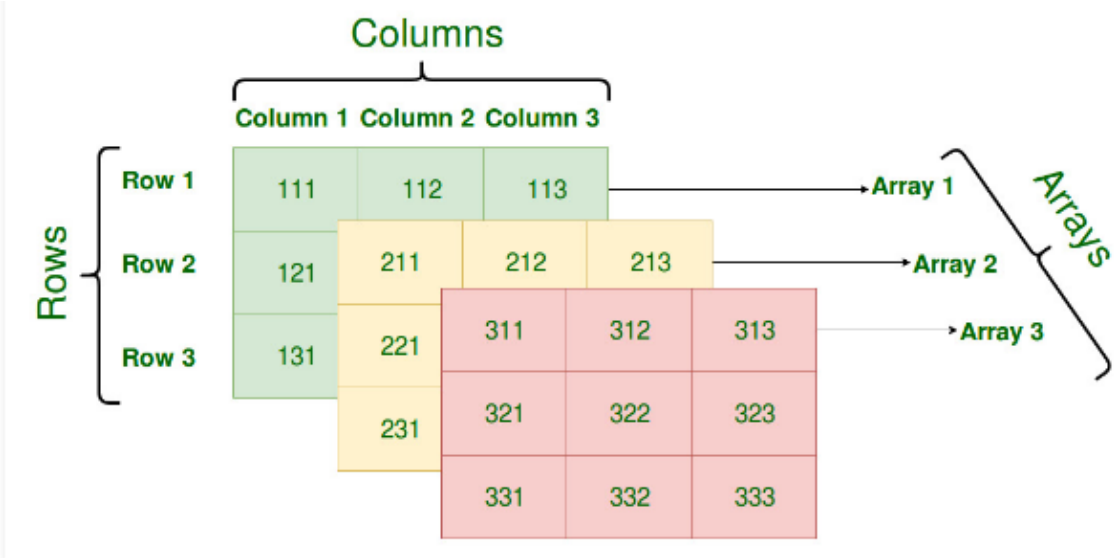
Elements in three-dimensional arrays are commonly referred by **x[i][j][k]** where 'i' is the array number, 'j' is the row number and 'k' is the column number.

Syntax:

```
x[array_index][row_index][column_index]
```

For example:

```
int[][][] x = new int[3][3][3];
```




```
int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };
```

```
class ThreeDimensionalArray {
   public static void main(String[] args)
   {
      int[][][] arr = {
                { { 1, 2 }, { 3, 4 } },
                    { { 5, 6 }, { 7, 8 } }
                };

      for (int i = 0; i < 2; i++) {

         for (int j = 0; j < 2; j++) {

            for (int k = 0; k < 2; k++) {

               System.out.print(arr[i][j][k] + " ");
            }

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

Jagged Array

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

                int arr[][] = new int[2][];

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

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

```
            int count = 0;
            for (int i = 0; i < arr.length; i++)
                for (int j = 0; j < arr[i].length; j++)
                    arr[i][j] = count++;

            // Displaying the values of 2D Jagged array
            System.out.println("Contents of 2D Jagged Array");
            for (int i = 0; i < arr.length; i++) {
                for (int j = 0; j < arr[i].length; j++)
                    System.out.print(arr[i][j] + " ");
                System.out.println();
            }
        }
}
```

String

In Java, string is basically an object that **represents sequence of char values.**

The Java String is **immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal
2. By new keyword

```
String s="welcome";   // string literal
String s=new String("Welcome")  // create using new keyword
```

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
```

```
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);  //converting char array to string
String s3=new String("example");  //creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

char charAt(int index)	It returns char value for the particular index
int length()	It returns string length
String substring(int beginIndex)	It returns substring for given begin index.
String replace(char old, char new)	It replaces all occurrences of the specified char value.
int indexOf(char ch)	It returns the specified char value index.
String toLowerCase()	It returns a string in lowercase.
String toUpperCase()	It returns a string in uppercase.
String trim()	It removes beginning and ending spaces of this string.
boolean equals(Object another)	It checks the equality of string with the given object.

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


        String text = "    HI, This is Sample Text    ";
        int len=text.length();
        //System.out.println(len);
       // System.out.println(text.charAt(4));
       // System.out.println(text.length());
        // System.out.println(text.substring(6, 10));
         //System.out.println();
        // String text2 = text.replace('I', 'i');
        // text = text.replace('I', 'i');
//         System.out.println(text);
        // System.out.println(text.trim());
String text2="java";
System.out.println(text.equals(text2));


    }
}
```

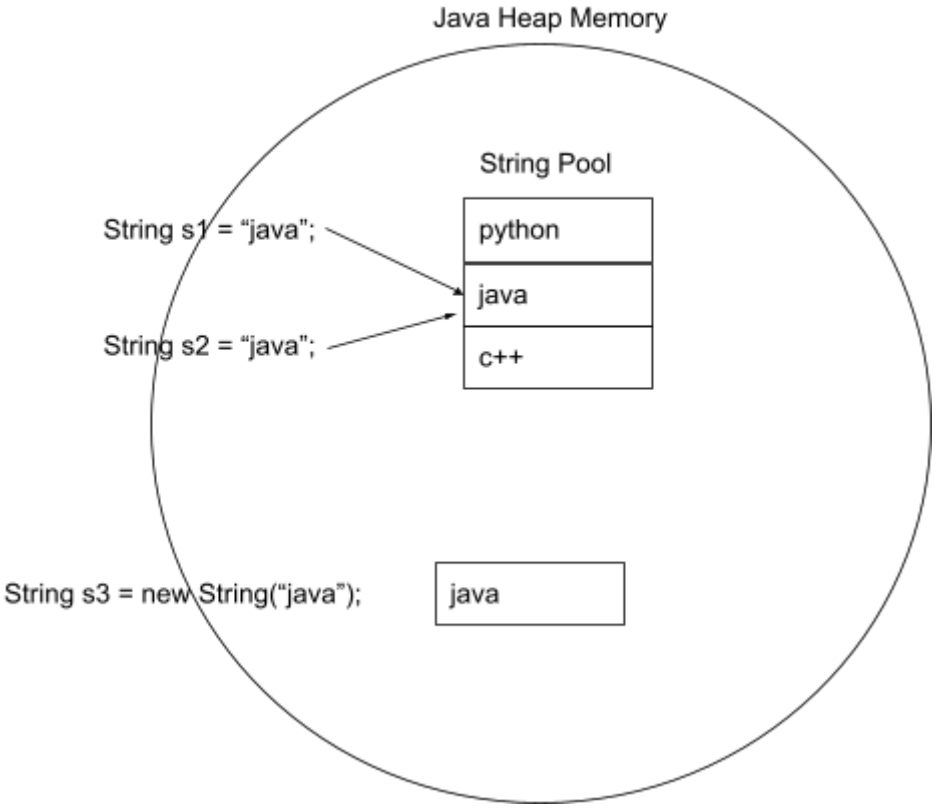
equals() vs ==

The Java String class **equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

```
String s1=”java”;
String s2=”java”;
System.out.println(s1.equals(s2)); // true
```

In simple words, **==** checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

```
String s1=”java”;
String s2=”java”;
String s3 = new String(“java”);
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
```



StringBuffer

Java StringBuffer class is used to create **mutable** (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Text = “credo”

text.append(“systems”); //Credo systems

text.insert(2,”systems”); // Crsystemsedo

- append(String s)
- insert(int offset, String s)
- replace(int startIndex, int endIndex, String str)
- delete(int startIndex, int endIndex)
- reverse()
- substring(int beginIndex)
- length()

StringBuilder

Java StringBuilder class is used to create **mutable** (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

- append(String s)
- insert(int offset, String s)
- replace(int startIndex, int endIndex, String str)
- delete(int startIndex, int endIndex)
- reverse()
- substring(int beginIndex)
- length()

Exception Handling

The Exception Handling in Java is one of the powerful *mechanism to* ***handle the runtime errors*** so that the normal flow of the application can be maintained.

Exception is an **abnormal condition**.In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at **compile-time**.

