Module 2

Introducing Classes

Class

A mechanism that allows to combine data and operations on those data into a single unit is called a class. A class is a plan, a blueprint, a template, a logical construct. A class defines the state and behavior of its instances called objects. The data fields defined in the class are called instance variables or member variables because they are created when the objects are instantiated.

General form of a class:

```
class classname {
        type instance-variable1;
        type instance-variable2;

        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
        // body of method
        }
        type methodname2(parameter-list) {
        // body of method
        }
        // ...
        type methodnameN(parameter-list) {
        // body of method
}
}
```

Variables defined within a class are called instance variables because each instance of theclass (that is, each object of the class) contains its own copy of these variables. Thus, the datafor one object is separate and unique from the data for another.

Creating Objects

An object is essentially a block of memory that combines space to store all the instance variables. Creating an object is also referred as instantiating an object. Objects are created using the new operator. The new operator creates an object of the specified class and returns a reference of that object.

```
Rectangle rect1;   // declares an object
rect1=new Rectangle(); // instantiates the object
```

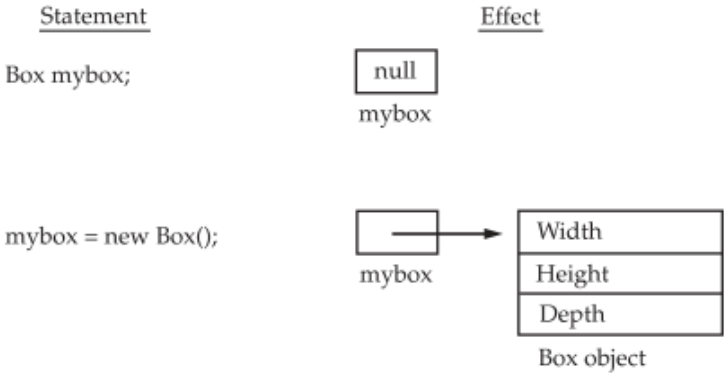
The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to it.

Consider the following example

// **This program declares two Box objects.**

```
class Box {
        double width;
```

```
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // assign different values to mybox2'sinstance variables
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```



The first line declares **mybox** as a reference to an object of type **Box**. After this line executes,**mybox** contains the value **null**, which indicates that it does not yet point to an actual object.Any attempt to use **mybox** at this point will result in a compile-time error. The next lineallocates an actual object and assigns a reference to it to **mybox**.The class name followed by parentheses(for eg., Box()) specifies the *constructor*for the class. Aconstructor defines what occurs when an object of a class is created.

It is important to understand that **new** allocates memory for an object during run time.The advantage of this approach is that your program can create as many or as few objects asit needs during the execution of your program. However, since memory is finite, it is possiblethat **new** will not be able to allocate memory for an object because insufficient memory exists.

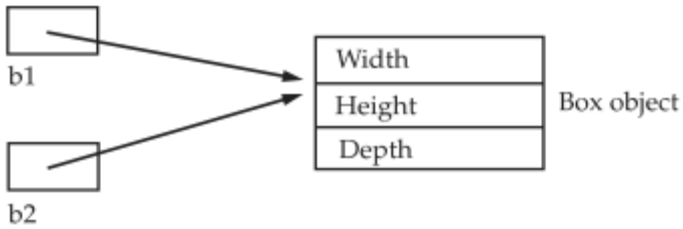
That is, a class creates a logical frameworkthat defines the relationship between its members. When you declare an object of a class, youare creating an instance of that class. Thus, a class is a logical construct. An object has physicalreality.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takesplace. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by**b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However,this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the*same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of theoriginal object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changesmade to the object through **b2** will affect the object to which **b1** is referring, since they are thesame object.



Adding a Method to the Box class

```
// This program includes a method to compute the volume  inside the box class.
        class Box {
        double width;
        double height;
        double depth;
        // display volume of a box
                void volume() {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
}
        }
        class BoxDemo3 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
        // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
        /* assign different values to mybox2'sinstance variables */
                mybox2.width = 3;
                mybox2.height = 6;
```

```
            mybox2.depth = 9;
        // display volume of first box
            mybox 1.volume();
        // display volume of second box
            mybox2.volume();
        }
    }
```

Returning a value

