Unit 1

Uu1.Introduction about java- The Genesis of Java

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registeredu company, James Gosling and his team changed the Oak name to Java.

Sample Program

```
class Simple
{

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

2.Features of JAVA -Buzzwords

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*.

1. Simple

Object-Oriented

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

3.Basic concept of oops(Object Oriented language)

OOP concepts in Java are the main ideas behind Java's Object Oriented Programming. They are

- Class
- Object
- Polymorphism
- Inheritance

Encapsulation

- Abstraction

Class:

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. Modifiers: A class can be public or has default access (Refer this for details).
2. Class name: The name should begin with a initial letter (capitalized by convention).
3. Body: The class body surrounded by braces, { }.

Example

Public Class class_name

{---------

}

Object

An entity that has state and behavior is known as an object e.g., chair, bike, marker. A Java object is a combination of data and procedures working on the available data. An object has a state and behavior.

The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.Objects are created from templates known as classes.

Example

```
aaa ai = new aaa();
```

Polymorphism

Polymorphism in Java is a concept by which we can perform a single action in different ways. ... So polymorphism means many forms. Polymorphism is considered as one of the important features of Object Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

Example

- Method overloading
- Method overriding

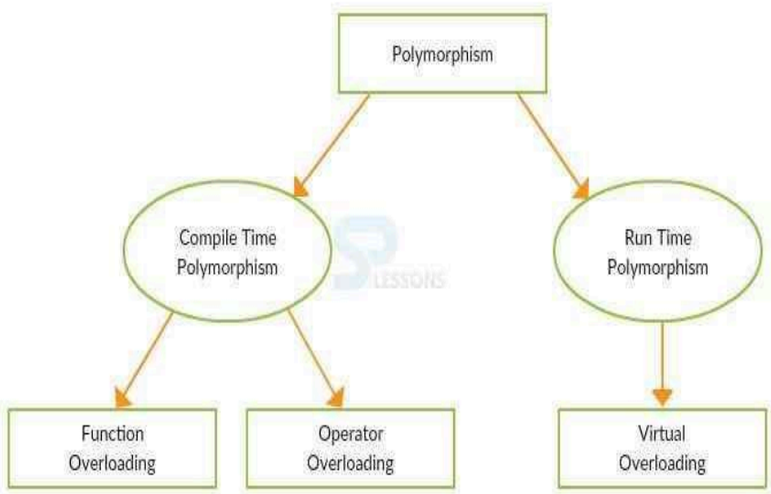
In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism

- Runtime Polymorphism

Uses

It used in implementing inheritance. It plays an important role in allowing objects having different internal structures to share the same external interface. polymorphism as stated clear by itself.



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.

- **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.

Syntax

```
class Subclass-name extends Superclass-name

{

  //methods and fields

}
```

Example

Class aaa extends bbb

{

}

Abstraction

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces. ... Abstraction Is one of the four major concepts behind object-oriented programming (OOP)

Abstraction can be of two types, namely, data abstraction and control abstraction. Data abstraction means hiding the details about the data and control abstraction means hiding the implementation details. In object-oriented approach, one can abstract both data and functions

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java −

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.



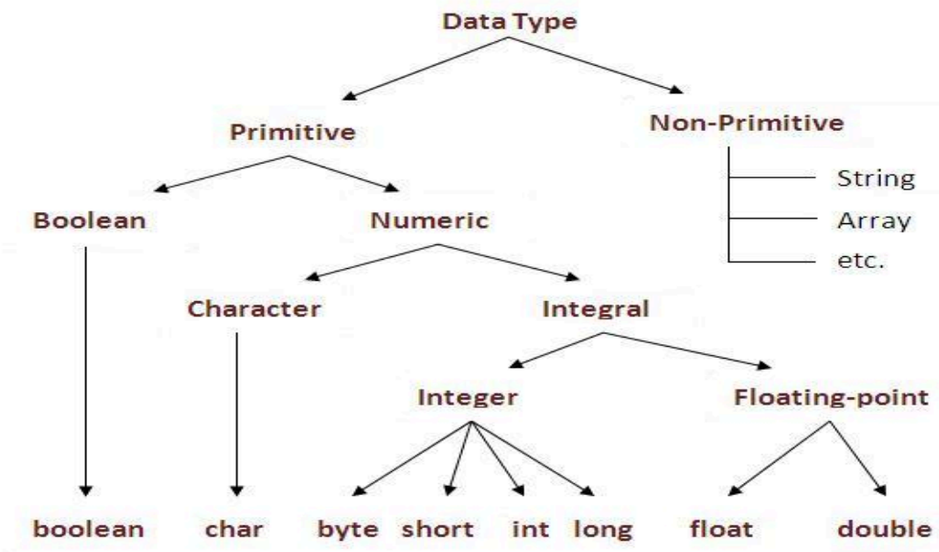
4.Data types in JAVA

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.

- o boolean data type
- o byte data type
- o char data type
- o short data type

- int data type
- long data type
- float data type,double data type



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

5.Java Tokens – Lexical Issues

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows

- Keywords
- Identifiers
- Constants
- Special Symbols
- Operators

Keyword:

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. Java language supports following keywords

abstract assert boolean

break byte case

catch char class

const continue default

do double else

enum exports extends

final finally float

for goto if

implements import instanceof

int interface long

module native new

Identifiers:

Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character.

Rules of Identifiers:

- A valid identifier has characters [A-Z],[a-z] or numbers [0-9], $ (dollar sign) and _ (underscore). ...
- We can't declare a variable with space. ...
- We can't start an identifier with a number. ...

- As there is no limit on the length of an identifier but it is 4 – 15 letters only appropriate to use.

Constants/Literals:

Constants are also like normal variables. But, the only difference is, their values can not be modified by the program once they are defined. Constants refer to fixed values. They are also called literals.

Syntax

final data_type variable_name;

Example

final int a=10;

Special Symbols:

The following special symbols are used in Java having some special meaning and thus, cannot be used for some other purpose.

[] () {}, ; * =

Brackets[]: Opening and closing brackets are used as array element reference.

Parentheses(): These special symbols are used to indicate function calls and function parameters.

Braces{}: These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.

comma (,): It is used to separate more than one statement like for separating parameters in function calls.

semicolon : It is an operator that essentially invokes something called an initialization list.

asterisk (*): It is used to create a pointer variable.

assignment operator: It is used to assign values.

Operators: Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

Arithmetic Operators:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)
- ++ (Increment)(Decrement)

Relational Operators

- == (equal to)
- != (not equal to)

- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)

Assignment Operator

=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

The Bitwise Operators

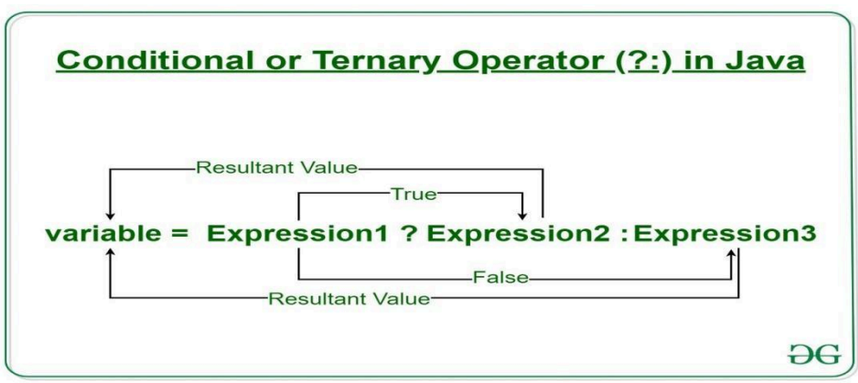
- & (bitwise and)
- | (bitwise or)
- ^ (bitwise XOR)
- ~ (bitwise complement)
- << (left shift)
- >> (right shift)
- >>> (zero fill right shift)

Logical Operators

- && (logical and)
- || (logical or)
- ! (logical not)

Ternary Operator

?,:



Shift Operators

Bitwise right shift operator in Java. Java supports two type of right shift operator. The>> operator is a signed right shift operator and >>> is an unsigned right shift operator. The left operand's value is moved right by the number of bits specified by the right operand.

6.Variables

- A Java variable is a piece of memory that can contain a data value. A variable thus has a data type. Data types are covered in more detail in the text on Java data types. Variables are typically used to store information which your Java program needs to do its job.
- A variable is a quantity that may change within the context of a mathematical problem or experiment. Typically, we use a single letter to represent a variable. The letters x, y, and z are common generic symbols used for variables.

int a=10;

2double b=12.768757578578;

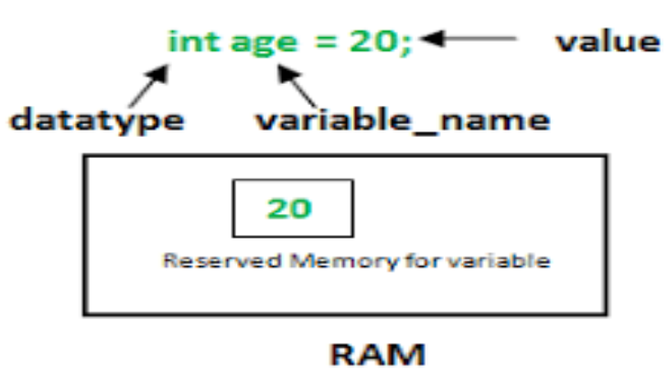
float c=244.5867;

char = name ;

To declare (create) a variable, you will specify the type, leave at least one space, then the name for the variable and end the line with a semicolon (;). Java uses the keyword int for integer, double for a floating point number



- **type:** Type of data that can be stored in this variable.
- name: Name given to the variable.
- In this way, a name can only be given to a memory location. It can be assigned values in two ways:
- Variable Initialization
- Assigning value by taking input
- **variable_name**: Name given to the variable.
- value: It is the initial value stored in the variable.
- int a------> Declare
- int a,b-------> more than one variable in same datatype
- int a = 10;
- int sum=total;---------> init value for variable
- Default variable value is 0



Types of variable

There are three types of variables in Java

- Local Variables

- Instance Variables,
- Static Variables

Local Variables:

A variable defined within a block or method or constructor is called a local variable.

- These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variables only within that block.
- Initialization of Local Variables is Mandatory.

Example

```
public class Age
{
public void StudentAge()
   {
      // local variable age
      int age = 0;
      age = age + 5; --------> local variable
      System.out.println("Student age is : " + age);
   }
 }
```

Instance variables

- instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initialization of Instance Variable is not Mandatory. Its default value is 0
- Instance Variable can be accessed only by creating objects.

Example

```
public class Age
{
int age = 0; --------> instance variables
public void StudentAge()
   {
```

```
int a= 10;
age = age + 5; --------> local variable
System.out.println("Student age is : " + age);
 }
public void StudentAge1()
{
age = age + 10;
System.out.println("Student age is : " + age);
}
}
```

Static Variables:

- Static variables are also known as Class variables.
- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of Static Variables is not Mandatory. Its default value is 0

Example

```
public class Age
{
static int age = 0; --------> static variables
public void StudentAge()
{
int a= 10;
age = age + 5; --------> local variable
System.out.println("Student age is : " + age);
}
public void StudentAge1()
{

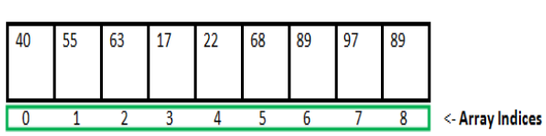
age = age + 10;
System.out.println("Student age is : " + age);
}}
```

Instance variable Vs Static variable

- Each object will have its own copy of instance variables whereas We can only have one copy of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In case of static, changes will be reflected in other objects as static variables are common to all objects of a class.
- We can access instance variables through object references and Static Variables can be accessed directly using class names.

8.Array in Java

- An array is a group of like-typed variables that are referred to by a common name.Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.
- In Java all arrays are dynamically allocated.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning from 0.
- Java arrays can also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int value and not long or short.



Array Length = 9
First Index = 0
Last Index = 8

One-Dimensional Arrays

The general form of a one-dimensional array declaration is

Syntax

type var-name[];

OR

type[] var-name;

Example

```
int Arr[];    //declaring array
Arr = new int[4];  // allocating memory to array
```

(or)

```
int[] Arr = new int[4];
 int[] Arr = new int[]{ 1,2,3,4 };
```

Example

```
class GFG
{
   public static void main (String[] args)
   {
      int[] arr = new int[2];
      arr[0] = 10;
      arr[1] = 20;

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

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array.