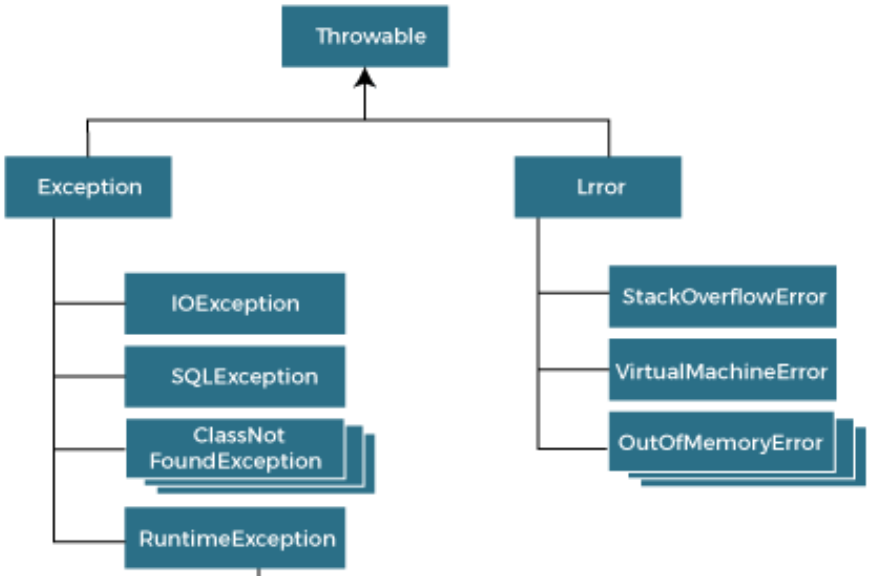
Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at **runtime**.

Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:




```
import java.util.*;
class ExceptionHandlingExample {
    public static void main(String[] args) {
        int n, result=0;
        Scanner scan=new Scanner(System.in);
        System.out.println("Type n value");
        n=scan.nextInt();
        try{
        result = 100/n;
        }
        catch(Exception e){
            System.out.println("Can't divide by Zero. So plz enter valid number");
            n=scan.nextInt();
            result = 100/n;
```

```
            System.out.println(result);
        }
        finally{
            System.out.println(result);
        }

    }
}
```

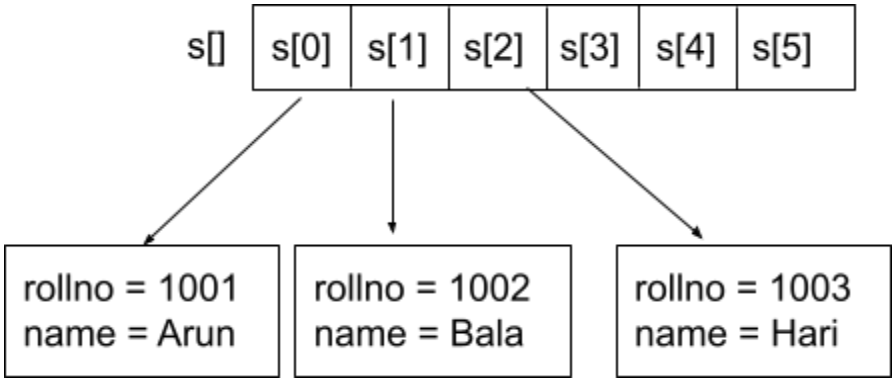
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

```
 import java.util.*;
public class ExceptionHandling {
    public static void main(String[] args)
    {
        int age;
        Scanner scan=new Scanner(System.in);
        System.out.println(“Type your age”);
```

```
        age=scan.nextInt();
        try {
            if(age<18){
            throw new ArithmeticException("You are not eligible");
            }
        }
        catch (ArithmeticException e) {
            e.printStackTrace();
           // System.out.println("Type age");
            //age=scan.nextInt();
        }
    }
}
```

Array of objects

```
class Student
{
int rollno;
String name;
}
```




```
class ArrayOfObjectsExample {

        public static void main(String args[])
        {

                Student[] arr;

                arr = new Student[2];
                arr[0] = new Student(1701289270, "Satyabrata");

                arr[1] = new Student(1701289219, "Omm Prasad");

                System.out.println("Student data in student arr 0: ");
                arr[0].display();

                System.out.println("Student data in student arr 1: ");
                arr[1].display();
        }
}

class Student {

        public int id;
```

```
    public String name;
    Student(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
    public void display()
    {
        System.out.println("Student id is: " + id + " "
                                    + "and Student name is: "
                                    + name);
        System.out.println();
    }
}
```

Java Updated Features

Varargs

The varrags allows the method **to accept zero or multiple arguments**. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Syntax

Three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName)
{
}
```

Wrapper Class

The wrapper class in Java provides the mechanism *to convert primitive into object and object into primitive*.

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

Autoboxing

The automatic **conversion of primitive data type into its corresponding wrapper class** is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

AutoUnboxing

The automatic **conversion of wrapper type into its corresponding primitive type** is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

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

        int a = 20; Integer i = Integer.valueOf(a);// converting int into Integer  explicitly
        Integer j = a;// autoboxing, now compiler will write Integer.valueOf(a) internally

        System.out.println(a + " " + i + " " + j);

// ---------- unboxing----------
        Integer a = new Integer(3);
        int i = a.intValue();// converting Integer to int explicitly
        int j = a;// unboxing, now compiler will write a.intValue() internally

        System.out.println(a + " " + i + " " + j);


}}
```

Enum

The Enum in Java is a data type which contains a **fixed set of constants**.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java.

```
class EnumExample{

	public enum Season {
		WINTER, SPRING, SUMMER, FALL
	}

	public static void main(String[] args) {

		System.out.println(Season.values());

		for (Season s : Season.values()) {
			System.out.println(s);
		}

		System.out.println("Value of WINTER is: " + Season.valueOf("WINTER"));
		System.out.println("Index of WINTER is: " +
Season.valueOf("WINTER").ordinal());
		System.out.println("Index of SUMMER is:
"+Season.valueOf("SUMMER").ordinal());

	}}
```

Annotation

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

Custom annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The @*interface* element is used to declare an annotation.

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

Marker Annotation

An annotation that has no method, is called marker annotation.The @Override and @Deprecated are marker annotations.

```
@interface MyAnnotation
{
}
```

Single value Annotation

An annotation that has one method, is called single-value annotation.

```
@interface MyAnnotation{
int value();
}
```

Multi value Annotation

An annotation that has more than one method, is called Multi-Value annotation.

```
@interface MyAnnotation{
int value1();
String value2();
String value3();
}
```

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as,

TYPE,
METHOD,
FIELD
LOCAL_VARIABLE
ANNOTATION_TYPE
PARAMETER

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

Assertion

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing the assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purposes.

Syntax

assert expression;

or

assert expression1 : expression2;

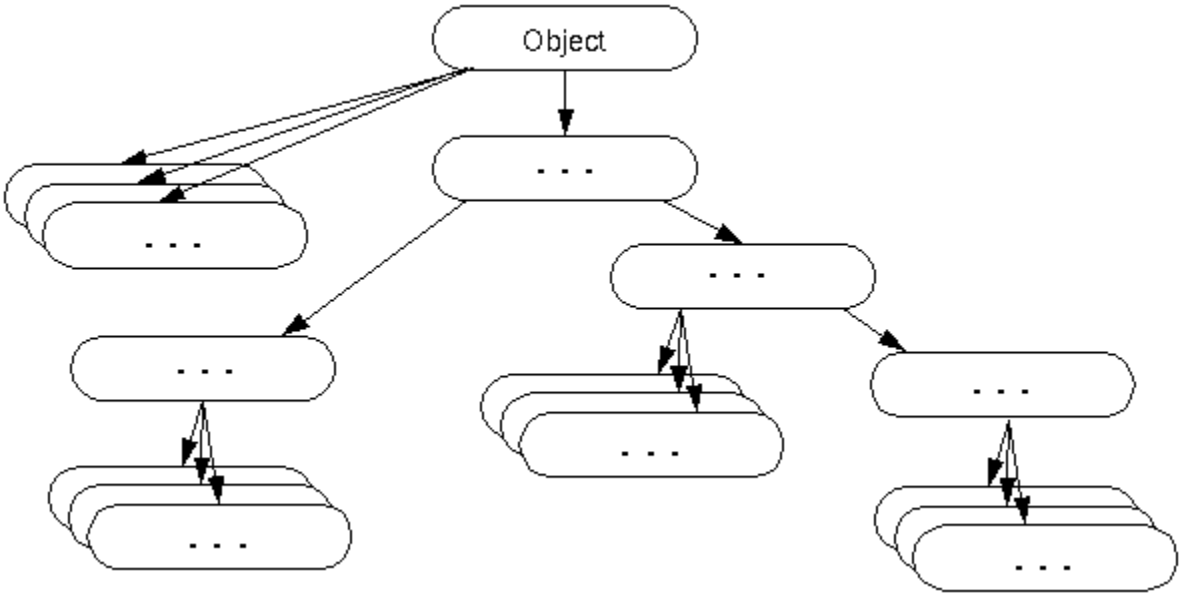
If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, -ea or -enableassertions switch of java must be used.
Compile it by: **javac AssertionExample.java**
Run it by: **java -ea AssertionExample**

```
import java.util.Scanner;

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

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }
}
```

OOPS Miscellaneous

Object class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes.



