MODULE 2

CLASS FUNDAMENTALS

A class is a user-defined data type. this template can be used to create objects of that type. An object is an *instance* of a class.

A class consists of data and the code that operates on that data. **class** keyword is used to declare a class.

General form of a class –

```
class class name
{
        datatype variable1;
        datatype variable2;          } Instance Variables
         ...
        datatype variable N;
        type methodname1(parameter-list)
        {                                  } Members of a class
        // body of method                  } Methods
        }
        …… ...
        type methodnameN(parameter-list)
        {
        // body of method
        }
}
```

The instance variables are accessed and changed by the methods defined in the same class. Each instance (each object) of the class contains its own separate copy of these variables in memory. Hence they are called as instance variables. The methods are defined inside the class only.

Eg –

```
class Room
{
        int width;
        int length;          } Instance Variables
        int height;

        int Area( )
        {
                int area = length * width;
        }                                          } Methods

        void getData(int x, int y, int z)
        {
                length = y;
```

```
        width = x;
        height = z;
    }
}
```

This is a simple example of class. Here a class called **Room** is defined. Here **Room** is a user-defined data type with 3 instance variables and 2 methods. The methods are using the instance variables in its definition. The Room data type can be used to create objects, this is called instantiating an object.

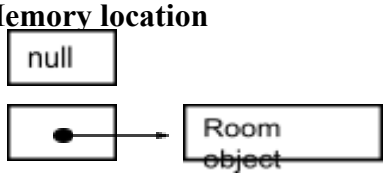
To actually create an object

An object is created or instantiated in Java using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object.

Eg –

```
Room room1;                    //declare the object
room1 = new Room();//instantiate the object
```

Memory location



The first statement declares a variable to hold the object reference and the second statement actually assigns the object reference to the variable. The **new** operator allocates memory at run time.

Each time an object is created, it contains its own copy of each instance variable defined by the class. Thus, every **Room** object will contain its own copies of the instance variables **width**, **height**, and **length**. The method **Room()** is a default constructor of class.

To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **room1** the value 100, the following statement is used –

```
room1.width = 100;
```

This statement tells the compiler to assign the **width** variable of **room1** object with the value of 100. The dot operator is also used methods within an object.

Syntax of Accessing Class Members

```
Object name. variable_ name = value;
object name. method name (parameter _ list);
```

```
// A program to demonstrate the working of class variables.
class Room
{
    double width;
    double length;

    // display Area of a room
    double area()
    {
```

```
                return(width * length);
        }
}
class RoomDemo
{
        public static void main(String args[])
        {
                Room room1 = new Room();
                Room room2 = new Room();

                // assign values to room1's instance variables
                room1.width = 10;
                room1.length = 15;

                // assign different values to room2's instance variables
                room2.width = 10;
                room2.length = 12;

                // display area of first box
                double result;
                result = room1.area();
                System.out.print("Area is ");
                System.out.println(result);

                // display area of second box
                result = room2.area();
                System.out.print("Area is ");
                System.out.println(result);
        }
}
```

Output –
Area is 150.0
Area is 120.0

In this program the following lines of code:
result = room1.area();
result = room2.area();
calls the method **area()** of the class.

The first line here invokes the **area( )** method on **room1**, that is, with reference to the **room1** object, using the object's name followed by the dot operator. Thus, the call to **room1.area( )** displays the area of the room defined by **room1**. Similarly the call to **room2.area( )** displays the area of the room defined by **room2**. As a method is always invoked relative to some object of its class, there is no need to use the dot operator to access variables in method definition.

The file name of this program must be **RoomDemo.java**, because the **main( )** method is in the class called **RoomDemo**.

Each object has its own copies of the instance variables. That is, the two **Room** objects **room1** and **room2** have its own copy of **length** and **width**. Any changes to the instance variables of one object have no effect on the instance variables of another.

The **area( )** method is called, by putting the statement on the right side of an assignment statement. On the left is a variable that will receive the value returned by **area( )**.
Here in the following statement

result = mybox1.area();

the function area() is executed to return a value of 150 or 120 and this value is stored in the variable **result**.

There are **two important things** to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, then an integer cannot be returned.
- The variable receiving the value returned by a method (such as **result**) must also be compatible with the return type specified for the method.