```
// Now, volume() returns the volume of a box.
        class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
            double volume() {
            return width * height * depth;
        }
        }
        class BoxDemo4 {
            public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            // assign values to mybox1's instance variables
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            /* assign different values to mybox2's
            instance variables */
            mybox2.width = 3;
            mybox2.height = 6;
            mybox2.depth = 9;
            // get volume of first box
            vol = mybox1.volume();
            System.out.println("Volume is " + vol);
            // get volume of second box
            vol = mybox2.volume();
            System.out.println("Volume is " + vol);
        }
        }
```

Adding methods that takes parameters

```
// This program uses a parameterized method.
        class Box {
        double width;
        double height;
```

```
double depth;
// compute and return volume
        double volume() {
        return width * height * depth;
}
// sets dimensions of box
        void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
}
}

class BoxDemo5 {
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
}
}
```

Example: Program to print the default values of instance variables.

```
class example{
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  char c;
  boolean boo;
void putData()
{
  System.out.println(“byte default value”+b);
System.out.println(“short default value”+s);
System.out.println(“int default value”+i);
System.out.println(“long default value”+l);
System.out.println(“float default value”+f);
System.out.println(“double default value”+d);
```

```
    System.out.println("character default value"+c);
    System.out.println("boolean  default value"+boo);
    }
    public static void main(Stroing args[])
    {
      example ob=new example();
      ob.putData();
    }
  }
```

The dot (.) operator

- It enables you to access instance variables of any objects within a class
- It enables you to store values in instance variables of an object
- It is used to call object methods

The new operator

The **'new'** operator in java is responsible for the creation of **new object** or we can say instance of a class. The dynamically allocation is just means that the memory is allocated at the run time of the program. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.

For example :

Before creating a new object of the class we create the 'reference' variable of the class as

Demo obj; // Where Demo is the class whose object we want to create.

If we use the new operator than the actual or physical creation of the object is occurred.

The statement is :

obj = new Demo();

This statement does two things :

1. It creates the actual or physical object in the heap with the variable 'obj' pointing to the object from the stack.
2. It calls the constructor of the "Demo" class for the initialization of the object.

Constructors

- A Constructor is a special method used to initialize the object of a class.
- Constructor is automatically invoked when the object of its class is created.
- Constructor name must be same as that of a class.
- Constructor does not have a return type not even void.
- The constructor can have default arguments.
- Constructor can not be inherited.
- Constructor overloading is possible.
- Constructor overriding is not possible.

Types of Constructors

1. Default constructor

2. Parameterized constructor
3. Copy constructor

Default Constructor : A constructor which does not contain any parameter is called default constructor.

```
class test
{  int a, b;
        test()
        {  a=10;
           B=20;
        }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class consructordemo
{
        public static void main(String args[])
        {
        Test ob=new test( );
        ob.putdata( );
        }
}
```

Parameterized Constructor: The constructor which contains parameters is called parameterized constructor.

```
class test
{  int a, b;
        test(int m, int n)
        {  a=m;
           b=n;
        }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class consructordemo
{
        public static void main(String args[])
        {
        test ob=new test(10,20 );
        ob.putdata( );
        }
}
```

Copy Constructor

// Program to demonstrate copy constructor and constructor overloading

```
class copy
{  int a, b;
        copy()
        {  a=0;
           b=0;
        }
        copy(int m, int n)
        { a=m;
          b=n;
        }
        copy(copy ob)
        {
          a=ob.a;
        b=ob.b;
          }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class Demo
{
        public static void main(String args[])
        {
        copy c1=new copy( );
        copy c2 = new copy(10,20);
        copy c3=new copy(c2);
        c2.putdata();
        c3.putdata();
        }
}
```

Differences between Constructors and methods

Constructors	**Methods**
Constructor should be of the same name as that of class	Method name should not be of the same name as that of class.
Constructor is used to initialize an object	Method is used to exhibits functionality of an object.
Constructors are invoked implicitly	methods are invoked explicitly
Constructor does not return any value	method must return a value
In case constructor is not present, a default constructor is provided by java compiler	In the case of a method, no default method is provided.
Constructor overriding is not possible	Method overriding is possible

The "this" keyword

There are two uses of this keyword. They are

1. **this can be used inside any method (except static) to refer to the *current* object**. That is, this is always a reference to the object on which the method was invoked.

