Programming in Java

Inheritance

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

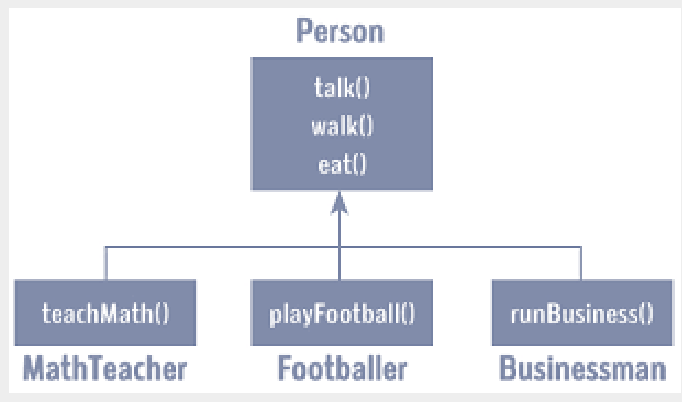
1. Inheritance Basics
2. Using super
3. Creating a Multilevel Hierarchy
4. When Constructors are Executed
5. Method Overriding
6. Dynamic Method Dispatch
7. Using Abstract Classes
8. Using Final with Inheritance
9. The Object Class

Inheritance

- One of the important concept/feature of Object Oriented Programming.
- It allows/facilitates Reusability through the Hierarchical Classification.

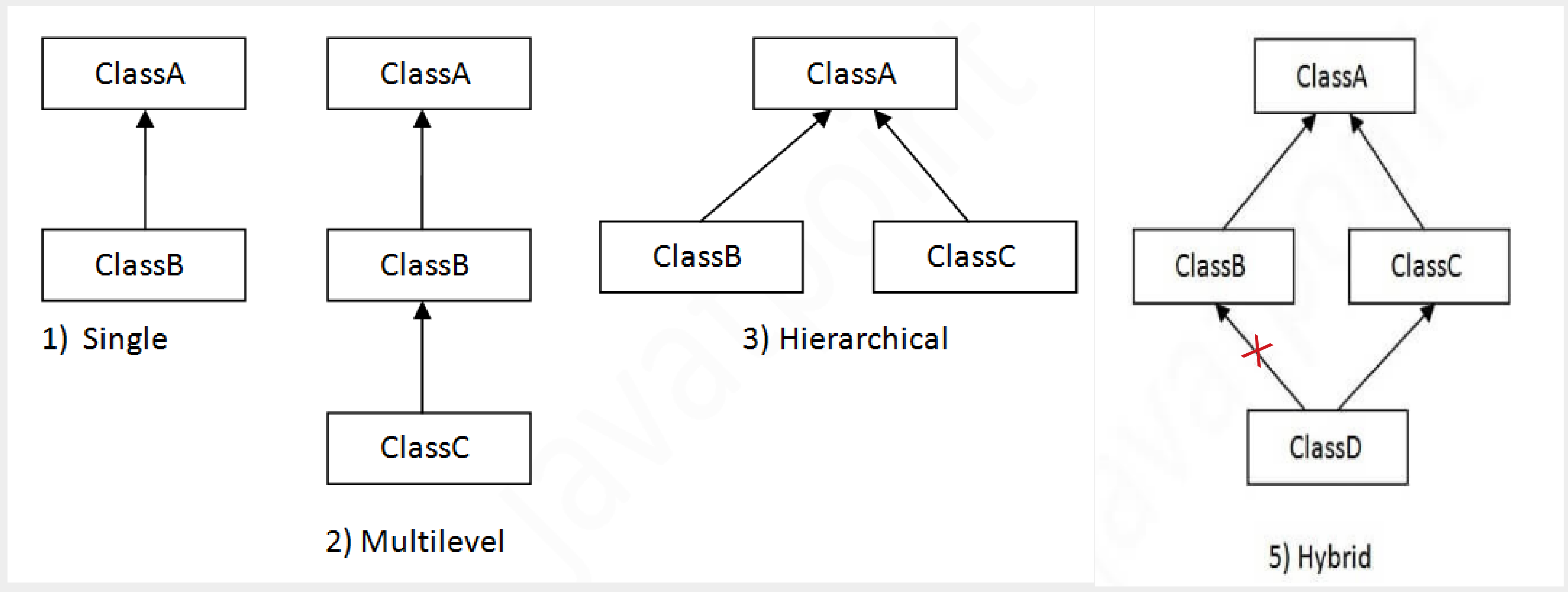
1. *Superclass*
   - *Defines the general aspects of an object (attributes common to a set of objects) .*
   - *It can be used to create any number of more specific subclasses.*
   - *Also known as base class or parent class.*

