Unit – II Introduction to Object-oriented Programming

- Basic concepts of object-oriented programming
- Classes, instances, methods
- Static and non-static members
- Packages
- Inheritance and polymorphism, method overriding
- Final and abstract classes, abstract methods
- Interfaces
- Generics, enumeration
- Inner classes and anonymous classes
- Class loaders, class path

Basic Concepts of OOP

- **Class**
  - A blueprint/template that describes the behavior/state that the **object** of its type support.
  - **E.g.** A **class Car** that has
    - State (Attributes) :: Color, Brand, Weight, Model
    - Behaviors (Methods) :: Break, Accelerate, Slow Down, Gear Change
  - This is just a blueprint, it does not represent any Car, however using this we can create Car objects (or instances) that represents the car.
- **Object**
  - It is an instance of class.
  - When the objects are created, they inherit all the variables and methods from the class.

  E.g.
  - **Objects** : Audi, Toyota, Volvo, Maruti, Honda

Basic Concepts of OOP

- Class : Fruit
- Objects : Orange, Mango, Banana etc.

- Class : Shape
- Objects : Square, Rectangle, Triangle, Circle etc.

- Class : Pen
- Objects : Reynold, Parker , Camlin, Luxor etc.

- Class : Bicycle
- Objects : SportsBicycle, MountainBicycle , TouringBicycle etc.

- Class : Department
- Objects : Sales , Purchase, Finance, HR etc.

Basic Concepts of OOP

- **Abstraction**
  - A process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
  - **E.g.**

    The person only knows that pressing the accelerators will increase the speed of car **or** Applying brakes will stop the car

    But person does not know about the inner mechanism of the accelerator, brakes etc in the car.
  - **E.g.**

    When you login to your bank account online, you enter your user_id and password and press login.

    But what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you
  - In Java, we use **abstract class and interface** to achieve abstraction.

Basic Concepts of OOP

- **Encapsulation**
  - Binding object state(fields / attributes) and behavior(methods) together.
  - **E.g.**
    - A **Class** is an example of encapsulation.
    - Inform what the object does instead of how it does.
    - Keep data and methods safe from outside interference or misuse.

Basic Concepts of OOP

- **Inheritance**
  - The process by which one class acquires the properties and functionalities of another class is called inheritance.
  - Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.
  - **E.g.** A **parent class Teacher** and a **child class MathTeacher**. In the MathTeacher class we need not to write the same code which is already present in the parent class.
  - **E.g. In Class Teacher methods are**
    - Result_Info
    - Attendance_Info
    - Classwork_Video etc.
  - **E.g. In Class MathTeacher methods are**
    - Word_Problems
    - Patterns_Problems

Basic Concepts of OOP

- **Polymorphism**
  - It allows us to perform a single action in different ways.
  - E.g. **Addition of two numbers.**

    Both numbers may be int, float, double etc.
  - E.g. **Area of 2D shape :**

    Formula is different for Rectangle, Square, Triangle
  - In Java, we use **method overloading and method overriding** to achieve polymorphism.

General form of a class definition

```
class      class-name
{
dat type variable 1;
dat type variable 2;
……..
dat type variable N;

data type method-name 1 ( parameter –list 1 )  { body of method }
data type method-name 2 ( parameter –list 1 )  { body of method }
……..

data type method-name N ( parameter –list 1 )  { body of method }
}
```

Instance of a Class

class-name instance-name = **new** class-name( )

E.g. Object of inbuilt classes in Java

```
Integer obj1 = new Integer();
Scanner sc = new Scanner(System.in);
String str1 = new String();
```

Example of Garbage Collection

```
Integer obj1  =  new Integer (5);

System.out.println(obj1);

obj1 = new Integer(6);

System.out.println(“ Integer (5) can be recycled”);
```

Program : To create a **product class** having two fields **product number and product name**. Also create **two methods set_data()** to assign value to the class fields and **get_data** to retrieve the values of class fields.

```
class Product {
int prod_no;
String prod_name ;

// Set data
void set_data(int p , String p_name)    {
        prod_no = p;
        prod_name = p_name;
}
// Get Data
void get_data(){
                System.out.println("Product No. : " + prod_no);
                System.out.println("Product Name : " + prod_name);
}
}
```

```
class Test {
public static void main ( String args[] )
{
                Product p1 = new Product();
                p1.set_data(1,"Mousse");
                p1.get_data();

                Product p2 = new Product();
                p2.set_data(2,"Printer");
                p2.get_data();
}

}
```

Exercise

1. A program to create a **class named Rectangle** having member variable **length and breadth.** Create two methods **area** and **perimeter**.(**Formulas :** Area = length * breadth & Perimeter = 2 * (length + breadth)
2. A program to create a **class named MathOp** having two member variable **num1 and num2**. Class having the methods like **addition, subtraction, division and multiplication.**
3. A program to create a **class named Student** having members like student id, student name and marks of three subjects.. Also it having the methods like **set_data and get_result.**
4. A program to create a **class named MathFun** having methods like minimum, maximum, sum, average, poscount, negcount, mode and median.

Constructor

- Special method that create and initialize an object of particular class.

- It has the same name as its class & may accepts arguments.

- It does not have return type.

- If you do not declare explicit constructor, Java compiler automatically generate a default constructor that has no arguments.

- It does not call directly, it is invoked via the new operator.