```
class test1
{
  void create()
  {

        System.out.println(this);// prints memory refernce of ob1 object eg., test@a26fc
   }
 public static void main(String arg[])
 {
        test1 ob1=new test1();
        ob1.create();
}
}
```

2. **Instance variable hiding**.If there is a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of this keyword.

```
class test1
{
 int a;
 int b;
  test1(int a,int b)
  {
        this.a=a;
        this.b=b;
    }

        void show()
        {
          System.out.println("a="+a+"b="+b);
   }
        public static void main(String arg[])
         {
        test1 ob1=new test1(10,20);
        ob1.show();
        }
}
```

Garbage Collection

C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* Garbage Collection is process of reclaiming the runtime unused memory automatically.It works like this: when no references to an object exist, that object is assumed to be no

longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. Garbage Collector is a Daemon thread that keeps running in the background. Basically, it frees up the heap memory by destroying the unreachable objects.

Advantages of Garbage Collection

1. The manual memory management done by the programmers is time consuming and error prone.Hence automatic memory management is done.
2. Reusability of memory is achieved.

Disadvantages of Garbage Collection

1. The execution of the program is paused or stopped during the process of garbage collection.
2. Sometimes, situations like thrashing may occur due to garbage collection.

The finalize( ) method

Ifan object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, we candefine specific actions that will occur when an object is just about to be reclaimed by thegarbage collector.
To add a finalizer to a class, simply define the **finalize( )** method. The Java run timecalls that method whenever it is about to recycle an object of that class. Inside the **finalize( )**method, specify those actions that must be performed before an object is destroyed. It is important to understand that **finalize( )** is only called just prior to garbage collection.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here

}
```

A Stack class

A *stack* stores data using first-in, last-outordering. That is, a stack is like a stack of plates on a table—the first plate put down on thetable is the last plate to be used. Stacks are controlled through two operations traditionallycalled *push* and *pop.* To put an item on top of the stack, we will use push. To take an item off the stack, we will use pop.
Here is a class called **Stack** that implements a stack for integers:

```
// This class defines an integer stack that can hold 10 values.
```

```
class Stack {
        int stck[] = new int[10];
        int tos;
// Initialize top-of-stack
        Stack() {
                tos = -1;
        }
// Push an item onto the stack
        void push(int item) {
                if(tos==9)
        System.out.println("Stack is full.");
else
        stck[++tos] = item;
        }
// Pop an item from the stack
        int pop() {
                if(tos < 0) {
                System.out.println("Stack underflow.");
                return 0;
                }
else
                return stck[tos--];
        }
        }
class TestStack {
        public static void main(String args[]) {
                Stack mystack1 = new Stack();
// push some numbers onto the stack
                for(int i=0; i<10; i++) mystack1.push(i);
// pop those numbers off the stack
                System.out.println("Stack in mystack1:");
                for(int i=0; i<10; i++)
                System.out.println(mystack1.pop());
}
        }
```

Nested classes

It is possible to define a class within another class such classes are called as known as nested classes. There are two types of nested classes

1) Static class 2) non-static class

1.Static nested class

A nested class can be made as static. The nested static class can be accessed without having an object of the outer class. A static nested class can not access non-static data members and methods. It can access static data members of outer class including private. Static nested classes are rarely used.

class example

```
{
  //static class
        static class X
        {
          static String str=" Inside static class";
        }
        public static void main(String args[])
        {
          X.str="Inside class example";
        System.out.println("String stored in str"+X.str);
        }
}
```

2.Non-static nested class or Inner class

An inner class is a non static nested class. It has access to all the data members and the members of its outer class and it can refer them directly without using an object. Inner classes are a security mechanism in java**.** An inner class can be made private.

```
class outer
{
  int x=10;
        class inner
        {
          int y=20;
          void display() {
        System.out.println(x);
          System.out.println(y);
         }
        }
        void show() {
          System.out.println(x);
          // System.out.println(y); member of inner class is not accessible by outer class
          inner ob=new inner();
          ob.display();
          }
class demo{
public static void main(String args[])
        {
        outer ob1=new outer();
          ob1.show();
        }
}
```

Static keyword

1. It is used to make block of code as static
2. It is used for a static variable
3. It can be used for a method

4. A class can be made static if it is a nested class. A nested class can be accessed without having an object of outer class.

