

## UNIT-I

### HISTORY OF JAVA

**James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

Originally designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called "**Greentalk**" by James Gosling, and file extension was .gt.

After that, it was called **Oak** and was developed as a part of the Green project.

Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania, etc.

In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies.

Java is an island of Indonesia where first coffee was produced (called java coffee).

Notice that Java is just a name, not an acronym.

Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

Many java versions have been released till now. The current stable release of Java is Java SE 10.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th March 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th March 2018)

## FEATURE OF JAVA

The primary objective of [Java programming](#) language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*

A list of most important features of Java language is given below

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

### Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, 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.

### Object-oriented

Java is an [object-oriented](#) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism

5. Abstraction
6. Encapsulation

## Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

## Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**

## Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

## Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

## Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

## High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

## Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

## Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

## Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

## JVM (JAVA VIRTUAL MACHINE)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

Jvm is:

- **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
- **An implementation** Its implementation is known as JRE (Java Runtime Environment).
- **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

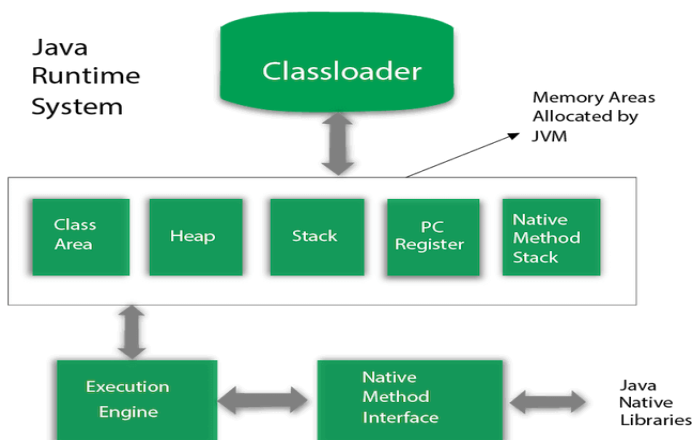
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

### JVM Architecture

It contains classloader, memory area, execution engine etc.



## ClassLoader

ClassLoader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader:** It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader:** It loads the jar files located inside *\$JAVA\_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader:** It loads the classfiles from classpath. By default, classpath is set to current directory

## Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

## Heap

It is the runtime data area in which objects are allocated.

## Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

## Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

## Native Method Stack

It contains all the native methods used in the application.

## Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

## Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

## 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:

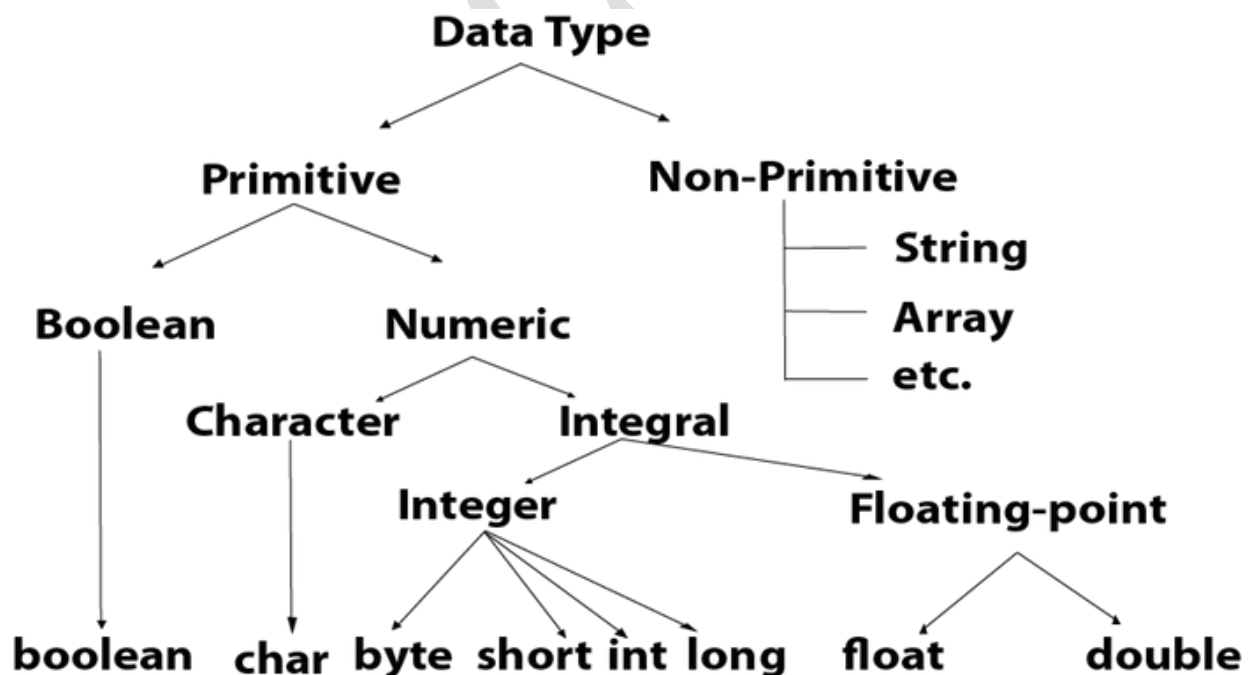
1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

### Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- ❖ boolean data type
- ❖ byte data type
- ❖ char data type
- ❖ short data type
- ❖ int data type
- ❖ long data type
- ❖ float data type
- ❖ double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

### Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

**Example:** Boolean one = false

### Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

**Example:** byte a = 10, byte b = -20

### Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

**Example:** short s = 10000, short r = -5000

### Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 ( $-2^{31}$ ) to 2,147,483,647 ( $2^{31} - 1$ ) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

**Example:** int a = 100000;

int b = -200000;



## Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808( $-2^{63}$ ) to 9,223,372,036,854,775,807( $2^{63}-1$ )(inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

**Example:** long a = 100000L, long b = -200000L

## Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

**Example:** float f1 = 234.5f

## Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

**Example:** double d1 = 12.3

## Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

**Example:** char letterA = 'A'

## TYPE CASTING

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size

byte -> short -> char -> int -> long -> float -> double

Widening conversion takes place when two data types are automatically converted. This happens when:

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

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type

double -> float -> long -> int -> char -> short -> byte

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

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

## Conditional statements /branching statements

A programming language uses control statements to control the flow of execution of program based on certain conditions. These are used to alter the flow of execution and branch the statements based on the condition result.

### Java's Selection statements:

- **if**
- **if-else**
- **nested-if**
- **if-else-if**
- **switch-case**
- **jump – break, continue, return**

These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

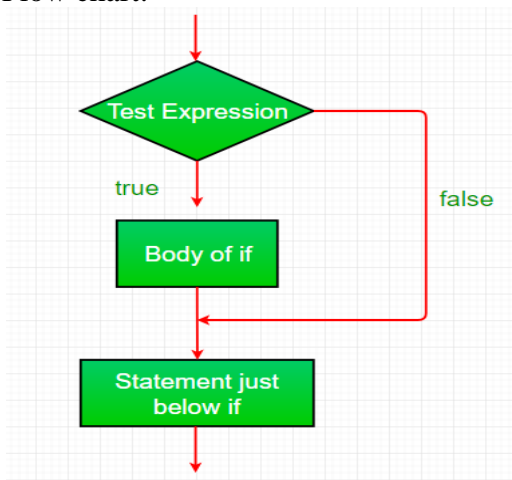
- **if:** if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

#### Syntax:

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

Here, **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.

Flow chart:



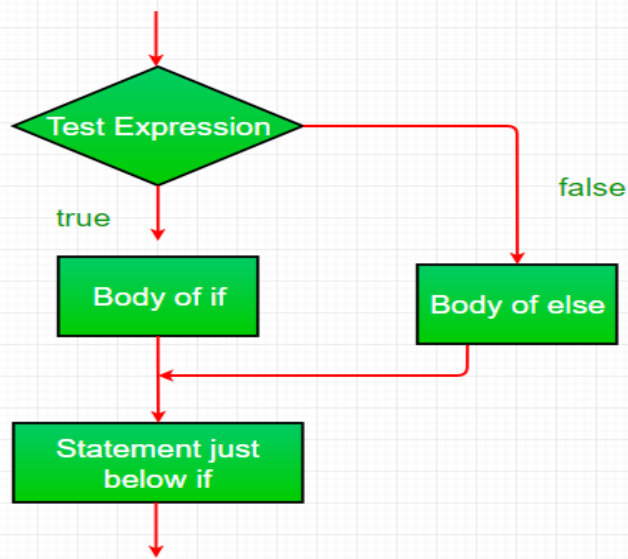
**if-else:** The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

**Syntax:**

```

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

```



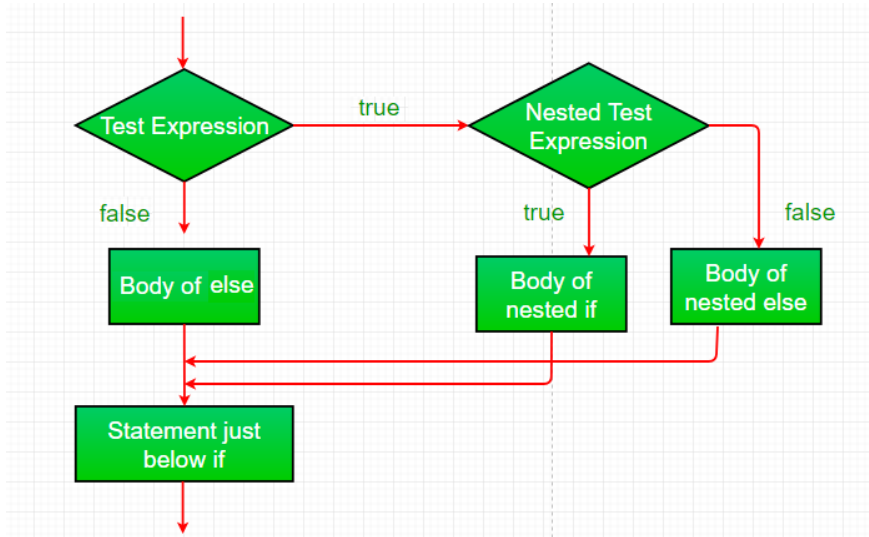
**nested-if:** A nested if is an if statement that is the target of another if or else. Nested if statements means an if statement inside an if statement. Java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

**Syntax:**

```

if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}

```



**if-else-if ladder:** Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

if (condition)

statement;

else if (condition)

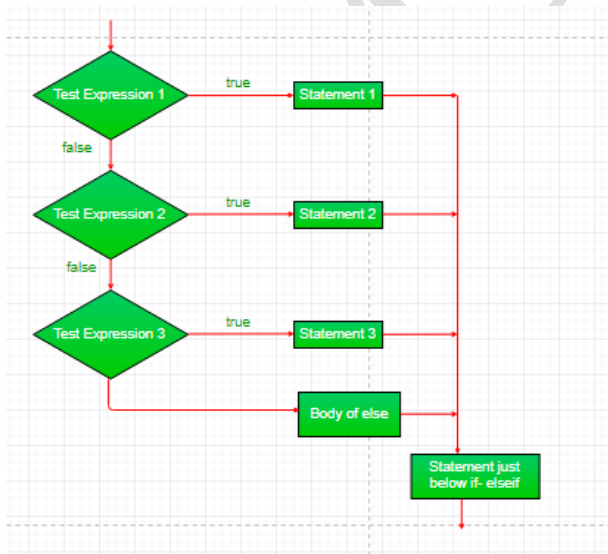
statement;

.