2. *Subclass*
   - *Specialized version of a Superclass.*
   - *Inherits the Superclass (common traits/properties).*
   - *Adds things that are unique to it (its own, unique elements).*
   - *Also known as derived class or child class.*



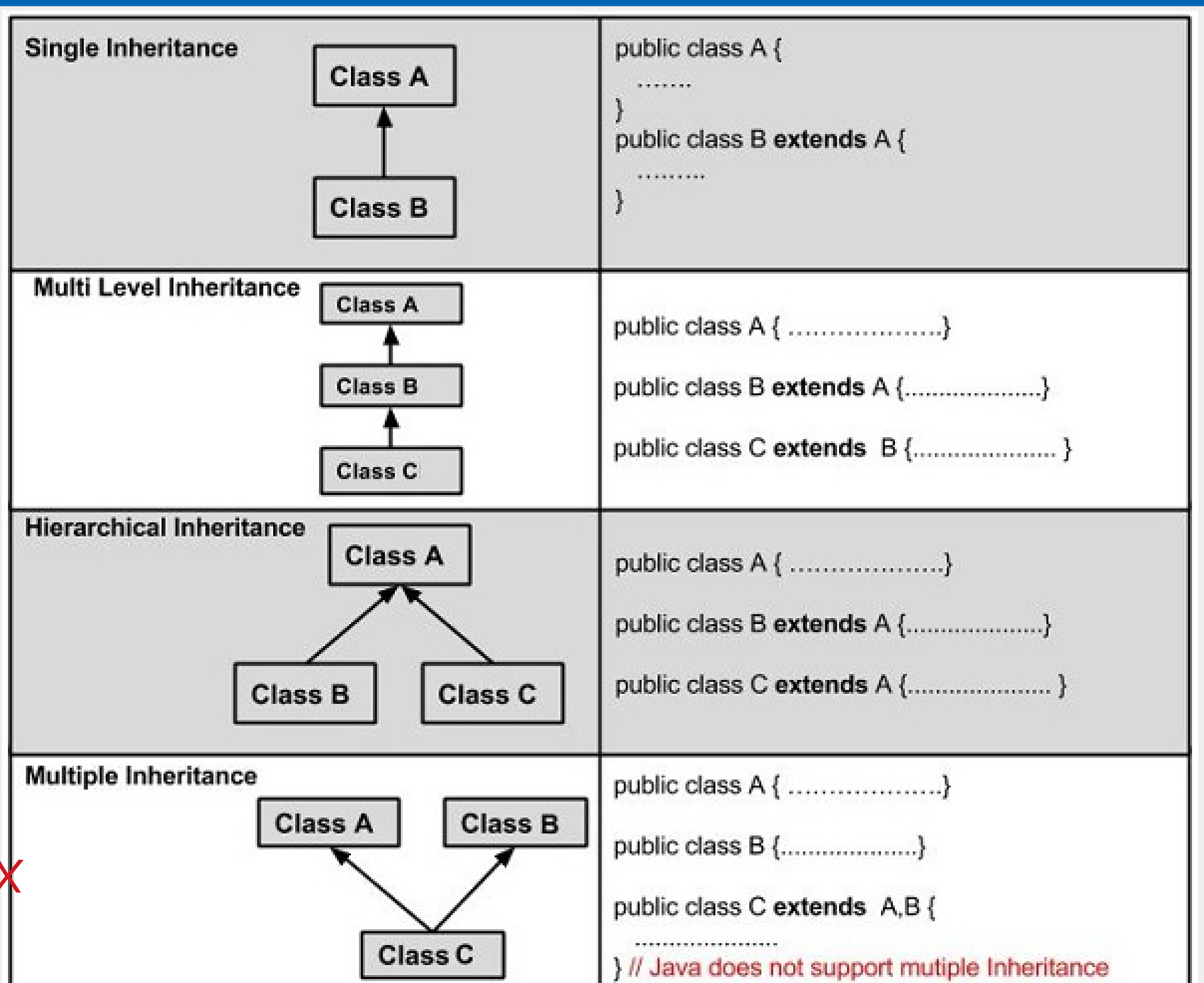
Example of Inheritance

Types of Inheritance in Java



Types of Inheritance [3]

Types of Inheritance in Java



Types of Inheritance [2]

Inheritance Basics

- The extends keyword is used to inherit a class.
- The general form of a class declaration that inherits a Superclass is shown here:

```
class subclass-name extends superclass-name {

    // body of class

}
```

#Note:
superclass is also a completely independent, stand-alone class, can be used by itself.

1. You can only specify one superclass for any subclass that you create.
   - Multiple inheritance is not supported in Java.
2. You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.
3. However, no class can be a superclass of itself.

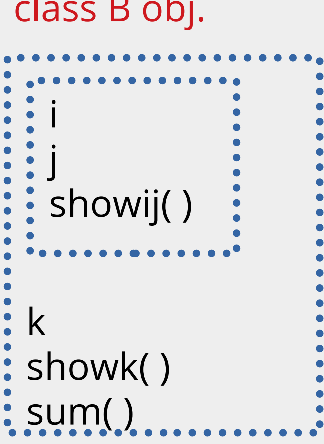
Inheritance Basics – A simple example of Inheritance

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

// Create a subclass by extending class A.
class B extends A {
	int k;
	void showk( ) {
		System.out.println("k: " + k);
	}
	void sum( ) {
		System.out.println("i+j+k: " + (i+j+k));
	}
}
```

class A obj.

i
j
showij( )

class B obj.



Inheritance Basics – A simple example of Inheritance

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

// Create a subclass by extending class A.
class B extends 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.
                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;
                subOb.j = 8;
                subOb.k = 9;
                System.out.println("Contents of subOb: ");
                subOb.showij( );
                subOb.showk( );
                System.out.println( );
                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum( );
        }
}
```

Inheritance Basics – A simple example of Inheritance

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
```

superOb

```
I = 10
J = 20