Example

```
int[][] intArray = new int[10][20];
int[][][] intArray = new int[10][20][10];
```

Example

```
class multiDimensional
{
   public static void main(String args[])
   {
      int arr[][][] = { {2,7,9},{3,6,1},{7,4,2} };
      for (int i=0; i< 3 ; i++)
      {
         for (int j=0; j < 3 ; j++)
            System.out.print(arr[i][j] + " ");
         System.out.println();
      } }}
```

out put

```
2 7 9
3 6 1
7 4 2
```

9.Control Statements in java

Decision Making

- ❖ if statement

- if-else statement
- if-else-if ladder
- nested if statement

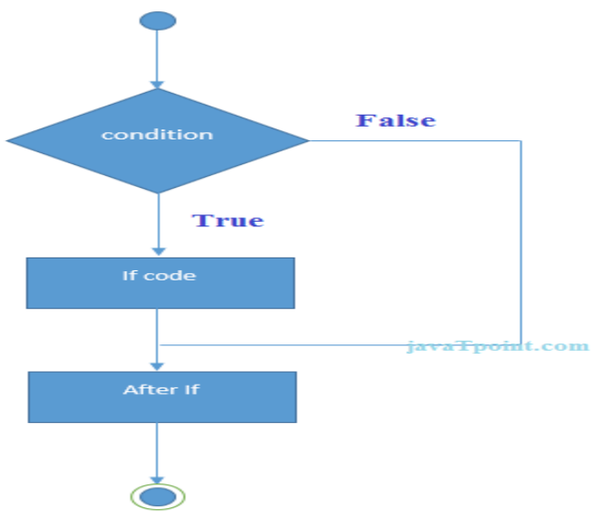
Looping Statement

- For loop
- While loop
- Do while loop

if statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

```
if(condition)
{
//code to be executed
}
```

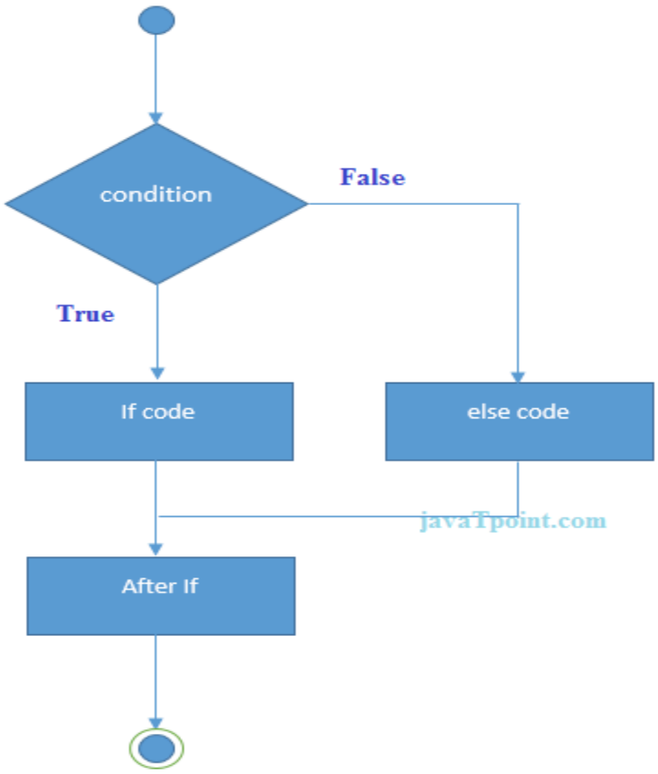


if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax

```
if(condition)
{
  //code if condition is true
  }else{
  //code if condition is false
  }
```



if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax

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

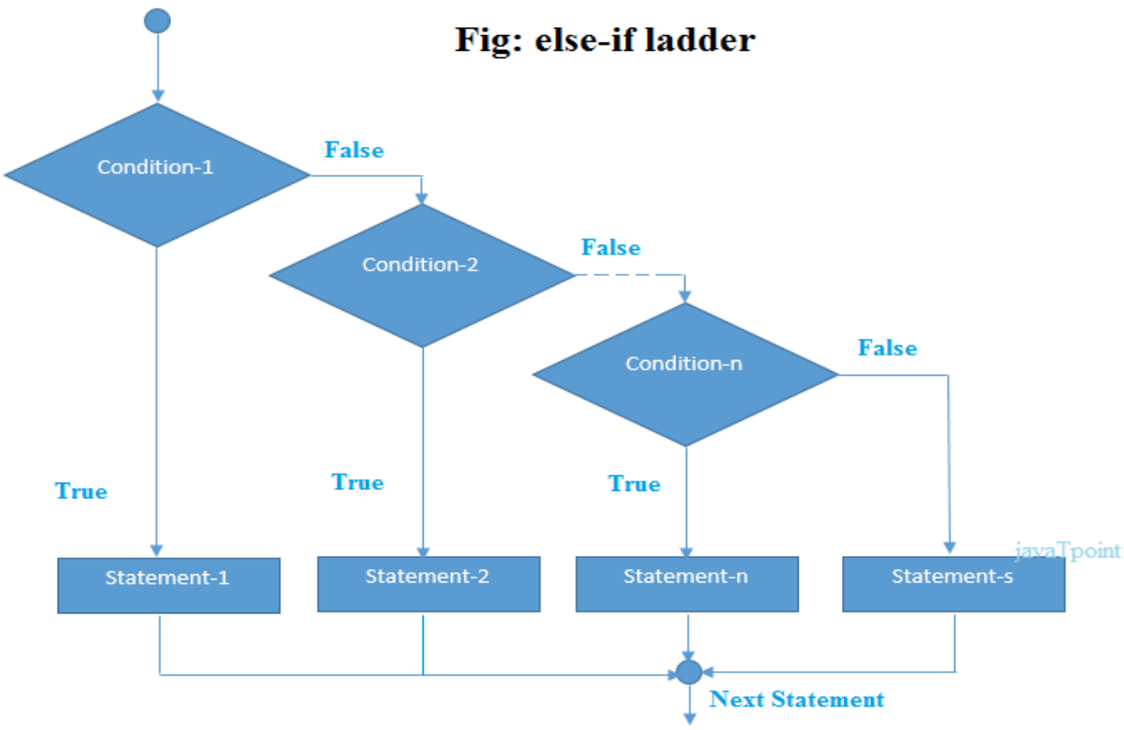


Fig: else-if ladder

Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when the outer if block condition is true.

Syntax

```
if(condition)
{
//code to be executed
 if(condition)++
{
//code to be executed
     }
}
```

Example program

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

    int age=20;
     if(age>18){
     System.out.print("Age is greater than 18");
    }
    }
    }
```

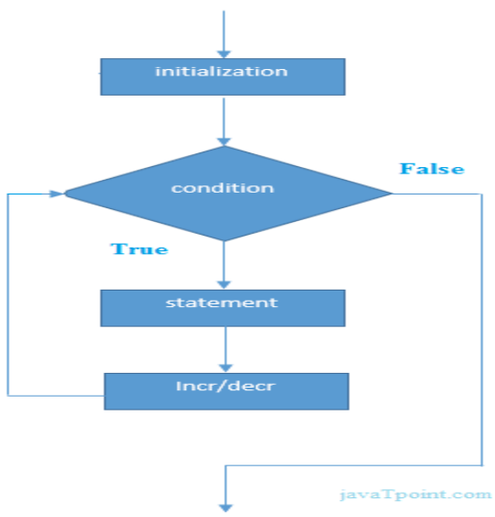
For Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return a boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

Syntax

```
For(initialization;condition;incr/decr)
{
//statement or code to be executed
}
```



Example

```
public class ForExample {
public static void main(String[] args) {
//Code of Java for loop
for(int i=1;i<=10;i++){
System.out.println(i);
    }
  }
}
```

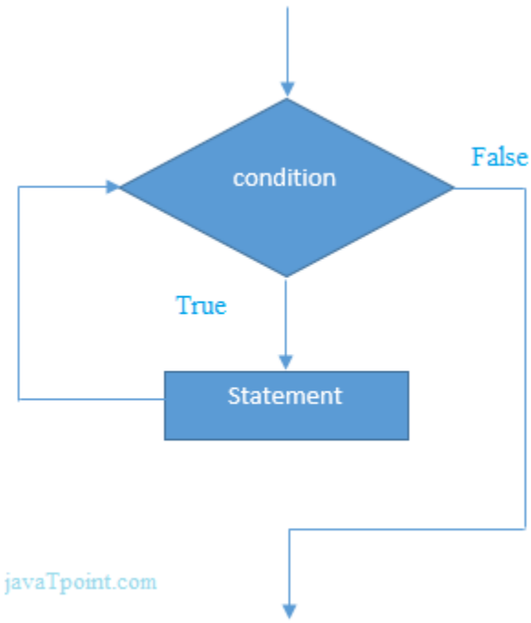
While Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

```
while(condition)
```

{
/ /code to be executed
}



Example

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

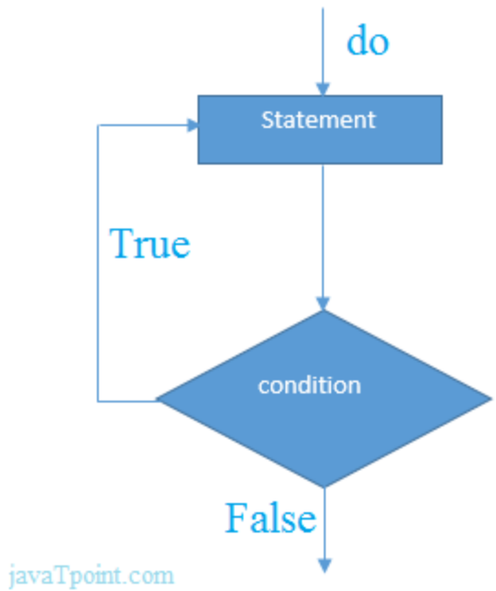
do-while Loop

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

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



Example

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

Unit II

1.Class and object in java

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access (Refer this for details).
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Super class**(if any): The name of the class's parent (super class), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body**: The class body surrounded by braces, { }.

Class declaration

Class declaration must contain the class keyword and the name of the class that you are defining. ... More specifically, within the class declaration you can: Declare what the class's superclass is. List the interfaces implemented by the class. Declare whether the class is public, abstract,

Syntax

Class class-name

Example

Class aaa

Field declaration in class

The class body component of a class implementation can itself contain two different sections: variable declarations and methods. ... For example, every class in the Java environment is a descendent (direct or indirect) of the Object class, that is, every class in Java inherits variables and methods from Object .

Method Declaration

Modifier-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 type of the access specifiers.

Example

Void sum()

Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Syntax

Class- name object name=new classname ();

Example

aaa ai = new aaa();

Example program

```
import java.io.*;

class Student

{

int regno,total;

String name;

int mark[]=new int[3];

void readinput() throws IOException

{

BufferedReader din=new BufferedReader(new InputStreamReader(System.in));
```

```
System.out.print("\nEnter the Reg.No: ");

regno=Integer.parseInt(din.readLine());

System.out.print("Enter the Name: ");

name=din.readLine();

System.out.print("Enter the Mark1: ");
mark[0]=Integer.parseInt(din.readLine()); System.out.print("Enter
the Mark2: "); mark[1]=Integer.parseInt(din.readLine());
System.out.print("Enter the Mark3: ");
mark[2]=Integer.parseInt(din.readLine());
total=mark[0]+mark[1]+mark[2];

}

void display()

{

System.out.println(regno+"\t"+name+"\t"+mark[0]+"\t"+mark[1]+"\t"+

mark[2]+"\t"+total);

}

}

class Mark

{

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

{

Student s[]=new Student[5];

 for(inti=0;i<5;i++)

{

s[i]=new Student();

s[i].readinput();

}

System.out.println("\t\t\tMark List");

System.out.println("\t\t\t*********");

System.out.println("RegNo\tName\tMark1\tMark2\tMark3\tTotal");

for(inti=0;i<5;i++)

s[i].display();

}

}
```

OUTPUT:

```
D:\java>javac Mark.java

D:\java>java Mark

Enter the Reg.No: 1

Enter the Name: Balu

Enter the Mark1: 67

Enter the Mark2: 90

Enter the Mark3: 56

Enter the Reg.No: 2

Enter the Name: Geetha

Enter the Mark1: 87

Enter the Mark2: 79

Enter the Mark3: 92

Enter the Reg.No: 3

Enter the Name: Vimal

Enter the Mark1: 87

Enter the Mark2: 60

Enter the Mark3: 71
```

2.Method in java

Basically we have two types of methods in java.

i) Built in methods

ii) User defined methods.

User Define Method

A **Java method** is a collection of statements that are grouped together to perform an operation. 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.

Method Declaration

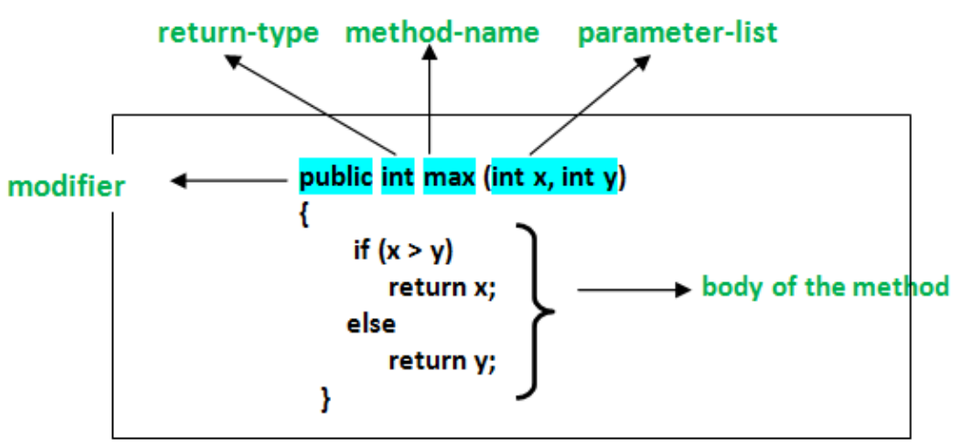
- **Modifier**-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of the access specifiers.
  - **public:** accessible in all classes in your application.
  - **protected**: accessible within the class in which it is defined and in its **subclass(es)**
  - **private:** accessible only within the class in which it is defined.

- **default** (declared/defined without using any modifier) : accessible within the 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 parentheses. 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 operations.

Example

```
   Public void sum()
{
A=b+c;
}
```



Calling Method

All you need is the name of your object, a dot, and the void method you want to call.Java will then just get on with executing the code inside of your method. Run your code and your Output window should display the following: In the next part, we'll take a closer look at passing values to methods.

Syntax

Object- name. method- name(para value);

Example+

Aa . display();

Example Program

Lab program ex no 2(class and object)

3.Method overloading in java

Method Overloading is a feature that allows a class to have more than one **method** having the same name, if their argument lists are different. It is similar to

constructor **overloading in Java**, that allows a class to have more than one constructor having different argument lists. **Overloading** is an **example** of compile time polymorphism

Example

Class aaa

{

Sum()

Sum(int a ,int b)

Sum(int x,int y,float z)

}

Example Program

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

4. Method overriding in java

Method overriding in java with example. Declaring a method in subclass which is already present in parent class is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. Overriding is an example of run time polymorphism

Example

Class aaa

{

Sum()

}

Class bbb extends aaa

{

Sum()

}

Example Program

```
class Parent
 {
private void m1()
{
System.out.println(“From parent m1”);
}
protected void m2()
{
System.out.println(“From parent m2”);
}
}
class Child extends Parent
{
private void m1()
{
System.out.println(“From child m1”);
}
public void m2()
{
System.out.println(“From child m2”);
```

```
}
}
class Main

{
public static void main(String[] args)
{
Parent obj1 = new Parent();
obj1.m2();
Child obj2 = new Child();
obj2.m2();

obj2.m1():
}
}
```

Output:

From parent m2

From child m2

From child m1// parent class mi() method is overridden

5.Constructor in java

Constructor is a special member function. Its having same name of class name Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it's not a method as it doesn't have a return type

Example

```
class aaa

{

  aaa()

{

----

}

}
```

Rules for writing Constructor.

- Constructor(s) of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of constructorconstructor with arguments or no-arguments then the compiler does not create a default constructor.

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.

Constructor Overloading

Constructor overloading is a concept of having more than one **constructor** with different parameters list, in such a way so that each **constructor** performs a different task.

Difference between constructor and method

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any number of times.

Example program

```
class add
{
   int x,y,z;
   add(int a, int b, int c)
   {
      x= a;
      y= b;
      z = c
      int  tot=x+y+z;
      System.out .println(“ The ans”+tot);
   }
   add()
   {
      x = y = z = 3;
     int  tot1= x+y+z;
      System.out .println(“ The ans”+tot1);
   }
    add(int m)
   {
      x = y = z = m;
      int  tot2= x+y+z;
```