In the above program, the call to **area( )** can be used in the **println( )** statement directly, as follows:

System.out.println("Area is " + room1.area());

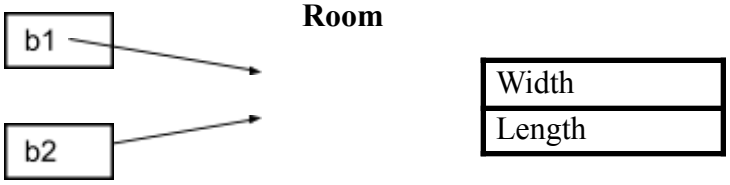
In this case, when **println( )** is executed, **room1.area( )** will be called automatically and its value will be passed to **println( )**.

Assigning object reference variables

An object can take the values or reference of another object.

Eg –
Room b1 = new Room ();
Room b2 = b1;

Here **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 the original object. It simply makes **b2** refer to the same object as done by **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. Here a copy of the reference is done.



Note
Object references are similar to pointers. The main difference—and the key to Java's safety—is that you cannot manipulate references. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

Constructors

A constructor is a special method that initializes an object immediately upon creation. Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

- It has the same name as the class in which it resides.
- It is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the memory is allocated for the object.
- It does not have any return type, not even **void**, because it implicitly returns the class type itself.

Normal methods can be used to initialize values, but it is tedious to call the method each time an object is created. It would be simpler and more concise to have all of the setup done at the time the object is first created.

Default Constructor

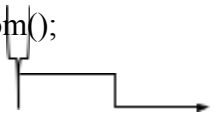
Every class has a default Constructor. Default Constructor for **class Room** is **Room()**. If it is not provided in the program, the compiler creates it and initializes the variables to default values. No values can be sent to default Constructor, as it has an empty argument-list. All the objects calling the default constructor will initialize the same value to its variables.

Eg –

```
Room()
{
        // Default Constructor
        width = 12;
        length = 10;
}
```

Objects are created using the statement

```
Room room1 = new Room();
```



Default Constructor

```
// A program to demonstate the working of default Constructor.
class Room
{
        double width;
        double length;
```

```
        Room()  //Default Constructor
        {
                width = 12;
                length = 10;
        }
        // display Area of a room
        double area()
        {
                return(width * length);
        }
}
class RoomDemo
{
        public static void main(String args[])
        {
                Room room1 = new Room();
                Room room2 = new Room();

                // display area of first box
                double result;
                result = room1.area();
                System.out.print("Area is " + result);

                // display area of second box
                result = room2.area();
                System.out.print("Area is " + result);
        }
}
```

Output –
Area is 120.0
Area is 120.0

Since the constructor gives all objects the same dimensions, 12 by 10, both **room1** and **room2** will have the same area.

Parameterized Constructors

By using Default Constructors, all the objects will have the same dimensions. To construct object of various initialization values, parameterized constructors are used. In this constructor values are sets as specified by the parameters.

```
// A program to demonstrate the working of parameterized Constructors.
class Room
{
        double width;
        double length;

        Room(double w, double l)  //Parameterized Constructor
```

```
        {
                width = w;
                length = l;
        }
        // display Area of a room
        double area()
        {
                return(width * length);
        }
}
class RoomDemo
{
        public static void main(String args[])
        {
                Room room1 = new Room(15,10);
                Room room2 = new Room(12,10);

                // display area of first box
                double result;
                result = room1.area();
                System.out.print("Area is " + result);

                // display area of second box
                result = room2.area();
                System.out.print("Area is " + result);
        }
}
```

Output –
Area is 150.0
Area is 120.0

In this program the following lines of code:

```
Room room1 = new Room(15,10);
Room room2 = new Room(12,10);
```

initializes values as specified in the parameters to its constructor. For the object **room1** the values 15 and 10 are assigned to width and length variables respectively. Similarly for the object **room2**, the values 12 and 10 are assigned to width and length variables.

The 'This' Keyword

The keyword ' **this**' is used in a method to refer to the object that invoked it. **this** can be used inside any method to refer to the *current* object. The '**this**' keyword can also be used to hide the instance variables.