- Class may have multiple constructor it is called **constructor overloading.**

Default Constructor

```
class   Point2D {
double   x ;
double   y ;

point2D (  )  {
        x  =  0;
        y  =  0;
}
}
```

Parameterized Constructor

```
class   Point2D {
double   x ;
double   y ;

point2D (  double ax , double  ay ) {
      x  =  ax ;
      y  =  ay ;
}
}
```

Constructor Overloaded

```
class   Point2D {
double   x ;
double   y ;

point2D (  ) {
        x  =  0;
        y  =  0;
}

point2D (  double ax ,  double  ay ) {
        x  =  ax ;
        y  =  ay ;
}
}
```

Example of Constructor Overloaded

```
class   Point2D {
double   x ;
double   y ;

Point2D (  )  {
            x  =  0;
            y  =  0;
}

Point2D (  double ax ,  double  ay )  {
            x  =  ax ;
            y  =  ay ;
}
void displayPointCoordinates ( )  {
            System.out.println ( " X co-ordinate value is :: " + x ) ;
            System.out.println ( " Y co-ordinate value is :: " + y ) ;
}
}

Class point2DConstructor {
public static void main (  String args [  ]  )   {
            Point2D  p1  = new Point2D ();    p1.displayPointCoordinates();
             Point2D  p2  = new Point2D ( 3 , 4 );    p2.displayPointCoordinates() ;
}
}
```

Example of Array

```
package javaapplication3;
import java.util.Scanner;

class Intdata {

public int search_data(int iarray[],int search_ele)    {
          for (int i = 0; i < iarray.length; i++) {
          if (iarray[i] == search_ele)
                    return i;
          }
    return (-1);
}
public void display_data(int iarray[])    {
for (int i = 0; i < iarray.length; i++) {
   System.out.print(" Value of element-- "+ i+1 +"::"+iarray[i]);
          } }
}
```

Example of Array

```
public class JavaApplication3 {
public static void main(String[] args) {
int ans , search_ele;
Scanner sc = new Scanner(System.in);
System.out.print(" Enter Number of elements:: ");
int no = sc.nextInt();
// Allocating memory for 5 object of type integer
int intarr[] = new int[no];
for (int i = 0; i < intarr.length; i++) {
   System.out.print(" Enter the value of element-- ("+ i +")::");
   intarr[i] = sc.nextInt();
}
Intdata i1;  i1 = new Intdata();   i1.display_data(intarr);
System.out.print(" Enter the search element in an array");
search_ele = sc.nextInt();
ans = i1.search_data(intarr, search_ele);
 if (ans != -1)
    System.out.print(" Search element at postion -- "+ ans );
 else        System.out.print(" Search element not found.");     }  }
```

Array of an Object.

```
import java.util.Scanner;
class Emp {
private int eno;
private String ename;

public void read_data()    {
Scanner sc = new Scanner(System.in);
System.out.print(" Enter Employee No :: ");
eno = sc.nextInt();
System.out.print(" Enter Employee Name :: ");
sc.nextLine();  // To clear the scanner buffer.
ename = sc.nextLine();
}
void display_data()    {
System.out.println("Emp No :: " + eno);
System.out.println("Emp Name :: " + ename); }
}
```

Array of an Object.

```
public class Test {
public static void main(String[] args) {
// Declaring an array of employee
Emp emparr[];
// Allocating memory for 5 object of type employee
emparr = new Emp[5];
for (int i = 0; i < emparr.length; i++) {
        // Initializing the elements of the array
        emparr[i] = new Emp();
        emparr[i].read_data();  }
for (int i = 0; i < emparr.length ; i++) {
        emparr[i].display_data();  }
//For Each loop
// for (Emp  e : emparr)       { emparr[i].display_data();  }
}
}
```

Access Class Member Variables

```
class Product{
int prod_no;
String prod_name ;    }

public class Test {
        public static void main ( String args[] ) {
            Product p1 = new Product();
            p1.prod_no = 1;
            p1.prod_name="Mouse";
     System.out.println("P1 object Product No. ::" + p1.prod_no);
     System.out.println("P1 object Product Name. ::" + p1.prod_name);

            Product p2 = new Product();
            p2.prod_no = 2;
            p2.prod_name="Printer";
            System.out.println("P2 object Product No. ::" + p2.prod_no);
      System.out.println("P2 object Product Name ::" + p2.prod_name);
      }
```

Exercise

1. Write a java program to create a class named Rectangle having member variable **length and breadth.** Define various constructors with and without arguments. Also create two methods **area** and **perimeter**.
   (**Formulas :** Area = length * breadth &
   Perimeter = 2 * (length + breadth)

2. Write a java program to create a class named **Square** having member variable **length.** Define constructor with and without arguments. Also create two methods **area** and **perimeter**.
   (**Formulas :** Area = length * length
   Perimeter = 4 * (length)

Exercise

3. Write a java program to create a class named circle having member variable **radius.** Also create two methods **area** and **circumference**. Initialize the radius value by constructor or **setRadius() method**.
   (**Formulas :** Area = PI * radius * radus &
   Circumference = 2 * PI * radius)
4. Define a **Sphere class** with **two constructors** and **one method**. The first form of constructor accepts no arguments. The second form of constructor accepts only radius of the sphere. The method is to find the area of sphere. (**Formula:** Area of sphere = 4* PI * radius * radius)
5. Declare a **three class rectangle, square and triangle**. Each class having **two methods perimeter and area**. Create an object of each class & find its area and perimeter. Use constructor overloading concepts in each classes.