```
    System.out .println(“ The ans”+tot2);
  }
    public class Test
  {
   public static void main(String args[])
   {
       add a1= new add (10, 20, 15);
       add a2 = new add ();
        add  a3 = new add (7);
  }
  }
```

Output

The ans 45

The ans 9

The ans 21

6.this keyword.

this is a keyword in Java. this keyword in java can be used inside the Method or constructor of Class. It(this) works as a reference to the current Object, whose Method or constructor is being invoked. This keyword can be used to refer to any member of the current object from within an instance Method or a constructor.

Uses of this keyword

1. Using ‘this’ keyword to refer current class instance variables
2. Using this() to invoke the current class constructor.
3. Using ‘this’ keyword to return the current class instance
4. Using ‘this’ keyword as method parameter
5. Using ‘this’ keyword to invoke current class method
6. Using ‘this’ keyword as an argument in the constructor call

Example program

```
class Test
{

  void display()
  {
    // calling function show()
    this.show();

    System.out.println("Inside display function");
  }
```

```
    void show()
    {
        System.out.println("Inside show funcion");
    }

    public static void main(String args[])
    {
        Test t1 = new Test();
        t1.display();
    }
}
```

***this* Keyword with Constructor**

"this" keyword can be used inside the constructor to call another overloaded constructor in the same Class. It is called the Explicit Constructor Invocation.

This occurs if a Class has two overloaded constructors, one without argument and another with the argument. Then "this" keyword can be used to call the constructor with an argument from the constructor without argument

7.Garbage collection and finalize() method

Garbage collection

In **Java**, the programmers don't need to take care of destroying the objects that are out of use. Garbage Collection is the process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy unused objects. The process of removing unused objects from heap memory is known as **Garbage collection** and this is a part of memory management in Java

Finalize method

finalize() method is a protected and non-static method of **java.lang.Object** class. This method will be available in all objects you create in java. This method is used to perform some final operations or clean up operations on an object before it is removed from the memory. you can override the finalize() method to keep those operations you want to perform before an object is destroyed

finalize()**method** is called by garbage collection thread before collecting object. It is invoked each time before the object is garbage collected

Example

```
protected void  ()
{
   //Keep some resource closing operations here
}
```

Example program

```
public class aaa
```

```
{
public static void main(String[] args)
{
    aaa obj = new aaa();
    System.out.println(obj.hashCode());
    obj = null;
    System.gc();
    System.out.println("end of garbage collection");
}
protected void finalize()
{
    System.out.println("finalize method called");
  }
}
```

System.gc() is used to invoke the garbage collector and on invocation garbage collector will run to reclaim the unused memory space. It will attempt to free the memory that is occupied by the discarded objects. The Java Language Specification does not guarantee that the JVM will start a GC when you call System.gc().

8.Use object as argument

The first parameter is a Data object. If you pass an object as an argument to a method, the mechanism that applies is called pass-by-reference, because a copy of the reference contained in the variable is transferred to the method, not a copy of the object itself.

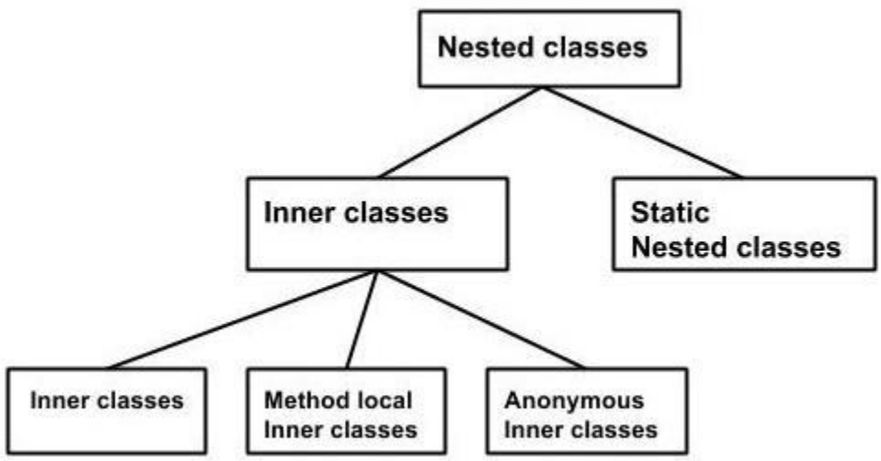
A method can take an object as a parameter. For example, in the following program, the method **setData( )** takes three parameter. The first parameter is an **Data** object. If you pass an object as an argument to a method, the mechanism that applies is called **pass-by-reference**, because a copy of the reference contained in the variable is transferred to the method, not a copy of the object itself.

```
class SetData
{
  void setData(Data da,int d1,int d2)
  {
    da.data1 = d1;
    da.data2 = d2;
  }
```

```
    void getData(Data da)
    {
        System.out.println("data1 : "+da.data1);
        System.out.println("data2 : "+da.data2);
    }
}
public class Javaapp
 {
    public static void main(String[] args)
 {
        Data da = new Data();
        SetData sd = new SetData();
        sd.setData(da,50,100);
        sd.getData(da);
    }
}
```

9.Inner class in java

Inner class means one class which is a member of another class. There are basically four types of inner classes in java. ... As a side note, 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.



1. **Nested Inner class // Non static**
    a) Method Local inner classes
    b) Anonymous inner classes
2. **Static nested classes**

The class written within is called the nested class, and the class that holds the inner class is called the outer class..

Syntax

Here, the class Outer_Demois the outer class and the class Inner_Demo is the nested class.

```
class Outer_Demo
 {
  class Inner_Demo
   {
   }
}
```

Nested classes are divided into two types −

- Non-static nested classes − These are the non-static members of a class.
- Static nested classes − These are the static members of a class.

1.Non static Nested Class (or) Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Method-local Inner Class

A method-local inner class can be instantiated only within the method where the inner class is defined.

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we declare and instantiate them at the same time.

Example Program

```
public class localInner1

{

 private int data=30;//instance variable

 void display()

{

System.out.println (“hello”);

}

class Local

  {

      void msg()

     {
```

```
        System.out.println(data);
    }
  }
   Local l=new Local();
   l.msg();
 }
Class main1
{
  public static void main(String args[])
{
   localInner1 obj=new localInner1();
   obj.display();
 }
}
```

2. Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class

Example program

```
class TestOuter1
{
  static int data=30;
  static class Inner
{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[])
{
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
```

```
  }

}
```

10.Explain String class in java.

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

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on strings such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in a window based, web based or mobile application.

1.char charAt(int index) :Returns the character at the specified index.

2 int compareTo(Object o) : Compares this String to another Object.

3 int compareTo(String anotherString): Compares two strings lexicographically.

4 int compareToIgnoreCase(String str);Compares two strings lexicographically, ignoring case differences.

5 String concat(String str);Concatenates the specified string to the end of this string.

6.boolean endsWith(String suffix);Tests if this string ends with the specified suffix.

7. boolean equals(Object anObject);Compares this string to the specified object.

8 boolean equalsIgnoreCase(String anotherString):

Compares this String to another String, ignoring case considerations.

9. int hashCode();Returns a hash code for this string.

10.int indexOf(int ch):

Returns the index within this string of the first occurrence of the specified character.

11.int indexOf(int ch, int fromIndex):

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

12 int indexOf(String str):

Returns the index within this string of the first occurrence of the specified substring.

13:int lastIndexOf(int ch):

Returns the index within this string of the last occurrence of the specified character.

14. int lastIndexOf(int ch, int fromIndex):

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

15.int lastIndexOf(String str):

Returns the index within this string of the rightmost occurrence of the specified substring.

16 int length(): Returns the length of this string.

17 String replace(char oldChar, char newChar):

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

18 String replaceAll(String regex, String replacement):

Replaces each substring of this string that matches the given regular expression with the given replacement.

19 .String[] split(String regex):

Splits this string around matches of the given regular expression.

20.boolean startsWith(String prefix): Tests if this string starts with the specified prefix..

21.String substring(int beginIndex): Returns a new string that is a substring of this string.

22 String substring(int beginIndex, int endIndex): Returns a new string that is a substring of this string.

23 String toLowerCase()

Converts all of the characters in this String to lowercase using the rules of the default locale.

24 String toLowerCase(Locale locale):

Converts all of the characters in this String to lowercase using the rules of the given Locale.

25 .String toString(): This object (which is already a string!) is itself returned.

26 String toUpperCase():

Converts all of the characters in this String to upper case using the rules of the default locale.

27.String toUpperCase(Locale locale):

Converts all of the characters in this String to upper case using the rules of the given Locale.

28. String trim(): Returns a copy of the string, with leading and trailing whitespace omitted.

Example Program

```
class strMethod
{
static String str="object";
public static void main(String arg[])
{
        System.out.println("original string: "+str);
        int str1=str.length();
        System.out.println("length of string:  "+str1);
        String str2=str+" Oriented";
        System.out.println("Modified string: "+str2);
        String str3=str2.toUpperCase();
        System.out.println("String :"+str3);
        char ch;
        ch="abc".charAt(2);
        System.out.println("CharAt :"+ch);
        byte x[]={66,67,68,69,70};
        String s4=new String(x);
        System.out.println(s4);
        String str4="Hello".replace('l','w');
        System.out.println(str4);
        String s1=”BCA”;
        String s2=”bca”;
        System.out.println(s1.equals(s2));
        System.out.println(s1.equalsIgnoreCase(s2));
        String s5=”Foot ball”;
        System.out.println(s5.startsWith(“foot”));
        System.out.println(s5.endsWith(“ball”));
}
}
```

OUTPUT:

```
original string: object
```

length of string: 6

Modified string : object Oriented

String : OBJECT ORIENTED

CharAt :c

BCDEF

Hewwo

false

true

true

true

true

11.Final keyword in java

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

- variable
- method
- class

Final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant)

Syntax (ex)

// final data-type variable-name = value   //Final int a=10;

Example program

```
class finalVariable

{

public static void main(String arg[])

{

int b=5;

final int a=20;

a=b+10;

System.out.println(a+" "+b);

}

}
```

OUTPUT:

finalVariable:java:7:error:cannot assign a value to final variable a.

Final method

In **method** declaration to indicate that the **method** cannot be overridden by subclasses.

Syntax

Final method-mname()

Example program

```
class A
{
final void display()
{
System.out.println('this is final class");
}
}
class B extends A
{
void display()
{
System.out.println("outside of final");
}
}
class FinalWithMethod
{
public static void main(String args[])
{
B ob=new B();
ob.display();
}
```

Final class

A **final class** is simply a **class** that can't be extended. (This does not mean that all references to objects of the **class** would act as if they were declared as **final** .) Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Syntax

Final class class-name

Example program

```
final class A
{
void display()
{
System.out.println("this is final class");
}
}
class B extends A
{
void display()
{
System.out.println("outside of final");
}
}
class FinalWithClass
{
public static void main(String s[])
{
B ob=new B();
ob.display();
}
}
```

OUTPUT:

finalWithClass:java:9:error:cannot inherit from final A

12.Static keyword

The **static keyword** is used in **java** mainly for memory management. It is used with variables, methods, blocks and nested classes. It is a **keyword** that are used for share the same variable or method of a given class. This is used for a constant variable or a method that is the same for every instance of a class

The **static keyword in Java** means that the variable or function is shared between all instances of that class as it belongs to the type, not the actual objects themselves.

Static variable

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class.

Syntax

Static int a=10;

Static method

A static method is a method that belongs to a class rather than an instance of a class. The method is accessible to every instance of a class, but methods defined in an instance are only able to be accessed by that member of a class.

Syntax

Static method-name();

Example program

```
class demo
{
static int a;
int b;
static
{
System.out.println("inside static block");
}
demo(int x,int y)
{
a=x;
b=y;
}
static void callme()
{
System.out.println("inside static is calling");
}
void display()
{
System.out.println("A :"+a);
```

```
System.out.println("B :"+b);
}
}
class staticByName
{
public static void main(String arg[])
{
demo.callme();
demo d1=new demo(5,2);
d1.display();
demo d2=new demo(10,12);
d2.display();
d1.display();
}
}
```

13.Recursion method in java

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using recursive algorithms, certain problems can be solved quite easily.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

The difference between direct and indirect recursion

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly

Direct recursion:

```
void directRecFun()
{
   // Some code....
```

```
    directRecFun();
    // Some code...
}
```

Indirect recursion:

```
void indirectRecFun1()
{
    // Some code...
    indirectRecFun2();
    // Some code...
}
void indirectRecFun2()
{
    // Some code...
    indirectRecFun1();
    // Some code...
}
```

14.Command Line argument in java

The java command-line argument is an argumenti.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.

The command line argument is the argument passed to a program at the time when you run it. To access the command-line argument inside a java program is quite easy, they are stored as string in String array passed to the args parameter of main() method.

Command Line Argument is information passed to the program when you run the program. The passed information is stored as a string array in the main method. Later, you can use the command line arguments in your program. java Demo arg1 arg2 arg3 …

Example Acquires

```
java Demo arg1 arg2 arg3 …
```

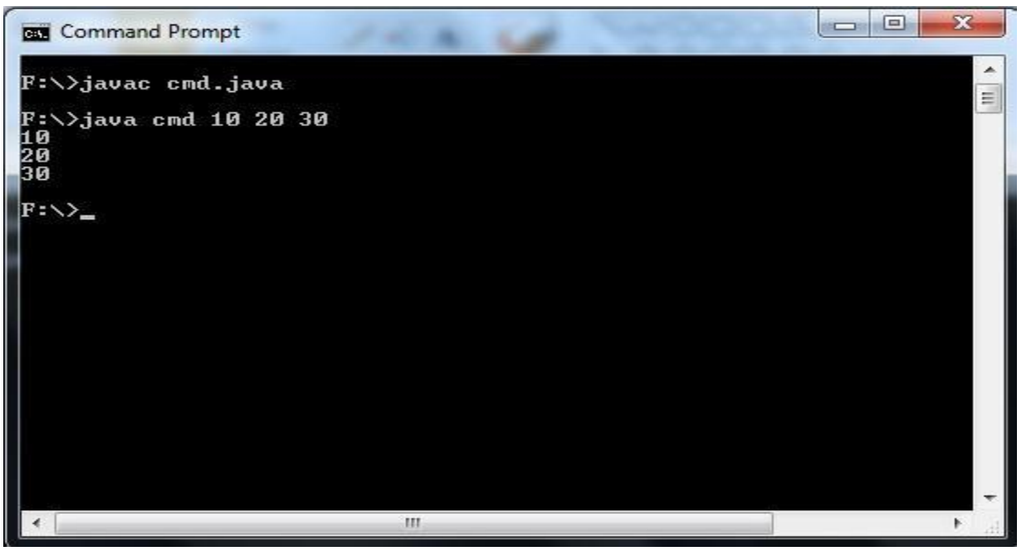
Command Line Arguments in Java: Important Points

- Command Line Arguments can be used to specify configuration information while launching your application.
- There is no restriction on the number of java command line arguments.You can specify any number of arguments
- Information is passed as Strings.
- They are captured into the String args of your main method

Example Program