The Object class provides multiple methods which are as follows:

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method
- clone() method
- wait(), notify() notifyAll() methods

Object cloning

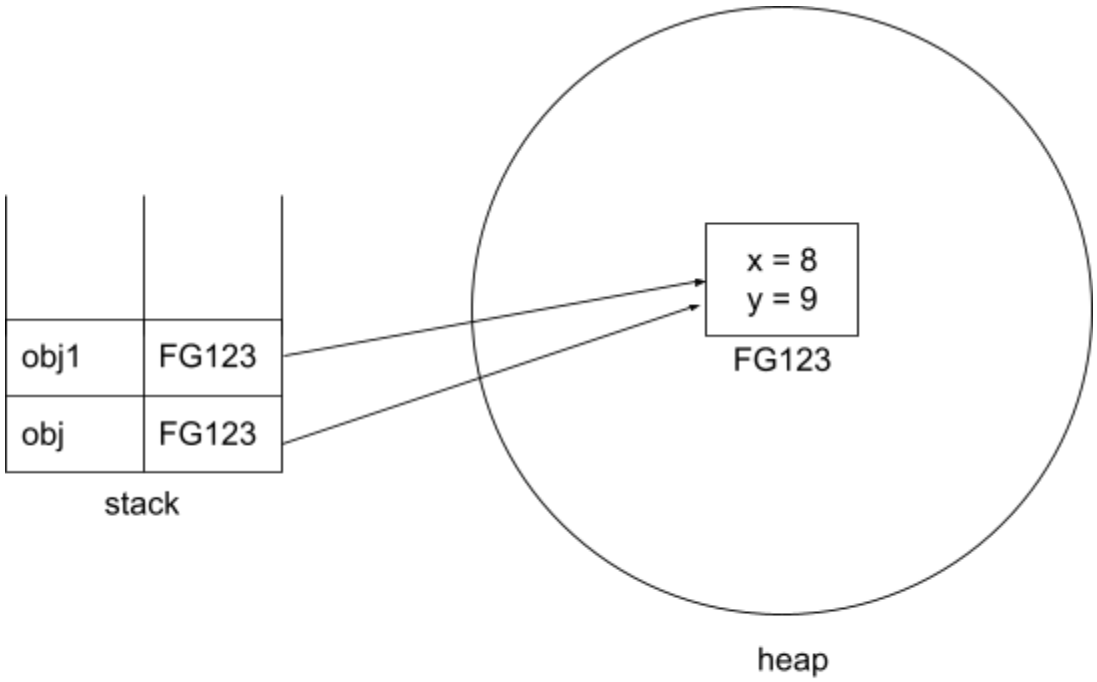
There are three types of object copying.

1. Shallow copy
2. Deep copy
3. Clone

Shallow copy

A shallow copy of an object is a new object whose instance variables are identical to the old object. For example, a shallow copy of a Set has the same members as the old Set and shares objects with the old Set through pointers. Shallow copies are sometimes said to use reference semantics.

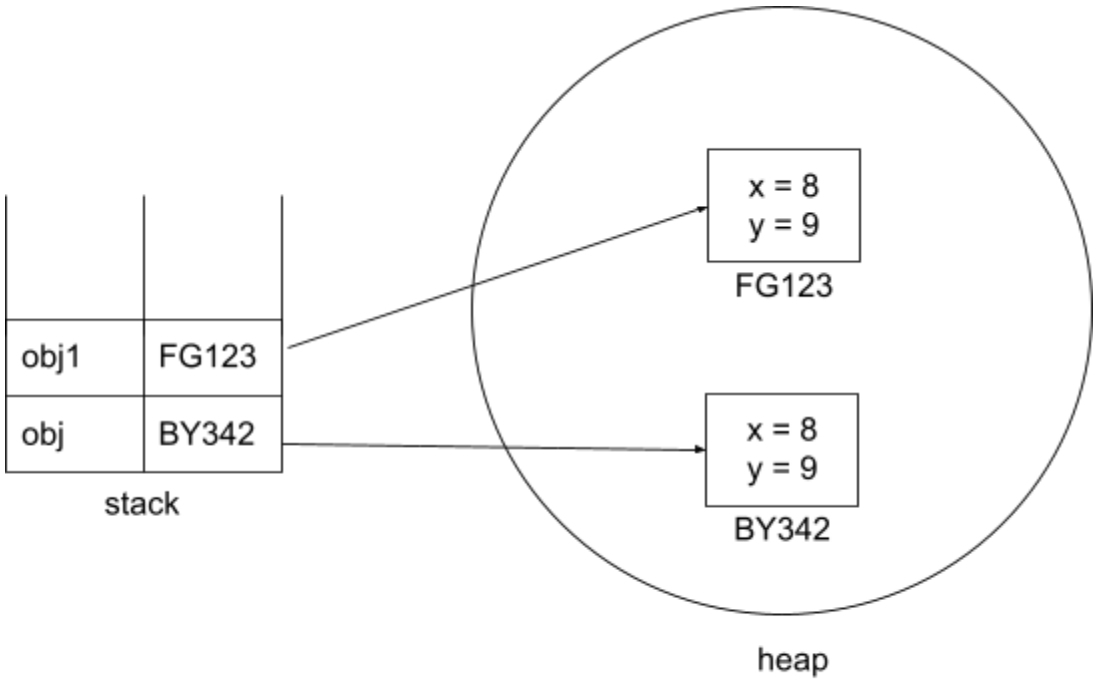
```
class Abc
{
int x;
int y;
}
Abc obj = new Abc();
obj.x = 8;
obj.y = 9;

Abc obj1 = obj;    // shallow copy
```



Shallow copy

Deep copy

When we do a copy of some entity to create two or more than two entities such that changes in one entity are not reflected in the other entities, then we can say we have done a deep copy. In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference.



Deep copy

IO Package

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform file handling in Java by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, three streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

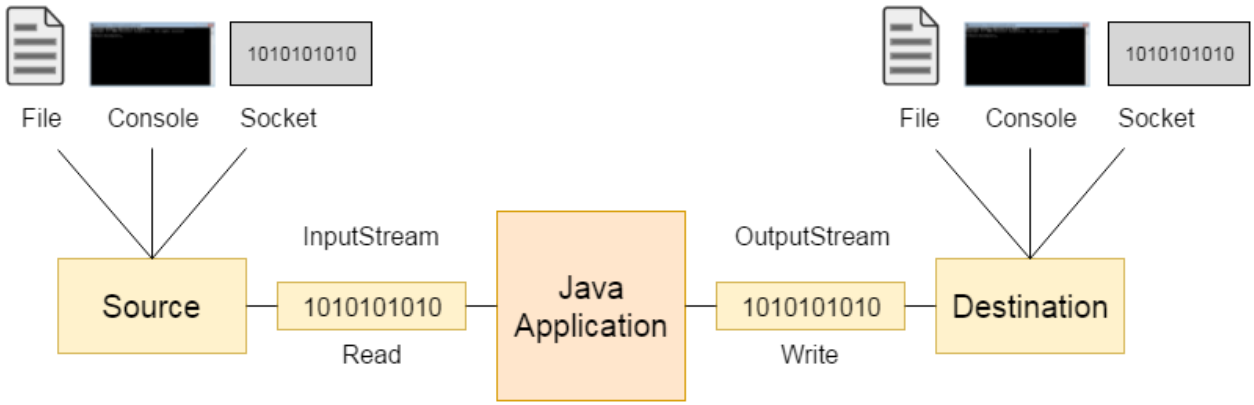
2) System.in: standard input stream

3) System.err: standard error stream

InputStream

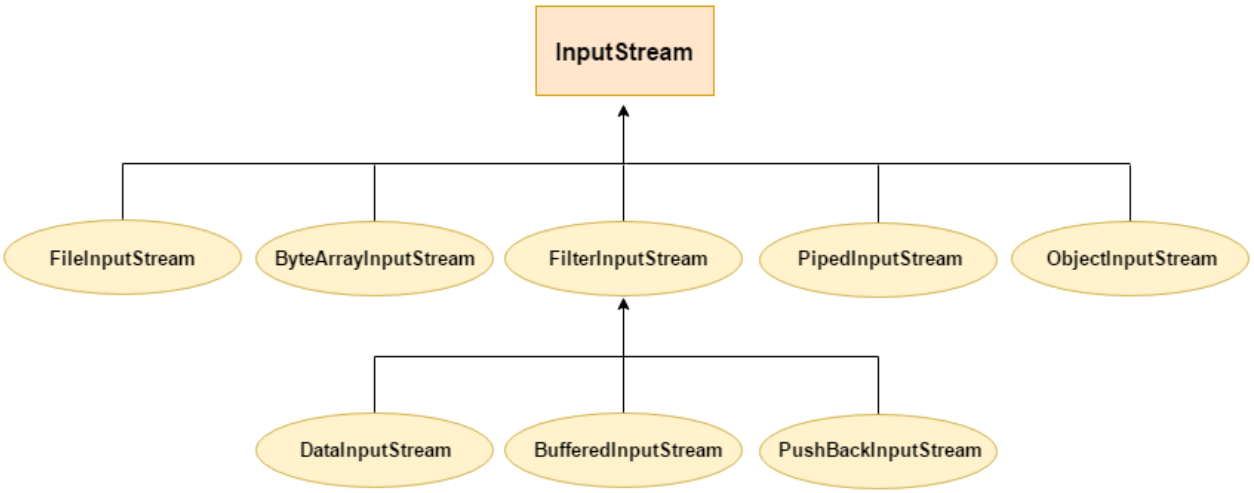
Java applications use an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