showij( )
```

subOb

```
I = 7
J = 8
showij( )

K = 9
showk( )
sum( )
```

```
class SimpleInheritance {
	public static void main(String args [ ]) {
		A superOb = new A( );
		B subOb = new B( );

		// The superclass may be used by itself.
		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;
		subOb.j = 8;
		subOb.k = 9;
		System.out.println("Contents of subOb: ");
		subOb.showij( );
		subOb.showk( );
		System.out.println( );
		System.out.println("Sum of i, j and k in subOb:");
		subOb.sum( );
	}
}
```

Member Access and Inheritance

```
// Create a superclass.
class A {
        int i;              // default access
        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( );
                subOb.setij(10, 12);
                subOb.sum( );
                System.out.println("Total is " + subOb.total);
        }
}
```

1. 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.
2. In a class hierarchy, private members remain private to their class.
3. #REMEMBER A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) {
        width = ob.width;
        height = ob.height;
        depth = ob.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;
    }
}
```

A More Practical Example

```
// Here, Box is extended to include weight.

class BoxWeight extends Box {

    double weight;  // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h,
              double d, double wt) {
        width = w;
        height = h;
        depth = d;
        weight = wt;
    }
}
```

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

        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

        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);
    }
}
```

A More Practical Example

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

                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

                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);
        }
}
```

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
```

A More Practical Example

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- Each subclass can precisely tailor its own classification.