```
class cmd
{
  public static void main(String[] args)
  {
    for(int i=0;i< args.length;i++)
    {
    System.out.println(args[i]);
    }
  }
}
```

Out put



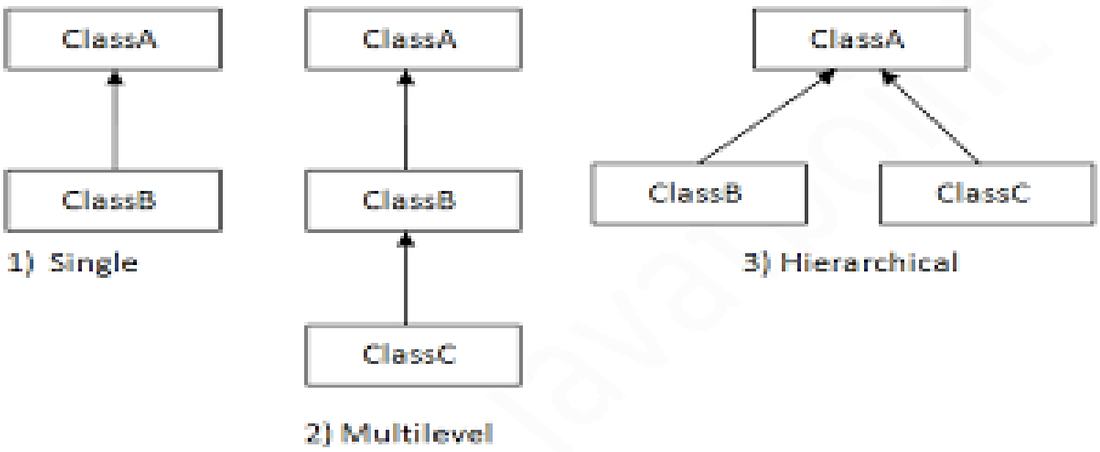
UNIT 3

1.Inheritance

- Inheritance in Java is a mechanism in which one object all the properties and behaviors of a parent object. ...
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.

extends Keyword

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



Syntax

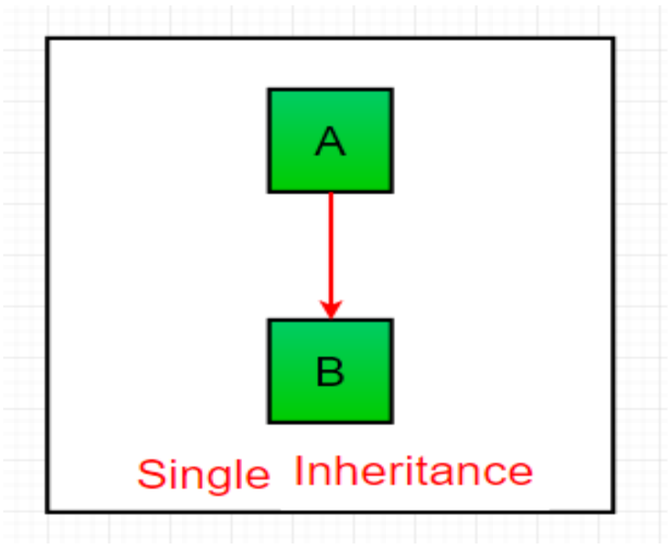
```
class Super
{
.....
}
class Sub extends Super
{
  .....
  .....
}
```

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class

Single Inheritance :(One base ,One subclass):

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



Syntax

class Super-class name

{

}

class Sub-classname extends Super-class name

{

}

Example

class aaa

{

}

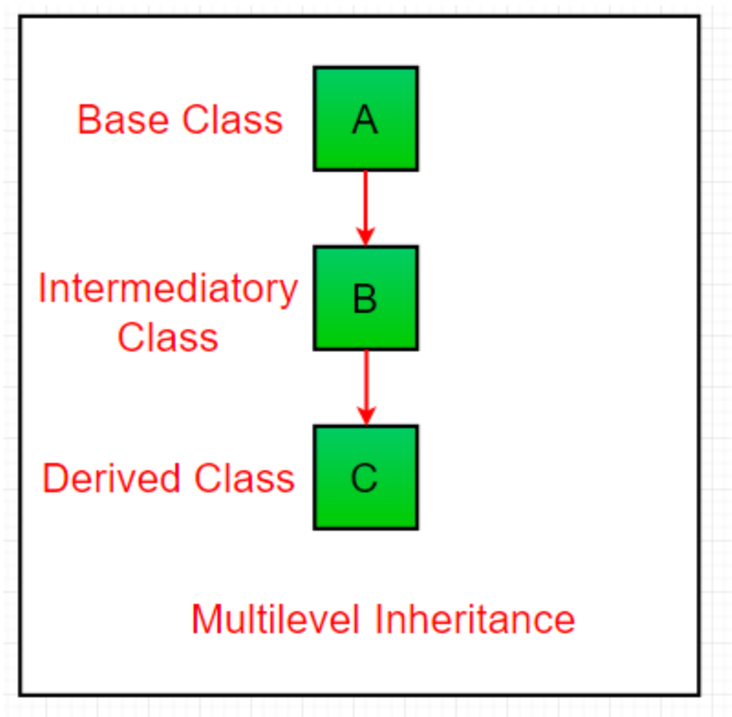
class bbb extends aaa

{

}

Multilevel Inheritance-(One base One sub one or more than one intermediate)

- **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.

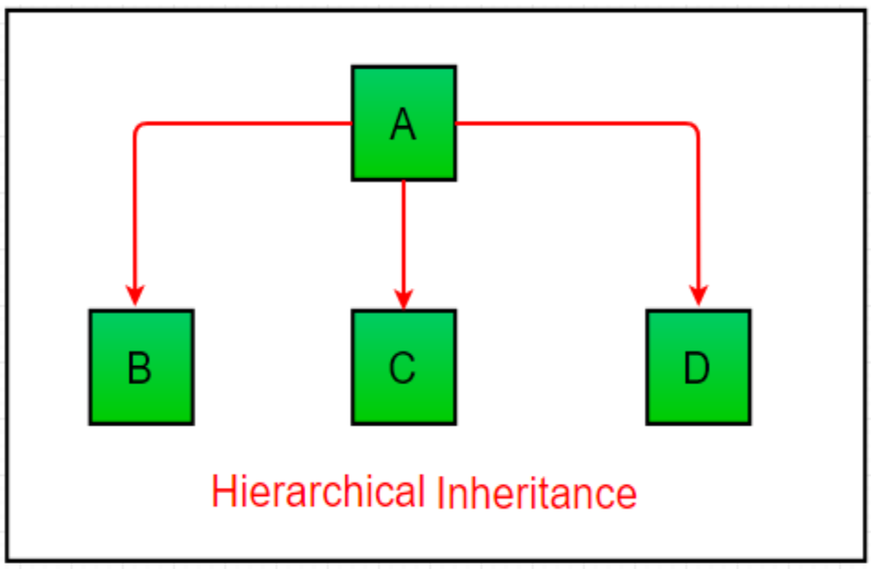


Example

```
class aaa
{
}
class bbb extends aaa
{
}
class ccc extends bbb
{
}
```

Hierarchical- (One base ,More than one sub)

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.

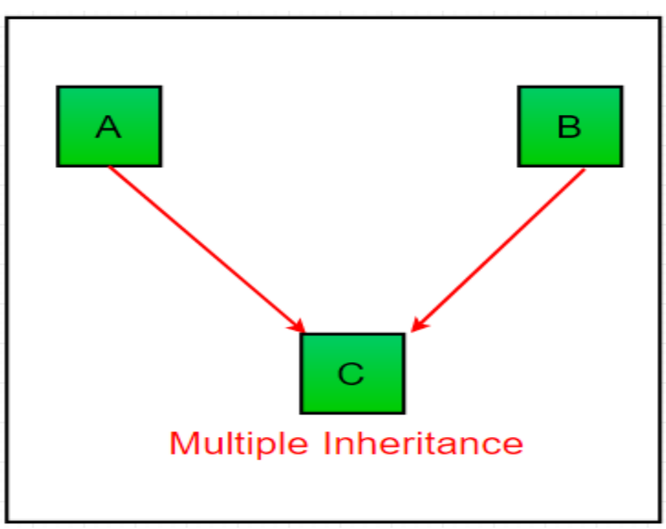


Example

```
class aaa
{
}
class bbb extends aaa
{
}
class ccc extends aaa
{
}
```

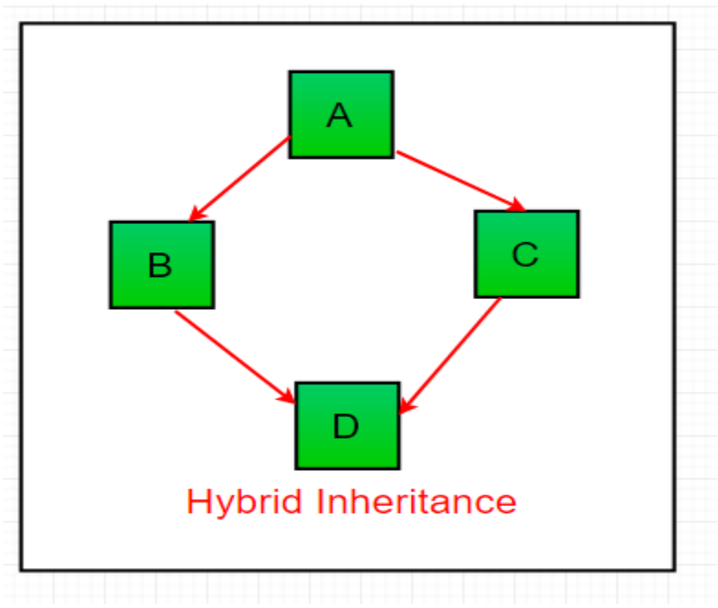
Multiple Inheritance (Through Interfaces) :

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



Hybrid Inheritance-More than one type of Inheritance

- **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.



Access Modifiers in Java

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example Program-1

```
class Teacher
{
 String designation = "Teacher";
String collegeName = "Beginnersbook";
 void aaa()
{
System.out.println("Teaching");
}
}
```

```
public class PhysicsTeacher extends Teacher
{
String mainSubject = "Physics";
public static void main(String args[])
{
 PhysicsTeacher obj = new PhysicsTeacher(); System.out.println(obj.collegeName);
 System.out.println(obj.designation);
System.out.println(obj.mainSubject);
obj.aaa()
} }
```

Example Program-2

```
Import java.io.*;
class Animal
 {
   public void eat()
 {
      System.out.println("I can eat");
   }
   public void sleep()
 {
      System.out.println("I can sleep");
   }
}
class Dog extends Animal
{
 public void bark()
{
      System.out.println("I can bark");
  } }
class Main
{
   public static void main(String[] args)
 {
      Dog dog1 = new Dog();
```

```
    dog1.eat();
    dog1.sleep();
    dog1.bark();
  }}
```

2.Interface

- An interface is a completely "**abstract class**" that is used to group related methods with empty bodies.
- To access the interface methods, the interface must be "implemented“ by another class with the implements keyword (instead of extends).
- The body of the interface method is provided by the "implement" class
- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method bodies. It is used to achieve abstraction and multiple inheritance in Java.

Example:

```
interface Animal
 {
public void animalSound();
public void run();
}
class dog implements Animal
{
}
```

Relationship between class and Interface

- Interface ----class (implements)
- class ---- class (extends)
- Interface ---- Interface (extends)

Multiple Interfaces:

To implement multiple interfaces, separate them with a comma

Multiple Interface

```
interface aaa
 {
}
```

```
interface bbb
{
}
interface ccc implements aaa,bbb
{
}
```

Example Program

```
interface Animal
{
  public void animalSound();
  public void sleep();
}
class Pig implements Animal
{
  public void animalSound()
 {
    System.out.println("The pig  says: wee wee");
  }
public void sleep()
{
    System.out.println("Zzz");
  }
}
class MyMainClass
 {
  public static void main(String[]  args)
{
    Pig myPig = new Pig();
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Output
The pig says: wee wee
Zzz

3.Package in java .

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 in java can be categorized in two forms, 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.

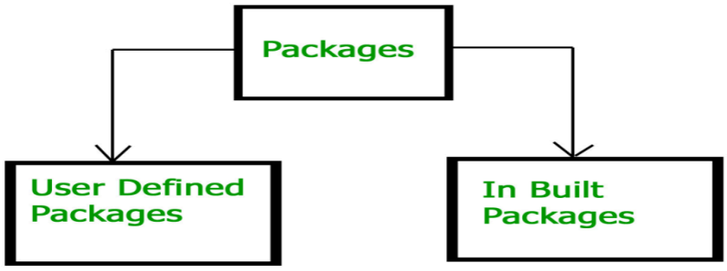
Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

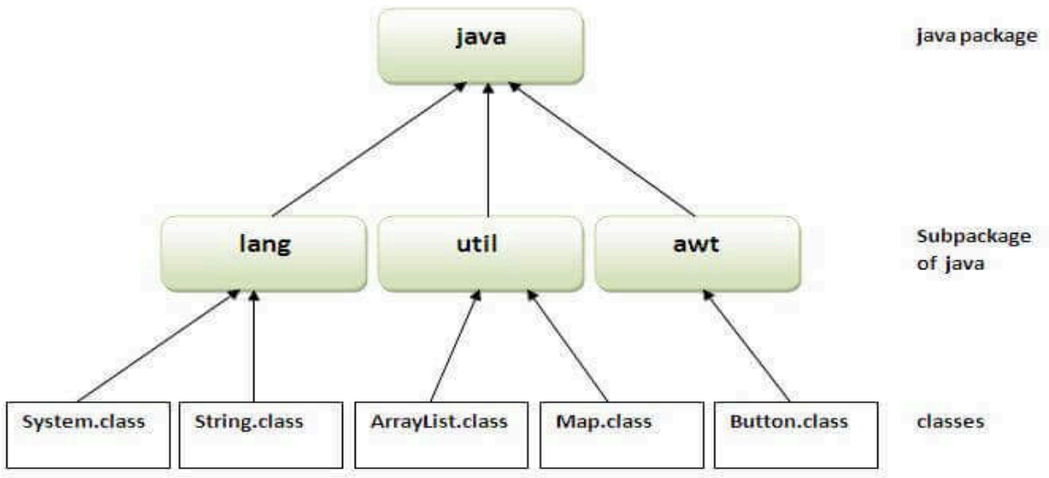
2) Java package provides access protection.