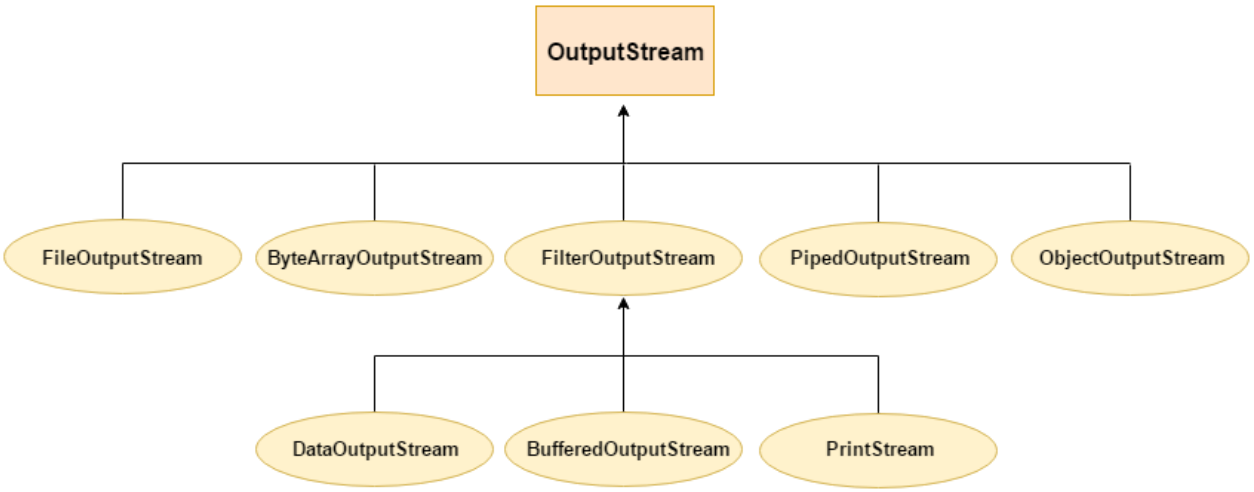


OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.



FileOutputStream

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
          try{
            FileOutputStream fout=new FileOutputStream("testout.txt");
            String s="Java Full Stack course";
            byte b[]=s.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
           }
          catch(Exception e)
          {
           System.out.println(e);
          }
     }
}
```

FileInputStream

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

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

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

ByteArrayOutputStream

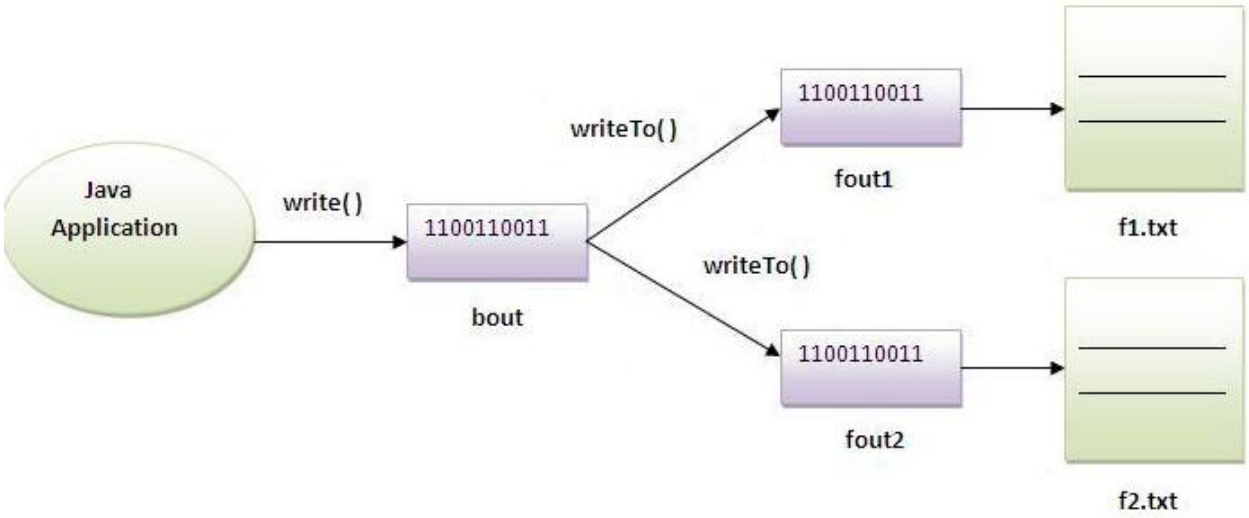
Java ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte array

which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len	It is used for writing **len** bytes from specified byte array starting from the offset **off** to the byte array output stream.
void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.




```
import java.io.*;

public class ByteArrayOutputStreamExample {
	public static void main(String args[]) throws Exception {
		FileOutputStream fout1 = new FileOutputStream("f1.txt");
		FileOutputStream fout2 = new FileOutputStream("f2.txt");

		String s="This text will be written in two files.";

		ByteArrayOutputStream bout = new ByteArrayOutputStream();

		byte[] b=s.getBytes();
		bout.write(b);
		bout.writeTo(fout1);
		bout.writeTo(fout2);

		bout.flush();
		bout.close();
		System.out.println("Success...");
	}
}
```

ByteArrayInputStream

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream.

Java ByteArrayInputStream class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

The buffer of ByteArrayInputStream automatically grows according to data.

Methods	Description
int available()	It is used to return the number of remaining bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read up to len bytes of data from an array of bytes in the input stream.
boolean markSupported()	It is used to test the input stream for mark and reset method.
long skip(long x)	It is used to skip the x bytes of input from the input stream.
void mark(int readAheadLimit)	It is used to set the current marked position in the stream.
void reset()	It is used to reset the buffer of a byte array.
void close()	It is used for closing a ByteArrayInputStream.

```
import java.io.*;

public class ByteArrayInputStreamExample {
	public static void main(String[] args) throws IOException {
		byte[] buf = { 35, 36, 37, 38 };

		ByteArrayInputStream byt = new ByteArrayInputStream(buf);
		int k = 0;
		while ((k = byt.read()) != -1) {

			char ch = (char) k;
			System.out.println("ASCII value of Character is:" + k + "; Special
character is: " + ch);
		}
	}
}
```

Serialization

Serialization in Java is a mechanism of ***writing the state of an object into a byte-stream*.** It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called ***deserialization*** where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream* class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the ***Serializable*** interface for serializing the object.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	It writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	It flushes the current output stream.
3) public void close() throws IOException {}	It closes the current output stream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	It reads an object from the input stream.
2) public void close() throws IOException {}	It closes ObjectInputStream.

```
class Game implements Serializable{
	int score;
	int completedLevel;
}

public class ObjectOutputStreamExample {

	public static void main(String[] args) throws IOException, ClassNotFoundException {

		Game obj = new Game();
		obj.score = 3400;
		obj.completedLevel = 3;

		File f= new File("GameLogDetails.txt");
		FileOutputStream fo = new FileOutputStream(f);
		ObjectOutputStream oos = new ObjectOutputStream(fo);
		oos.writeObject(obj);


		FileInputStream fi = new FileInputStream(f);
		ObjectInputStream ois = new ObjectInputStream(fi);
		Game obj1 = (Game) ois.readObject();
		System.out.println(obj1.completedLevel+"\t "+obj1.score);
	}
}
```

transient keyword

During the serialization, when we do not want an object to be serialized we can use a transient keyword.

Why use the transient keyword?

The transient keyword can be used with the data members of a class in order to avoid their serialization. For example, if a program accepts a user's login details and password. But we don't want to store the original password in the file. Here, we can use the transient keyword and when the JVM reads the transient keyword it ignores the original value of the object and instead stores the default value of the object.

Syntax

```
	private transient <member variable>;
```

Character Streams

FileReader

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

It is character-oriented class which is used for file handling in java.

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.
void close()	It is used to close the FileReader class.