```
// Here, Box is extended to include color.

class ColorBox extends Box {

        int color; // color of box

        ColorBox(double w, double h, double d, int c) {
                width = w;
                height = h;
                depth = d;
                color = c;
        }
}
```

A Superclass Variable Can Reference a Subclass Object

```
class RefDemo {
    public static void main(String args[ ]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);     // weightbox is a reference to BoxWeight objects
        Box plainbox = new Box( );                              // plainbox is a reference to Box objects.
        double vol;
        vol = weightbox.volume( );
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " + weightbox.weight);

        // assign BoxWeight reference to Box reference, since BoxWeight is a subclass of Box
        plainbox = weightbox;
        vol = plainbox.volume( );      // OK, volume( ) defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox does not define a weight member.
            1. 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.
            2. Because the superclass has no knowledge of what a subclass adds to it   */

        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}
```

Using Super

- So far inheritance were not implemented as efficiently or as robustly as they could have been. For example:

```
class BoxWeight extends Box {

        double weight;  // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h,
                    double d, double wt) {

                width = w;
                height = h;
                depth = d;
                weight = wt;
        }
}
```

Using Super

- So far inheritance were not implemented as efficiently or as robustly as they could have been. For example:

```
class BoxWeight extends Box {

    double weight;  // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h,
              double d, double wt) {

        width = w;
        height = h;
        depth = d;
        weight = wt;
    }
}
```

1. The constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box.**
2. Two issues of concern:
   - *Duplicate code in its superclass (inefficient)*
   - *But it implies that a subclass must be granted access to these members.*
3. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).
4. In this case, there would be no way for a subclass to directly access or initialize these variables on its own.

Using Super

- So far inheritance were not implemented as efficiently or as robustly as they could have been. For example:

```
class BoxWeight extends Box {

	double weight;  // weight of box

	// constructor for BoxWeight
	BoxWeight(double w, double h,
			double d, double wt) {

		width = w;
		height = h;
		depth = d;
		weight = wt;
	}
}
```

5. Since ***encapsulation*** is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem.

6. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

7. **super** has two general forms:
   - *Can be used to Call the superclass' constructor*
   - *Can be used to access a member of the superclass*

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(arg-list);

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {

    double weight;          // weight of box

    // initialize width, height, and depth using super( )
    BoxWeight(double w, double h, double d, double wt) {

        super(w, h, d);       // call superclass constructor
        weight = wt;
    }
}
```

0. Here, arg-list specifies any arguments needed by the constructor in the superclass.
1. When a subclass calls super( ), it is calling the constructor of its immediate superclass.
2. Thus, super( ) always refers to the superclass immediately above the calling class.
3. This is true even in a multileveled hierarchy.
4. Also, super( ) must always be the first statement executed inside a subclass constructor.

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(arg-list);

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {

    double weight;          // weight of box

    // initialize width, height, and depth using super( )
    BoxWeight(double w, double h, double d, double wt) {

        super(w, h, d);        // call superclass constructor
        weight = wt;
    }
}
```

5. Here, BoxWeight( ) calls super( ) with the arguments **w, h,** and **d**. This causes the Box constructor to be called, which initializes width, height, and depth using these values.

6. BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**.

7. This leaves Box free to make these values private if desired.

8. Since constructors can be overloaded, super( ) can be called using any form defined by the superclass.

Using Super to Call Superclass Constructors

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

	// construct clone of an object
	Box(Box ob) {
		width = ob.width;
		height = ob.height;
		depth = ob.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;
	}
}
```