3) Java package removes naming collisions.

Types of packages:



Predefined Package



a.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:

java.lang:

Contains language support classes(e.g classes which define primitive data types, math operations). This package is automatically imported.

java.io:

Contains classed for supporting input / output operations.

java.util:

Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

java.applet:

Contains classes for creating Applets.

java.awt:

Contain classes for implementing the components for graphical user interfaces (like buttons , ;menus etc).

java.net:

Contain classes for supporting networking operations.

b.User-defined packages

These are the packages that are defined by the user. First we create a package using the package keyword. While creating a package, programmers must choose a name for the package and include a package statement program that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

Creating a Package

To create a package, you choose a name for the package (and put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

Syntax

package nameOfPackage;

class class-name

{-i---}

Example

```
package  mypack;
public class Simple
{
public static void main(String args[])
{
System.out.println("Welcome to package");
}
}
```

Importing package

import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name

Using package names.*

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

import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Using packagename.classname

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

import pack.A;
class B
{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Sub package

Package inside the package is known as a subpackage. The Packages that come lower in the naming hierarchy are called "sub package" of the corresponding package higher in the hierarchy i.e. the package that we are putting into another package is called "sub package"

Example program

```
package mypack;

public class balance

{

String name;
```

```
double bal;
public balance(String n, double b) {
name=n;
bal=b;
}
public void show() {
if(bal>0) {
System.out.print("□");
System.out.println(name+"$"+bal);
}
else
System.out.println("negative value");
}
}
import mypack.*;
class testBalance
{
public static void main(String arg[]) {
balance test=new balance("j.j.jaspers",-99.88);
test.show();
}
}
```

OUTPUT:

□

j.j.jaspers $ 99.88

- **Unit -4**

1.Exception handling

Exception

An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. ... This block of code is called an **exception handler**.

Exception handling?

Exception is the place where a problem has occurred, Handling is the place for the solution to the specific exception.

Use try block to write code that may disrupts the normal program flow Use catch block to handle it

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

Types of Exception

- RunTimeException or Unchecked Exception
- CompileTimeException or checked Exception
- Error

Example Program-1

```
public class JavaExceptionExample
{
public static void main(String args[])
{
Try
{
int data=100/0;
}
catch(ArithmeticException e)
{
System.out.println(e);
}
//rest code of the program.
```

Example Program-2

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

```
{
int myNumbers[] = {1, 2, 3};
 System.out.println(myNumbers[1]);
System.out.println(myNumbers[2]);
 System.out.println(myNumbers[10]);
// error!
}
}
```

Example Program-3

```
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.");
} }
}
```

RuntimeException or Unchecked Exception

These types of exceptions can be easily validated during code development. A developer can avoid this exception completely in their application by writing good validation. Exception that are subclass of RuntimeException are called as Unchecked Exception

Example

ArithmeticException

NumberFormatException

NullpointerException

CompileTimeException or checked Exception

Exceptions that cannot be validated during development are called checked Exceptions, the compiler will force you to use try catch blocks or to throw the exception.

Example

FileNotFoundException

SQLException

InterruptedException

Error

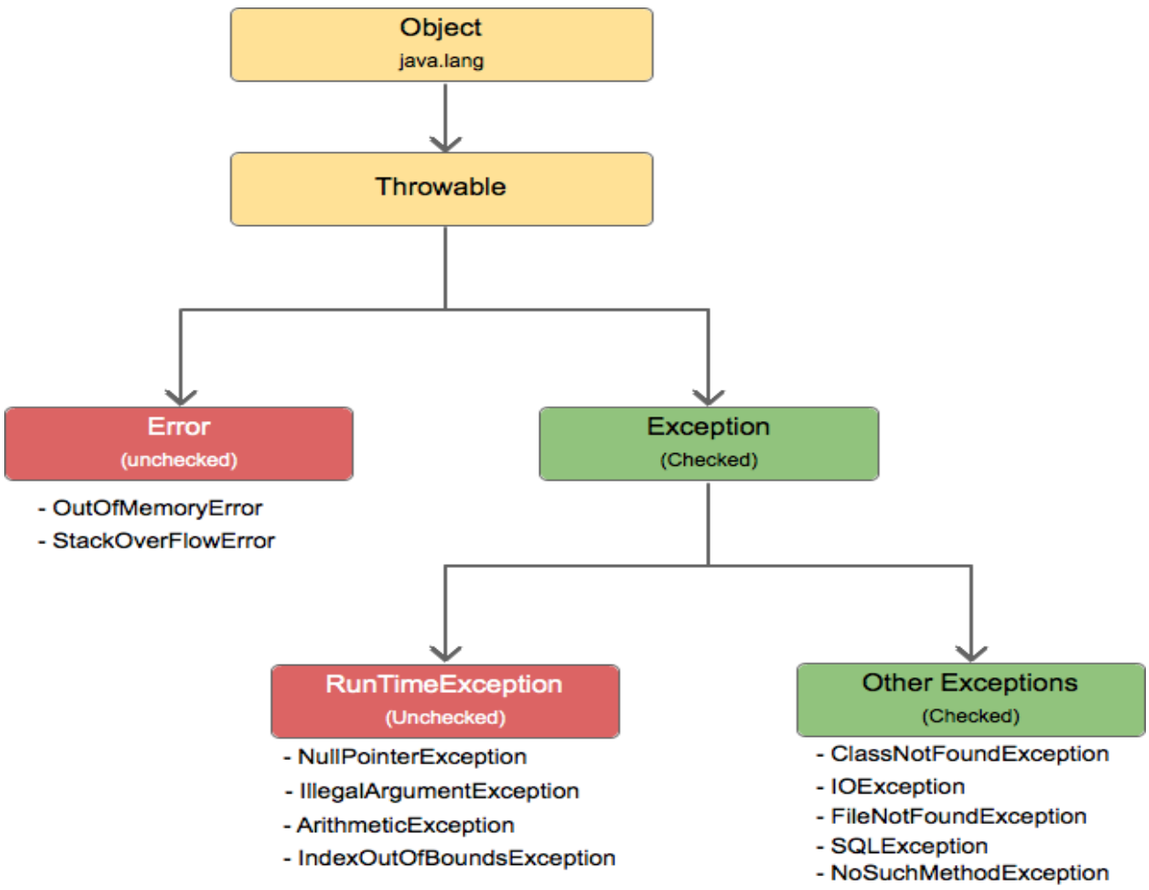
An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. That is, Error and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.

Example

IOError

LinkageError

VirtualMachineError



Try block

The try block contains a set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block. A try block must be followed by catch blocks or finally blocks or both.

Syntax:

```
try
{
 //statements that may cause an exception
 }
```

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in a try block, the corresponding catch block that handles that particular exception executes.

Syntax

```
try
{
//statements that may cause an exception
 } catch (exception(type) e(object))
{
//error handling code
}
```

Multiple catch blocks

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

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception

Finally block

A finally block contains all the crucial statements that must be executed whether exception occurs or not.

```
try
{
//Statements that may cause an exception
}
 Catch
 {
 //Handling exception
 }
finally
{
//Statements to be executed
}
```

Throw and Throws

 =

Finally and Close()

close() statement is used to close all the open streams in a program. Its a good practice to use close() inside the finally block.

```
finally{

Obj.close();

}
```

Finally block and System.exit()

System.exit() statement behaves differently than return statement. Unlike return statement whenever System.exit() gets called in try block then Finally block doesn't execute.

```
try

 {

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

System.exit(0)

 }
```

Types of Exception in Java with Examples

- **ArithmeticException**
  It is thrown when an exceptional condition has occurred in an arithmetic operation.
- **ArrayIndexOutOfBoundsException**
  It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- **ClassNotFoundException**
  This Exception is raised when we try to access a class whose definition is not found
- **FileNotFoundException**
  This Exception is raised when a file is not accessible or does not open.
- **IOException**
  It is thrown when an input-output operation failed or interrupted
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
- **NoSuchFieldException**
  It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException**
  It is thrown when accessing a method which is not found.
- **NullPointerException**
  This exception is raised when referring to the members of a null object. Null represents nothing

- **NumberFormatException**
  This exception is raised when a method could not convert a string into a numeric format.

- **RuntimeException**
  This represents any exception which occurs during runtime.

- **StringIndexOutOfBoundsException**
  It is thrown by String class methods to indicate that an index is either negative than the size of the string

2.Thread

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

Each process has an address in memory. In other words, each process allocates a separate memory area.

The process is heavyweight.

Cost of communication between the processes is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the threads is low.

Thread

- A thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution.

- Threads are independent. If an exception occurs in one thread, it doesn't affect other threads. It uses a shared memory area

Multithreading

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

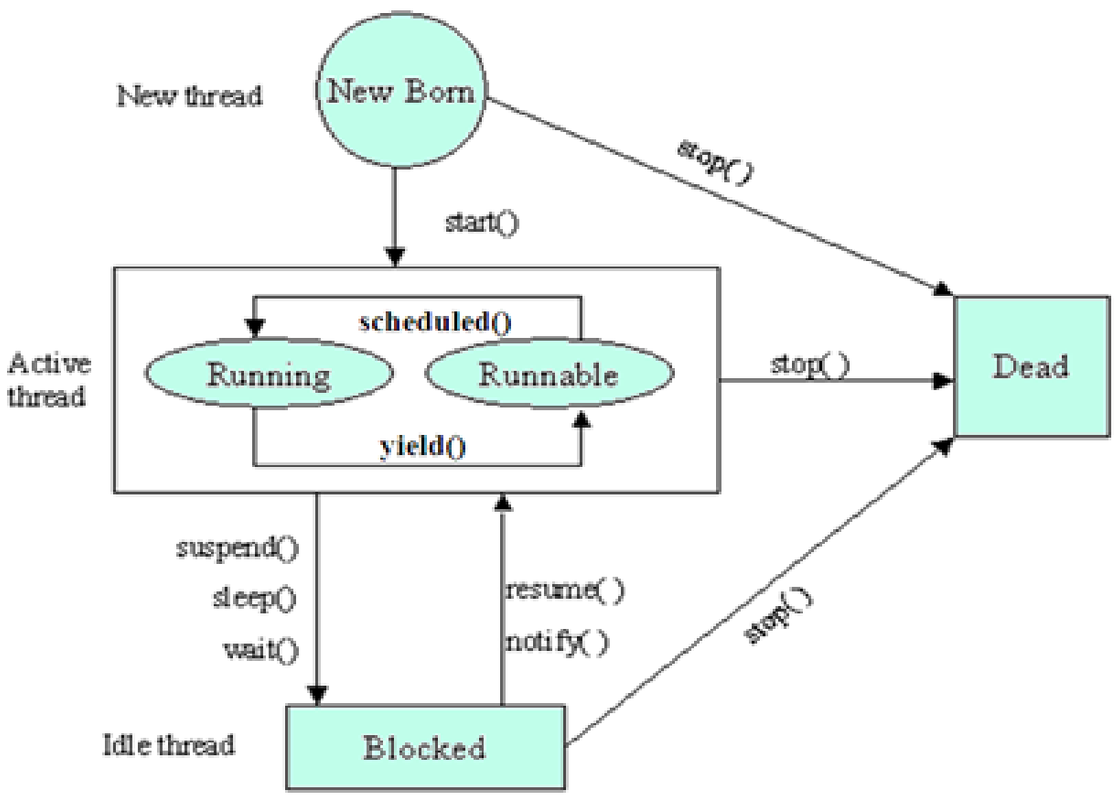
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

- However, we use multithreading rather than multiprocessing because threads use a shared memory area. They don't allocate a separate memory area so saves memory, and context-switching between the threads takes less time than process.

Advantages of Java Multithreading

- 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.

Thread Life Cycle



New

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it's a state internal to Java programming.

Runnable

When we call start() function on the Thread object, its state is changed to Runnable. The control is given to the Thread scheduler to finish its execution. Whether to run this thread instantly or keep it in a runnable thread pool before running, depends on the OS implementation of thread scheduler.

Example

```
MyThread t1 = new MyThread();

t1.start();
```

Running

- When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the threads from the runnable thread pool and changes its state to Running.
- Then the CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depending on time slicing, thread completion of run() method or waiting for some resources.
- Schedule()
- yield(()

Blocked/Waiting state:

When a thread is temporarily inactive, then it’s in one of the following states.

- Blockedrg
- Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread.

- **suspend()**
- **wait()**
- **sleep(t)**
- **notify()**
- **notifyAll()**
- **resume()**

A thread in this state cannot continue its execution any further until it is moved to a runnable state.

Terminated (Dead)

A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Dead state means that a **thread** has finished its execution or its run() method and when the stop() method is invoke. The stop() method kills the **thread** and the **thread** doesn't work further

Example

t1.stop();

<u>Thread Methods</u>

- public void run(): is used to perform action for a thread.
- public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
- public void sleep(long milliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- public void join(): waits for a thread to die.

- public void join(long milliseconds): waits for a thread to die for the specified milliseconds.
- public int getPriority(): returns the priority of the thread.
- public int setPriority(int priority): changes the priority of the thread.
- public String getName(): returns the name of the thread.
- public Thread currentThread(): returns the reference of currently executing thread.
- public int getId(): returns the id of the thread.
- public boolean isAlive(): tests if the thread is alive.
- public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public void suspend(): is used to suspend the thread(deprecated).
- public void resume(): is used to resume the suspended thread(deprecated).
- public void stop(): is used to stop the thread(deprecated)

Create a Thread

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Thread class

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

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

By extending the Thread class

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

Example

class class_name extends Thread

Example Program-extends thread

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

By implementing the Runnable interface.

- We create a new class which implements java.lang.Runnable interface and overrides run() method.
- Then we instantiate a Thread object and call start() method on this object.
- Runnable is an interface name .