Exercise

6. Write a program to accept two integer numbers and one operator from the user and according to input data apply the operation on numbers and display the result. (**Hint:** Make a **class MathOp** having methods for addition, subtraction, division and multiplication.)
7. Write a program to create a class Employee having two member variables emp_id and emp_name. Also methods to store input data & retrieve output data. Accepts data of 5 employees from the user & display in proper format.
8. Write a program to create a class for MathFun having methods even_odd_check, prime_no_check, palindrome_check, armstrong_check. Create an object of class, accept one integer number from the user and list menu for the above methods and according to user choice display the proper output.

Exercise

9. Write a program to Arr_operation having two methods sort_data and search_element. Accept N numbers from the user and choice of operation.
10. Write a program to create a class MatrixOperation having three methods Mat_Add, Mat_Sub & Mat_Mul. Accepts two matrice from the user and choice of operation.
11. Create a class of student to store the roll_no, stud_name and marks of 3 subjects. Accepts the data of 5 students from the user and display the marksheet in proper format.
12. Create a class of product to store product id, product name, product price and product quantity. Accepts the data of N products from the user. Also fetch product details by passing product id or product price.

Exercise

13. Calculate the net salary of an employee by considering the parameters called HRA(House rent allowance), DA(Dearness allowance), GS (Gross salary) and income tax. Let us assume some parameters.

    HRA = 10% of basic salary
    DA = 73% of basic salary
    GS = basic salary + DA + HRA
    Income tax = 30% of gross salary
    net salary = GS - income tax

Take the input from the user **N** employees name, id and basic salary and display output in proper format

Access Modifiers in Java

- There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.

- **There are four types of Java access modifiers:**
    - **Private**
    - **Default**
    - **Protected**
    - **Public**

    - There are many **non-access modifiers,** such as static, abstract, synchronized, native, volatile, transient, etc.

Access Modifiers in Java

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

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

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

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

Access Modifiers in Java

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Access Modifiers in Java

1) Private

The private access modifier is accessible only within the class.

E.g. Two classes A and Test.

A class contains private data member and private method.

Accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
public class Test {
 public static void main(String args[]) {
            A obj=new A();
            System.out.println(obj.data); //Compile Time Error
            obj.msg(); //Compile Time Error
   }
}
```

Access Modifiers in Java

2) Default

- If you don't use any modifier, it is treated as **default** by default.

- The default modifier is accessible only within package.

- It cannot be accessed from outside the package.

- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Access Modifiers in Java

Accessing the class A with access modifier as default.

```
class A{
int data=40;
void msg(){System.out.println("Hello java");}
}
public class Test {
 public static void main(String args[]) {
            A obj=new A();
            System.out.println(obj.data);
            obj.msg();
   }
}
```

Access Modifiers in Java

3) Protected

•The **protected access modifier** is accessible within package and outside the package but through inheritance only.

•The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

•It provides more accessibility than the default access modifer.

E.g.

•Create two packages pack1 and pack2.

•Create A.java file in package pack1 .

•Create B.java file in package pack2 .

Access Modifiers in Java

```
//save by A.java
package pack1;
public class A{
        protected void msg(){System.out.println("Hello");}
        }
//save by B.java
package pack2;
import pack1.*;
        class B extends A{
                }
//Test.java
import pack2.*;
public class Test {
 public static void main(String args[]) {
            B obj = new B();
            obj.msg();    //Compile time error
  }
```

Access Modifiers in Java

4) Public

- The **public access modifier** is accessible everywhere.
- It has the widest scope among all other modifiers.

E.g.

```
//save by A.java
package pack1;
public class A{  public void msg(){System.out.println("Hello");}       }
//save by B.java
package pack2;
import pack1.*;
          class B extends A{      }
//Test.java
import pack2.*;
public class Test {
 public static void main(String args[]) {
              B obj = new B();
              obj.msg();      }    }
```

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The ***static*** modifier for creating class methods and variables.

- The ***final*** modifier for finalizing the implementations of classes, methods, and variables.

- The ***abstract*** modifier for creating abstract classes and methods.

- The ***synchronized*** modifiers, which are used for threads.

Static Methods & Variables

Static methods in java belong to the class (not an instance of it)

To call a static methods or variable

classname.methodname(args);

classname.variablename;

public class Math

double abs(double a)	*absolute value of a*
double max(double a, double b)	*maximum of a and b*
double min(double a, double b)	*minimum of a and b*
double sin(double theta)	*sine of theta*
double cos(double theta)	*cosine of theta*
double tan(double theta)	*tangent of theta*
double toRadians(double degrees)	*convert angle from degrees to radians*
double toDegrees(double radians)	*convert angle from radians to degrees*
double exp(double a)	*exponential (e^a)*
double log(double a)	*natural log ($\log_e a$, or ln a)*
double pow(double a, double b)	*raise a to the bth power (a^b)*
long round(double a)	*round a to the nearest integer*
double random()	*random number in [0, 1)*
double sqrt(double a)	*square root of a*
double E	*value of e (constant)*
double PI	*value of π (constant)*

Static Variables & Methods of Integer Class

MIN_VALUE
MAX_VALUE

static Integer valueOf (String s)
static String toBinaryString (int i)
static String toOctalString (int i)
static String toHexString (int i)

static int parseInt (String s)

Example of Static Methods & Variable

E.g. :: To Access static variables and methods.

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

System.out.println("Value of pi =" + Math.PI );
System.out.println("Square of 5 is ="+ Math.pow(5,2));
System.out.println("Max. value of Integer ="+ Integer.MAX_VALUE);
System.out.println("Min. value of Integer ="+ Integer.MIN_VALUE);
System.out.println("Max. value of Double ="+ Double.MIN_VALUE);

String s = "100";
System.out.println("Value s in Integer="+ Integer.parseInt(s));
int i = 10;
System.out.println("Value s in Integer="+ Integer.toBinaryString(i));
}     }
```