```
public class FileReaderExample {

    public static void main(String args[])throws Exception{

        FileReader fr=new FileReader("test.txt");

        int i;

        while((i=fr.read())!=-1)

        System.out.print((char)i);

        fr.close();

    }

}
```

FileWriter

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

```
import java.io.FileWriter;

public class FileWriterExample {

    public static void main(String args[]){

        try{

          FileWriter fw=new FileWriter("testt.txt");

          fw.write("Welcome All.");

          fw.close();

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

        System.out.println("Success...");

    }
```

```
}
```

DATABASE

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

SQL

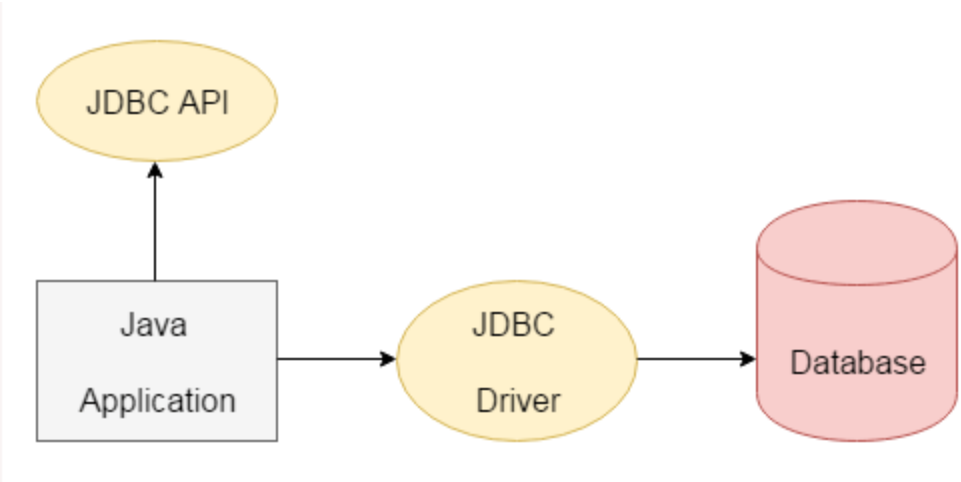
SQL tutorial provides basic and advanced concepts of SQL. Our SQL tutorial is designed for both beginners and professionals.SQL (Structured Query Language) is used to perform operations on the records stored in the database, such as updating records, inserting records, deleting records, creating and modifying database tables, views, etc.

SQL is not a database system, but it is a query language.Suppose you want to perform the queries of SQL language on the stored data in the database. You are required to install any database management system in your systems, for example, Oracle, MySQL, MongoDB, PostgreSQL, SQL Server, DB2, etc.

JDBC

JDBC stands for **Java Database Connectivity**. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver
3. Network Protocol driver
4. Thin driver

JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

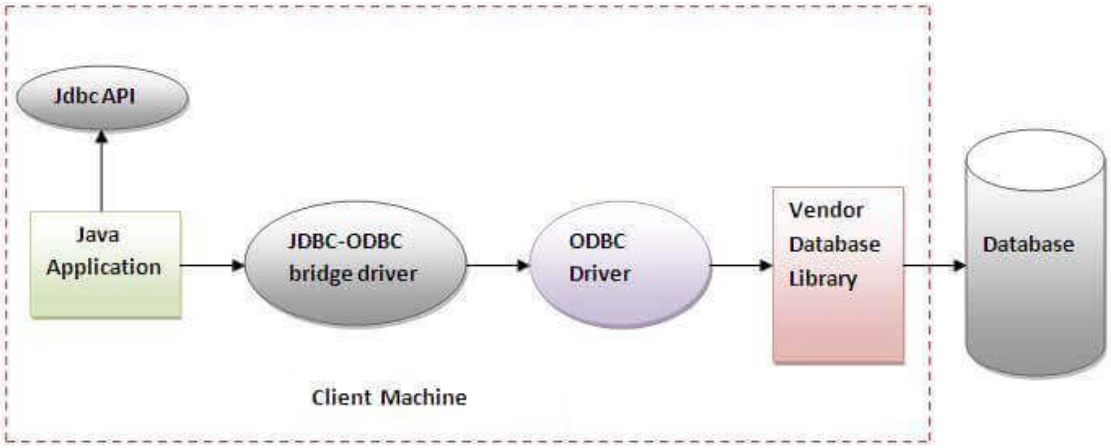


Figure- JDBC-ODBC Bridge Driver

Native API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

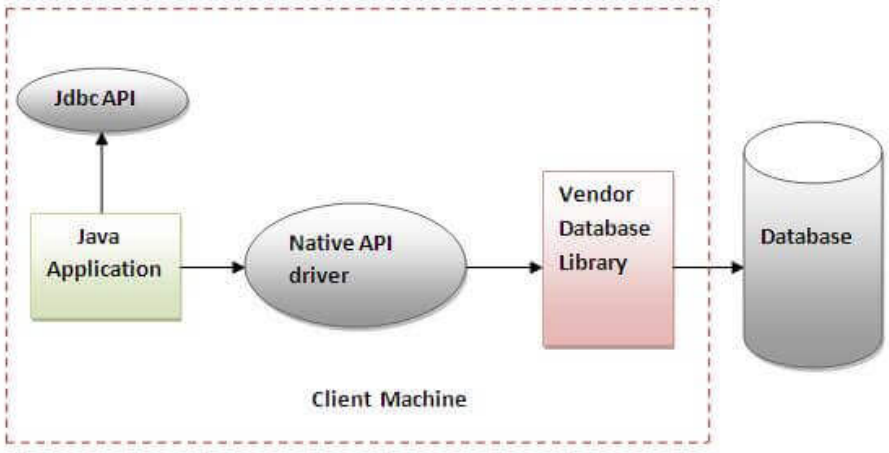


Figure- Native API Driver

Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

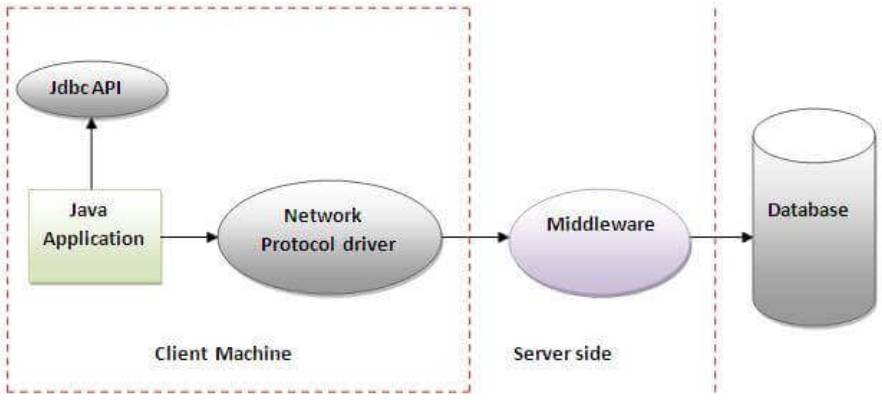


Figure- Network Protocol Driver

Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

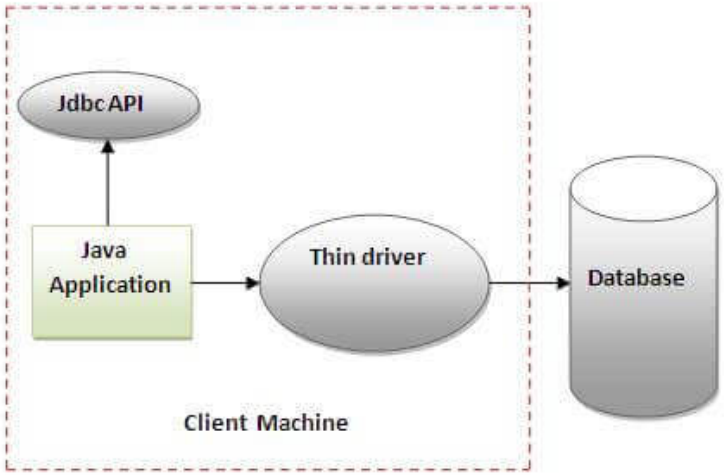


Figure- Thin Driver

Java Database Connectivity

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

https://www.javatpoint.com/steps-to-connect-to-the-database-in-java

```
import java.beans.Statement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Scanner;

public class JdbcExample {

	public static void main(String[] args) throws ClassNotFoundException, SQLException {

		Class.forName("oracle.jdbc.driver.OracleDriver");

		Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "123");

		java.sql.Statement st = con.createStatement();

		ResultSet rs = st.executeQuery("select * from student");
		while (rs.next()) {
			System.out.println(rs.getInt(1) + " " + rs.getString(2));
		}

		/*
		 * Scanner scan = new Scanner(System.in);
		 *
		 * System.out.println("Type rollno");
		 * int rn = scan.nextInt();
		 * System.out.println("Type Name");
		 * scan.nextLine();
		 * String name = scan.nextLine();
		 * int rs = st.executeUpdate("insert into student values (" +rn + ",'" + name + "')");
		 * if (rs > 0)
		 * System.out.println("Inserted ");
		 * String query = "insert into student values (?, ?)";
		 * java.sql.PreparedStatement pt = con.prepareStatement(query);
		 * pt.setInt(1, rn);
		 * pt.setString(2, name);
		 * int rs = pt.executeUpdate();
		 * if (rs > 0)
		 * System.out.println("Inserted");
```

```
                 */
                con.close();

        }

}
```

Multithreading

Multithreading is a Java feature that allows **concurrent execution of two or more parts of a program** for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

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.