Eg – // Use this to resolve name-space collisions.

```
Box(double width, double height, double depth)
{
        this.width = width;
        this.height = height;
```

```
        this.depth = depth;
}
```

Here the variables this.width, this.height and this.depth are the instance variables and width, height and depth are the local parameters. If '**this'** keyword is not used, there can be no differentiation between the local and instance variables.

Garbage Collection

The unused memory are released automatically in Java, this released memory is later reallocated during the execution. 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. This technique is called ***garbage collection***.

This process is done when

- All references of that object are explicitly set to null, that is object = null;
- Object becomes out of scope, that is the execution comes out of the scope where object is declared.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.*

The finalization is defined by **finalize( )** method in Java. The Java run time calls this method **whenever the object is about to be reclaimed** by the garbage collector. In this method the programmer can specify those actions that must be performed before an object is destroyed.

The general form of **finalize( )** method:

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

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.

Inheritance

Inheritance is an object-oriented property. It allows the reusability of existing code. The mechanism of deriving a new class from an old class is called inheritance. The old class is known as the **super class** or **parent class** or **bass class** and the new class is called as **sub class** or **child class** or the **extended class**.

The idea of inheritance is simple and powerful. When a new class is to be coded and there is already a class that includes some of the code that is required, then the new class can be derived from the existing class. By using this mechanism the user can reuse the fields and methods of the existing class without having to write (and debug) them.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the Constructor of the superclass can be invoked from the subclass.

Syntax of inheritance

```
class subclass name extends superclass name
{
        variables of subclass;
        methods of subclass;
}
```

```
// A simple example of inheritance.
class A //superclass
{
        int i, j;
        void showij()
        {
             System.out.println("i and j: " + i + " " + j);
        }
}

class B extends A //subclass B, by extending class A
{
        int k;
        void showk()
        {
            System.out.println("k: " + k);
        }
        void sum()
        {
             System.out.println("i+j+k: " + (i+j+k));
        }
}
class SimpleInheritance
{
        public static void main(String args[])
        {
                A superOb = new A();
                B subOb = new B();
```

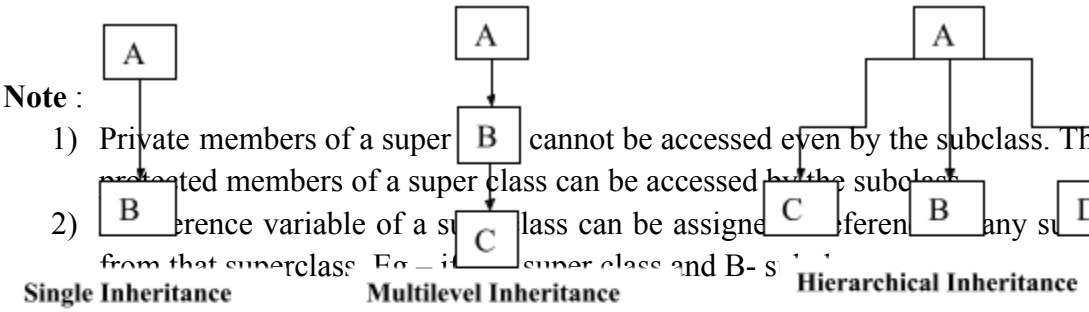
```
        // The superclass may be used by itself, independent
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of its superclass. */
        subOb.i = 7; // i & j are variables of super class initialized here using subclass obj.
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();  // method of super class
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

Output –
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24

here, the subclass **B** includes all of the members of its superclass **A**. s**ubOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred directly, as if they are part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. A subclass can be a superclass for another subclass.
Java supports Single Inheritance, Multilevel Inheritance and Hierarchical Inheritance. It indirectly supports Multiple Inheritance, by using interfaces.

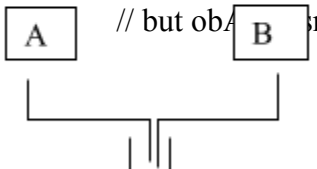


Single Inheritance **Multilevel Inheritance** **Hierarchical Inheritance**

Note :

1) Private members of a super cannot be accessed even by the subclass. The public and protected members of a super class can be accessed by the subclass.
2) erence variable of a su lass can be assigne eferen any su s derived from that superclass. Eg – i super class and B- s

```
B obB = new B();
obA =    // but obA not define the members defined in B
```



Super class

A subclass can refer to its **immediate superclass** explicitly by using the keyword **super**. The keyword **super** has two general forms –

- To call the superclass constructor
- To access a member of the superclass that has been hidden by a member of a subclass(overriding)

Using super to Call Superclass Constructors

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

```
super();   // Default constructor
OR
super(parameter-list);    //Parameterized constructor
```

super( ) must always be the first statement executed inside a subclass constructor and it calls only the immediate superclass.