.

else

statement;



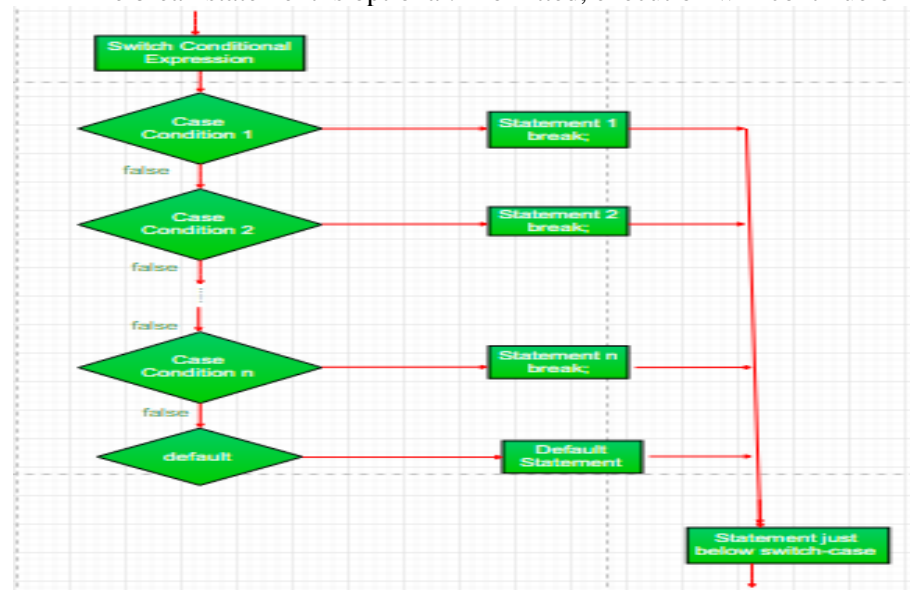
**switch-case** The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

**Syntax:**

```
switch (expression)
```

```
{
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    .
    .
    case valueN:
        statementN;
        break;
    default:
        statementDefault;
}
```

- Expression can be of type byte, short, int, char or an enumeration., *expression* can also be of type String.
- Duplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.



**jump:** Java supports three jump statement: **break**, **continue** and **return**. These three statements transfer control to other part of the program.

I. **Break:** In Java, break is majorly used for:

- Terminate a sequence in a switch statement (discussed above).
- To exit a loop.
- Used as a “civilized” form of goto.

#### Using break to exit a Loop

Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

Note: Break, when used inside a set of nested loops, will only break out of the innermost loop

- II. **Continue:** Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.
- III. **Return:** The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.

// Java program to illustrate using continue in an if statement

```
class ContinueDemo
```

```
{
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            // If the number is even
            // skip and continue
            if (i%2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```

**// Java program to illustrate using break to exit a loop**

**class BreakLoopDemo**

```
{  
    public static void main(String args[])  
    {  
        // Initially loop is set to run from 0-9  
        for (int i = 0; i < 10; i++)  
        {  
            // terminate loop when i is 5.  
            if (i == 5)  
                break;  
  
            System.out.println("i: " + i);  
        }  
        System.out.println("Loop complete.");  
    }  
}
```

# Loops

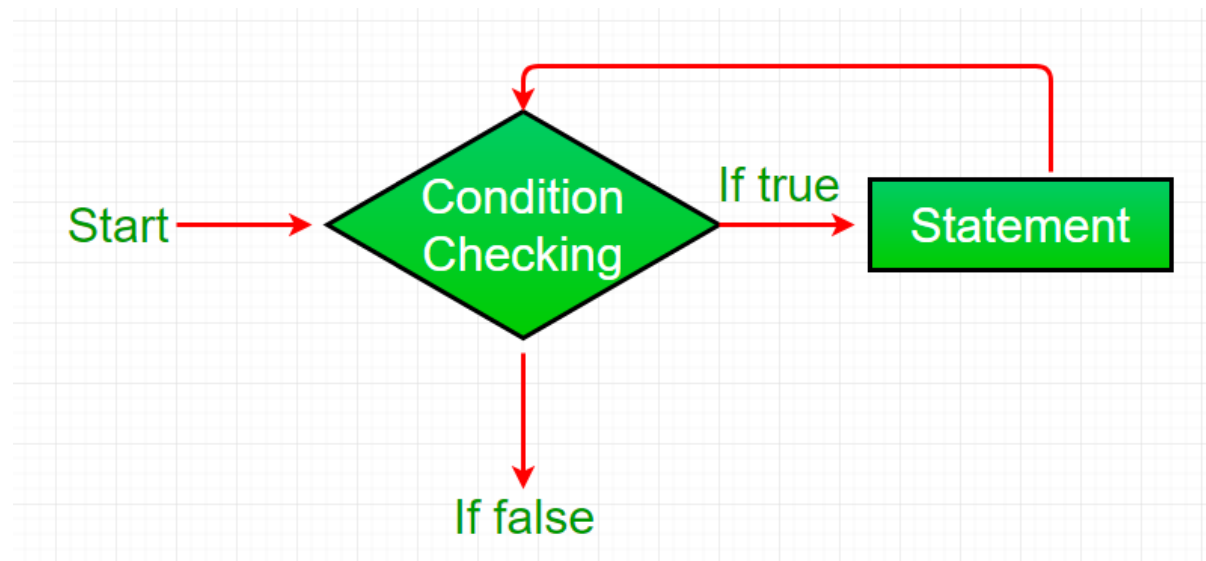
Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.

Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

1. **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement. **Syntax :**

```
while (boolean condition)
{
    loop statements...
}
```

Flowchart:



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

// Java program to illustrate while loop

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



```

int x = 1;

// Exit when x becomes greater than 4
while (x <= 4)
{
    System.out.println("Value of x:" + x);

    // Increment the value of x for
    // next iteration
    x++;
}
}

```

**Output:**

```

Value of x:1
Value of x:2
Value of x:3
Value of x:4

```

**2. for loop:** for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

**Syntax:**

```

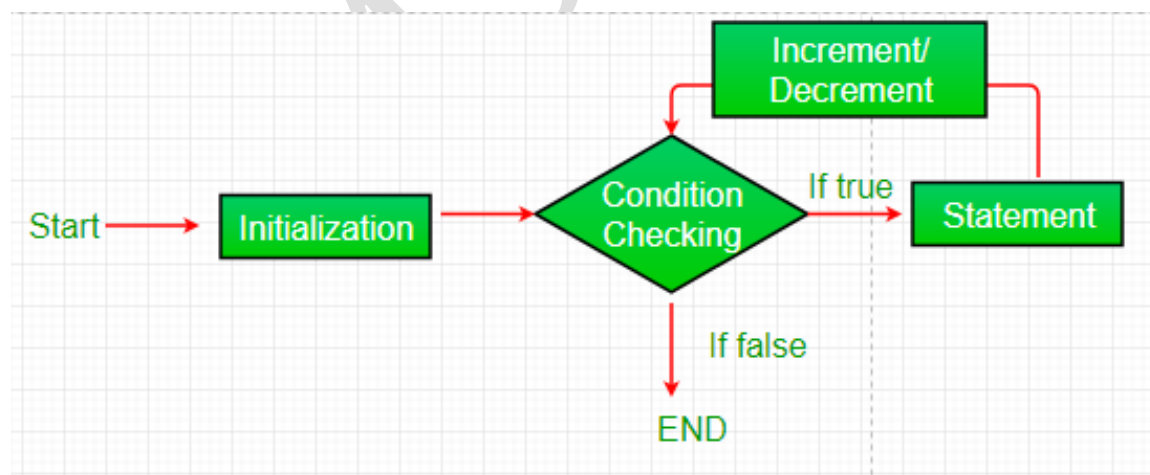
for (initialization condition; testing condition; increment/decrement)

```

```

{
    statement(s)
}

```



- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.

- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle.

// Java program to illustrate for loop.

```
class forLoopDemo
{
    public static void main(String args[])
    {
        // for loop begins when x=2
        // and runs till x <=4
        for (int x = 2; x <= 4; x++)
            System.out.println("Value of x:" + x);
    }
}
```

Value of x:2

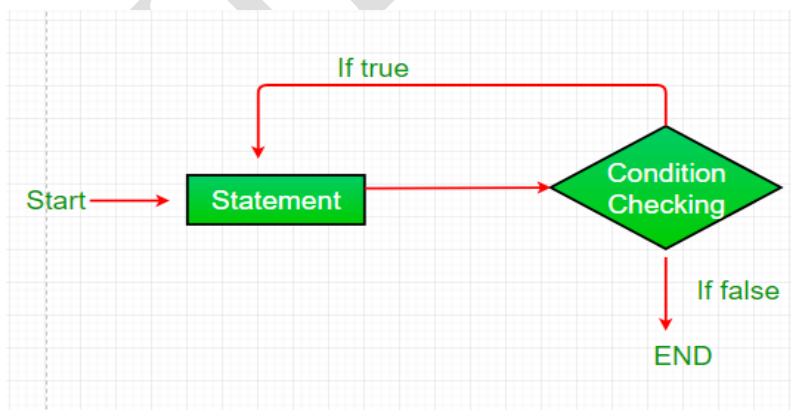
Value of x:3

Value of x:4

3. **do while:** do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop**.

#### Syntax:

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



- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

// Java program to illustrate do-while loop

```
class dowhileloopDemo
{
    public static void main(String args[])
    {
        int x = 21;
        do
        {
            // The line will be printed even if the condition is false
            System.out.println("Value of x:" + x);
            x++;
        } while (x < 20);
    }
}
```

#### Output:

Value of x:21

## CLASSES AND OBJECTS

A **class** is a **blueprint or prototype** that defines the variables and the methods (functions) common to all objects of a certain kind. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

A class in Java can contain:

- ❖ **Fields**
- ❖ **Methods**
- ❖ **Constructors**
- ❖ **Blocks**
- ❖ **Nested class and interface**

## Syntax to declare a class

```

1  :class <class_name>{
2      field;
3      method;
    }

```

## Objects

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

### Object Definitions:

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

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

## Object and Class Example

```

class Student{
    int id;
    String name;
}
class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="LAXMINARAYANA";
        System.out.println(s1.id+" "+s1.name);//printing members with a white space
    }
}

```

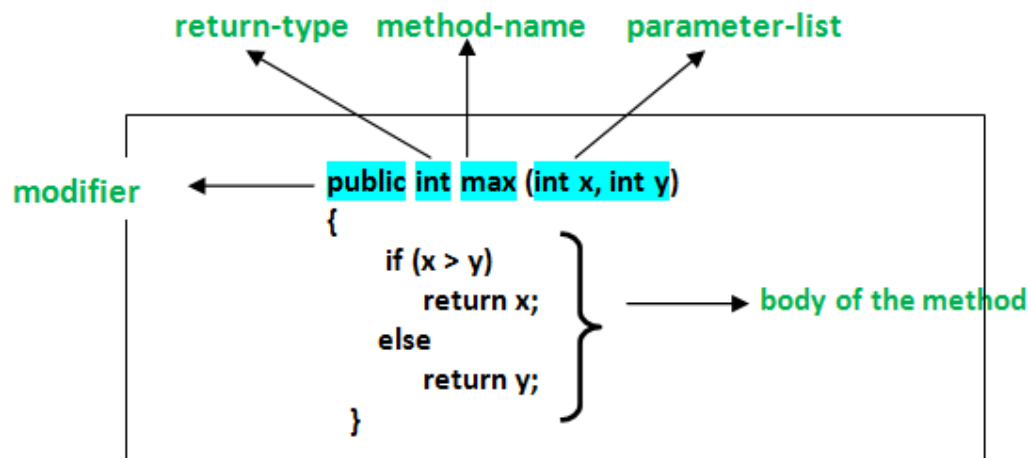
## Methods

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class

Methods are **time savers** and help us to **reuse** the code without retyping the code

In general, method declarations has six components :

- **Modifier-**: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
  - ❖ public: accessible in all class in your application.
  - ❖ protected: accessible within the class in which it is defined and in its **subclass(es)**
  - ❖ private: accessible only within the class in which it is defined.
  - ❖ default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- **The return type** : The data type of the value returned by the method or void if does not return a value.
- **Method Name** : the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list** : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list** : The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body** : it is enclosed between braces. The code you need to be executed to perform your intended



**Method signature:** It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.

Method Signature of above function:

```
max(int x, int y)
```

## Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;. We pass parameters in this parenthesis to Method. A can be called any number of times. To call a method which is member of another class, we use . operator along with object.

```
public class MyClass {
    static void myMethod() {
        System.out.println("I just got executed!");
    }

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

## Method Overloading

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

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

Method overloading *increases the readability of the program*.