```
class A extends Thread {
   public void run()
   {
     int sum=0;
      try {
         for(int i=1;i<=1000;i++){
          sum = sum+i;
         System.out.println(sum);
Thread.sleep(1000);
         }
      }
      catch (Exception e) {
         System.out.println("Exception is caught");
      }
   }
}

class B extends Thread {
```

```
   public void run()
   {

      try {
         for(int i=1000;i>=1;i=i-5){
         System.out.println(i);
Thread.sleep(1000);
         }
      }
      catch (Exception e) {
         System.out.println("Exception is caught");
      }
   }
}

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

         A obj1 = new A();
         B obj2 = new B();

         obj1.start();
        obj2.start();


   }
}
```

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.

```
class ThreadA implements Runnable{
    public void run()
    {
       int sum=0;
```

```
        try {
            for(int i=1;i<=1000;i++){
           sum = sum+i;
            System.out.println(sum);
Thread.sleep(1000);
            }
        }
        catch (Exception e) {
            System.out.println("Exception is caught");
        }
    }
}

class ThreadB implements Runnable{
    public void run()
    {

        try {
            for(int i=1000;i>=1;i=i-5){
            System.out.println(i);
Thread.sleep(1000);
            }
        }
        catch (Exception e) {
            System.out.println("Exception is caught");
        }
    }
}


public class MultiThreadDemo1 {
    public static void main(String[] args)
    {

          ThreadA obj1 = new ThreadA();
                    ThreadB obj2 = new ThreadB();

                    obj1.run();
                    obj2.run();



    }
}
```

Life cycle of a Thread

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

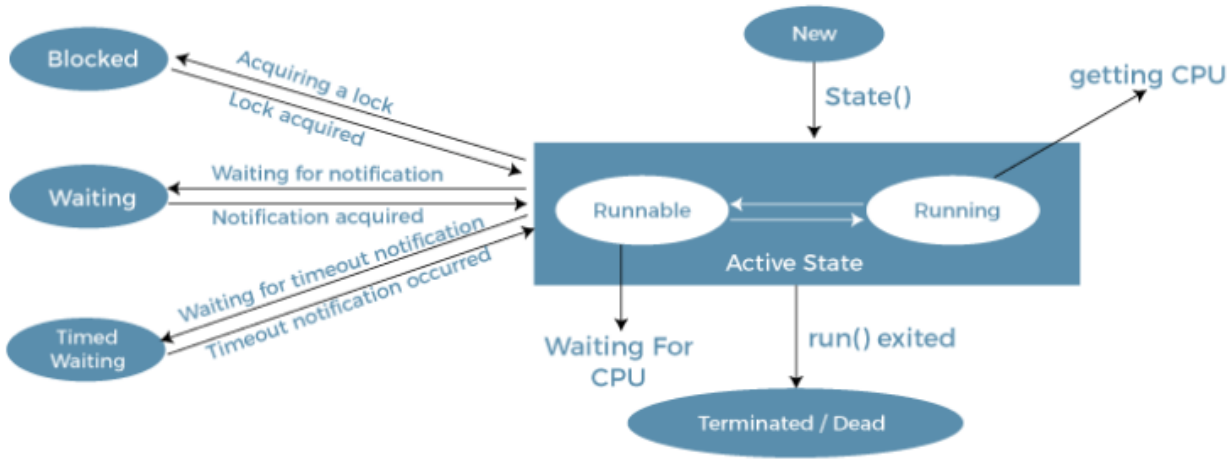
Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting

state.

Timed Waiting: Sometimes, waiting for leads to starvation.A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.



Life Cycle of a Thread

Synchronization

Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class Table {
	 void printTable(int n) {
		for (int i = 1; i <= 20; i++) {
			System.out.println(n * i);
			try {
				Thread.sleep(400);
			} catch (Exception e) {
				System.out.println(e);
			}
		}

	}
}

class MyThread1 extends Thread {
	Table t;

	MyThread1(Table t) {
		this.t = t;
	}

	public void run() {
		t.printTable(2);
	}

}

class MyThread2 extends Thread {
	Table t;
```

```
	MyThread2(Table t) {
		this.t = t;
	}

	public void run() {
		t.printTable(5);
	}
}

public class SynchronizationExample {
	public static void main(String args[]) {
		Table obj = new Table();
		MyThread1 t1 = new MyThread1(obj);
		MyThread2 t2 = new MyThread2(obj);
		t1.start();
		t2.start();
	}
}
```

Util Package

The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

Matcher class

It implements the **MatchResult** interface. It is a *regex engine* which is used to perform match operations on a character sequence.

No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.
3	boolean find(int start)	finds the next expression that matches the pattern from the given start number.
4	String group()	returns the matched subsequence.
5	int start()	returns the starting index of the matched subsequence.
6	int end()	returns the ending index of the matched subsequence.
7	int groupCount()	returns the total number of the matched subsequence.

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

```
import java.util.regex.*;
class RegexExample4{
public static void main(String args[]){
System.out.println("? quantifier ....");
System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
```

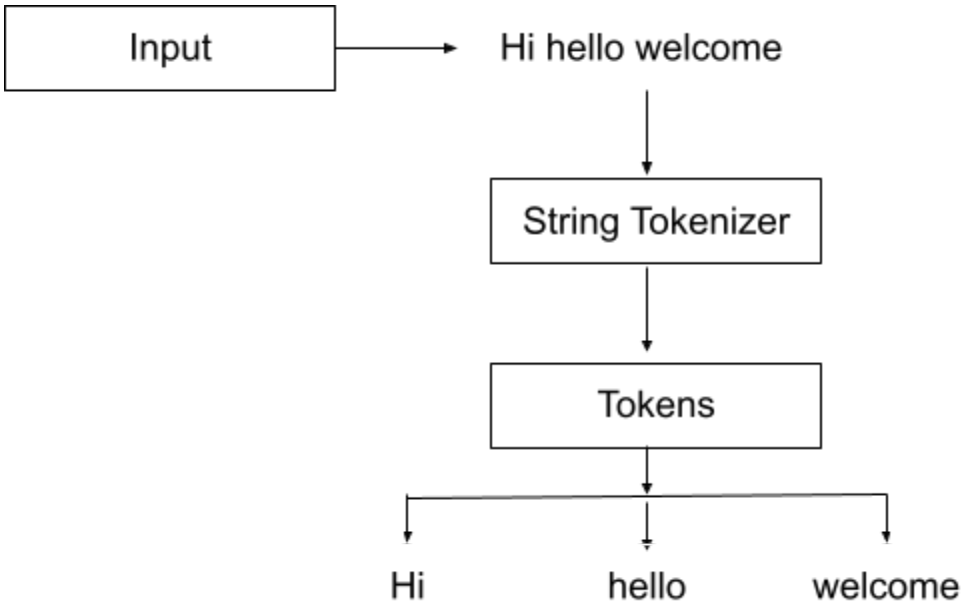
```
System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)

System.out.println("+ quantifier ....");
System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)
System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)

System.out.println("* quantifier ....");
System.out.println(Pattern.matches("[amn]*", "ammmna"));//true (a or m or n may come zero or more times)

}}
```

StringTokenizer

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.



Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.
int countTokens()	It returns the total number of tokens.

Date

Java LocalDate class is an immutable class that represents Date with a default format of yyyy-mm-dd. It inherits Object class and implements the ChronoLocalDate interface.