```
// A complete implementation of BoxWeight.
class Box
{
        private double width;
        private double height;
        private double depth;

        // constructor used when all dimensions specified
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }

        // constructor used when no dimensions specified
        Box()
        {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }

        // constructor used when cube is created
        Box(double len)
        {
                width = height = depth = len;
        }
```

```
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box
{
        double weight; // weight of box

        // constructor when all parameters are specified
        BoxWeight(double w, double h, double d, double m)
        {
                super(w, h, d); // call superclass constructor
                weight = m;
        }

        // default constructor
        BoxWeight()
        {
                super();
                weight = -1;
        }

        // constructor used when cube is created
        BoxWeight(double len, double m)
        {
                super(len);
                weight = m;
        }
}

class DemoSuper
{
        public static void main(String args[])
        {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                BoxWeight mybox3 = new BoxWeight(); // default
                BoxWeight mycube = new BoxWeight(3, 2);

                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                System.out.println("Weight of mybox1 is " + mybox1.weight);
                System.out.println();

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
```

```
                System.out.println("Weight of mybox2 is " + mybox2.weight);
                System.out.println();

                vol = mybox3.volume();
                System.out.println("Volume of mybox3 is " + vol);
                System.out.println("Weight of mybox3 is " + mybox3.weight);
                System.out.println();

                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
                System.out.println("Weight of mycube is " + mycube.weight);
                System.out.println();
        }
}
```

Output –

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of mycube is 27.0
Weight of mycube is 2.0
```

Here, when a subclass calls **super( )**, it is calling the constructor of its immediate superclass. This is the case even in a multileveled hierarchy.

Using super to access the members of super class

The **super** keyword can be used to access the members of the super class from the subclass.

Syntax –

```
        super.member
```

here, *member* can be either a method or an instance variable.
This is used in situations in which member of a subclass hide members in the superclass.

```
class A // Super class
{
        int i;
}

// Create a subclass by extending class A.
class B extends A
{
        int i; // this i hides the i in A
        B(int a, int b)
        {
                super.i = a; // i in A
```

```
        i = b; // i in B
    }
    void show()
    {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

class UseSuper
{
    public static void main(String args[])
    {
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```

Output –
i in superclass: 1
i in subclass: 2

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. **super** can also be used to call methods that are hidden by a subclass.

Multilevel Hierarchy

A subclass can be a superclass of another class. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. Each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.

Eg -

```
Class A
{
    int x,y;
    A(int x1,int y1)
    {

        x = x1;
        y=y1;
    }

    Void displayA();
    {
        System.out.println (“ value of x is “ + x);
        System.out.println (“ value of y is “ + y);
    }
}
```

```
Class B extends A
{
	int m;
	B(int x1, int y1, int m1)
	{
		super(x1,y1);
		m=m1;
	}

	Void displayB()
	{
		System.out.println(“ value of m is “ + m);
	}
}


Class C extends B
{
	int a;
	B(int x1, int y1, int m1,int a1)
	{
		super(x1,y1,m1);
		a=a1;
	}

	Void displayC()
	{
		displayA();
		displayB();
		System.out.println(“ value of a is “ + a);
	}
}

class DemoMultilevel
{
	public static void main(String args[])
	{
		C objc = new C(23,5,4,8);
		objc.displayC();

	}
}
```

Output –

value of x is 23
value of y is 5
value of m is 4
value of x is 8

Because of inheritance, class **C** can make use of the previously defined members of **B** and **A,** that is inheritance allows the reuse of code. **super( )** always refers to the constructor in the closest superclass. The **super( )** in **C** calls the constructor in **B** and the **super( )** in **B** calls the constructor in **A**.

Note : If **super( )** is not used, then the default constructor of immediate super class will be executed first and then the sub class constructor.

Execution of Constructors

The constructors are executed in order of derivation. As a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