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

**// Java program to demonstrate working of method overloading in Java.**

```
public class Sum {

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

    // Overloaded sum(). This sum takes three int parameters
```

```
public int sum(int x, int y, int z)
{
    return (x + y + z);
}

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

// Driver code
public static void main(String args[])
{
    Sum s = new Sum();
    System.out.println(s.sum(10, 20));
    System.out.println(s.sum(10, 20, 30));
    System.out.println(s.sum(10.5, 20.5));
}
}
```

**Output :**

**30**

**60**

**31.0**

## UNIT-II

### Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

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

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

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

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

Rules 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

### Types of Java constructors

There are two types of constructors in Java:

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

**Default Constructor:** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor. Default constructor provides the default values to the object like 0, null, etc. depending on the type..

#### Syntax of default constructor:

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

```
// Java Program to illustrate calling a no-argument constructor
```

```
import java.io.*;
```



```

class MyClass
{
    int num;
    String name;
    // this would be invoked while an object of that class is created.
    MyClass()
    {
        System.out.println("Constructor called");
    }
}
Class Test
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        MyClass m = new MyClass();
        // Default constructor provides the default values to the object like 0, null
        System.out.println(geek1.name);
        System.out.println(geek1.num);
    }
}

```

Constructor called

null

0

**Parameterized Constructor:** A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

```

class Student{
    int id;
    String name;
    //creating a parameterized constructor
    Student(int i,String n){
        id = i;
        name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
        //creating objects and passing values
        Student s1 = new Student(111,"LAXMAN");
    }
}

```

```

Student s2 = new Student(222,"RAMU");
//calling method to display the values of object
s1.display();
s2.display();
}
}

```

## Constructor Overloading

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

//Java program to overload constructors

```

class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
        id = i;
        name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
        Student5 s1 = new Student5(111,"Laxman");
        Student5 s2 = new Student5(222,"Ramu",25);
        s1.display();
        s2.display();
    }
}

```

### Output:

```

111 Laxman 0
222 Ramu 25

```

## Cleaning up unused objects

Java lacks a destructor element, and instead uses a garbage collector for resource deallocation.

- **Garbage Collector:** The garbage collector is a program that runs on the JVM and recovers memory by **deleting objects that are not used anymore or are not accessible from the code** (and are considered garbage, hence the name). It runs automatically and periodically checks the references in contrast to the objects in the memory heap. If an unreferenced object is found, that means that there is no way to access it anymore and it is useless, so the garbage collector gets rid of it and frees the memory.

## static keyword

The **static keyword** in Java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

### Java static variable

If you declare any variable as static, it is known as a static variable.

- ❖ The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- ❖ The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

//Java Program to demonstrate the use of static variable

```
class Student{
    int rollno;//instance variable
    String name;
    static String college ="MIMS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
```

```
void display () {System.out.println(rollno+" "+name+" "+college);}
}
```

//Test class to show the values of objects

```
public class TestStaticVariable1 {
    public static void main(String args[]){
        Student s1 = new Student(111,"Laxman");
        Student s2 = new Student(222,"Chinna");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}
```

**Output:**

```
111 Laxman MIMS
222 Chinna MIMS
```

### Java static method

If you apply static keyword with any method, it is known as static method.

- ❖ A static method belongs to the class rather than the object of a class.
- ❖ A static method can be invoked without the need for creating an instance of a class.
- ❖ A static method can access static data member and can change the value of it.

//Java Program to demonstrate the use of a static method.

```
class Student{
    int rollno;
    String name;
    static String college = "MIMS";
    //static method to change the value of static variable
    static void change(){
        college = "MIMSDC";
    }
    //constructor to initialize the variable
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
```

```

}
//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
        Student.change();//calling change method
        //creating objects
        Student s1 = new Student(111,"Laxman");
        Student s2 = new Student(222,"Ramu");
        Student s3 = new Student(333,"Chinna");
        //calling display method
        s1.display();
        s2.display();
        s3.display();
    }
}

```

**Output:**

```

111 Laxman MIMSDC
222 Ramu MIMSDC
333 Chinna MIMSDC

```

**Java static block**

- ❖ Is used to initialize the static data member.
- ❖ It is executed before the main method at the time of classloading.

```

class A2{
    static{System.out.println("static block is invoked");}
    public static void main(String args[]){
        System.out.println("Hello main");
    }
}

```

**Output:**

```

static block is invoked
Hello main

```

## This keyword

this is a **reference variable** that refers to the current object.

## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
    public static void main(String args[]){
        Student s1=new Student(111,"shiva",5000f);
        Student s2=new Student(112,"krishna",6000f);
        s1.display();
        s2.display();
    }
}
```

### Output:

```
111 shiva 5000
112 krishna 6000
```

## Arrays

an array is a collection of similar type of elements which have a contiguous memory location.

**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.

### Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

### Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

### Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

### Single Dimensional Array in Java

#### Syntax to Declare an Array in Java

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

#### Instantiation of an Array in Java

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

We can declare, instantiate and initialize the java array together by:

```
1 int a[]={33,3,4,5}; //declaration, instantiation and initialization
```

### Multidimensional Array in Java

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

**Syntax to Declare Multidimensional Array in Java**

```

dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];

```

**Example to instantiate Multidimensional Array in Java**

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

**Example to initialize Multidimensional Array in Java**

```

arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;

```

//Java Program to multiply two matrices

```

public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};
//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3]; //3 rows and 3 columns
//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++){
{
c[i][j]+=a[i][k]*b[k][j];
} //end of k loop
System.out.print(c[i][j]+" "); //printing matrix element
} //end of j loop
System.out.println();//new line
}
}}

```

**Output:**

```

6 6 6
12 12 12
18 18 18

```



## Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass N(1,2,3 and so on) numbers of arguments from the command prompt.

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

```
for(int i=0;i<args.length;i++)
System.out.println(args[i]);
```

```
}
}
```

compile by > javac A.java

run by > java A hi hello Namaste adab abc

**output:**

```
hi
hello
Namaste
Adab
abc
```

## Inner Classes

**Java inner class** or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

*Syntax of Inner class*

```
class Java_Outer_class{
//code
class Java_Inner_class{
//code
}
}
```

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization:** It requires less code to write.

**Example:**

```
class Outer {  
    // Simple nested inner class  
    class Inner {  
        public void show() {  
            System.out.println("In a nested class method");  
        }  
    }  
}  
class Main {  
    public static void main(String[] args) {  
        Outer.Inner in = new Outer().new Inner();  
        in.show();  
    }  
}
```

**Output:**

In a nested class method

- we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself.
- Inner class can be declared within a method of an outer class
- Static nested classes are not technically an inner class. They are like a static member of outer class.
- Anonymous inner classes are declared without any name at all.

## Inheritance

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

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

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

### Advantages of inheritance in java

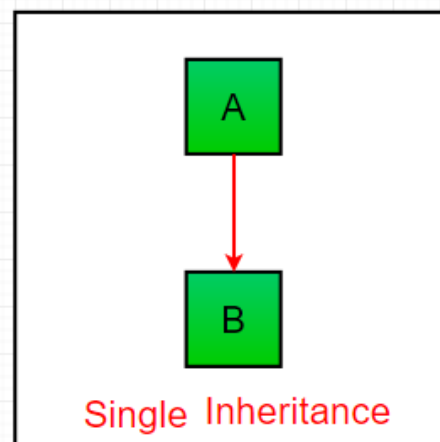
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

### Terms used in Inheritance

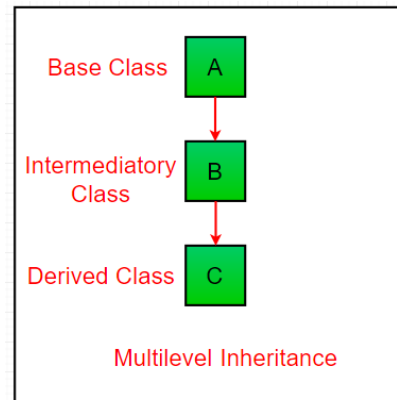
- ❖ **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- ❖ **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- ❖ **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- ❖ **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### Types of Inheritance

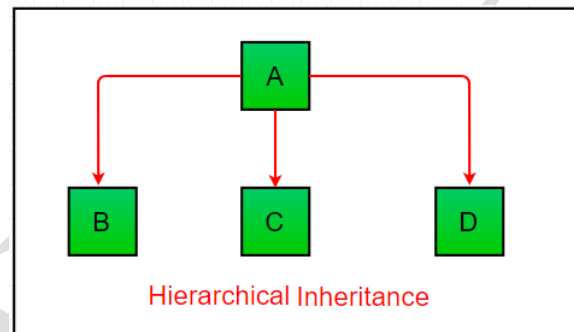
**Single Inheritance :** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



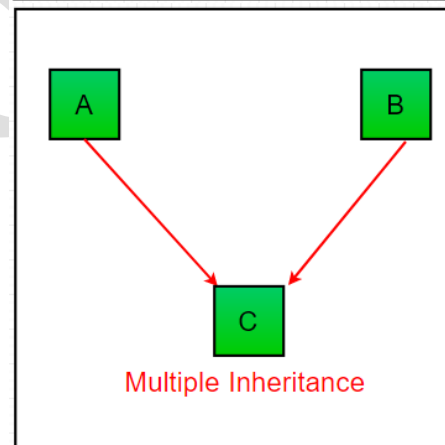
**Multilevel Inheritance :** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



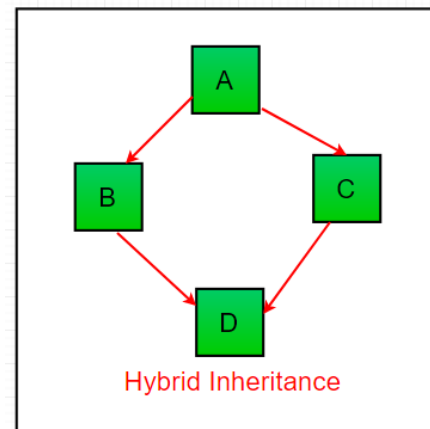
**Hierarchical Inheritance :** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B, C and D.



**Multiple Inheritance (Through Interfaces) :** In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



**Hybrid Inheritance(Through Interfaces) :** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



## extends Keyword

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

### Syntax

```
class Super {
    ....
    ....
}
class Sub extends Super {
    ....
    ....
}
```

Example:

```
class Calculation {
    int z;
    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }
    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}
```

OUTPUT:

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

## Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

// A Simple Java program to demonstrate method overriding in java

```
// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Parent type reference refers to a Parent object, then
        //Parent's show is called
        Parent obj1 = new Parent();
    }
}
```

```

        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}

```

**Output:**

```

Parent's show()
Child's show()

```

**Super Keyword**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

**Usage of Java super Keyword**

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

```

class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}

class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5{
    public static void main(String[] args){
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
    }
}

```

## Final keyword

In java final is a keyword or reserved word and can be applied to variables, methods, classes etc. The reason behind final keyword is to make entity non modifiable. It means when you make a variable or class or method as final you are not allowed to change that variable or class or method and compiler will verify this and gives compilation error if you try to re-initialized final variables in java.

The final keyword in java is used to give restriction to the user. The java final keyword can be used in many contexts. Final can be:

- variable
- method
- class

### Final Variable

Any variable which is declared by using the final keyword is called final variable. Final variables can be declare with static keyword in java and treated as constant. A final variable can only be explicitly assigned once.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

```
class FinalVariable1
{
    public static void main(String args[])
    {

        final int i = 30;

        i = 60;

    }
}
```

### Output:

Compiler Error: cannot assign a value to final variable i

### Final method

Sometimes we may want to prevent a childclass to overriding a method from parentclass. To do this we use final keyword with method declaration. It means a method with final keyword is called final method. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time.

The main reason behind making a method final would be that the content of the method should not be changed by any outsider.

### Program Example of Final Method:

Let us take an example to understand the use of final method.



**Step 1:** First we create a class **X** in which we declare the final method **getMethod()**:

```
class X
{
    final void getMethod()
    {
        System.out.println("X method has been called");
    }
}
```

**Step 2:** Second we create a class **Y** which extends the class **X** and here we try to override the method of above class:

```
class Y extends X
{
    void getMethod() //cannot override
    {
        System.out.println("Y method has been called");
    }
}
```

**Step 3:** Third we create a class **FinalMethod** in which we create an object of class **Y**:

```
class FinalMethod
{
    public static void main(String[] args)
    {
        Y obj = new Y();
        obj.getMethod();
    }
}
```

### Explanation:

On compiling the above program, it will display an compilation error "getMethod() in Y cannot override getMethod() in X; overridden method is final."

### Final Class

A class with final keyword is known as final class in java. Final class is complete in nature and cannot be inherited. Several classes in Java are final e.g. String, Integer and other wrapper classes.

The main purpose or reason of using a final class is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

### Program Example of Final Class:

Let us take a program example to show the use of final class.

**Step 1:** First we create a class **X** and make it final class by using final keyword:

```
final class X
```

```
{  
  //properties and methods of class X  
}
```

**Step 2:** Second we create a class **Y** which is trying to extend final class **X**:

```
class Y extends X  
{  
  //properties and methods of class Y  
}
```

**Step 3:** Third we create a class **FinalClass**:

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

**Output:**

Compiler Error: cannot inherit from final X

## Abstraction in Java

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

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

## Abstract class in Java

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.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

### Example of abstract class

```
abstract class A{ }
```

### Abstract Method

A method which is declared as abstract and does not have implementation is known as an abstract method.

### Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

### Example Program

```
abstract class Shape{
    abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
    public static void main(String args[]){
        Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
        s.draw();
    }
}
```

### Interfaces

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

An **interface** is a completely "**abstract class**" that is used to group related methods with empty bodies:

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. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**. It cannot be instantiated just like the abstract class.

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

```
interface <interface_name>{

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

### Example

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class A7 implements Printable,Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
        A7 obj = new A7();
        obj.print();
        obj.show();
    }
}
```

Output:Hello  
Welcome

## abstract class Vs interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance</b> .	Interface <b>supports multiple inheritance</b> .
3) Abstract class <b>can have final, non-final, static and non-static variables</b> .	Interface has <b>only static and final variables</b> .
4) Abstract class <b>can provide the implementation of interface</b> .	Interface <b>can't provide the implementation of abstract class</b> .
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.
6) An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface only.
7) An <b>abstract class</b> can be extended using keyword "extends".	An <b>interface</b> can be implemented using keyword "implements".
8) A Java <b>abstract class</b> can have class members like private, protected, etc.	Members of a Java interface are public by default.

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

## Packages

**Package** in [Java](#) is a mechanism to encapsulate a group of classes, sub packages and interfaces

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

Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

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.

### Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

- 1) **java.lang**: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
- 2) **java.io**: Contains classed for supporting input / output operations.
- 3) **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- 4) **java.applet**: Contains classes for creating Applets.
- 5) **java.awt**: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
- 6) **java.net**: Contain classes for supporting networking operations.

### User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
    public static void main(String args[])
    {
        // Initializing the String variable with a value
        String name = "Laxminarayana";

        // Creating an instance of class MyClass in the package.
        MyClass obj = new MyClass();

        obj.getNames(name);
    }
}
```

## Access Protection

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

**Default:** When no access modifier is specified for a class, method or data member – It is said to be having the **default** access modifier by default.

- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

**Private:** The private access modifier is specified using the keyword **private**.

- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of same package will not be able to access** these members.
- Top level Classes or interface can not be declared as private because
  1. private means “only visible within the enclosing class”.
  2. protected means “only visible within the enclosing class and any subclasses”

**protected:** The protected access modifier is specified using the keyword **protected**.

- The methods or data members declared as protected are **accessible within same package or sub classes in different package**.

**public:** The public access modifier is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods or data members which are declared as public are **accessible from every where** in the program. There is no restriction on the scope of a public data members.

## Wrapper classes

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

**autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	<a href="#"><u>Boolean</u></a>
char	<a href="#"><u>Character</u></a>
byte	<a href="#"><u>Byte</u></a>
short	<a href="#"><u>Short</u></a>
int	<a href="#"><u>Integer</u></a>
long	<a href="#"><u>Long</u></a>
float	<a href="#"><u>Float</u></a>
double	<a href="#"><u>Double</u></a>



## 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.

we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.

### Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
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);
}}
```

OUTPUT:  
20 20 20

## Unboxing

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.

### Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
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);
}}
Output:
3 3 3
```

## String class

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

### Creating a String

There are two ways to create string in Java:

#### String literal

```
String s = "mims";
```

#### Using new keyword

```
String s = new String ("mims");
```

### String Methods

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

```
"mims degree college".length(); // returns 19
```

**Char charAt(int i):** Returns the character at ith index.

```
"mimscollege".charAt(3); // returns 's'
```

**String substring (int i):** Return the substring from the ith index character to end.

```
"mimscollege".substring(3); // returns "scollege"
```

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

```
"mimscollege".substring(2, 5); // returns "msc"
```

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

```
String s1 = "mims";
```

```
String s2 = "college";
```

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

**int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

```
String s = "welcome to java programing";
```

```
int output = s.indexOf("java"); // returns 11
```

**int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "welcome to java programing";
```

```
int output = s.indexOf("em",3); // returns 18
```

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

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

```
Boolean out = "Mims".equals("mims"); // returns false
```

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

```
Boolean out= "Mims".equalsIgnoreCase("Mims"); // returns true
```

```
Boolean out = "Mims".equalsIgnoreCase("mims"); // returns true
```

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

```
int out = s1.compareTo(s2); // where s1 and s2 are strings to be compared
```

This returns difference s1- s2.

If : out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

**int compareToIgnoreCase( String anotherString):** Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareToIgnoreCase(s2); // where s1 ans s2 are strings to be compared
```

This returns difference s1-s2.

If : out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

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

```
String word1 = "HeLLo";
```

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

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

```
String word1 = "HeLLo";
```

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

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

```
String word1 = " mims college ";
```

```
String word2 = word1.trim(); // returns "mims college"
```

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

```
String s1 = "degree college";
```

```
String s2 = s1.replace('e', 'k'); // returns "dkgrkk collkgk"
```

## StringBuffer class

**StringBuffer** is a peer class of **String** that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

### Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

### Methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

## UNIT-III

### Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

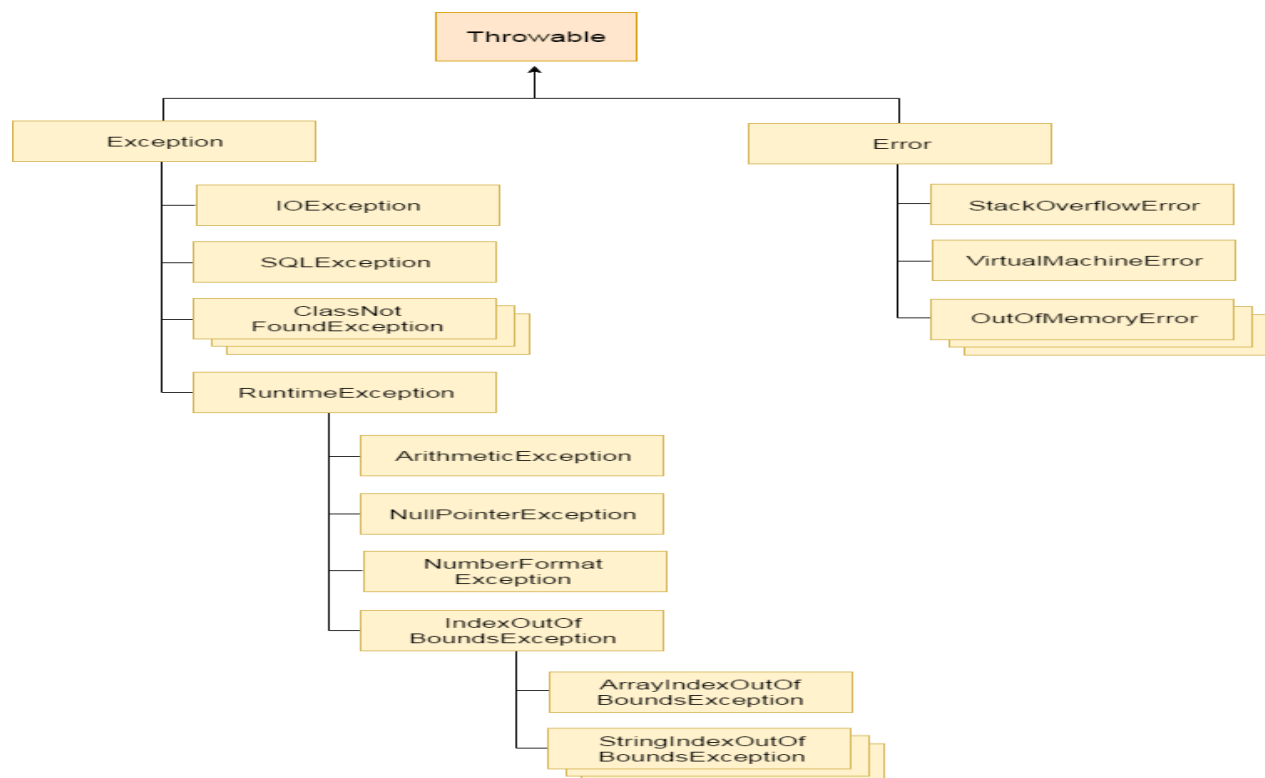
- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

### Exception Hierarchy

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.

`NullPointerException` is an example of such an exception. Another branch, **Error** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). `StackOverflowError` is an example of such an error.



## Types of exceptions

There are three types of Exceptions

- **Checked exceptions** – A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

- **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6<sup>th</sup> element of the array then an *ArrayIndexOutOfBoundsException* occurs.

- **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## Exception Handling

### Try Catch Block

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs. If an error occurs, we can use **try...catch** to catch the error and execute some code to handle it.

The **finally** statement lets you execute code, after **try...catch**, regardless of the result

### Syntax

```
try {
    // Block of code to try
}
catch(Exception e) {
    // Block of code to handle errors
}
Finally {
    // Block of code to
}
```

**Try catch example program**

```
public class MyClass {  
  
    public static void main(String[] args) {  
  
        try {  
  
            int[] myNumbers = { 1, 2, 3 };  
  
            System.out.println(myNumbers[10]);  
  
        } catch (Exception e) {  
  
            System.out.println("Something went wrong.");  
  
        } finally {  
  
            System.out.println("The 'try catch' is finished.");  
  
        }  
  
    }  
  
}
```

**The output will be:**

Something went wrong.  
The 'try catch' is finished.

**Java throws keyword**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

**Syntax of java throws**

```
return_type method_name() throws exception_class_name{  
    //method code  
}
```



Example of throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n()throws IOException{
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:  
exception handled  
normal flow...

## Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

### Syntax:

```
throw exception;
```

Example of throw IOException.

```
throw new IOException("sorry device error);
```

### Example Program using throw keyword

```
public class TestThrow1 {
    static void validate(int age){
        if(age<18)
            throw new ArithmeticException("not valid");
        else
            System.out.println("welcome to vote");
    }
    public static void main(String args[]){
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

### User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes –

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below –

```
class MyException extends Exception {
```

```
// class code
}
```

### Example Program

```
class JavaException{
    public static void main(String args[]){
        try{
            throw new MyException(2);
            // throw is used to create a new exception and throw it.
        }
        catch(MyException e){
            System.out.println(e) ;
        }
    }
}
```

```

class MyException extends Exception{
    int a;
    MyException(int b) {
        a=b;
    }
    public String toString(){
        return ("Exception Number = "+a) ;
    }
}

```

Output:

Exception Number = 2

## Multithreading

**Multithreading in java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

### Advantages

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- 2) You **can perform many operations together, so it saves time**.
- 3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

### Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

### Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	<a href="#"><u>start()</u></a>	It is used to start the execution of the thread.
2)	void	<a href="#"><u>run()</u></a>	It is used to do an action for a thread.
3)	static void	<a href="#"><u>sleep()</u></a>	It sleeps a thread for the specified amount of time.

4)	static Thread	<a href="#"><u>currentThread()</u></a>	It returns a reference to the currently executing thread object.
5)	void	<a href="#"><u>join()</u></a>	It waits for a thread to die.
6)	int	<a href="#"><u>getPriority()</u></a>	It returns the priority of the thread.
7)	void	<a href="#"><u>setPriority()</u></a>	It changes the priority of the thread.
8)	String	<a href="#"><u>getName()</u></a>	It returns the name of the thread.
9)	void	<a href="#"><u>setName()</u></a>	It changes the name of the thread.
10)	long	<a href="#"><u>getId()</u></a>	It returns the id of the thread.
11)	boolean	<a href="#"><u>isAlive()</u></a>	It tests if the thread is alive.
12)	static void	<a href="#"><u>yield()</u></a>	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	<a href="#"><u>suspend()</u></a>	It is used to suspend the thread.
14)	void	<a href="#"><u>resume()</u></a>	It is used to resume the suspended thread.
15)	void	<a href="#"><u>stop()</u></a>	It is used to stop the thread.
16)	void	<a href="#"><u>destroy()</u></a>	It is used to destroy the thread group and all of its subgroups.
17)	void	<a href="#"><u>interrupt()</u></a>	It interrupts the thread.
18)	void	<a href="#"><u>notify()</u></a>	It is used to give the notification for only one thread which is waiting for a particular object.
19)	void	<a href="#"><u>notifyAll()</u></a>	It is used to give the notification to all waiting threads of a particular object.

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.

// Java code for thread creation by extending the Thread class