Example

```
 class class_name implements Runnable
```

Example Program-implements Runnable

```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Priority of a Thread (Thread Priority):

- Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the 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.
- The minimum thread priority in java is 1 and maximum or highest thread priority is 10. We will see a program example to set and get thread priority. **Default** priority of thread in java is = 5. We can set priority of a thread within this range only.

Thread Priority

- public static int MIN_PRIORITY(1)
- public static int NORM_PRIORITY(5)
- public static int MAX_PRIORITY(10)

Get and Set methods in Thread priority

- getPriority()
- setPriority(int Priority)

Example Program(1) –Thread Priority

```
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

Example Program(2) –Thread Priority

```
class MyThread extends Thread
{
 public void run()
{
System.out.println("Thread Running...");
 }
public static void main(String[]args)
{
MyThread p1 = new MyThread();
p1.start(); // Starting thread
p1.setPriority(2); //Setting priority
 int p = p1.getPriority();// Getting priority
 System.out.println("thread priority : " + p);
} }
```

Synchronization

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- Java provides a way of creating threads and synchronizing their tasks by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object.
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.
- This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Why use Synchronization

- To prevent thread interference.
- To prevent consistency problems.

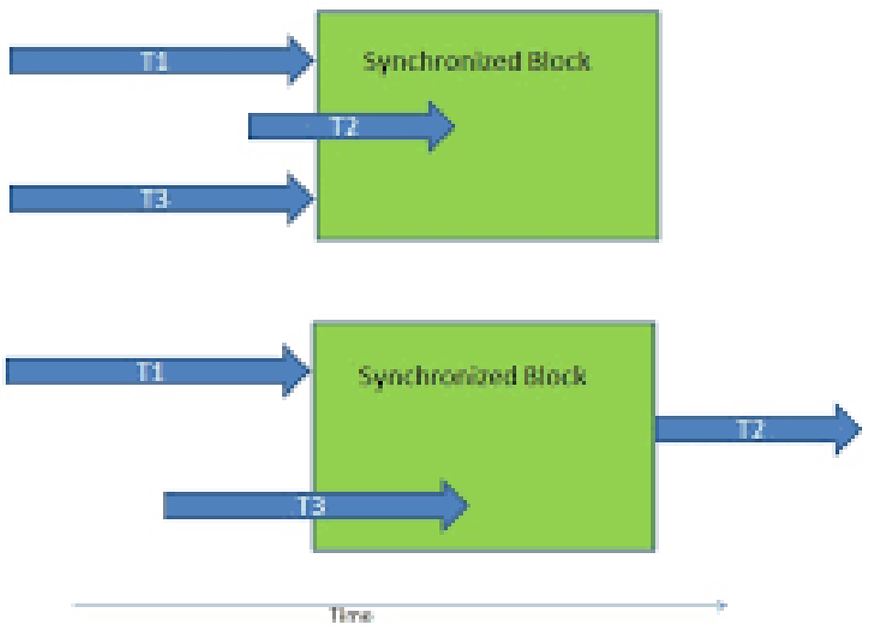
Types of Synchronization

- Process Synchronization
- Thread Synchronization

Mutual Exclusive

- Synchronized method.
- Synchronized block.
- static synchronization.

Synchronization block



In Synchronized Block, other threads will have to wait when one thread is in

Synchronized Blocks

- synchronized block you may lock on an object other than "***this***" which allows to be much more flexible.
- **Example:**

```
public void add(int value){
Student s=new Student();
        synchronized(s) {
                this.count += value;
        }
 }
```

SynchronizationBlockDemo

Example Program

```
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
```

3.Applet

- An applet is a Java program that runs in a Web browser. ... An applet is a Java class that extends the java. applet. Applet class. A main() method is not invoked on an applet, and an applet class will not define main().
- Applet is a special type of program that is embedded in the webpage to generate dynamic content. It runs inside the browser and works at the client side.

Advantage

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

- 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().

Difference between Applet and Application

Java Applications are the stand-alone programs which can be executed independently	Java Applets are small Java programs which are designed to exist within HTML web document
Java Applications must have main() method for them to execute	Java Applets do not need main() for execution
Java Applications just needs the JRE	Java Applets cannot run independently and require API's
Java Applications do not need to extend any class unless required	Java Applets must extend java.applet.Applet class
Java Applications can execute codes from the local system	Java Applets Applications cannot do so
Java Applications has access to all the resources available in your system	Java Applets has access only to the browser-specific services

Example Program

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("welcome",150,150);
} }
```

Html Program

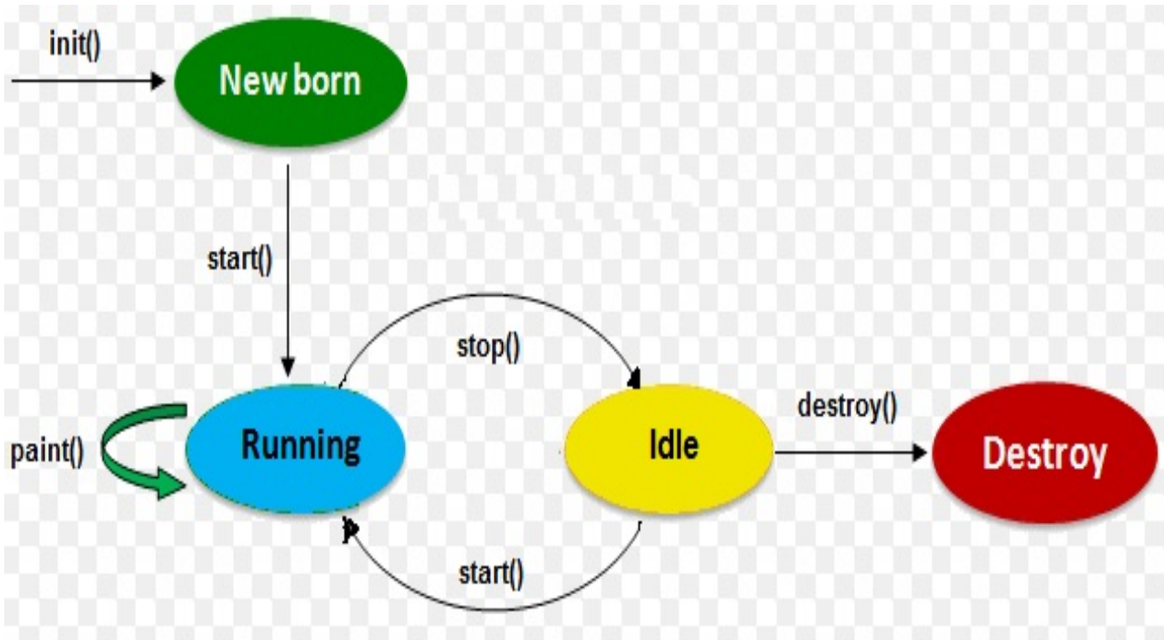
```
<html>
<body>
<applet code="First.class" width="300" height="300">
```

</applet>

</body>

</html>

Applet Life Cycle



Applet Methods

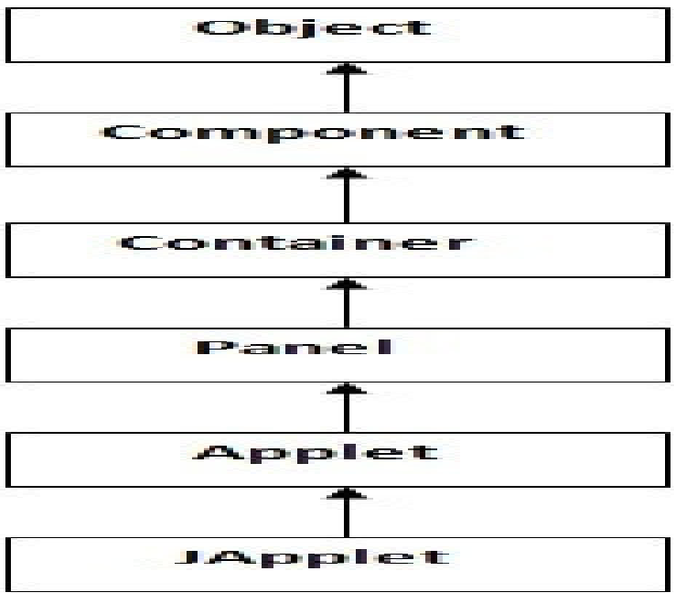
- **init( ) :** The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
- **start( ) :** The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Note that init( ) is called .
- **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called.
- **stop( ) :** The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page. 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.
- **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( ).

Applet Hierarchy in Java

- class java.lang.**Object**
  - class java.awt.**Component**
    - class java.awt.**Container**

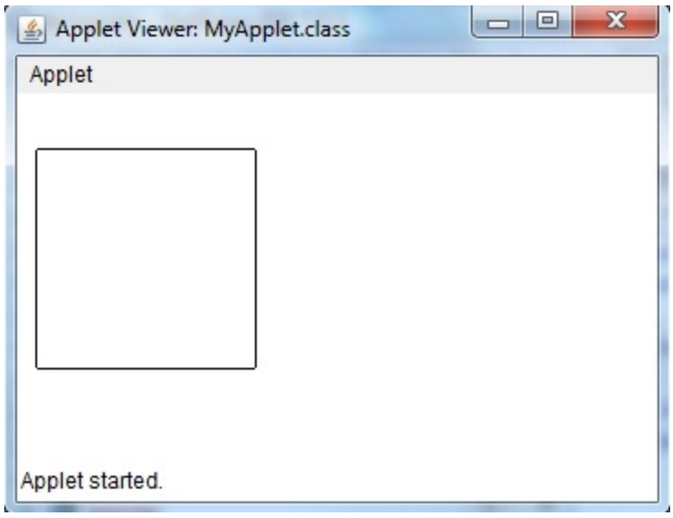
class java.awt.**Panel**

class java.applet.**Applet**



Example Program-1

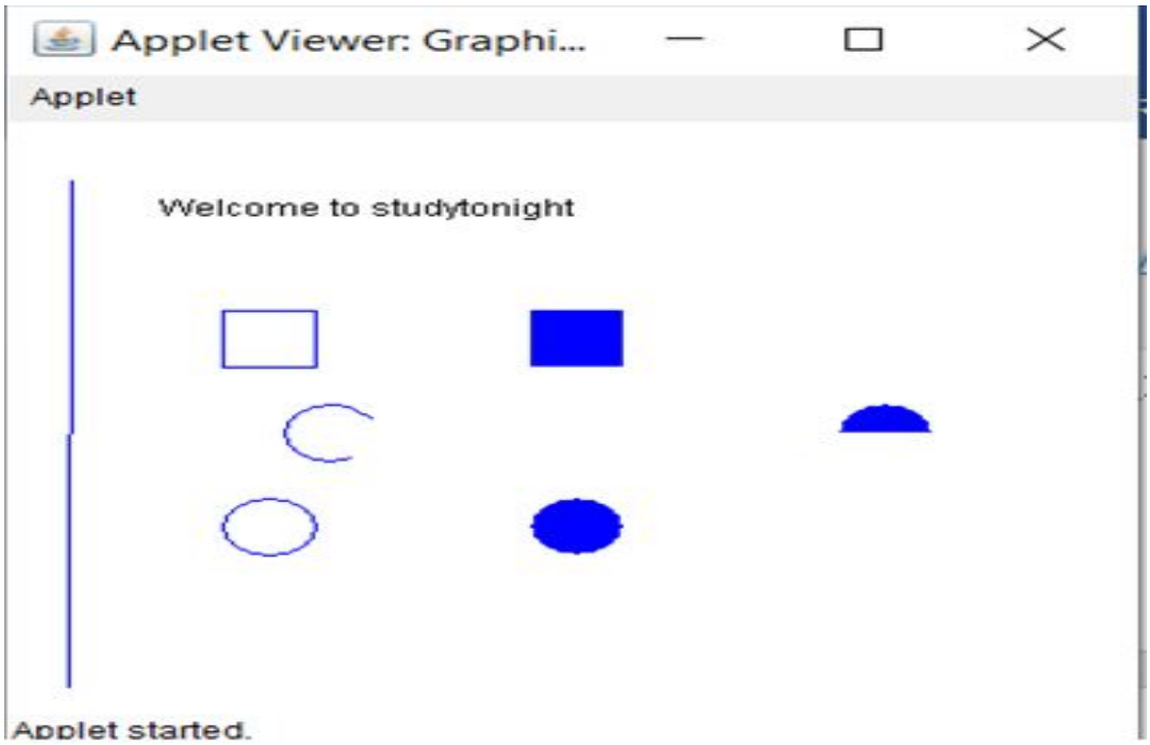
```
import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
int height, width;
public void init()
{
height = getSize().height;
width = getSize().width;
setName("MyApplet");
 }
public void paint(Graphics g)
{
g.drawRoundRect(10, 30, 120, 120, 2, 3);
} }
<html>
<body>
<applet code=“MyApplet.class" width="300" height="">
</applet>
</body>
</html>
```



Example Program-2

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo1 extends Applet
{
public void paint(Graphics g)
{
g.setColor(Color.black);
g.drawString("Welcome to studytonight",50, 50); g.setColor(Color.blue);
g.fillOval(170,200,30,30); g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
g.drawLine(21,31,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
 g.drawOval(70,200,30,30);
} }
<html>
<body>
<applet code="GraphicsDemo1.class" width="300" height="300">
 </applet>
</body>
 </html>
```

OutPut



Passing Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax

public String getParameter(String parameterName)

Steps

- To pass the parameters to the Applet we need to use the param attribute of <applet> tag.
- To retrieve a parameter's value, we need to use the getParameter() method of Applet class.