Static Block

Static block is mostly used to change the default values of static variables. This block gets executed when the class is loaded in the memory (before the execution of the main method). A class can have multiple static blocks. They will execute in the same sequence in which they have been written in the program.

Example:
```
class test1
{
        static int num;
        static{
                num=40;
                System.out.println("Static Block");
                }
        public static void main(String args[])
        {
                System.out.println(" Value of num="+num);
                }
}
```

Ouput

Static Block
Value of num=40

Using static keyword for instance variables

- If all the objects of a class want to share a common instance variable, then we have to make the instance variable as static
- Local variables can not be declared as static
- Memory allocation for static variables in done during class loading
- Static variable gets life as soon as class is loaded into JVM and it is accessible through out the class.
- A static variable belongs to the class not to the object(instance).

Example:
```
class staticdata{
        int a;
        static int b;
        void getData(int m)
        {
          a=m;
          b++;
        }
        void putData()
        {
                System.out.println(a);
                System.out.println( b);
        }
```

```
    }
class staticeg {
        public static void main(String args[])
        {
                staticdata ob1=new staticdata();
                staticdata ob2=new staticdata();
                ob1.getData(10); ob2.getData(20);
                ob1.putData(); ob2.putData();
        }
  }
```

Static method

- It is a method which belongs to the class not to the object (instance).
- A static method can access only static data.
- A static method can access only other static method and can't invoke a non static method.
- A static method can be directly invoked through the class name and dose not need any object
- A static method can't refer to "this" or "super" in anyway.
- Please note, the main method is static because it must be accessible for an application to run before any instantiation takes place.

Example:

```
class staticmethod{
        int a;    static int b;
        void getData(int m)
        {
          a=m;  b++;
        }
        Static void printData()
        {
                //System.out.println( a); can not access a since it is non static
                System.out.println( b);
        }

        void putData()
        {
                System.out.println( a);
                System.out.println( b);
        }
    }
class staticeg {
        public static void main(String args[])
        {
                staticmethod.printData();
                staticmethod ob1=new staticmethod();
                staticmethod ob2=new staticmethod();
                ob1.getData(10); ob2.getData(20);
                ob1.putData(); ob2.putData();
        }
```

```
    }
```

Static class

A nested class can be made as static. The nested static class can be accessed without having an object of the outer class.

```
class example
{
  //static class
        static class X
        {
           static String str=" Inside static class";
        }
        public static void main(String args[])
        {
          X.str="Inside class example";
        System.out.println("String stored in str"+X.str);
        }
}
```

Inheritance

It is the mechanism in java by which one object acquires the properties of the other object. This is important because it supports the concept of hierarchical classification. The class whose features are inherited is known as super class (or a base class or a parent 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 super class fields and methods. 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.

Defining a sub class:

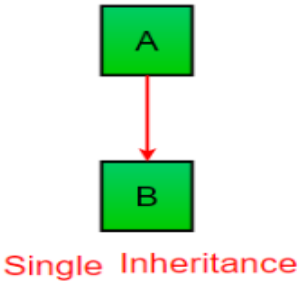
```
class subclassname extends superclassname
{
 Variables    declarartion;
 Methods declarartion;
 }
```

The keyword extends specifies that the properties of the superclassname are extended to the subclassname.

Types of Inheritance

1. Single inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

Single Inheritance: Deriving a new class from a single class is called single inheritance



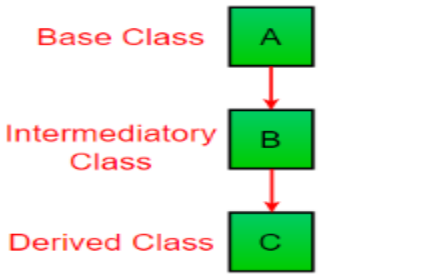
Example:

```
import java.util.*;
        class A
        {  int m.n;
          void getData()
          {
          Scanner s=new Scanner(System.in);
          System.out.println("Enter the values of m and n");
          m=s.nextInt();
          n=s.nextInt();}
        void putData()
        {
```

```
                System.out.println("m="+m+"n="+n);
            }
        }
class B extends A
{
  int sum;
 void add()
 {
 sum=m+n;
  System.out.println("  SUM="+sum);
 }
}
        class SingleInherit
        {
         public static void main(String args[])
         {
          B ob=new B();
          ob.getData();
          ob.putData();
          ob.add();
         }
        }
```

Multilevel Inheritance

Deriving a class from another derived class is called multilevel inheritance.