LocalDateTime atTime(int hour, int minute)	It is used to combine this date with a time to create a LocalDateTime.
int compareTo(ChronoLocalDate other)	It is used to compares this date to another date.
boolean equals(Object obj)	It is used to check if this date is equal to another date.
String format(DateTimeFormatter formatter)	It is used to format this date using the specified formatter.
int get(TemporalField field)	It is used to get the value of the specified field from this date as an int.
boolean isLeapYear()	It is used to check if the year is a leap year, according to the ISO proleptic calendar system rules.
LocalDate minusDays(long daysToSubtract)	It is used to return a copy of this LocalDate with the specified number of days subtracted.
LocalDate minusMonths(long monthsToSubtract)	It is used to return a copy of this LocalDate with the specified number of months subtracted.
static LocalDate now()	It is used to obtain the current date from the system clock in the default time-zone.
LocalDate plusDays(long daysToAdd)	It is used to return a copy of this LocalDate with the specified number of days added.

```
import java.time.LocalDate;

public class LocalDateExample {
	public static void main(String[] args) {
		LocalDate date = LocalDate.now();
		LocalDate yesterday = date.minusDays(1);
		LocalDate tomorrow = yesterday.plusDays(2);
		System.out.println("Today date: " + date);
		System.out.println("Yesterday date: " + yesterday);
		System.out.println("Tomorrow date: " + tomorrow);

		LocalDate date2 = LocalDate.of(2022, 11, 02);
	   System.out.println(date.isLeapYear());
	}
}
```

LocalTime

Java LocalTime class is an immutable class that represents time with a default format of hour-minute-second. It inherits Object class and implements the Comparable interface.

Method	Description
LocalDateTime atDate(LocalDate date)	It is used to combine this time with a date to create a LocalDateTime.
int compareTo(LocalTime other)	It is used to compare this time to another time.
String format(DateTimeFormatter formatter)	It is used to format this time using the specified formatter.
int get(TemporalField field)	It is used to get the value of the specified field from this time as an int.
LocalTime minusHours(long hoursToSubtract)	It is used to return a copy of this LocalTime with the specified number of hours subtracted.
LocalTime minusMinutes(long minutesToSubtract)	It is used to return a copy of this LocalTime with the specified number of minutes subtracted.
static LocalTime now()	It is used to obtain the current time from the system clock in the default time-zone.
static LocalTime of(int hour, int minute, int second)	It is used to obtain an instance of LocalTime from an hour, minute and second.
LocalTime plusHours(long hoursToAdd)	It is used to return a copy of this LocalTime with the specified number of hours added.
LocalTime plusMinutes(long minutesToAdd)	It is used to return a copy of this LocalTime with the specified number of minutes added.

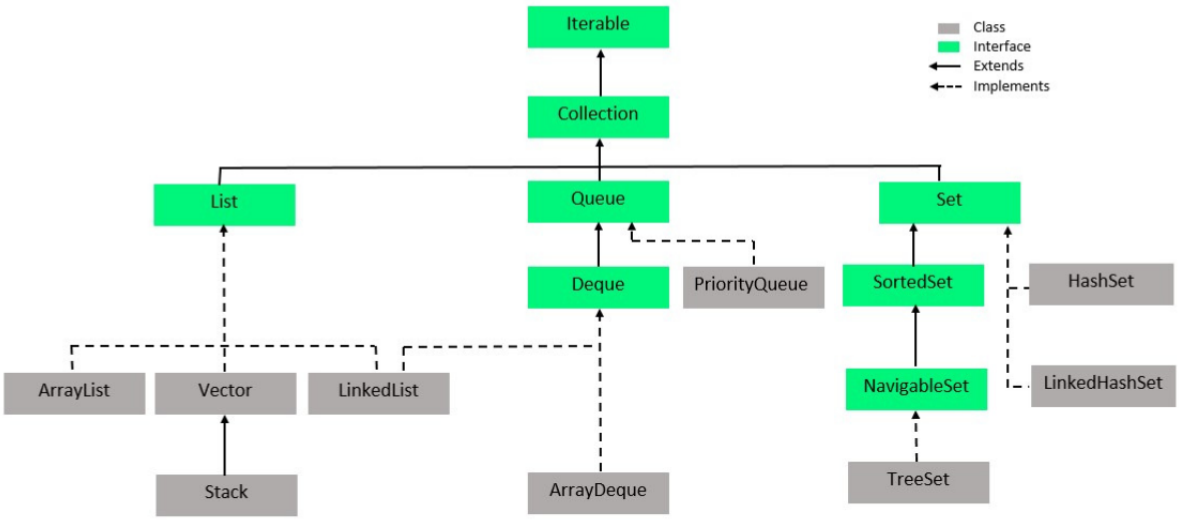
```
import java.time.LocalTime;

public class LocalTimeExample {
	public static void main(String[] args) {
		LocalTime time1 = LocalTime.of(10, 43, 12);
		System.out.println(time1);
		LocalTime time2 = time1.minusHours(2);
		LocalTime time3 = time2.minusMinutes(34);
		System.out.println(time3);
	}
}
```

Util Package - Collection Framework

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

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

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



List

List in Java provides the facility to maintain the ***ordered collection***. It contains the **index-based** methods to insert, update, delete and search the elements. It **can have the duplicate element**s also. We can also store the null elements in the list. The List interface is found in the java.util package and inherits the Collection interface.The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming.

ArrayList

Java ArrayList class uses a ***dynamic array*** for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package.

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- Default capacity 10. Load factor is 0.75

```
import java.util.*;
class ArrayListExample {
    public static void main(String[] args) {
        List<String> names=new ArrayList<String>();

        names.add("arun");
        names.add("bala");
        names.add("Charan");
        names.add("Devi");

        for(String n:names){
            System.out.println(n);
        }

        names.set(0,"Ajay");
        System.out.println(names.get(0));

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

LinkedList

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

- 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.

```
import java.util.*;
public class LinkedListExample{
 public static void main(String args[]){

  LinkedList<String> train=new LinkedList<String>();
  train.add("carriage 1");
  train.add("carriage 2");
  train.add("carriage 3");
  train.add("carriage 4");

  Iterator<String> itr=train.iterator();
  while(itr.hasNext()){
  // System.out.println(itr.next());
  }

  train.removeFirst();
  System.out.println(train.getFirst());
 }
}
```

Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework.

Stack

The stack is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the Stack class that provides different operations such as push, pop, search, etc.

```
import java.util.*;
public class StackExample{
 public static void main(String args[]){

  Stack<Integer> stock= new Stack<>();

  stock.push(50);
```

```
  stock.push(100);
  stock.push(70);
  stock.push(90);
  System.out.println("Stack:"+stock );
  System.out.println("Top of the Stack:"+stock.peek() );
  stock.pop();
  System.out.println("Stack:"+stock );
    System.out.println("Top of the Stack:"+stock.peek() );
   }
}
```

Queue

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list

Priority Queue

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

```
import java.util.*;
public class QueueExample{
 public static void main(String args[]){
  PriorityQueue<Integer> stock= new PriorityQueue<>();
  stock.add(50);
  stock.add(100);
```

```
  stock.add(70);
  stock.add(90);

  System.out.println("Queue:"+stock );
  System.out.println("Top of the Queue:"+stock.peek() );
  stock.remove();
  System.out.println("Queue:"+stock );

  System.out.println("Top of the Queue:"+stock.peek() );

 }
}
```

Deque

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

ArrayDeque

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

```
import java.util.*;
public class DequeueExample{
 public static void main(String args[]){
  Deque<String> deque=new ArrayDeque<String>();
    deque.offer("arvind");
    deque.offer("vimal");
    deque.add("mukundhan");
    deque.offerFirst("jai");
    System.out.println("After offerFirst Traversal...");
    for(String s:deque){
        System.out.println(s);
    }

    deque.pollLast();
    System.out.println("After pollLast() Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
 }
}
```

Set

The set is an interface available in the java.util package. The set interface extends the Collection interface. An **unordered collection** or list in which **duplicates are not allowed** is referred to as a collection interface. The set interface is used to create the mathematical set. The set interface use collection interface's methods to **avoid the insertion of the same elements**. SortedSet and NavigableSet are two interfaces that extend the set implementation.

HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.

- HashSet doesn't maintain the insertion order. Here, elements are **inserted on the basis of their hashcode.**
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16

```
import java.util.*;
public class SetOperations
{
   public static void main(String args[])
   {
      Integer[] A = {22, 45,33, 66, 55, 34, 77};
      Integer[] B = {33, 2, 83, 45, 3, 12, 55};
      Set<Integer> set1 = new HashSet<Integer>();
      set1.addAll(Arrays.asList(A));
      Set<Integer> set2 = new HashSet<Integer>();
      set2.addAll(Arrays.asList(B));

      Set<Integer> union_data = new HashSet<Integer>(set1);
      union_data.addAll(set2);
      System.out.print("Union of set1 and set2 is:");
      System.out.println(union_data);

      Set<Integer> intersection_data = new HashSet<Integer>(set1);
      intersection_data.retainAll(set2);
      System.out.print("Intersection of set1 and set2 is:");
      System.out.println(intersection_data);

      Set<Integer> difference_data = new HashSet<Integer>(set1);
      difference_data.removeAll(set2);
      System.out.print("Difference of set1 and set2 is:");
      System.out.println(difference_data);
   }
}
```

LinkedHashSet

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

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

- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.