Using Super to Call Superclass Constructors

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

	// construct clone of an object
	BoxWeight(BoxWeight ob) {
		super(ob);
		weight = ob.weight;
	}

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

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

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

Using Super to Call Superclass Constructors

```
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);
        BoxWeight myclone = new BoxWeight(mybox1);  // clone

        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( );
```

Using Super to Call Superclass Constructors

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

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

Using Super to Call Superclass Constructors

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 myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0
```

Using Super to Call Superclass Constructors

```
// construct clone of an object
BoxWeight(BoxWeight ob) {

        super(ob);
        weight = ob.weight;
}
```

- Notice that super( ) is passed an object of type BoxWeight—not of type Box.
- This still invokes the constructor Box(Box ob).
- As mentioned earlier, a superclass variable can be used to reference any object derived from that class.
- Thus, we are able to pass a BoxWeight object to the Box constructor. Of course, Box only has knowledge of its own members.

Using Super to access member of Superclass

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

  super.member

- Here, member can be either a method or an instance variable.
- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Using Super to access member of Superclass

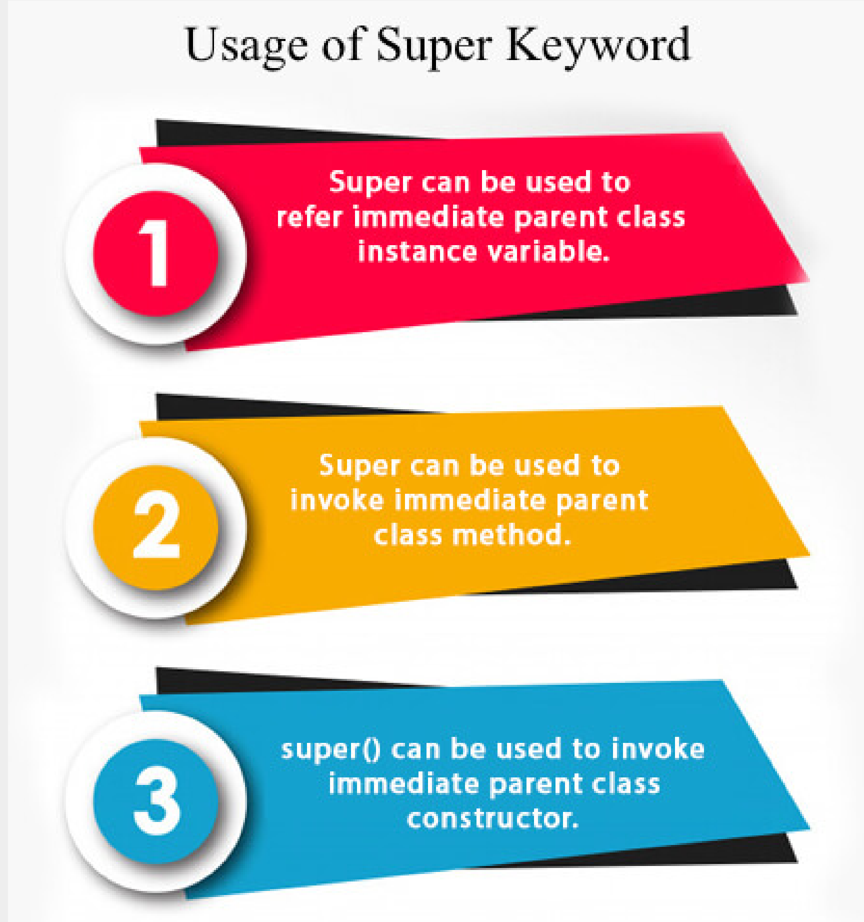
```
// Using super to overcome name hiding.
class A {
        int i;
}