Example of Static Methods & Variable

E.g. :: To find area & circumference of a circle.

```
class  AreaCircumference {
public static void main (String args[]) {
double radius=15;
//       double area = Math.PI * radius * radius;
double area = Math.PI * Math.pow (radius,2);

double cf = 2 * Math.PI * radius;

System.out.println("Radius is : \t" + radius);
System.out.println("Area is : \t" + area);
System.out.println("Circumfernce is : \t" + cf);
}     }
```

Instance Methods & Variables

Method or Variable defined in a class which is only accessible through the Object of the class

E.g. To access methods of Byte, Short, Integer, Long, Float, Double, Char, String Class, the object of corresponding class is needed.

`int`	`length()`	*number of characters*
`char`	`charAt(int i)`	*the character at index* `i`
`String`	`substring(int i, int j)`	*characters at indices* `i` *through* `(j-1)`
`boolean`	`contains(String substring)`	*does this string contain* `substring`*?*
`boolean`	`startsWith(String pre)`	*does this string start with* `pre`*?*
`boolean`	`endsWith(String post)`	*does this string end with* `post`*?*
`int`	`indexOf(String pattern)`	*index of first occurrence of* `pattern`
`int`	`indexOf(String pattern, int i)`	*index of first occurrence of* `pattern` *after* `i`
`String`	`concat(String t)`	*this string with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`toLowerCase()`	*this string, with lowercase letters*
`String`	`toUpperCase()`	*this string, with uppercase letters*
`String`	`replaceAll(String a, String b)`	*this string, with* `a`*s replaced by* `b`*s*
`String[]`	`split(String delimiter)`	*strings between occurrences of* `delimiter`
`boolean`	`equals(Object t)`	*is this string's value the same as* `t`*'s?*

Instance Methods of Integer Class

byteValue() : returns the value of this Integer as a byte

doubleValue() : returns the value of this Integer as an double

floatValue() : returns the value of this Integer as a float

intValue() : returns the value of this Integer as an int

longValue() : returns the value of this Integer as a long

shortValue() : returns the value of this Integer as a short

toString() : returns a String object representing the value of this Integer

Example of Instance Methods

```
class  Rectangle
{
private double length = 10.50;
private double width = 20.70;

void setData( double l, double w)   {
            length = l;
            width = w;
}

void  displayData ()    {
            System.out.println("Length :  "+ length);
            System.out.println("Width :  "+ width);
}

public static void main (String args[])            {
            Rectangle r = new Rectangle();
            r. displayData();
            r.setData(11.25,33.77);
            r. displayData();
}
}
```

Wrapper Classes

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

- Wrapper classes for each of the eight simple types.
  - Byte, Short, Integer, Long, Float, Double, Character, Boolean

- Common **static variables** in all classes are MIN_VALUE, MAX_VALUE

- **Static methods** of wrapper classes are used to represent the data in various format as below.

E.g. toBinaryString (datatype var)

toBinaryString (int i)
toOctalString (double i)
toHexString (long i)

Wrapper Classes

E.g. To get the value of an Integer Object the intValue() method is used.

- Same way for other objects we may use corresponding methods as below to get its values.

```
byte    byteValue ( )
short   shortValue ( )
int     intValue ( )
long    longValue ( )
float   floatValue ( )
double doubleValue ( )
String  toString ( )
```

Wrapper Classes

Example

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

        Integer iOb = new Integer(100);

          //Conversion from wrapper class type to primary data type
          int i = iOb.intValue();
          System.out.println(" Value of i ::" + i);

          //Conversion from primary data type to wrapper class type
          iOb = i;
          }
}
```

Output: Value of i :: 100

Autoboxing & Auto-unboxing

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.


```
Integer iOb = 100; // autobox an int

int i = iOb; // auto-unbox
```

Example of autoboxing/unboxing.