```
// Demonstrate when constructors are called.
// Create a super class.
class A
{
        A()
        {
           System.out.println("Inside A's constructor.");
        }
}

// Create a subclass by extending class A.
class B extends A
{
        B()
        {
            System.out.println("Inside B's constructor.");
        }
}

// Create another subclass by extending B.
class C extends B
{
        C()
        {
            System.out.println("Inside C's constructor.");
        }
}

class CallingCons
{
        public static void main(String args[])
        {
           C c = new C();
        }
}
```

Output –
Inside A's constructor
Inside B's constructor
Inside C's constructor

Abstract methods and classes

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), eg - abstract void moveTo(int x, int y);

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. If a method or a class is specified as abstract, then they must be compulsorily redefined.

If a class contains one or more abstract methods, class should also be declared as **abstract**.

Eg –

```
abstract class Animal
{
        ……………
        ……….
        abstract class dog();
        …….
        abstract class cat();
        ……..
        …
}
```

An abstract class cannot instantiate objects directly. It has to be subclassed and the methods of abstract class must be defined in the subclass. However, if it does not, the subclass must also be declared `abstract`.

Multiple Inheritance

A Java class cannot be a subclass of more than one superclass. So Java provides an alternate approach known as **interfaces** to support the concept of multiple inheritance.

Method Overriding

A method in a subclass with the same name, parameter-list and return type as in its superclass, then this method in the subclass is said to *override* the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Eg –// Method overriding.

```
class A
{
        int i, j;
        A(int a, int b)
        {
                i = a;
                j = b;
        }
        // display i and j
        void show()
        {
                System.out.println("i and j: " + i + " " + j);
        }
}

class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a, b);
                k = c;
        }
        // display k – this overrides show() in A
        void show()
        {
                System.out.println("k: " + k);
        }
}

class Override
{
        public static void main(String args[])
        {
                B subOb = new B(1, 2, 3);
                subOb.show(); // this calls show() in B
        }
}
```

Output –
k: 3

Here, the **show( )** method inside **B** overrides the method declared in **A**. The hidden method of the superclass can be accessed by using the keyword **super**, as shown below –

```
class B extends A
{
        int k;
        B(int a, int b, int c)
```

```
	{
		super(a, b);
		k = c;
	}
	void show()
	{
		super.show(); // this calls A's show()
		System.out.println("k: " + k);
	}
}
```

Now the output will be –

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )** method. Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Using final to Prevent Overriding

The keyword **final** can be specified at the start of method declaration, to disallow a method from being overridden. Methods declared as **final** cannot be overridden.

```
class A
{
	final void meth()
	{
		System.out.println("This is a final method.");
	}
}

class B extends A
{
	void meth()// ERROR! Can't override.
	{
		System.out.println("Illegal!");
	}
}
```

As the function **meth( )** is declared as **final**, it cannot be overridden in **B**. If such an attempt is made, a compile-time error will result.

Using final to Prevent Inheritance

To prevent a class from being further inherited, the class declaration is preceded with **final** keyword. Declaring a class as **final** implicitly declares all of its methods as **final**, too.

It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
```

```
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

It is illegal for **B** to inherit **A** since **A** is declared as **final**.

Method overloading

Methods with the same name, but different parameter-list or return type are called overloaded methods. The overloaded methods can be in the same class or in extended classes. It is one of the ways by which Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

Eg –

```
class A
{
        int i, j;
        A(int a, int b)
        {
                i = a;
                j = b;
        }

        void show()// display i and j
        {
                System.out.println("i and j: " + i + " " + j);
        }
}

// Create a subclass by extending class A.
class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a, b);
                k = c;
        }
        void show(String msg) // overloaded show()
        {
                System.out.println(msg + k);
        }
}
```

```
class Override
{
	public static void main(String args[])
	{
		B subOb = new B(1, 2, 3);
		subOb.show("This is k: "); // this calls show() in B
		subOb.show(); // this calls show() in A
	}
}
```

Output –
This is k: 3
i and j: 1 2

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Thus the methods are overloaded.