Multilevel Inheritance

Example:

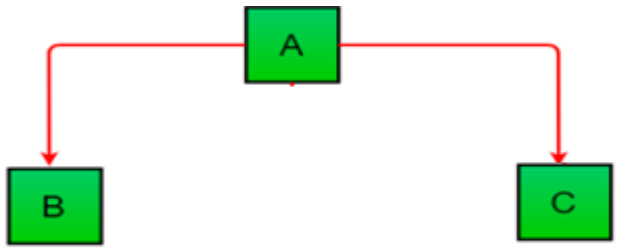
```
import java.util.*;
        class A
        {  int m.n;
         void getData()
         {
         Scanner s=new Scanner(System.in);
         System.out.println("Enter the values of m and n");
         m=s.nextInt();
         n=s.nextInt();}
        void putData()
        {
                System.out.println("m="+m+"n="+n);
        }
```

```
    }
class B extends A
{
  int sum;
 void add()
 {
 sum=m+n;
  System.out.println("  SUM="+sum);
 }
}
class C extends B
{
  int prod;
 void product()
 {
 prod=m*n;
  System.out.println("  Product="+prod);
 }
}

        class Multilevel
        {
          public static void main(String args[])
          {
           C ob=new C();
           ob.getData();
           ob.putData();
           ob.add();
           ob.product();
          }
        }
```

Hierarchical Inheritance

Deriving many classes from a single super class is called hierarchial inheritance.



Hierarchical Inheritance

Example: import java.util.*;

```
class A
{  int m.n;
  void getData()
  {
  Scanner s=new Scanner(System.in);
  System.out.println("Enter the values of m and n");
  m=s.nextInt();
  n=s.nextInt();}
void putData()
{
        System.out.println("m="+m+"n="+n);
}
}
class B extends A
{
  int sum;
 void add()
 {
 sum=m+n;
  System.out.println("  SUM="+sum);
 }
}
class C extends A
{
  int prod;
 void product()
 {
 prod=m*n;
  System.out.println("  Product="+prod);
 }
}

class Hierachial
{
  public static void main(String args[])
  {
        B ob1=new B();
        ob1.getData();
        ob1.putData();
        ob1.add();
        C ob=new C();
          ob.getData();
          ob.putData();
          ob.product();
  }
}
```

Access Specifiers / Modifiers

Sometimes, we may not like the objects of a class directly altering the value of a variable or accessing a method. This is achieved by applying visibility modifiers to the instance variables and methods. These visibility modifiers are called access modifiers.

public access : A variable or method declared as public has the widest possible visibility and accessable everywhere i.e., within the package as well as outside the package.
Example : public int a;

```
public void sum()
{
    // statements
}
```

Friendly access : when no modifier is specified, the member defaults to a limited version of public accessibility known as "friendly " level of access. The difference between the public access and friendly access is that, the public modifier makes fields visible in all classes regardless of their packages, while the friendly access makes fields visible only in the same package, but not in other packages.
protected access: The visibility of a protected field lies in between the public and friendly access i.e., the protected modifier makes the fields visible in the same package but also to sub classes in other packages.
private access : private fields are accessible only within their classes. A method declared as private behaves like a method declared final. It prevents the method from being sub classed.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class.
*/
This program contains an error and will not
compile.
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
class Access {
public static void main(String args[]) {
B subOb = new B();
```

```
        S  ubOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total is " + subOb.total);
         }
        }
```

Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. In the following program, **obj** is a reference to **NewData** object, Since **NewData** is a subclass of **Data**, it is permissible to assign **obj** a reference to the **NewData** object. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is way **obj** can't access **data3** and **data4** even it refers to a **NewData** object.