```
import java.util.*;
class LinkedHashSetExample{
public static void main(String args[]){
 LinkedHashSet<String> al=new LinkedHashSet<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
 }
 }
}
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about the Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and **retrieval times are quiet fast.**
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains **ascending** order.

Map

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

HashMap

Java HashMap class implements the Map interface which allows us *to* ***store key and value pair*****, where keys should be unique**. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16

```
import java.util.*;
public class HashMapExample{
public static void main(String args[]){
 HashMap<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"Mango");
map.put(2,"Apple");
map.put(3,"Banana");
map.put(4,"Grapes");
 System.out.println("Iterating Hashmap...");
for(Map.Entry m : map.entrySet()){
System.out.println(m.getKey()+" "+m.getValue());
```

```
  }
}
}
```

LinkedHashMap

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

TreeMap

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

Generics

Generics means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Types of Java Generics

Generic Classes:

A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

```
/*class Employee<T, U>
{
	T empId;
	U salary;

	public Employee(T empId, U salary) {
		this.empId = empId;
		this.salary = salary;
	}

	public void display()
	{
		System.out.println("EmpID :"+this.empId+"\nSalary :"+this.salary);
	}
}*/

class Student<T>
{
	T rollno;

	public Student(T rollno) {

		this.rollno = rollno;
	}
```

```
	public void display()
	{
		System.out.println("Rollno :"+this.rollno);
	}

}


public class GenericClassExample {

	public static void main(String[] args) {

		Student<Integer> s1 = new Student<Integer>(1001);
		s1.display();
	}

}
```

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

```
class A<T>
{

	 <T> void display(T num)
	 {
		System.out.println("the given value is :"+ num);

	 }
}

public class GenericMethodExample {

	public static void main(String[] args) {

		A obj = new A();
		obj.display(10);

	}
```

```
}
```

Advantages

Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Type casting is not required: There is no need to typecast the object.

Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Networking

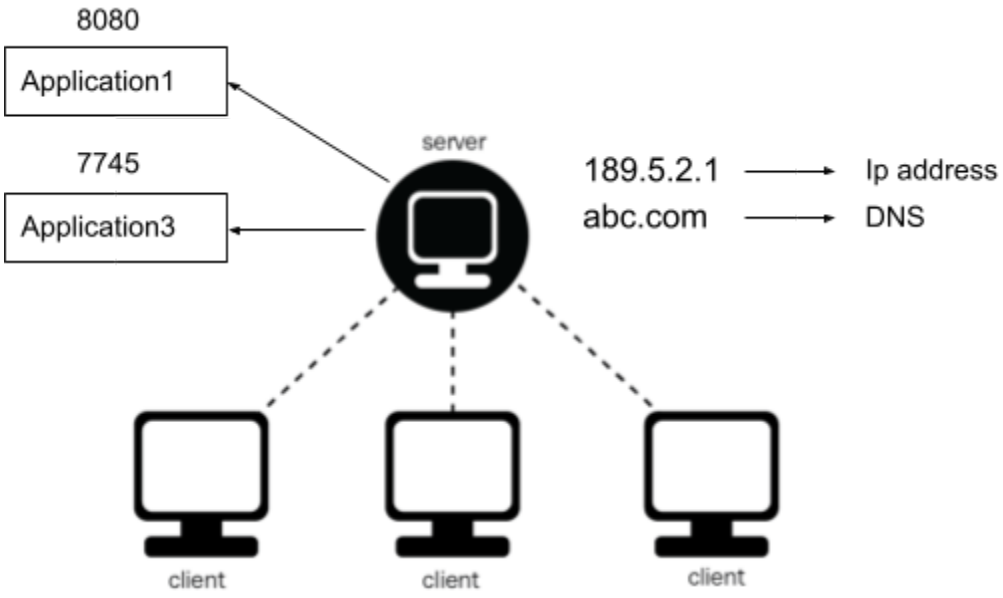
Basics of Networking

IP Address:

An IP address is a unique address that identifies a device on the internet or a local network. IP stands for "Internet Protocol," which is the set of rules governing the format of data sent via the internet or local network.

DNS

The Domain Name System (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.



abc.com - > 189.5.2.1:8080

When client said abc.com it means 189.5.2.1:8080

Socket

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less. Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server
2. Port number.

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

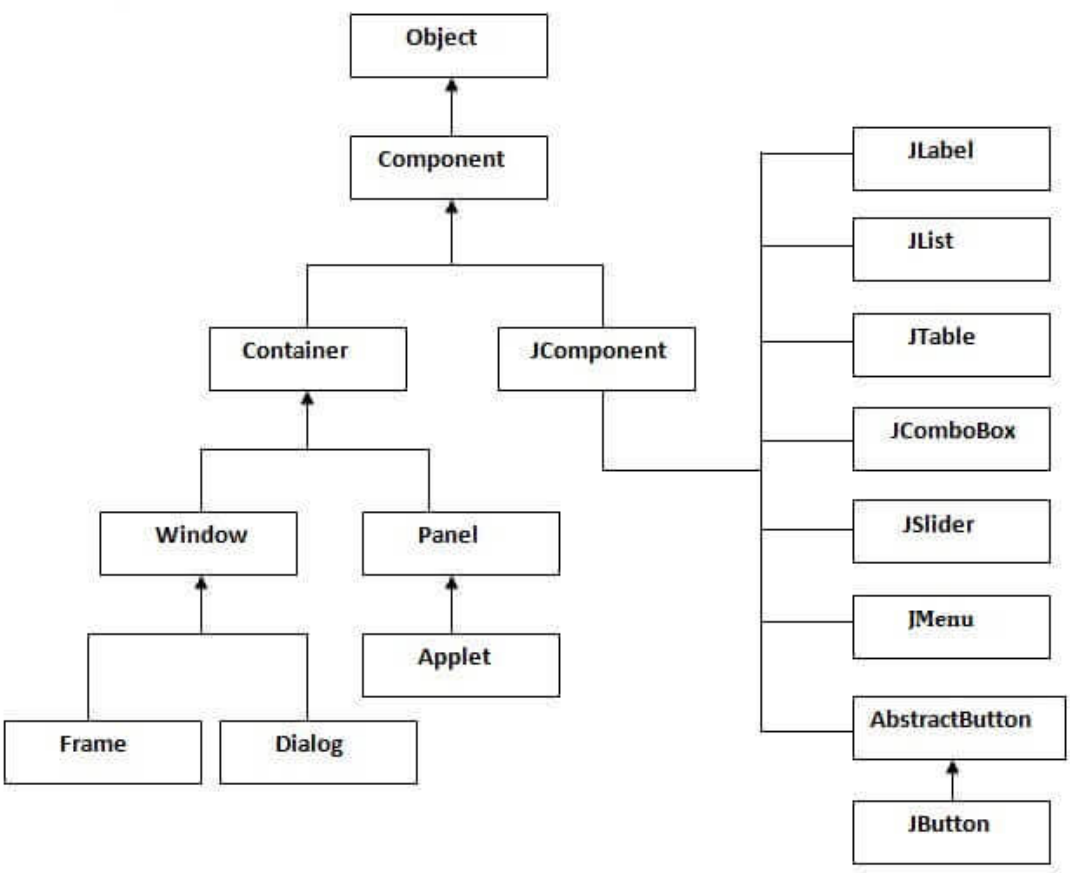
Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Swing

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.



Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

JLabel

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

Methods	Description
String getText()	t returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.
Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or null if no Action is set.
void setFont(Font f)	It is used to set the current font.
void removeActionListener(ActionListener l)	It is used to remove the specified action listener so that it no longer receives action events from this textfield.

Default Method

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

```
interface Sayable{

    default void say(){
        System.out.println("Hello, this is default method");
    }

    void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();
        dm.sayMore("this is definition of abstract method");

    }
}
```

Functional Interface

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

```
@FunctionalInterface
interface sayable{
   void say(String msg);
}
```

Lambda expression ->

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in the collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has a functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so the compiler does not create a .class file.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```

```
@FunctionalInterface
interface Drawable{
   public void draw();
}

public class LambdaExpressionExample {
   public static void main(String[] args) {
      int width=10;

      Drawable d2=()->{
         System.out.println("Drawing "+width);
      };
      d2.draw();
   }
}
```