```
class AutoBox {

public static void main(String args[]) {

Integer iOb = 100; // autobox an int

int      i  =  iOb; // auto-unbox

System.out.println(i + " " + iOb);}
}

// Output:  100 100
```

StringBuffer Class

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
- Allow to change the character sequence after string object created.
- Also provides methods to append or insert characters.

Constructor

StringBuffer( ) : Default capacity is 16 chars.

StringBuffer (int size) : Capacity to size of chars.

StringBuffer (String s): Capacity is size of s plus 16 chars.

StringBuffer Class

Instance Methods

StringBuffer append (char ch)
StringBuffer append (String str)
int capacity( )
int length ( )
StringBuffer reverse ( )
char charAt (int i)
StringBuffer insert (int i, String str)

StringBuilder Class

StringBuilder:

•The StringBuilder in Java represents a mutable sequence of characters.

•Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

•Syntax:

```
StringBuilder str = new StringBuilder("Good");
str.append("Morning");
```

Difference between String, Stringbuffer and Stringbuilder

- **Immutable** objects are instances whose state doesn't change after it has been initialized.
- The **String class is an immutable** class whereas **StringBuffer and StringBuilder classes are mutable**.

E.g.

// concatenate two strings with **String**

```
String fullName = firstName.concat(" ").concat(lastName);
```

The concatenation does not modify firstname or lastname string, rather create a new object which will be referenced by fullName.

//Concate two string with **Stringbuffer**

```
sb.append("How r u?");
System.out.println(sb);
```

Difference Between Stringbuffer and Stringbuilder

StringBuffer	StringBuilder
StringBuffer operations are thread-safe and synchronized	StringBuilder operations are not thread-safe & are not-synchronized
StringBuffer is to used when multiple threads are working on the same String	StringBuilder is used in a single-threaded environment.
StringBuffer performance is slower when compared to StringBuilder	StringBuilder performance is faster when compared to StringBuffer
Syntax: StringBuffer var = new StringBuffer(str);	Syntax: StringBuilder var = new StringBuilder(str);

‘this’ Keyword

- It refers to object that is currently executing.
- Access instance variable with

  this.varName

- Allows one constructor to explicitly invoke another constructor in the same class.

  this (args)

  -

Example of 'this' keyword

```
class   Point2D {
double   x ;
double   y ;
// parameter and class variable name is same.
point2D (  double x ,  double  y )  {
        this.x  =  x ;
        this.y  =  y ;
}
}
```

Example of ‘this’ keyword

```
class   Point2D {
double   x ;
double   y ;

point2D (  )  {
        this ( 0 , 0 );
}

point2D (  double x ,  double  y )  {
        this.x  =  x ;
        this.y  =  y ;
}
}
```

Example of ‘this’ keyword

```
class  Rectangle
{
private double length = 10;
private double width = 20;

void setData( double l, double w)   {
            this.length = l;
            this.width = w;
}
void  displayData ()    {
            System.out.println("Length :  "+ this.length);
            System.out.println("Width :  "+ this.width);
            this.area();
}
void  area()    {
            System.out.println("Area of rectangle  :  "+ this.length * this.width);
}
public static void main (String args[])     {
            Rectangle r = new Rectangle();
            r. displayData();
            r.setData(11,2.5);
            r. displayData();
}
}
```

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class
- It provides the reusability.

Types of Inheritance

- Simple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance

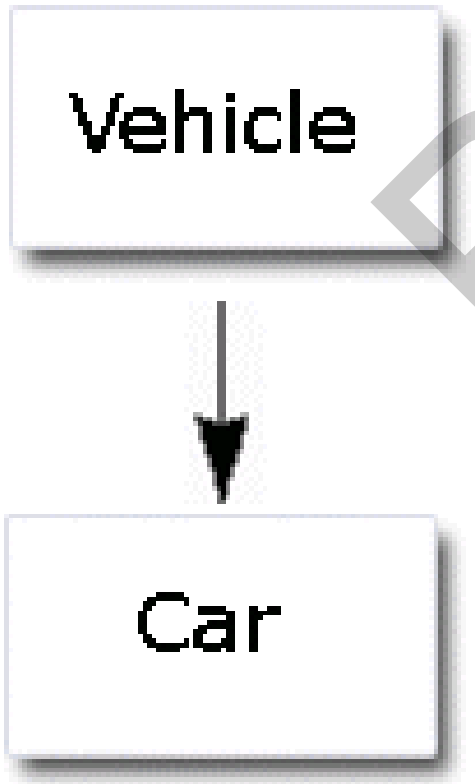
Defining a Subclass

- Syntax:
  - Class subclass-name **extends** superclass-name