Example

```
import java.applet.Applet;

import java.awt.Graphics;

public class UseParam extends Applet

{

public void paint(Graphics g)

{

String str=getParameter("msg");
```

```
g.drawString(str,50, 50);
}
}
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```

Output:

Welcome to Apple

Unit 5

1.Stream classes in java

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files. The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

Stream – A sequence of data.
Input Stream: reads data from source.
Output Stream: writes data to destination.

Uses of io streams in java.

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode.

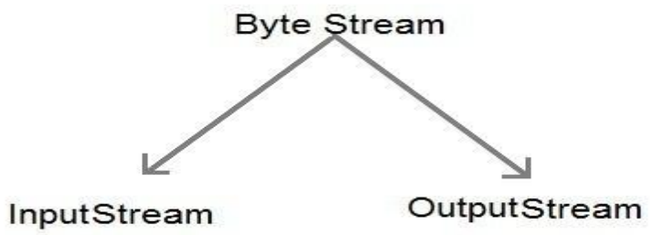
Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter.

Types of Streams in java

- Byte Stream classes
- Character Stream classes

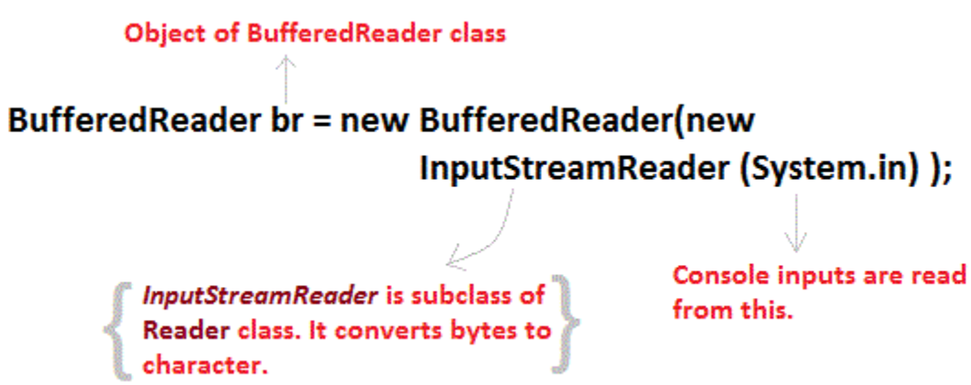
A) Byte stream classes.

Java Byte streams are used to perform input and output of 8-bit **bytes.** Byte stream is defined by using two abstract classes at the top of hierarchy, they are InputStream and OutputStream. For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

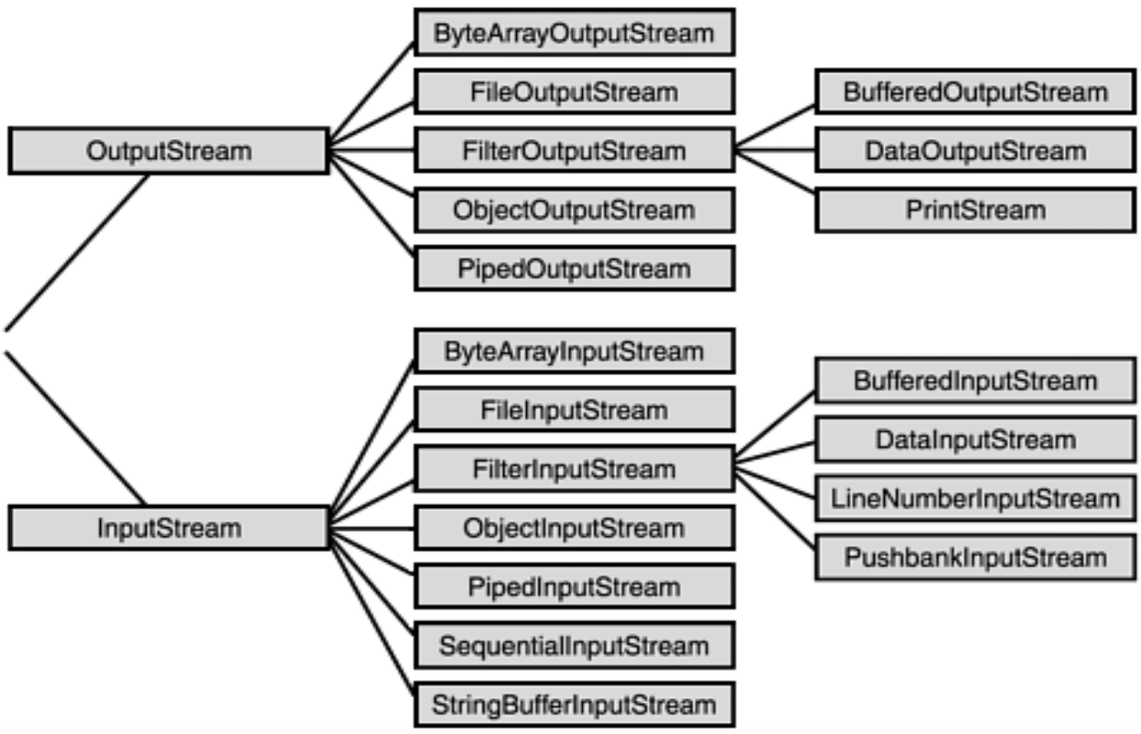


- **BufferedInputStream** -Used for Buffered Input Stream.
- **BufferedOutputStream** -Used for Buffered Output Stream.
- **DataInputStream** - Contains method for reading java standard data type
- **DataOutputStream-** - An output stream that contain method for writing java standard data type
- **FileInputStream**- Input stream that reads from a file
- **FileOutputStream** - Output stream that writes to a file.
- **InputStream** - Abstract class that describes stream input.
- **OutputStream** -Abstract class that describes stream output.
- **PrintStream** - Output Stream that contain print() and println() method

Read Console input in java



Sub classes for Input Stream and Output Stream



Input streams - for reading data from streams

Input Stream Class in Java. InputStream class is the superclass of all the io classes i.e. representing an input stream of bytes. It represents input stream of bytes. Applications that are defining subclass of InputStream must provide method, returning the next byte of input

Input stream Constructor :

- InputStream() : Single Constructor

Method in Input Stream

Method	Syntax	Description
mark()	public void mark(int arg)	marks the current position of the input stream. It sets readlimit i.e. maximum number of bytes that can be read before mark position becomes invalid.
read()	public abstract int read()	reads next byte of data from the Input Stream
close()	public void close()	closes the input stream and releases system resources associated with this stream to Garbage Collector.
read()	public int read(byte[] arg)	reads number of bytes of arg.length from the input stream to the buffer array arg. The bytes read by read() method are returned as int.
reset()	public void reset()	invoked by mark() method. It repositions the input stream to the marked position.
markSupported()	public boolean markSupported()	checks whether the input stream is supporting the mark() and reset() method or not.
skip()	public long skip(long arg)	skips and discards arg bytes in the input stream.

Output Stream

OutputStream class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

Output Stream constructor

- **OutputStream() :** Single Constructor

Methods in Output Stream

void close() : Closes this output stream and releases any system resources associated with this stream

void flush() : Flushes this output stream and forces any buffered output bytes to be written output

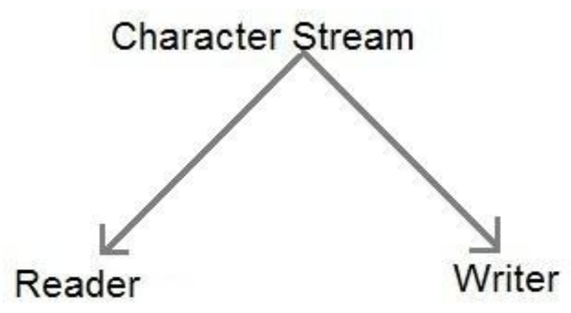
void write(byte[] b) : Writes b.length bytes from the specified byte array to this output stream

void write(byte[] b, int off, int len) : Writes len bytes from the specified byte array starting at offset off to this output stream.

abstract void write(int b) : Writes the specified byte to this output stream

B) Character Stream classes in java

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter.



BufferedReader - Handles buffered input stream.

BufferedWriter- Handles buffered output stream.

FileReader- Input stream that reads from file.

FileWriter- Output stream that writes to file.

InputStreamReader -Input stream that translate byte to character

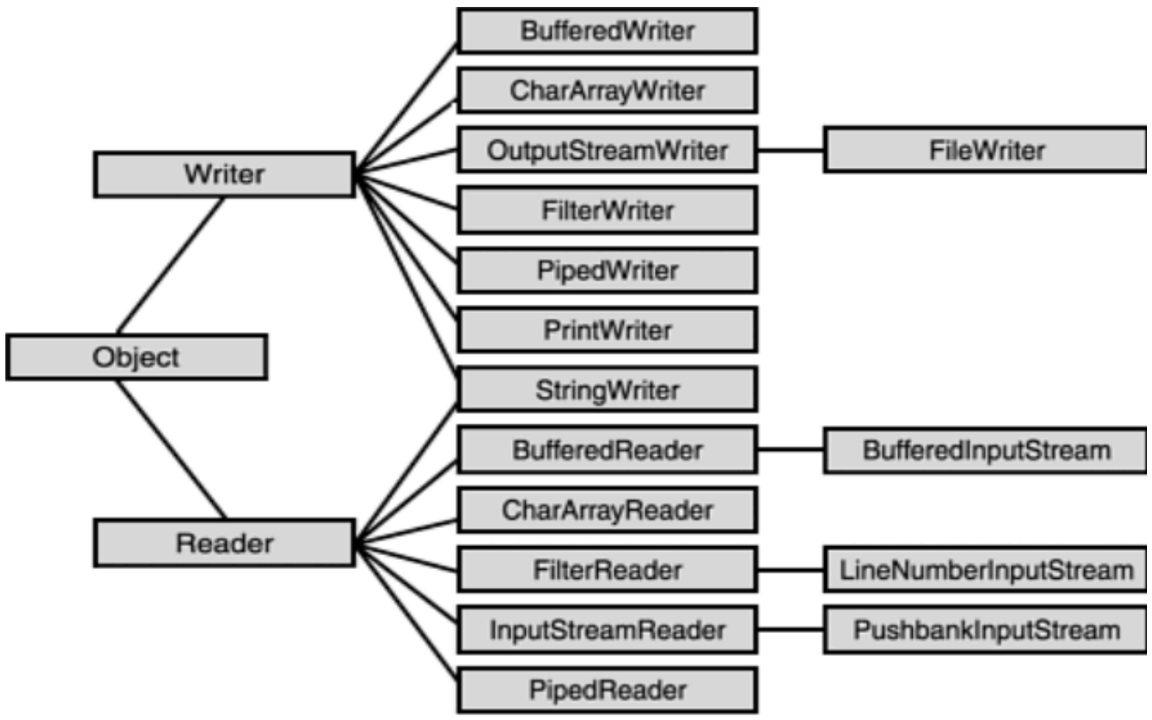
OutputStreamReader -Output stream that translates character to byte.

PrintWriter -Output Stream that contains print() and println() method.

Reader -Abstract class that define character stream input

Writer -Abstract class that define character stream output

Sub Classes in Reader and Writer class



Reader class

Reader classes are used to read 16-bit unicode characters from the input stream. The Reader class is the superclass for all character-oriented input stream classes. All the methods of this class throw an IOException. Being an abstract class, the Reader class cannot be instantiated hence its subclasses are used.

Constructor in Reader class

Reader() It creates a new character-stream reader whose critical sections will synchronize on the reader itself

Reader(Object lock) It creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods in Reader class;

close() It closes the stream and releases any system resources associated with it.

mark(int readAheadLimit) It marks the present position in the stream.

read() It reads a single character.

read(char[] cbuf) It reads characters into an array.

ready() It tells whether

reset() It resets the stream.r this stream is ready to be read.

Writer class

Writer classes are used to write 16-bit Unicode characters onto an outputstream. The Writer class is the superclass for all character-oriented output stream classes .All the methods of this class throw an IOException. Being an abstract class, the Writer class cannot be instantiated hence, its subclasses are used

Constructor in Writer class

Writer() It creates a new character-stream writer whose critical sections will synchronize on the writer itself.

Writer(Object lock) It creates a new character-stream writer whose critical sections will synchronize on the given object.

Writer class method:

append(char c) It appends the specified character to this writer.

close() It closes the stream, flushing it first.

flush() It flushes the stream.

write(char[] cbuf) : It writes an array of characters.

write(int c) It writes a single character.

2. Utility Classes in java

Utility class is classes that defines a set of methods that perform common, often re-used functions.Examples of utility classes include java.util.Collections which provides several utility methods (such as sorting) on objects that implement a Collection (java.util.collection)

Date class

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

Constructor in Date class.

- **Date()** : Creates date object representing current date and time.
- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
- **Date(int year, int month, int date)**
- **Date(int year, int month, int date, int hrs, int min)**
- **Date(int year, int month, int date, int hrs, int min, int sec)**
- **Date(String s).**

Methods in Date class.

- **boolean after(Date date) :** Tests if current date is after the given date.
- **boolean before(Date date) :** Tests if current date is before the given date.
- **int compareTo(Date date) :** Compares current date with given date. Returns 0 if the argument Date is equal to the Date

- **long getTime()** : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- **void setTime(long time)** : Changes the current date and time to given time.

Example Program:

```
import java.util.Date;

class DateDemo

{

public static void main(String args[])

{

Date date = new Date ();

System.out.println (date);

long msec = date.getTime();

System.out.println ("Milliseconds since Jan. 1, 1970 GMT = " + msec);

}

}
```

OUTPUT:

```
Fri Sep 25 20:11:45 IST 2015

Milliseconds since Sep.25, 2015 GMT = 1443192105482
```

Calendar class

Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.

As it is an Abstract class, we cannot use a constructor to create an instance. Instead, we will have to use the static method Calendar.getInstance() to instantiate and implement a subclass.

Methods used in Calendar class.

- **Calendar.getInstance():** return a Calendar instance based on the current time in the default time zone with the default locale.
- **Calendar.getInstance(TimeZone zone)**
- **Calendar.getInstance(Locale aLocale)**
- **Calendar.getInstance(TimeZone zone, Locale aLocale)**

Example program

```
import java.util.*;

class GregorianCalendarDemo

{

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

```
{
String months[ ] = {"Jan", "Feb", "Mar", "Apr","May", "Jun", "Jul", "Aug", "Sep", "Oct",
"Nov", "Dec"};
int year;
GregorianCalendar gcalendar = new GregorianCalendar();
System.out.print("Date: ");
System.out.print(months[gcalendar.get(Calendar.MONTH)]);
System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
System.out.println(year = gcalendar.get(Calendar.YEAR));
System.out.print("Time: ");
System.out.print(gcalendar.get(Calendar.HOUR) + ":");
System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
System.out.println(gcalendar.get(Calendar.SECOND));
if(gcalendar.isLeapYear(year)) {
System.out.println("The current year is a leap year");
}
else
{   System.out.println("The current year is not a leap year");
} } }
```

OUTPUT:

```
Date: Sep 25 2015
Time: 8:17:42
The current year is not a leap year
```

Random class

An instance of java Random class is used to generate random numbers. This class provides several methods to generate random numbers of type integer, double, long, float etc. Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

Constructors in random class:

- **Random()**: Creates a new random number generator
- **Random(long seed)**: Creates a new random number generator using a single long seed

Methods in Random class.

- **java.util.Random.doubles():** Returns an effectively unlimited stream of pseudo random double values,
- **java.util.Random.ints():** Returns an effectively unlimited stream of pseudo random int values
- **java.util.Random.longs():** Returns an effectively unlimited stream of pseudo random long values.