```
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() +
                               " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
```

```

        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}

```

### Output

```

Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running

```

### 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.

// Java code for thread creation by implementing the Runnable Interface

class MultithreadingDemo implements Runnable

```

{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

```

```
    }  
    }  
}  
  
// Main Class  
class Multithread  
{  
    public static void main(String[] args)  
    {  
        int n = 8; // Number of threads  
        for (int i=0; i<8; i++)  
        {  
            Thread object = new Thread(new MultithreadingDemo());  
            object.start();  
        }  
    }  
}
```

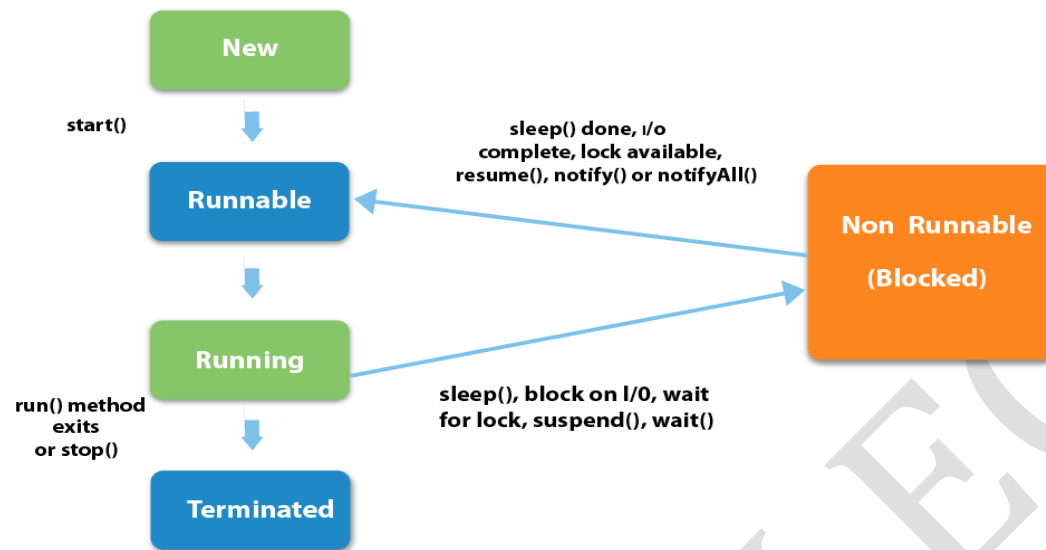
Output :

Thread 8 is running  
Thread 9 is running  
Thread 10 is running  
Thread 11 is running  
Thread 12 is running  
Thread 13 is running  
Thread 14 is running  
Thread 15 is running

## Life cycle of a Thread

There are five thread states in java

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



### 1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

## Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN\_PRIORITY



2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

### Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
    public void run(){
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
    public static void main(String args[]){
        TestMultiPriority1 m1=new TestMultiPriority1();
        TestMultiPriority1 m2=new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

Output:

```
running thread name is:Thread-0
    running thread priority is:10
    running thread name is:Thread-1
    running thread priority is:1
```

## Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Synchronization is achieved by three ways .

1. by synchronized method
2. by synchronized block
3. by static synchronization

### Java 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.

//example of java synchronized method

```
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;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(5);
    }
}

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

```
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output: 5

10  
15  
20  
25  
100  
200  
300  
400  
500

### **Synchronized block in java**

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### **Syntax to use synchronized block**

```
synchronized (object reference expression) {
//code block
}
```

### **Program of synchronized block**

```
class Table{

void printTable(int n){
    synchronized(this){//synchronized block
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }
}
}
//end of the method
}
```

```
class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}
```

```
class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(100);
    }
}
```

```
public class TestSynchronizedBlock1 {
    public static void main(String args[]){
```

```
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Test it Now

Output:5

```
10
15
20
25
100
200
300
400
500
```

## Java IO Stream

Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

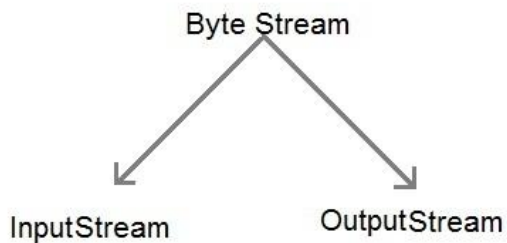
Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream** : It provides a convenient means for handling input and output of byte.
2. **Character Stream** : It provides a convenient means for handling input and output of characters.

Character stream uses Unicode and therefore can be internationalized.

## Java Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are `InputStream` and `OutputStream`.



*Some important Byte stream classes.*

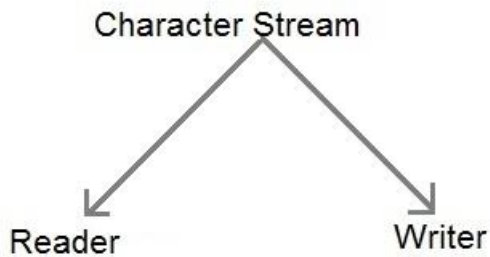
Stream class	Description
<b>BufferedInputStream</b>	Used for Buffered Input Stream.
<b>BufferedOutputStream</b>	Used for Buffered Output Stream.
<b>DataInputStream</b>	Contains method for reading java standard datatype
<b>DataOutputStream</b>	An output stream that contain method for writing java standard data type
<b>FileInputStream</b>	Input stream that reads from a file
<b>FileOutputStream</b>	Output stream that write to a file.
<b>InputStream</b>	Abstract class that describe stream input.
<b>OutputStream</b>	Abstract class that describe stream output.
<b>PrintStream</b>	Output Stream that contain <b>print()</b> and <b>println()</b> method

These classes define several key methods. Two most important are

1. **read()** : reads byte of data.
2. **write()** : Writes byte of data.

## Java Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



Some important Character stream classes

Stream class	Description
<b>BufferedReader</b>	Handles buffered input stream.
<b>BufferedWriter</b>	Handles buffered output stream.
<b>FileReader</b>	Input stream that reads from file.
<b>FileWriter</b>	Output stream that writes to file.
<b>InputStreamReader</b>	Input stream that translate byte to character
<b>OutputStreamReader</b>	Output stream that translate character to byte.
<b>PrintWriter</b>	Output Stream that contain <b>print()</b> and <b>println()</b> method.
<b>Reader</b>	Abstract class that define character stream input
<b>Writer</b>	Abstract class that define character stream output

## Java.io.File Class

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

```
File a = new File("/usr/local/bin/mims");
```

### Methods

1. **boolean canExecute()** : Tests whether the application can execute the file denoted by this abstract pathname.
2. **boolean canRead()** : Tests whether the application can read the file denoted by this abstract pathname.
3. **boolean canWrite()** : Tests whether the application can modify the file denoted by this abstract pathname.
4. **int compareTo(File pathname)** : Compares two abstract pathnames lexicographically.
5. **boolean createNewFile()** : Atomically creates a new, empty file named by this abstract pathname .
6. **boolean delete()** : Deletes the file or directory denoted by this abstract pathname.
7. **boolean equals(Object obj)** : Tests this abstract pathname for equality with the given object.
8. **boolean exists()** : Tests whether the file or directory denoted by this abstract pathname exists.
9. **String getAbsolutePath()** : Returns the absolute pathname string of this abstract pathname.
10. **long getFreeSpace()** : Returns the number of unallocated bytes in the partition .
11. **String getName()** : Returns the name of the file or directory denoted by this abstract pathname.
12. **String getParent()** : Returns the pathname string of this abstract pathname's parent.
13. **File getParentFile()** : Returns the abstract pathname of this abstract pathname's parent.
14. **String getPath()** : Converts this abstract pathname into a pathname string.
15. **boolean isDirectory()** : Tests whether the file denoted by this pathname is a directory.
16. **boolean isFile()** : Tests whether the file denoted by this abstract pathname is a normal file.
17. **boolean isHidden()** : Tests whether the file named by this abstract pathname is a hidden file.
18. **long length()** : Returns the length of the file denoted by this abstract pathname.
19. **String[] list()** : Returns an array of strings naming the files and directories in the directory .
20. **File[] listFiles()** : Returns an array of abstract pathnames denoting the files in the directory.
21. **boolean mkdir()** : Creates the directory named by this abstract pathname.
22. **boolean renameTo(File dest)** : Renames the file denoted by this abstract pathname.
23. **boolean setExecutable(boolean executable)** : A convenience method to set the owner's execute permission.
24. **boolean setReadable(boolean readable)** : A convenience method to set the owner's read permission.
25. **boolean setReadable(boolean readable, boolean ownerOnly)** : Sets the owner's or everybody's read permission.
26. **boolean setReadOnly()** : Marks the file or directory named so that only read operations are allowed.
27. **boolean setWritable(boolean writable)** : A convenience method to set the owner's write permission.
28. **String toString()** : Returns the pathname string of this abstract pathname.
29. **URI toURI()** : Constructs a file URI that represents this abstract pathname.



// In this program, we accept a file or directory name from command line arguments. Then the program  
 //will check if that file or directory physically exist or not and it displays the property of that file or  
 //directory.

```
*import java.io.File;
```

```
// Displaying file property
```

```
class fileProperty
```

```
{
```

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

```
        //accept file name or directory name through command line args
```

```
        String fname =args[0];
```

```
        //pass the filename or directory name to File object
```

```
        File f = new File(fname);
```

```
        //apply File class methods on File object
```

```
        System.out.println("File name :"+f.getName());
```

```
        System.out.println("Path: "+f.getPath());
```

```
        System.out.println("Absolute path:" +f.getAbsolutePath());
```

```
        System.out.println("Parent:"+f.getParent());
```

```
        System.out.println("Exists :"+f.exists());
```

```
        if(f.exists())
```

```
        {
```

```
            System.out.println("Is writeable:"+f.canWrite());
```

```
            System.out.println("Is readable"+f.canRead());
```

```
            System.out.println("Is a directory:"+f.isDirectory());
```

```
            System.out.println("File Size in bytes "+f.length());
```

```
        }
```

```
    }
```

```
}
```

### Output:

File name :file.txt

Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt

Parent:null

Exists :true

Is writeable:true

Is readabletrue

Is a directory:false

File Size in bytes 20

## FileInputStream Class

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.

Methods:

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>b.length</b> bytes of data from the input stream.

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

OUTPUT:

Welcome to javaTpoint

## FileOutputStream Class

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.

Methods

void write(byte[] ary)	It is used to write <b>ary.length</b> bytes from the byte <a href="#">array</a> to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write <b>len</b> bytes from the byte array starting at offset <b>off</b> to the file output stream.

void write(int b)

It is used to write the specified byte to the file output stream.