```
{
    Body of the class
}
```

Simple Inheritance

- When a subclass is derived simply from it's parent class then this mechanism is known as simple inheritance.
- In case of simple inheritance there is only a subclass and it's parent class.
- It is also called single inheritance or one level inheritance.



Base / Super / Parent Class

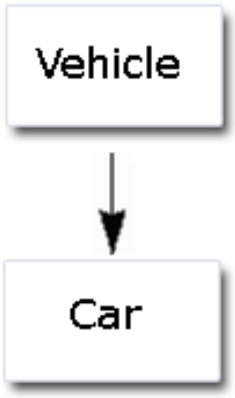
Derived / Sub / Child Class

Example of Inheritance

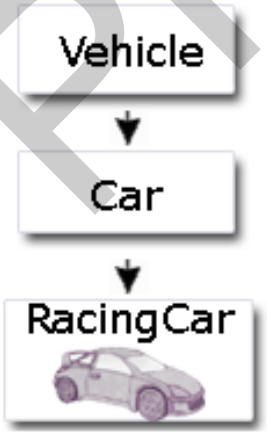
```
class A1 {
        int i=10  , j;
        void show() { System.out.println("Value of i and j is :" + i  + "," + j); }
}
class B1 extends A1 {
        int k;
        void display()     { System.out.println("Value of k is :" + k);       }
        void sum() { System.out.println("i+j+k" +" " + "is" + " :" + (i+j+k));  }
}
public class Test {
public static void main(String S[])     {
          A1 a = new A1();          a.j = 20;
                    a.show();
           B1 b = new B1();         b.j = 30;       b.k = 40;
                    b.show();       b.display();     b.sum();
          }
}
```

Multilevel Inheritance

- When a subclass is derived form a derived class than this mechanism is known as Multilevel Inheritance.
- Multilevel can go up to the any number of levels.



Simple Inheritance



Multilevel Inheritance

Supar Class

Derived Class

Derived of Derived Class

Example of Multilevel Inheritance

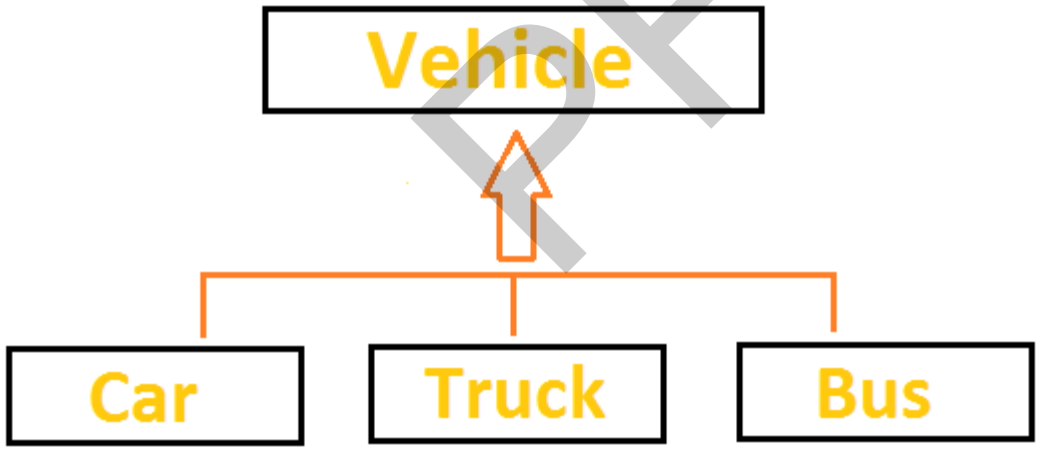
```
class A  {
int i = 10 , j ; // var declaration
void show() {  System.out.println("Inside Class A....");
        System.out.println("Value of i and j is :" + i + " , " +j );        }
}
class B extends A {
int k = 20 ;
void display() { System.out.println("Inside Class B....");
        System.out.println("Value of i , j and k is :" + i + " , " +j + " , " +k );   }
void sum() {
         System.out.println("i+j+k" +" " + "is" + " :" + (i+j+k));        }
}
class C extends B{
int l;
void view() { System.out.println("Inside Class C....");
System.out.println("Value of i , j , k and l is :" + i + " , " +j + " , " +k + " , " +l );  }
void add() {
        System.out.println("i+j+k+l" +" " + "is" + " :" + (i+j+k+l));  }
}
```

Example of Multilevel Inheritance

```
public class Test {
public static void main(String S[])          {
        A a = new A();
        B b = new B();
        C c = new C();

        a.j = 4;
        a.show();

        b.j = 3;
         b.display();
        b.sum();          // sum is 33

         c.j = 2;      c.l = 4;
         c.view();
         c.add();          // sum is 36      }
}
```

Hierarchical Inheritance

- When a more than one subclasses are derived from it's parent class then this mechanism is known as **hierarchical inheritance**.
- In case of hierarchical inheritance there is more than one subclass and one parent class.



Example of Hirarchical Inheritance

```
class A  {
int i , j ;
void show() { System.out.println("A :: Value of i and j is :" + i + " , " +j );  }
}
class B extends A {
int k;
void display()  {
System.out.println("B :: Value of i , j and k is :" + i + " , " +j + " , " +k ); }
void sum() { System.out.println("i+j+k" +" " + "is" + " :" + (i+j+k));  }
}
class C extends A {
int l;
void view() {
System.out.println(" C :: Value of i , j & l is :" + i + ", " + j + ", " + l ); }
void add() { System.out.println("i+j+l" +" " + "is" + " :" + (i+j+l)); }
}
```

Example of Hirarchical Inheritance (Cont.)

```
public class Hierarchical   {
public static void main(String S[])
{
        A a=new A();
        B b=new B();
        C c=new C();

        a.i=10;          a.j=20;
        a.show();

        b.i=2;           b.j=4;          b.k=6;
        b.display();
        b.sum();

        c.i=1;           c.j=3;          c.l=5;
        c.view();
        c.add();
}
}
```

Multiple Inheritance

- The mechanism of inheriting the features of more than one base class into a single class is knows as multiple inheritance.

- **Java does not support multiple Inheritance.**

- In java multiple inheritance is achieved through the use of interfaces.

- **A class can implement more than one interfaces.**

‘super’ Keyword

- super keyword is used to access the members of the super class (base class).
- Super must always be the first statement executed inside a subclass constructor.
- It used for following two purpose:
  - To call super class constructor
  - To access a member of the super class that has been hidden by a member of a subclass

- A subclass can call a constructor method defined by its superclass by use of the following form of **super:**

  super(*parameter-list*);

Example of 'super' Keyword