Scanner class

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings

Methods used in Scanner class

- **nextBoolean()** Reads a boolean value from the user
- **nextByte()** Reads a byte value from the user
- **nextDouble()** :Reads a double value from the user
- **nextFloat()** Reads a float value from the user
- **nextInt()** Reads a int value from the user
- **nextLine()** Reads a String value from the user
- **nextLong()** Reads a long value from the user
- **nextShort()** Reads a short value from the user

Example Program

```
import java.io.*;
import java.util.Scanner;
public class Add
{
public static void main(String[] args)
{
int num1, num2, sum;
Scanner sc = new Scanner(System.in);
System.out.println("Enter First Number: ");
num1 = sc.nextInt();
System.out.println("Enter Second Number: ");
 num2 = sc.nextInt();
sc.close();
sum = num1 + num2;System.out.println("Sum of these numbers: "+sum);
}
```

}

String Tokenizer class

String Tokenizer class in Java is used to break a string into tokens. Example: A String Tokenizer object internally maintains a current position within the string to be tokenized .A token is returned by taking a substring of the string that was used to create the String Tokenizer object.

Constructor used in String Tokenizer

- **String Tokenizer(String str):** creates StringTokenizer with specified string.
- **String Tokenizer(String str, String delim):** creates StringTokenizer with specified string and delimeter.
- **String Tokenizer(String str, String delim, boolean returnValue): c**reates String Tokenizer with specified string, delimeter and returnValue. If the return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods used in String Tokenizer.

- **boolean hasMoreTokens()** - bchecks if there are more tokens available.
- **String nextToken() -** returns the next token from the StringTokenizer object.
- **String nextToken(String delim) -**returns the next token based on the delimeter.
- **boolean hasMoreElements()** - same as hasMoreTokens() method.
- **Object nextElement()** - same as nextToken() but its return type is Object.
- **int countTokens() -** returns the total number of tokens.

Example Program

```
import java.util.StringTokenizer;
public class Simple
{
 public static void main(String args[])
{
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
     while (st.hasMoreTokens())
{
         System.out.println(st.nextToken());
     }
   }
}
```

Output:

my
name
is

khan

3. FILES IN JAVA

Some of the common file handling operations are;

1. Create file
2. Delete file
3. Open file
4. Close file
5. Read file
6. Write file
7. Change file permissions.
8. Check file permission
9. Rename of file
10. Getting the name of file
11. Getting the size of file

Create File

We can use the File class createNewFile() method to create a new file. This method returns true if the file is successfully created, otherwise it returns false. Once creating file following steps should be follow:

- Suitable name for file
- Data Types to be stored
- Purpose (Reading ,Writing, Updating)
- Methods of creating file

Delete File

File class delete method is used to delete a file or an empty directory

Read File

There are many ways to read a file in java. We can use BufferedReader, FileReader or Files class

FileReader

- This class inherits from the InputStreamReader Class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate.

- FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Constructors:

- **FileReader(File file) -** Creates a FileReader , given the File to read from

- **FileReader(FileDescriptor fd) -** Creates a new FileReader , given the File Descriptor to read from
- **FileReader(String fileName) -** Creates a new FileReader , given the name of the file to read from

Methods:

- **public int read () throws IOException** – Reads a single character. This method will block until a character is available
- **public int read(char[] cbuff) throws IOException** – Reads characters into an array. This method will block until some input is available

Write File

We can use FileWriter, BufferedWriter, Files or FileOutputStream to write file in java.

FileWriter

- This class inherits from the OutputStream class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable.

- FileWriter is meant for writing streams of characters.
- FileWriter creates the output file , if it is not present already.

Constructors:

- **FileWriter(File file)** - Constructs a FileWriter object given a File object.
- **FileWriter (File file, boolean append)** - constructs a FileWriter object given a File object.
- **FileWriter (FileDescriptor fd)** - constructs a FileWriter object associated with a file descriptor.
- **FileWriter (String fileName) -** constructs a FileWriter object given a file name.
- **FileWriter (String fileName, Boolean append)** - Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

Methods:

- **public void write (int c) throws IOException** – Writes a single character.
- **public void write (char [] stir) throws IOException** – Writes an array of characters.
- **public void write(String str)throws IOException –** Writes a string.

File Permissions

File class provides methods to get file permission details as well as change them. Java provides a number of method calls to check and change the permission of a file, such as a read-only file can be changed to have permissions to write.

File permissions are required to be changed when the user wants to restrict the operations permissible on a file. For example, a file permission can be changed from write to read-only because the user no longer wants to edit the file.

1. **Checking the file permissions**

Executable:

Tests whether the application can execute the file denoted by this abstract path name.

Syntax

public boolean canExecute()

Readable:

Tests whether the application can read the file denoted by this abstract path name.

Syntax

public boolean canRead()

Writable:

Tests whether the application can modify the file denoted by this abstract path name.

Syntax

public boolean canWrite()

2. **Changing file permissions**

setExecutable

A convenience method to set the owner's execute permission for this abstract path name

Syntax

public boolean setExecutable(boolean executable)

setReadable:

A convenience method to set the owner's read permission for this abstract path name

Syntax

public boolean setReadable(boolean readable)

setWritable:

A convenience method to set the owner's write permission for this abstract path name.

Syntax

public boolean setWritable(boolean writable)

4. Event in java

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 a result of user interaction with the graphical ucser interface components.

For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from a list, scrolling the page are the activities that cause an event to happen.

Types of Event

- **Foreground Events** - Those events which require the direct interaction of user. For example, clicking on a button, moving the mouse, entering a character through the keyboard, selecting an item from a 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, and operation completion are the examples 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 has the code which is known as an event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events.

- **Event Source**

The source is an object on which an event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides classes for source objects.

- **Event Listener**

It is also known as an event handler. Listener is responsible for generating responses to an event. From a 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 processes the event and then returns.

- **Event classes**

Event Classes in Java are the classes defined for almost all the components that may generate events

- **ActionEvent :** Button, TextField, List, Menu

- **WindowEvent :** Frame
- **ItemEvent :** Checkbox, List
- **AdjustmentEvent :** Scrollbar
- **MouseEvent :** Mouse
- **KeyEvent :** Keyboard

- **Event Adapters:**

Event Adapters classes are abstract classes that provide some methods used for avoiding the heavy coding. Adapter class is defined for the listener that has more than one abstract method.

Event and Listener (Java Event Handling)

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener

The component with the Listener, many classes provide the registration methods. For example:

Button

public void addActionListener(ActionListener a){}

◦**MenuItem**

public void addActionListener(ActionListener a){}

◦**TextField**

public void addActionListener(ActionListener a){}

◦public void addTextListener(TextListener a){}

◦**TextArea**

public void addTextListener(TextListener a){}

◦**Checkbox**

public void addItemListener(ItemListener a){}

◦**Choice**

public void addItemListener(ItemListener a){}

◦**List**

public void addActionListener(ActionListener a){}

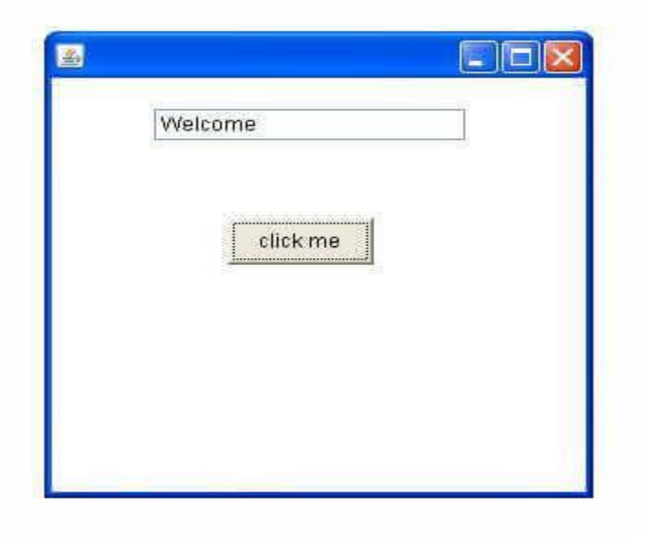
public void addItemListener(ItemListener a){}

Example Program

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){  .
create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
register listener
b.addActionListener(this);//passing current instance
add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
```

} }

Output



5.Legacy Classes and Interface

- Legacy classes and interfaces are the classes and interfaces that formed the collections framework in the earlier versions of Java and how now been restructured or re-engineered. They are fully compatible with the framework. ... All legacy classes were re-engineered to support generic in JDK5.
- Legacy Classes - Java Collections. ... It only defined several classes and interfaces that provide methods for storing objects. When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes.

The following are the legacy classes defined by **java.util** package

- Dictionary
- HashTable
- Properties
- Stack
- Vector
- There is only one legacy interface called **Enumeration**

Enumeration interface

- Enumeration interface defines method to enumerate(obtain one at a time) through collection of objects.
- This interface is superseded(replaced) by Iterator interface.
- However, some legacy classes such as Vector and Properties defines several method in which Enumeration interface is used.

- An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma.
- Note that they should be in uppercase letters

Vector class

- **Vector** is similar to **ArrayList** which represents a dynamic array.
- There are two differences between **Vector** and **ArrayList**. First, Vector is synchronized while ArrayList is not, and Second, it contains many legacy methods that are not part of the Collections Framework.
- With the release of JDK 5, Vector also implements Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the for-each loop.

Vector Constructor

Vector()

This creates a default vector, which has an initial size of 10.

Vector(int size)

This creates a vector whose initial capacity is specified by size.

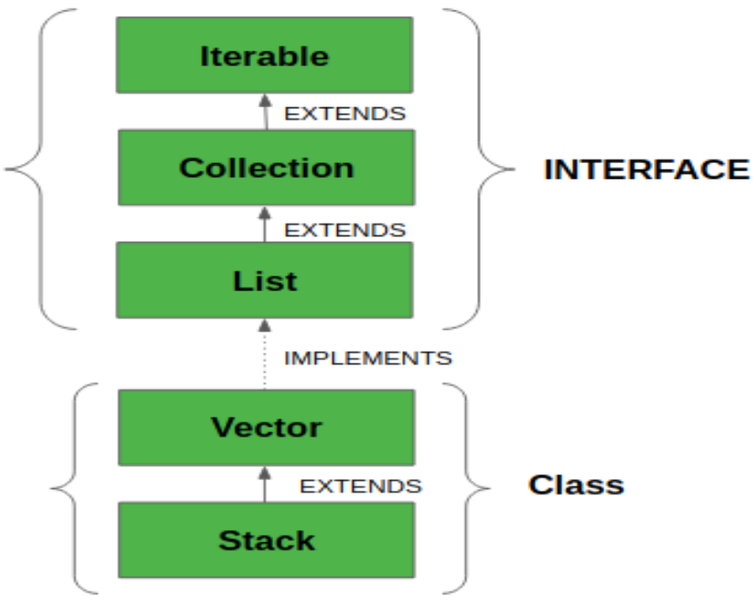
Vector(int size, int incr)

This creates a vector whose initial capacity is specified by size and whose increment is specified by incr. Vector(Collection c)

Vector Methods

Method	Description
void addElement(E element)	adds element to the Vector
E elementAt(int index)	returns the element at specified index
Enumeration elements()	returns an enumeration of element in vector
E firstElement()	returns first element in the Vector
E lastElement()	returns last element in the Vector
void removeAllElements()	removes all elements of the Vector

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.



Example Program

```
import java.io.*;

import java.util.*;

class VectorExample {

   public static void main(String[] args)

   {

     int n = 5;

     Vector<Integer> v = new Vector<Integer>(n);

     for (int i = 1; i <= n; i++)

     v.add(i);

     System.out.println(v);

     v.remove(3);

     System.out.println(v);


       for (int i = 0; i < v.size(); i++)

          System.out.print(v.get(i) + " ");

   } }
```

Output

[1, 2, 3, 4, 5]

[1, 2, 3, 5]

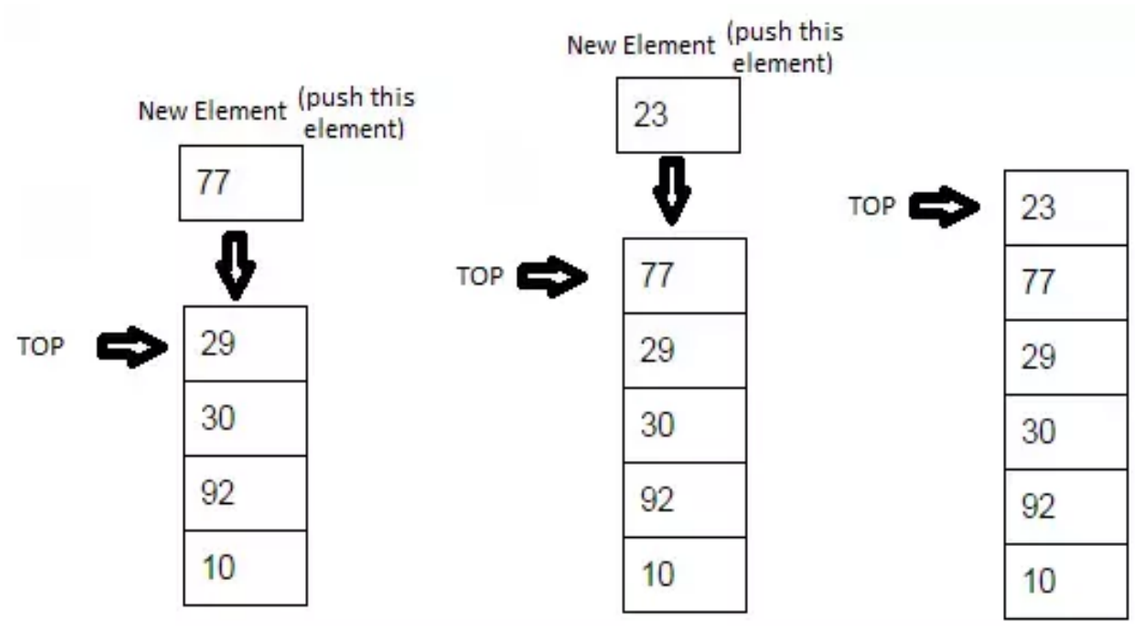
1 2 3

Stack class

- Stack class extends Vector.
- It follows last-in, first-out principle for the stack elements.
- It defines only one default constructor

Stack() //This creates an empty stack

If you want to put an object on the top of the stack, call push() method. If you want to remove and return the top element, call pop() method. An EmptyStackException is thrown if you call pop() method when the invoking stack is empty.



boolean empty()

Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.

Object peek( )

Returns the element on the top of the stack, but does not remove it.

Object pop( )

Returns the element on the top of the stack, removing it in the process.

Object push(Object element)

Pushes the element onto the stack. Element is also returned.

Example Program

```
import java.util.Stack;
```

```
class Main
 {
public static void main(String[] args)
{
Stack<String> animals= new Stack<>();
animals.push("Dog");
animals.push("Horse");
 animals.push("Cat");
System.out.println("Initial Stack: " + animals);
String element = animals.pop();
System.out.println("Removed Element: " + element);
} }
```