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
        try{
            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
            String s="Welcome to javaTpoint.";
            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);}
    }
}
```

## Scanner class

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

1)	BigInteger	<a href="#"><u>nextBigInteger()</u></a>	It scans the next token of the input as a BigInteger.
2)	boolean	<a href="#"><u>nextBoolean()</u></a>	It scans the next token of the input into a boolean value and returns that value.
3)	byte	<a href="#"><u>nextByte()</u></a>	It scans the next token of the input as a byte.
4)	double	<a href="#"><u>nextDouble()</u></a>	It scans the next token of the input as a double.

5)	float	<a href="#">nextFloat()</a>	It scans the next token of the input as a float.
6)	int	<a href="#">nextInt()</a>	It scans the next token of the input as an Int.
7)	String	<a href="#">nextLine()</a>	It is used to get the input string that was skipped of the Scanner object.
8)	long	<a href="#">nextLong()</a>	It scans the next token of the input as a long.
9)	short	<a href="#">nextShort()</a>	It scans the next token of the input as

```
import java.util.*;
public class ScannerExample {
    public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
    }
}
```

### Output:

Enter your name: sonoo jaiswal  
Name is: sonoo jaiswal

### Java BufferedInputStream Class

Java BufferedInputStream [class](#) is used to read information from [stream](#). It internally uses buffer mechanism to make the performance fast.

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer [array](#) is created.

### Methods

int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.

```

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

```

### BufferedOutputStream Class

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. the syntax for adding the buffer in an OutputStream:

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));
```

#### BufferedOutputStream class methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte <u>array</u> , starting with the given offset
void flush()	It flushes the buffered output stream.

```

import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
BufferedOutputStream bout=new BufferedOutputStream(fout);
String s="Welcome to javaTpoint.";
byte b[]=s.getBytes();
bout.write(b);
}
}

```

```

bout.flush();
bout.close();
fout.close();
System.out.println("success");
}
}

```

## RandomAccessFile

This [class](#) is used for reading and writing to random access file. A random access file behaves like a large [array](#) of bytes. There is a cursor implied to the array called file [pointer](#), by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is [thrown](#). It is a type of IOException.

### Constructors

<a href="#">Constructor</a>	Description
RandomAccessFile(File file, <a href="#">String</a> mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

### Methods

Modifier and Type	Method	Method
void	close()	It closes this random access file stream and releases any system resources associated with the stream.
FileChannel	getChannel()	It returns the unique <a href="#">FileChannel</a> object associated with this file.
int	readInt()	It reads a signed 32-bit integer from this file.
String	readUTF()	It reads in a string from this file.

void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

### Example

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
    static final String FILEPATH = "myFile.TXT";
    public static void main(String[] args) {
        try {
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I love my country and my people", 31);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static byte[] readFromFile(String filePath, int position, int size)
        throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
    }
}
```

```
return bytes;
}
private static void writeToFile(String filePath, String data, int position)
throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "rw");
    file.seek(position);
    file.write(data.getBytes());
    file.close();
}
}
```

## UNIT-IV

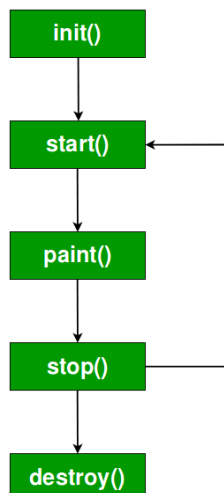
### Applet

An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.

Applets are used to make the web site more dynamic and entertaining.

- All applets are sub-classes (either directly or indirectly) of *java.applet.Applet* class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.
- Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

#### Life cycle of an applet :





It is important to understand the order in which the various methods shown in the above image are called. When an applet begins, the following methods are called, in this sequence:

1. `init()`
2. `start()`
3. `paint()`

When an applet is terminated, the following sequence of method calls takes place:

1. `stop()`
2. `destroy()`

1. **`init()`** : The **`init()`** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
2. **`start()`** : The **`start()`** method is called after **`init()`**. It is also called to restart an applet after it has been stopped. Note that **`init()`** is called once i.e. when the first time an applet is loaded whereas **`start()`** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **`start()`**.
3. **`paint()`** : The **`paint()`** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.  
**`paint()`** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **`paint()`** is called. The **`paint()`** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.
4. **`stop()`** : The **`stop()`** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **`stop()`** is called, the applet is probably running. You should use **`stop()`** to suspend threads that don't need to run when the applet is not visible. You can restart them when **`start()`** is called if the user returns to the page.
5. **`destroy()`** : The **`destroy()`** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **`stop()`** method is always called before **`destroy()`**.

### Creating Hello World applet :

// A Hello World Applet Save file as HelloWorld.java

```
import java.applet.Applet;
```

```
import java.awt.Graphics;
```

```
// HelloWorld class extends Applet
```

```
public class HelloWorld extends Applet
```

```
{
```

```
    // Overriding paint() method
```

```
    public void paint(Graphics g)
```

```
    {
```

```
        g.drawString("Hello World", 20, 20);
```

```
    }
```

```
}
```

**Running the HelloWorld Applet :**

We can use APPLET or OBJECT tag for this purpose. Using APPLET, here is the HTML file that executes HelloWorld :

```
<applet code="HelloWorld" width=200 height=60>
</applet>
```

**Using appletviewer :** This is the easiest way to run an applet.

```
appletviewer RunHelloWorld.html
```

**Event**

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

**Types of Event**

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

**Event Handling**

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

## AWT

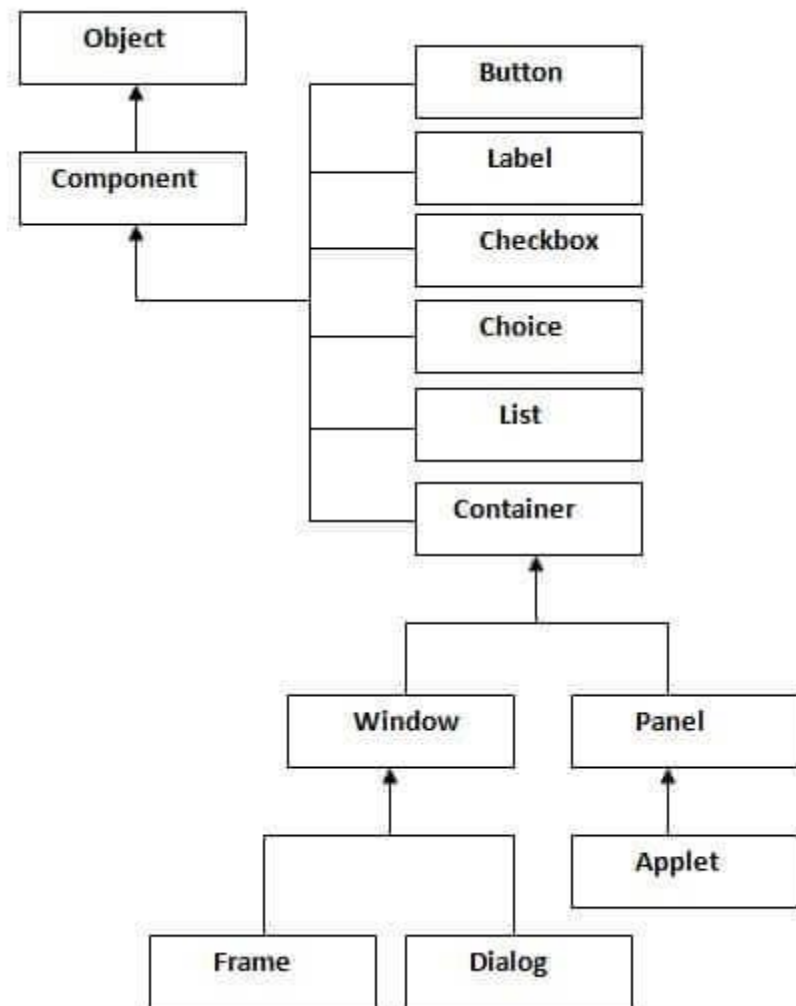
Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

### Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



## Container

The Container is a component in AWT that can contain another components like [buttons](#), textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

## Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

## Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

## Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

## Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

## Java AWT Example

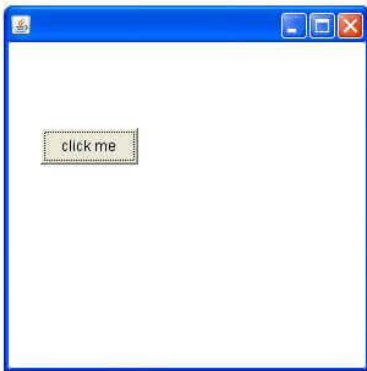
To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

## AWT Example by Inheritance

a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{
    First(){
        Button b=new Button("click me");
        b.setBounds(30,100,80,30);// setting button position
        add(b);//adding button into frame
        setSize(300,300);//frame size 300 width and 300 height
        setLayout(null);//no layout manager
        setVisible(true);//now frame will be visible, by default not visible
    }
    public static void main(String args[]){
        First f=new First();
    }
}
```



## AWT Example by Association

a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First2{
    First2(){
        Frame f=new Frame();
        Button b=new Button("click me");
        b.setBounds(30,50,80,30);
        f.add(b);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

```

}
public static void main(String args[]){
First2 f=new First2();
}}

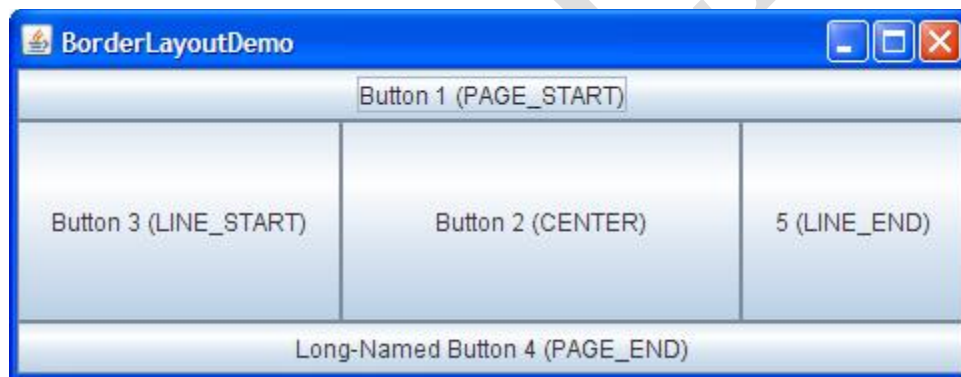
```

## Layout Managers

Several AWT and Swing classes provide layout managers for general use:

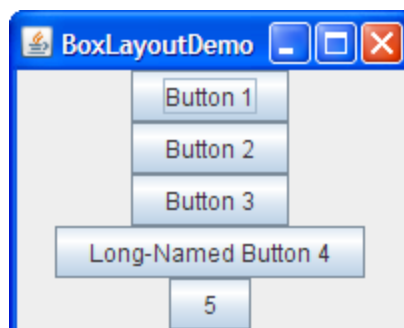
- **BorderLayout**
- **BoxLayout**
- **CardLayout**
- **FlowLayout**
- **GridBagLayout**
- **GridLayout**
- **GroupLayout**
- **SpringLayout**

### BorderLayout



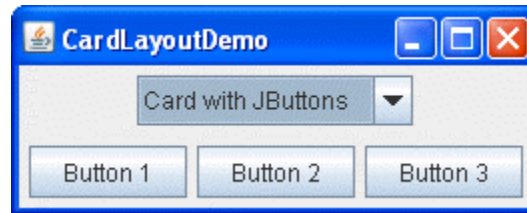
Every content pane is initialized to use a BorderLayout. (As [Using Top-Level Containers](#) explains, the content pane is the main container in all frames, applets, and dialogs.) A BorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using [JToolBar](#) must be created within a BorderLayout container, if you want to be able to drag and drop the bars away from their starting positions..

### BoxLayout



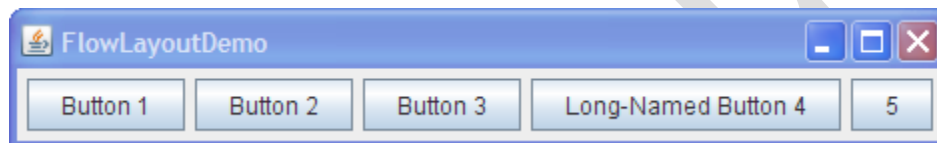
The BorderLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components.

### CardLayout



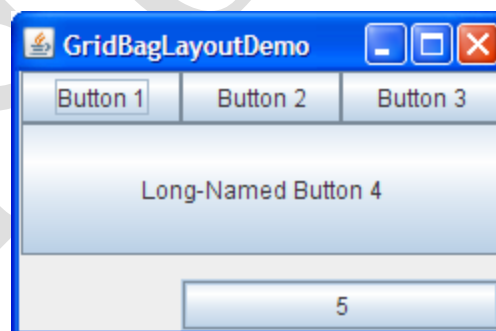
The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a [tabbed pane](#), which provides similar functionality but with a pre-defined GUI.