```
class A {
double l = 10 ;
double b ;     }
class B extends A {
double area ()  {
          double ans = super.l * b;
        // double ans = l * b;
        return ans; }
}
class Test  {
    public static void main(String args[]) {
        B b1 = new B();    b1.b = 20;
        double ans = b1.area();
        System.out.println("Area :: " + ans);                }
}
```

Polymorphism

- Allows to define one interface and have multiple implementation.

- Types of polymorphism
    1) Method Overloading
    2) Method Overriding

Method Overloading

- Two or more methods of same name in a class, provided that there argument list or parameters are different. This concept is known as Method Overloading.

- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

- The parameters may differ in their type or number, or in both.

- They may have the same or different return types.

- It is also known as **compile time polymorphism.**

```
class Overload {
void add (int num1, int num2) {
      int ans = num1 + num2;
      System.out.println ("Addition (int)::" + ans);}
void add (float num1, float num2, float num3) {
      float ans = num1 + num2 + num3;
      System.out.println("Addition(float)::" + ans);}
double add (double num1, double num2) {
      return (num1 + num2); }
}
class MethodOverloading {
public static void main (String args []) {
     Overload Obj = new Overload();
     Obj.add(10,20);
     Obj.add(2.5f,3.5f,4.5f);
     double result = Obj.add(3e10,4e10);
     System.out.println("O/P : " + result); } }
```

Method Overriding

- Child class has the same method as of base class. In such cases child class overrides the parent class method. This feature is known as method overriding.

- Applies only to inherited methods
- Object type (NOT reference variable type) determines which overridden method will be used at runtime

```
void add (int num1, int num2) {
      int ans = num1 + num2;
      System.out.println ("Addition (int)::" + ans);}
}

public class DerivedClass extends BaseClass {
void add (double num1, double num2) {
      double ans = num1 + num2;
      System.out.println("Addition(double)::"+ ans);}
}
public class Test {
public static void main (String args []) {
      BaseClass obj1 = new BaseClass();
       DerivedClass obj2 = new DerivedClass();
      obj1.add(3,6);
      obj2.add(12e5,10e5);
      obj2.add(2,8);   }
}
```

Abstract Class

- Abstract class is used to declare functtionality that is implemented by one or more subclasses.

- **It specify what functionality is provided but not how that functionality is provided.**

- **Abstract class can not be instantiated** but you can declare as an object type.

- **The methods in abstract have often empty bodies.** You may define the concrete methods in abstract class.

Example of Abstract Class

```
abstract class T {
   // Method with empty body. Implementation in derived class.
   abstract void emptymethod();

   // Concrete methods are still allowed in abstract classes
   void concretemethod() {
      System.out.println("This is a concrete method.");    }
}
class S extends T {
void emptymethod() {
                  System.out.println(“ Implementation of emptymethod.");
          }
}
class Test {
          public static void main(String args[]) {
                  T b = new S();
                  b. emptymethod();
                  b. concretemethod();
   }      }
```

Example of Abstract Class

```
import java.io.*;
abstract class Shape {
   // abstract methods which will be implemented by its subclass(es)
   abstract public double area();
   abstract public void perimeter();
}
class Rectangle extends Shape {
   int length, width;
   // constructor
   Rectangle(int length, int width)   {
      this.length = length;
      this.width = width;
   }
       // Override method in abstract class
   public void perimeter()     {
      System.out.println("Rectangle perimeter ::" + 2 * (length+width));
   }
       // Override method in abstract class
   public double area()     {
      return (double)(length*width);    }       }
```

Example of Abstract Class

```
class Circle extends Shape {
   int radius;
//constructor
   Circle(int radius)   {
      this.radius = radius;
   }
       // Override method in abstract class
   public void perimeter()  {
            System.out.println("Circle perimeter ::" + 2 * Math.PI * radius);
   }
       // Override method in abstract class
   public double area()    {
      return (double)((Math.PI*radius*radius)/2);
   }
}
```

Example of Abstract Class

```
class Test {
public static void main(String[] args) {
    // creating the Object of Rectangle class and using shape class reference.
    Shape rect = new Rectangle(2,3);
    System.out.println("Area of rectangle: " + rect.area());
    rect.perimeter();

    System.out.println(" ");

    // creating the Objects of circle class
    Shape circle = new Circle(2);
    System.out.println("Area of circle: " + circle.area());
    circle.perimeter();
}
}
```

Final Class

- Final class cannot be extended.
- Methods implemented by this class can not be overridden.
- **The compiler gives an error if you attempt to extend a final class.**
- **E.g. Math Class and Its Methods.**
- **Math.pow , Math.abs, Math.max, Math.round etc.**

Interface

- **It provides the multiple inheritance. (Derived class may inherit multiple Interfaces.)**
- It is group of constants and method declaration.
- An interface can be used as type of variable. The variable can then reference any object that implements that interface.
- Interface may extend another interface.

Interface Syntax

intfModifier **interface** intName {
varModifier1 type1 varName1 = value1;
……
varModifierN typeN varNameN = valueN;

mathModifier1 type1 mthName1(params 1);
……
mathModifierN typeN mthNameN(params N);
}

- Implicitly var. modifier is public, static and final.
- If method Modifier is not specify than interface is limited to the code in same package.
- If method Modifier is public than other package may use interface.

Interface Example