```
class Data{
        int data1;
        int data2;
        }
class NewData extends Data{
{       int data3;
        int data4;
}
public class Javaapp{
    public static void main(String args[]){
        Data obj=new NewData();
        obj.data1=50;
        obj.data2=100;
        System.out.println("obj.data1 = "+obj.data1);
        System.out.println("obj.data2 = "+obj.data2);
        }
    }
```

Super

The super keyword is a reference variable that is used to refer parent class objects

Usage of super keyword

1. super() can be used to invoke immediate parent class constructor.
2. super can be used to refer immediate parent class instance variable.
3. super can be used to refer immediate parent class method.
4. Super() must be the first statement in the sub class constructor to invoke the super class constructor.

// Program to call immediate super class constructor using super

```
class A{
        A{
          System.out.println("Super class Constructor");
          }
}
class B extends A {
        B()
          {
                super();
                System.out.println("Sub class Constructor");
        }
}
class supereg1{
        public static void main(String args[])
        {
          B ob=new B();
        }
}
```

Ouput

Super class Constructor
Sub class Constructor

// Program to refer super class instance variable

```
class A {
int i;
  }
class B extends A{
  int i;
  B(int a, int b)
{
  super.i=a;
    i=b;
}
void show()
{
System.out.println(" i of super class"+ super.i);
System.out.pr intln(" i of sub class"+ i);
}
}
class supereg2
{
public static void main(String args[])
```

```
        {
          B ob=new B(10,20);
           ob.show();
        }}
```

// Program to invoke super class method using super

```
        class A{
                void display()
                {
                  System.out.println("Display of Super class");
                }
           }
        class B extends A{
                void display()
                {
                  System.out.println("Display of Sub class");
                }
                void printMsg()
                {
                display(); // overriding method is called
                super.display();  // overridden method is called
                }
          public static void main(String args[])
        {
                B ob=new B();
                ob.printMsg();
        }
    }
```

Method overloading

It is possible to define two or more methods within the same class that share the same name but differ in the signature i.e., difference in return type or number of parameters passed or data type of the parameters passed or the order of the parameters passed. Method overloading is used when we perform similar kind of operations on different data type. This is called **static polymorphism** because, which method to be invoked is decided at the time of compilation

Example:

```
class Geometry{
                void area(double r){
                double a=3.142*r*r;
                System.out.println("Area of circle is"+a);
                }
                void area(int b, int h){
                double a=0.5*b*h;
                System.out.println("Area of triangle is"+a);
```

```
        }}
class demo{
   public static void main(String args[])
        {
          Geometry g=new Geometry();
          g.area(2.5);
          g.area(10,20);
        }
}
```

Method overriding

When a method in a sub class has the same name and type signature as a method of super class, then the method in the sub class is said override the method in the super class. When a overridden method is called from within a subclass, it always refer to the version of that method defined by the sub class. The version of the method defined in the super class is hidden.

Method overriding allows a sub class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes. The call to overriden method is resolved at runtime, thus called **runtime**/**dynamic** polymorphism.

Example:

```
class A{
        void display()
        {
          System.out.println("Display of Super class");
        }
   }
class B extends A{
        void display()
        {
          System.out.println("Display of Sub class");
        }
class MethodOverride{
{
  public static void main(String args[]) {
  {
        B ob=new B();
   }
}
```

Dynamic Method Dispatch (DMD)

It is the mechanism by which a call to an overridden method is resolved at run time rather than compile time. It is important because this is how java implements run time polymorphism

A super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a super class

reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

```
// Program to demonstrate Dynamic Method Dispatch
class A
{
        void display()
        {
          System.out.println(" welcome to Nokia World");
        }
}
class B extends A
{
        void display()
        {
          System.out.println(" welcome to Samsung World");
        }
}
class DMD
{
        public static void main(String args[])
        {
          A x=new A( );
          x.display( );  // invokes super class display( )
          B y=new B( );
          x=y;
          x.display( );  // invokes sub class display( )
        }
}
```

Abstract Classes

There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

- Abstract class is an incomplete class that contains one or more abstract methods.
- A abstract method is a method that does not contain any implementation
- An abstract class may contain concrete method(with implementation) along with abstract method
- Abstract classes can be inherited

- Abstract class can't be instantiated
- Abstract constructors and abstract static method can't be declared
- An abstract class must have at least one subclass which overrides the abstract methods. If sub class does not override the methods, then it also becomes abstract.

Example:

```
abstract class A
{
        void display()
        {
          System.out.println("Hello");
        }
    abstract void show( );
}
class B extends A
{
        void show( )
        {
          System.out.println(" Show() implemented");
        }
}
class demo
{
        public static void main(String args[])
        {
          B Ob=new B();
          Ob.display();
          Ob.show();
        }
}
```

final Keyword

There are three uses of final

1. If a variable is declared as final, its contents can not be modified, it becomes a constant
2. It is used to prevent method overriding.
3. It is used to prevent inheritance.

// A final variable is a constant

```
class A
{
        int a=10;
        final int b=20;
        void change()
```

```
        {
          a=a+10;
        //b=b+10; final variable cant be modified
        System.out.println(a);
        System.out.println(b)
        }
  }
class finalvar
 {
        public static void main(String args[])
        {
                A ob=new A();
                ob.change():
        }
}
```

//A final method cannot be overridden

```
class A
{
        void display()
        {
          System.out.println("display of super method");
        }
        final void show( )
        {
          System.out.println(" Show() implemented");
        }
}
class B extends A
{
        void display()
        {
          System.out.println("display of sub method");
        }
        /* void show( )
        {
          System.out.println(" Show() implemented");
        }
        cant be inherited
       */
}
```

```
//A final class can't be inherited
final class A
{
        void display()
        {
          System.out.println("Hello");
        }
   }
/*final class A can't be inherited
class B extends A
{
        void show( )
        {
          System.out.println(" Show() implemented");
        }
}
 A final class can't be inherited
*/
```

Differences between a concrete class and an abstract class

Concrete Class	abstract Class
A regular class will have all implemented methods	An abstract class will have at least one abstract method
A normal class need not have a sub class	An abstract class must have at least one sub class which implements all the abstract methods
A regular class can be instantiated	An abstract class can not be instantiated as it as an incomplete class

Differences between an abstract class and a final class

abstract Class	final Class
An abstract class will have at least one abstract method	A final class does not contain any abstract methods rather it contains only concrete methods
An abstract class can not be instantiated as it as an incomplete class	A final class can be instantiated
An abstract class must be inherited	A final class cannot be inherited

Significance of main method in Java

public static main(String args[])

public : public is an access specifier. When a class is preceded by public, then the members of the class may be accessed by outside the class in which it is declared. main() method must be public since it must be called by code outside of its class when the program is started.

static : The keyword static allows main() to be called without having to create a particular instance (object). This is necessary since main() is called before any objects are created.
void : The keyword void simply tells the compiler that main() does not return a value to the operating system.
String args[] :This declares a parameter named args which is an array of instances of the class String. Objects of type String store character streams. args receives any command line arguments supplied when the program is executed.

Significance of System.out.println(): System is a predefined class that provides access to the system. out is the output stream that is connected to the console. println() displays the data on the screen.

Differences between method overloading and method overriding

Method Overlaoding	**Method overriding**
Method overloading occurs at compile time	Method overriding occurs at run time
It supports static polymorphism	It supports dynamic/run time polymorphism
Different number of parameters can be passes to methods	Same number of parameters are passed to the methods
The overloaded methods may have different return types	All methods will have the same return types
Method overloading is performed within the same class	Method overriding is performed between two classes that have inheritance relationship

INTERFACES

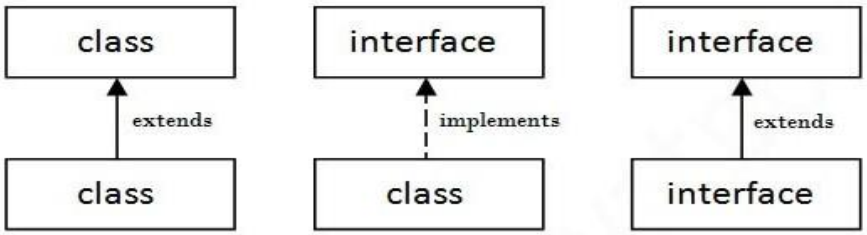
Interfaces are syntactically similar to class but defines only abstract method and final fields (must be static). Once interface is defined, any number of classes can implement. However, each class is free to determine the details of its own implementation. Java allows us to fully utilize the " one interface multiple forms" aspect of polymorphism. If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
Syntax:

```
interface interfacename
{
        Variable declaration;// variables declared must be static and final
        Method declaration;
}
```

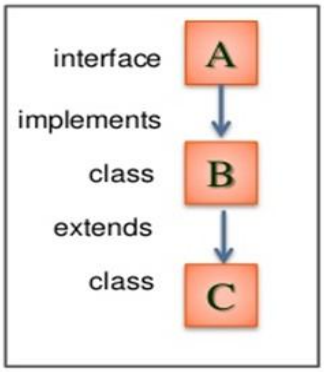
Here, the access is either public or not used. When no access specifier is included, then the default access specifier results and this interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. The methods declared inside an interface are abstract by default, the variables declared must be final and static.

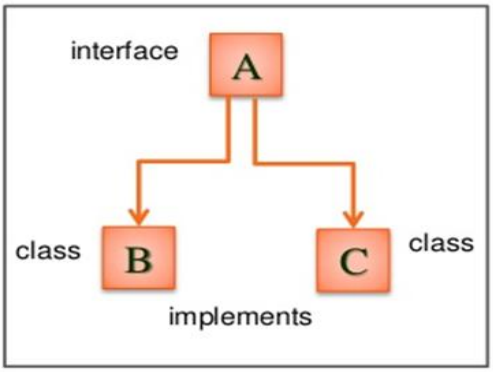
The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Various forms of Interfaces





//Program to demonstrate interface

```
interface printable
        {
        void print();
        }
class A implements printable{
        public void print()
        {
        System.out.println("Hello");
        }
class demo
{
public static void main(String args[]){
A ob = new A();
ob.print();
}
}
```

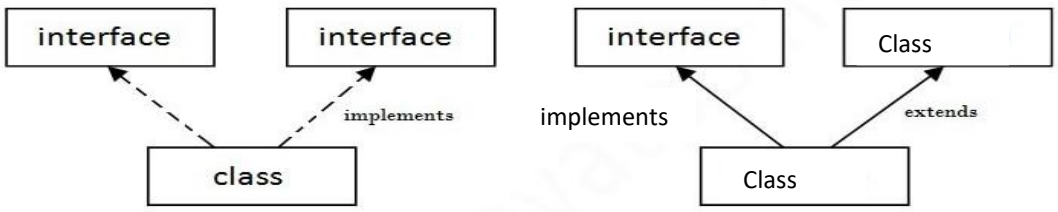
// Interfaces can be extended

```
interface inter1
```

```
{
  void show();
}
interface inter2 extends inter1
{
  void display();
}
class A implements inter2 // class A must override all the methods of inter1 and inter2
{
              public void show()
             {
               System.out.println(" Hai");
             }
            public void display()
             {
               System.out.println(" Hello");
             }
}
class interfaceextends
{
             public static void main(String args[]){
             A ob = new A();
             ob.show();
             ob.display();
            }
}
```

Multiple Inheritance

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class.



Multiple inheritance in java is implemented using interfaces as shown above.

Example 1: To implement Multiple Inheritance

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

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

Example 2: To implement Multiple Inheritance

```
interface Printable
{
        void print();
}
class B
{
        void show()
        {
        System.out.println("Welcome");
        }

}
class A extends B implements Printable
{
        public void print()
        {
        System.out.println("Hello");
        }
```

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

Differences between class and interface

Class	interface
A class is instantiated to create objects	An interface can never be instantiated as the methods are unable to perform any action on invoking
The members of a class can be private, public or protected	The members of an interface are always public
The methods of a class are defined to perform a specific action	The methods in an interface are purely abstract
A class can have constructors to initialize the variables	An interface can never have a constructor as there is hardly any variable to initialize
A class can implement any number of interface and can extend only one class	An interface can extend multiple interfaces but can not implement any interface

Differences between abstract class and interface

Abstract Class	interface
Abstract class can have abstract and non-abstract methods	Interface can have only abstract methods
An abstract class may contain non-final variables	Variables declared in a Java interface are by default final
Abstract class can provide the implementation of interface	Interface can't provide the implementation of abstract class
abstract class can be extended using keyword "extends"	A Java interface can be implemented using keyword "implements"
an abstract class can extend another Java class and implement multiple Java interfaces	An interface can extend another Java interface only