### FlowLayout



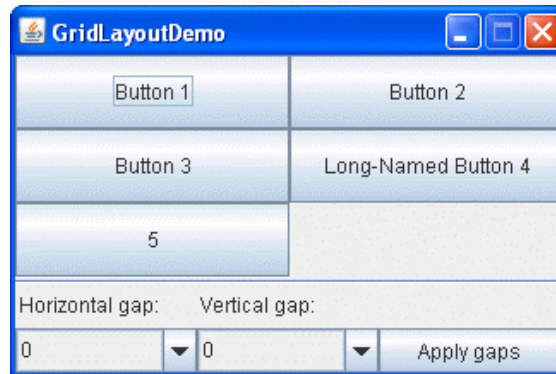
FlowLayout is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container is not sufficiently wide. Both panels in CardLayoutDemo, shown [previously](#), use FlowLayout.

### GridBagLayout



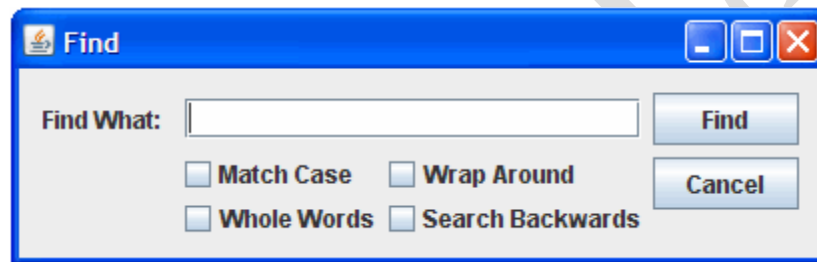
GridBagLayout is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

## GridLayout



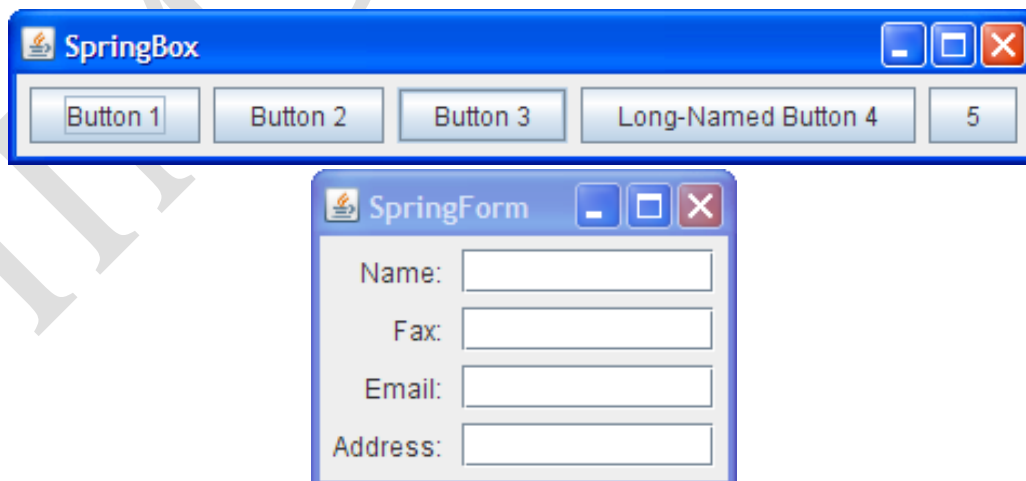
GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.

## GroupLayout



GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts separately. The layout is defined for each dimension independently. Consequently, however, each component needs to be defined twice in the layout. The Find window shown above is an example of a GroupLayout.

## SpringLayout





SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.

## Swings

Swing is a part of **JFC (Java Foundation Classes)**. Building Graphical User Interface in Java requires the use of Swings. Swing Framework contain a large set of components which allow high level of customization and provide rich functionalities, and is used to create window based applications. Java swing components are lightweight, platform independent, provide powerful components like tables, scroll panels, buttons, list, color chooser, etc.

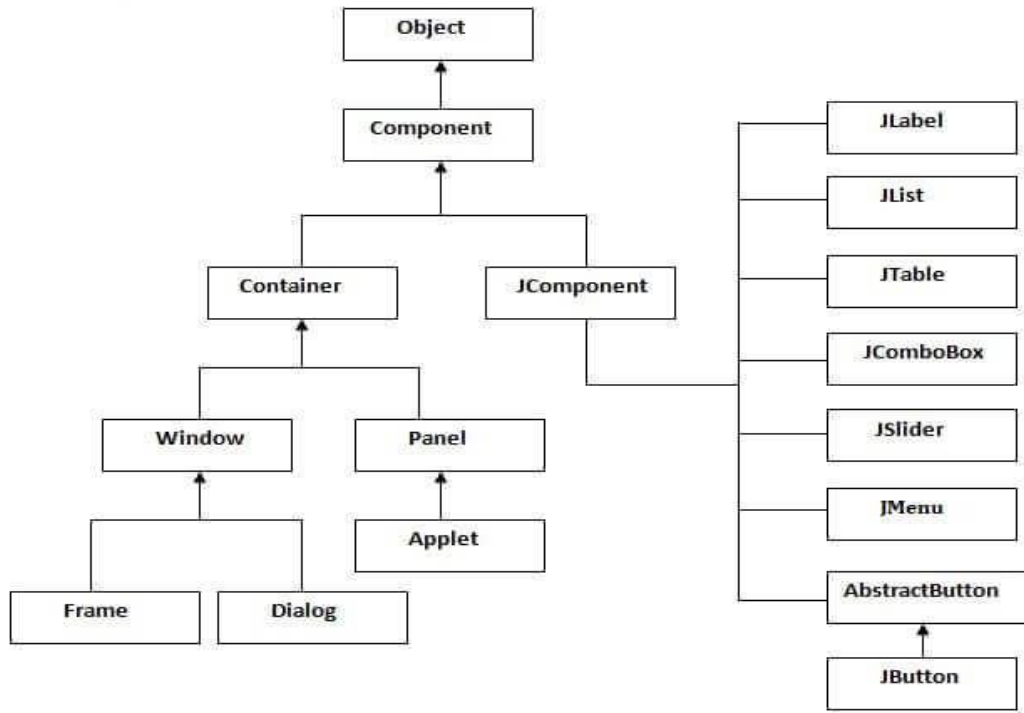
### Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are <b>platform-dependent</b> .	Java swing components are <b>platform-independent</b> .
2)	AWT components are <b>heavyweight</b> .	Swing components are <b>lightweight</b> .
3)	AWT <b>doesn't support pluggable look and feel</b> .	Swing <b>supports pluggable look and feel</b> .
4)	AWT provides <b>less components</b> than Swing.	Swing provides <b>more powerful components</b> such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT <b>doesn't follows MVC</b> (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing <b>follows MVC</b> .

### Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



```

import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame

```

```

JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height

```

```

f.add(b);//adding button in JFrame

```

```

f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}

```



## JTable

The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

### Constructors in JTable:

1. **JTable():** A table is created with empty cells.
2. **JTable(int rows, int cols):** Creates a table of size rows \* cols.
3. **JTable(Object[][] data, Object []Column):** A table is created with the specified name where []Column defines the column names.

### Methods

1. **addColumn(TableColumn []column) :** adds a column at the end of the JTable.
2. **clearSelection() :** Selects all the selected rows and columns.
3. **editCellAt(int row, int col) :** edits the intersecting cell of the column number col and row number row programmatically, if the given indices are valid and the corresponding cell is editable.
4. **setValueAt(Object value, int row, int col) :** Sets the cell value as 'value' for the position row, col in the JTable.

**Below is the program to illustrate the various methods of JTable:**

```
// Packages to import
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class JTableExamples {
    // frame
    JFrame f;
    // Table
    JTable j;

    // Constructor
    JTableExamples()
    {
        // Frame initialization
        f = new JFrame();

        // Frame Title
        f.setTitle("JTable Example");

        // Data to be displayed in the JTable
        String[][] data = {
```

```

        { "Kundan Kumar Jha", "4031", "CSE" },
        { "Anand Jha", "6014", "IT" }
    };

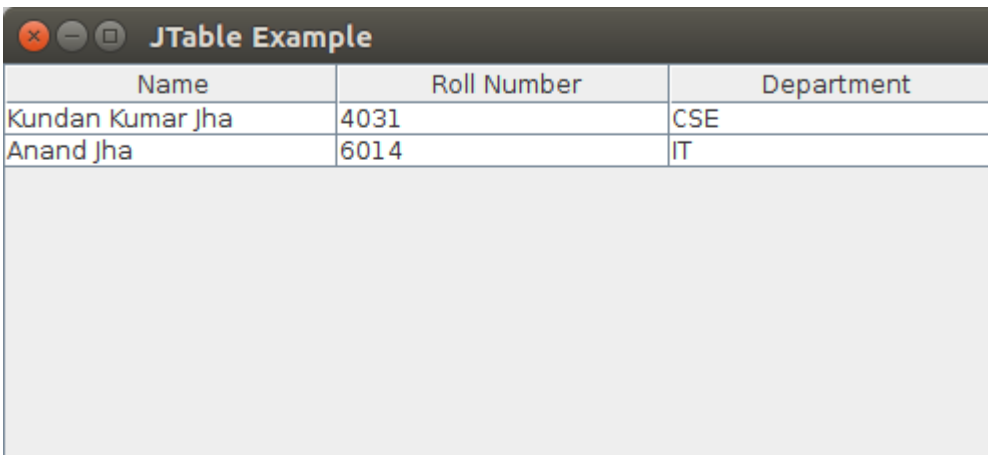
    // Column Names
    String[] columnNames = { "Name", "Roll Number", "Department" };

    // Initializing the JTable
    j = new JTable(data, columnNames);
    j.setBounds(30, 40, 200, 300);

    // adding it to JScrollPane
    JScrollPane sp = new JScrollPane(j);
    f.add(sp);
    // Frame Size
    f.setSize(500, 200);
    // Frame Visible = true
    f.setVisible(true);
}

// Driver method
public static void main(String[] args)
{
    new JTableExamples();
}
}

```



Name	Roll Number	Department
Kundan Kumar Jha	4031	CSE
Anand Jha	6014	IT

## Dialog Box

The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

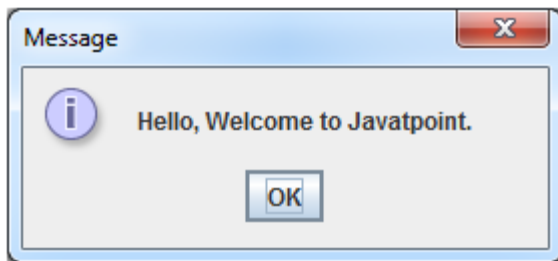
### Common Constructors of JOptionPane class

Constructor	Description
JOptionPane()	It is used to create a JOptionPane with a test message.
JOptionPane(Object message)	It is used to create an instance of JOptionPane to display a message.
JOptionPane(Object message, int messageType)	It is used to create an instance of JOptionPane to display a message with specified message type and default options.