```
// Multiple inheritance by Interface in Java
interface Shape2D {
   void area();
   void perimeter();
}
interface Shape3D{
   void volume();
}
class Rectangle implements Shape2D {
   private int l=10, b=20;
   public void area() {
      System.out.println("Area ::"+l * b);
   }
   public void perimeter() {
      System.out.println("Perimeter::"+2 * (l+b));   }
}
```

Interface Example

```
class Sphere implements Shape2D, Shape3D{
    int r=2;
  public void area() {
        System.out.println("Area ::"+ 4 * Math.PI * Math.pow(r,2));   }
  public void perimeter() {
    System.out.println("Perimeter::"+2 * Math.PI * r);   }
  public void volume() {
    System.out.println("Volume::"+4/3 * Math.PI * Math.pow(r,3));   }
}
public class Demo {
  public static void main(String args[]) {
        System.out.println("Rectangle");
        Rectangle r = new Rectangle();     r.area();     r.perimeter();
        System.out.println("Sphere");
        Sphere s = new Sphere();     s.area();     s.volume();          }
}
```

Difference Between Abstract Class and Interface

- The **abstract keyword** is used to declare abstract class.
  The **interface keyword** is used to declare interface.

- **Abstract CLASS** contains
  - **General methods**
  - **Abstract methods**

  Interface contains
  - **Only abstract methods**.

- An **abstract class** can be extended using keyword "extends".
  An **interface** can be implemented using keyword "implements".

Difference Between Abstract Class and Interface

- Abstract class **doesn't support multiple inheritance**.
  Interface **supports multiple inheritance**.

- **In abstract** by defaults methods are not abstract. It may be public, private, protected.
  In Interface all methods are public & abstract & variable are public static final.

- Abstract class **can provide the implementation of interface**.
  Interface **can't provide the implementation of abstract class**.

Difference Between Abstract Class and Interface

- Class can extend only one **abstract class**.

  A class can implement more than one **interface**.

- An **abstract class** can extend another Java class and implement multiple Java interfaces.

  An **interface** can extend another Java interface only.

- **Example of abstract class:**

```
public abstract class Shape    {
public abstract void perimeter();    }
```

Example of interface

```
public interface Shape   {
void perimeter();              }
```

Generics

- Introduce in JDK 5.
- It is possible to create classes, interfaces and methods that will work in a type-safe manner with various kinds of data.

E.g. The mechanism that supports a stack is the same whether that stack is storing items of type Integer, Float, Double, String, Object or Thread.

- With generic you can define an algorithm once, independently of any specific type of data, and than apply that algorithm to a wide variety of data type without any additional effort.
- With generics, all casts are automatic and implicit, which expands the ability to reuse code and easily.

Java Generic Type

- Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:
- E – Element
- K – Key
- N – Number
- T – Type
- V – Value
- S,U,V etc. – 2nd, 3rd, 4th types

Generic Method Example

```
// A Simple Java program to make a program of generic
   classes
// We use < > to specify Parameter type
class Gclass <T> {
// An object of type T is declared
T obj;
Gclass ( T obj ) {
                    this.obj = obj; }

   public T getObject() {
         return this.obj; }
}
```

Generic Method Example

```
public class Test  {
public static void main (String[] args) {
// instance of Integer type
Gclass <Integer> iObj = new Gclass<Integer>(15);
System.out.println( iObj.getObject() );

// instance of String type
Gclass <String> sObj = new Gclass<String>("GeeksForGeeks");
System.out.println( sObj.getObject() );}
}
```

Generic Method Example

E.g. Print an array of different type using a single Generic method

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }
```

Generic Method Example

```
public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array    } }
```

Enumeration

- It is a list of named constants.
- An enumeration define a class type.
- An enumeration can have constructors, methods and instance variable.
- Methods
  - public static enum-type [ ] values ()
  - public static enum-type valueOf (String str)

Example of Enumeration

```
class EnumExample1{
//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//creating the main method
public static void main(String[] args) {
//printing all enum
for (Season s : Season.values()){
      System.out.println(s);  }
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is:
   "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is:
   "+Season.valueOf("SUMMER").ordinal());    }   }
```

Inner class

- Inner class or nested class is a class which is declared within a class.
- It is used to logically group classes and interfaces in one place so that it can be more readable and maintainable.

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

ClassLoader in Java

- The Java ClassLoader is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine.
- Java classes aren't loaded into memory all at once, but when required by an application.
- At this point, the Java ClassLoader is called by the JRE and these ClassLoaders load classes into memory dynamically.
- Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided.
- To know the ClassLoader that loads a class the getClassLoader() method is used.
- All classes are loaded based on their names and if any of these classes are not found then it returns a NoClassDefFoundError or ClassNotFoundException.

Class Path in Java

- We don't need to set **PATH** and **CLASSPATH** to compile and run java program while using IDE like **Netbeans**.
- These environment variables are required to **compile** and **run** java program using **CMD** (Command prompt)
- System will understand java command with the help of **PATH** variable and find class using **CLASSPATH** variable to

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

Set Path Variable

To compile & run java program in any directory the path of java bin directory set into the path environment variable as below.

Windows

Select **Start**, select **Control Panel**. double click **System**

Select the **Advanced** tab.

Click **Environment Variables**.

In the section **System Variables**, find the PATH environment variable & Select it. Click **Edit**.

At the end of line add **path of Bin Directory.**