### Common Methods of JOptionPane class

Methods	Description
JDialog createDialog(String title)	It is used to create and return a new parentless JDialog with the specified title.
static void showMessageDialog(Component parentComponent, Object message)	It is used to create an information-message dialog titled "Message".
static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)	It is used to create a message dialog with given title and messageType.
static int showConfirmDialog(Component parentComponent, Object message)	It is used to create a dialog with the options Yes, No and Cancel; with the title, Select an Option.
static String showInputDialog(Component parentComponent, Object message)	It is used to show a question-message dialog requesting input from the user parented to parentComponent.
void setInputValue(Object newValue)	It is used to set the input value that was selected or input by the user.

```
import javax.swing.*;
public class OptionPaneExample {
    JFrame f;
    OptionPaneExample(){
        f=new JFrame();
        JOptionPane.showMessageDialog(f,"Hello, Welcome to Javatpoint.");
    }
    public static void main(String[] args) {
        new OptionPaneExample();
    }
}
```



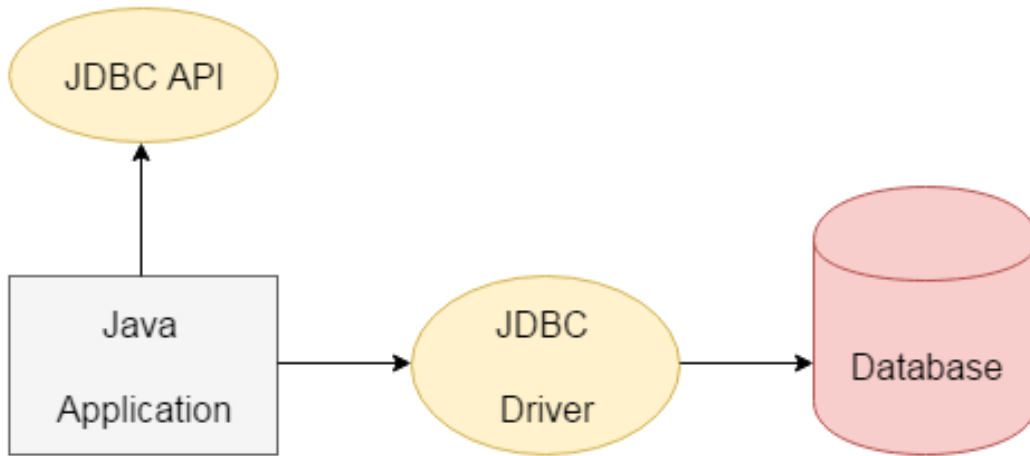
## JDBC

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.

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database



## Types of 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 (partially java driver)**
3. **Network Protocol driver (fully java driver)**
4. **Thin driver (fully java driver)**

### 1) 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.

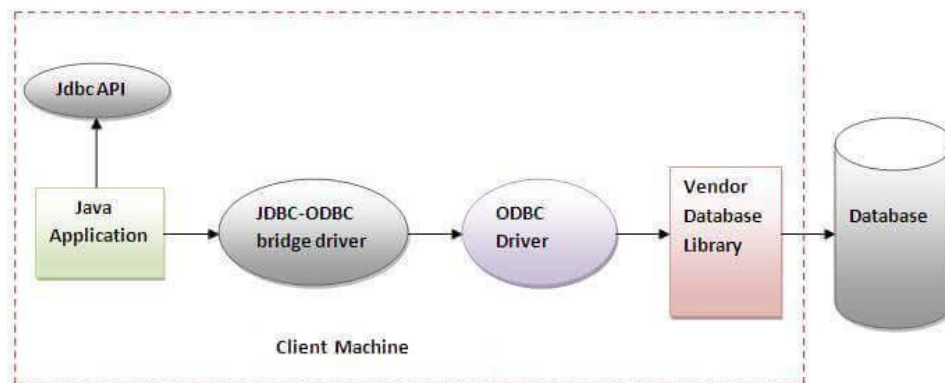


Figure- JDBC-ODBC Bridge Driver

#### Advantages:

- ✓ easy to use.
- ✓ can be easily connected to any database.

#### Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

## 2) 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.

#### Advantage:

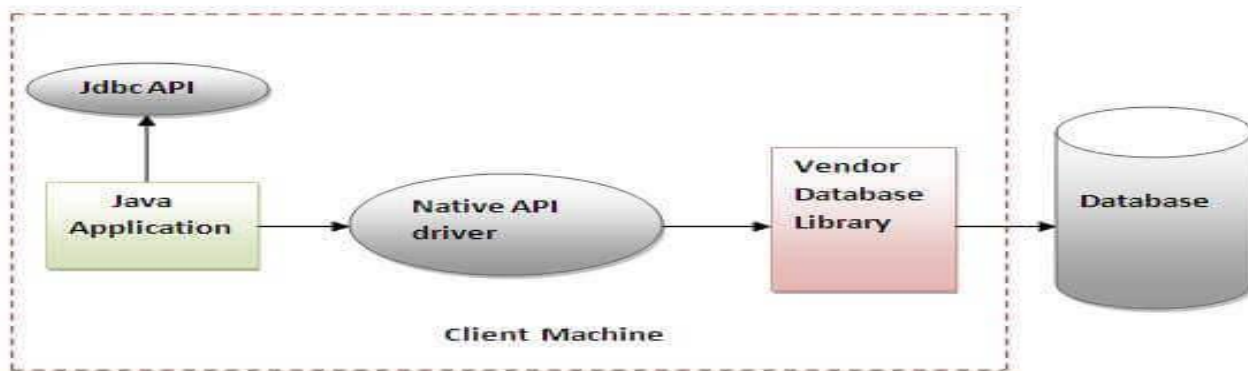


Figure- Native API Driver

- performance upgraded than JDBC-ODBC bridge driver.

#### Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

## 3) 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.



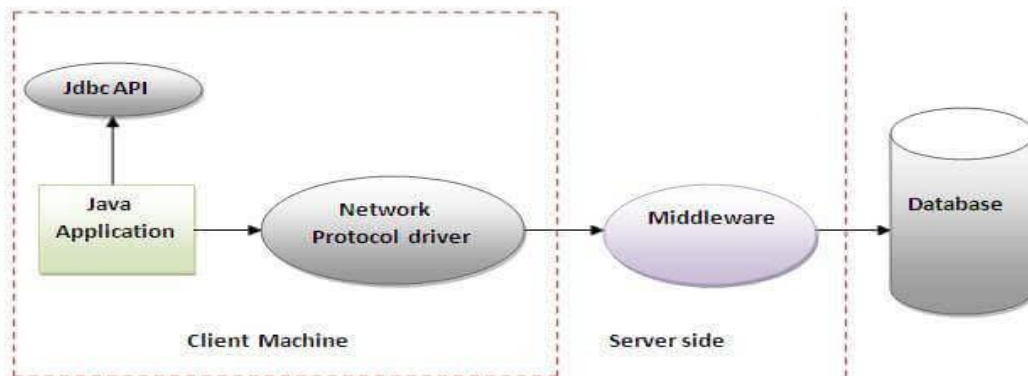


Figure- Network Protocol Driver

#### Advantage:

- ✓ No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

#### Disadvantages:

- ❖ Network support is required on client machine.
- ❖ Requires database-specific coding to be done in the middle tier.
- ❖ Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

#### 4) 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.

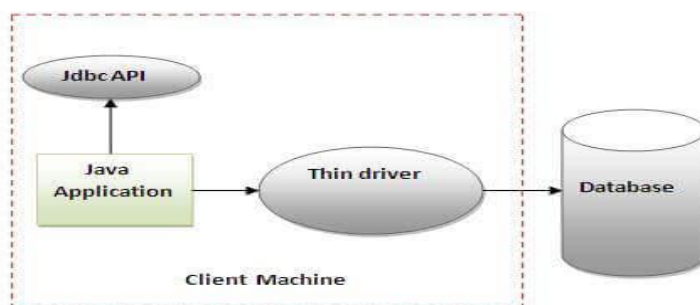


Figure- Thin Driver

**Advantage:**

- Better performance than all other drivers.
- No software is required at client side or server side.

**Disadvantage:**

- ❖ Drivers depend on the Database.

**Steps for connectivity between Java program and database****1. Loading the Driver**

To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- ❖ **Class.forName()** : Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –  
`Class.forName("oracle.jdbc.driver.OracleDriver");`

**2. Create the connections**

After loading the driver, establish connections using :

```
Connection con = DriverManager.getConnection(url,user,password)
```

**user** – username from which your sql command prompt can be accessed.

**password** – password from which your sql command prompt can be accessed.

**con:** is a reference to Connection interface.

**url** : Uniform Resource Locator. It can be created as follows:

```
String url = "jdbc:oracle:thin:@localhost:1521:xe"
```

Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

**3. Create a statement**

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step .

**4. Execute the query**

Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The executeUpdate(sql query) method of Statement interface is used to execute queries of updating/inserting .

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
```

## 5.Close the connections

So finally we have sent the data to the specified location and now we are at the verge of completion of our task . By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Example :

```
con.close();
```

## Developing Application

```
import java.sql.*;
import java.util.*;
class Main
{
    public static void main(String a[])
    {
        //Creating the connection
        String url = "jdbc:oracle:thin:@localhost:1521:xe";
        String user = "system";
        String pass = "12345";

        //Entering the data
        Scanner k = new Scanner(System.in);
        System.out.println("enter name");
        String name = k.next();
        System.out.println("enter roll no");
        int roll = k.nextInt();
        System.out.println("enter class");
        String cls = k.next();

        //Inserting data using SQL query
        String sql = "insert into student1 values('"+name+"','"+roll+"','"+cls+"')";
        Connection con=null;
        try
        {
            DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

```

//Reference to connection interface
con = DriverManager.getConnection(url,user,pass);

Statement st = con.createStatement();
int m = st.executeUpdate(sql);
if (m == 1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
con.close();
}
catch(Exception ex)
{
    System.err.println(ex);
}
}
}

```

## Scrollable ResultSet

Normal ResultSet allows fetching elements in forward only direction. However, Scrollable ResultSet allows us to easily move in forward/backward direction.

To create scrollable ResultSet, we must use a Statement/PreparedStatement object and provide scroll type to createStatement/prepareStatement method.

### Syntax :

```
PreparedStatement pstmt = conn.prepareStatement(sql, Scroll type constant, Concurrency constant);
```

```
Statement stmt = conn.createStatement(Scroll type constant, Concurrency constant);
```

### Scroll type constant

There are 3 scroll type constants can be used with ResultSets.

**ResultSet.TYPE\_FORWARD\_ONLY**

Default type.. only allows forward only fetching

*ResultSet.TYPE\_SCROLL\_INSENSITIVE*

Allows both forward and backward movement. Not sensitive to ResultSet updates.

*ResultSet.TYPE\_SCROLL\_SENSITIVE*

Allows both forward and backward movement. Not sensitive to ResultSet updates.

**Concurrency constant**

We can use following Concurrency constants for the ResultSets.

**ResultSet.CONCUR\_READ\_ONLY**

Default value .. ResultSet can not be updated.

*ResultSet.CONCUR\_UPDATABLE*

Signifies an updatable ResultSet.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
```

```
    ResultSet.CONCUR_UPDATABLE);
```

```
PreparedStatement pstmt = conn.prepareStatement(sql,ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

```
package com.topjavatutorial.jdbc;
```

```
import java.sql.Connection;
```

```
import java.sql.DriverManager;
```

```
import java.sql.PreparedStatement;
```

```
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
```

```
public class ScrollableResultSetDemo {  
    public static void main(String[] args) throws SQLException {  
        String url = "jdbc:mysql://localhost:3306/TestDB";  
        String user = "userid";//add your db user id here  
        String password = "password";//add your db password here  
        Connection conn = DriverManager.getConnection(url, user, password);  
        System.out.println("Successfully connected");  
        getEmployeeData(conn);  
    }  
  
    private static void getEmployeeData(Connection conn) throws SQLException{  
        String sql = "select id,name,age from employee";  
        try(PreparedStatement pstmt = conn.prepareStatement(sql,ResultSet.TYPE_SCROLL_INSENSITIVE,  
            ResultSet.CONCUR_READ_ONLY);){  
            ResultSet rs = pstmt.executeQuery();  
            //First Record  
            rs.first();  
            System.out.println("Emp Id : " + rs.getInt("id") + ", Name : " + rs.getString("name") + ", Age : " +  
rs.getInt("age"));  
            //Last Record  
            rs.last();  
            System.out.println("Emp Id : " + rs.getInt("id") + ", Name : " + rs.getString("name") + ", Age : " +  
rs.getInt("age"));  
            //Previous Record  
            rs.previous();  
            System.out.println("Emp Id : " + rs.getInt("id") + ", Name : " + rs.getString("name") + ", Age : " +  
rs.getInt("age"));  
  
            //Next Record
```

```
rs.next();  
System.out.println("Emp Id : " + rs.getInt("id") + ", Name : " + rs.getString("name") + ", Age : " +  
rs.getInt("age"));  
}  
}  
}
```

### Output

Successfully connected  
Emp Id : 8, Name : John Doe, Age : 21  
Emp Id : 13, Name : James, Age : 31  
Emp Id : 12, Name : James, Age : 23  
Emp Id : 13, Name : James, Age : 31