// 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
```

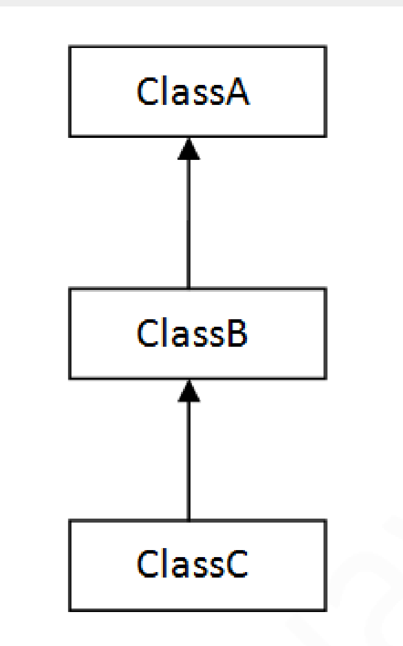
Using Super - Summary



[Source: (3)]

Creating a Multilevel Hierarchy

- You can build hierarchies that contain as many layers of inheritance as you like.
- As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A.
- When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses.
- In this case, C inherits all aspects of B and A.
- NOTE: The class hierarchy, including A, B, and C, can be in one file. In Java, all three classes can be placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.



Creating a Multilevel Hierarchy

```
// Extend BoxWeight to include shipping costs.
// A complete implementation of BoxWeight.
class Box {
	private double width;
	private double height;
	private double depth;

	// construct clone of an object
	Box(Box ob) {
		width = ob.width;
		height = ob.height;
		depth = ob.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;
}
}
```

Creating a Multilevel Hierarchy

```
// Add weight
class BoxWeight extends Box {
	double weight;	// weight of box

	// construct clone of an object
	BoxWeight(BoxWeight ob) {
		super(ob);
		weight = ob.weight;
	}

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

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

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

Creating a Multilevel Hierarchy

```
// Add shipping costs.
class Shipment extends BoxWeight {
	double cost;
	// construct clone of an object
	Shipment(Shipment ob) {
		super(ob);
		cost = ob.cost;
	}

	// constructor when all parameters are specified
	Shipment(double w, double h, double d, double wt, double c) {
		super(w, h, d, wt);     // call superclass constructor
		cost = c;
	}

	// default constructor
	Shipment( ) {
		super( );
		cost = -1;
	}

	// constructor used when cube is created
	Shipment(double len, double wt, double c) {
		super(len, wt);
		cost = c;
	}
}
```

Creating a Multilevel Hierarchy

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

        Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);

        double vol;

        vol = shipment1.volume( );
        System.out.println("Volume of shipment1 is " + vol);
        System.out.println("Weight of shipment1 is " + shipment1.weight);
        System.out.println("Shipping cost: $" + shipment1.cost);
        System.out.println( );

        vol = shipment2.volume( );
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "+ shipment2.weight);
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}
```

OUTPUT

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

When Constructors are Executed?

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed?
- For example, given a subclass called B and a superclass called A, is A's constructor executed before B's, or vice versa?
- The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.
- If super( ) is not used, then the default or parameterless constructor of each superclass will be executed.

When Constructors are Executed?

```
// Demonstrate when constructors are executed.
// 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
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

- When an overridden method is called from within its 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.

Method Overriding

```
// 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
        }
}
```

This program displays the following output :

k: 3

1. When show( ) is invoked on an object of type B, the version of show( ) defined within B is used.
2. That is, the version of show( ) inside B overrides the version declared in A.

Q. How to access the superclass version of an overridden method?

Method Overriding

- If you wish to access the superclass version of an overridden method, you can do so by using super.

```
// To access Superclass version of show( )

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);
        }
}
```

This program displays the following output :

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

1. Here, super.show( ) calls the superclass version of show( ).
2. Method overriding occurs only when the names and the type signatures of the two methods are identical.

Q. What if names and the type signatures of the two methods are non-identical?

Method Overriding

```
// Methods with differing type signatures are overloaded – not overridden.
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;
	}
	void show(String msg) {  // overload 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
	}
}
```

This program displays the following output :

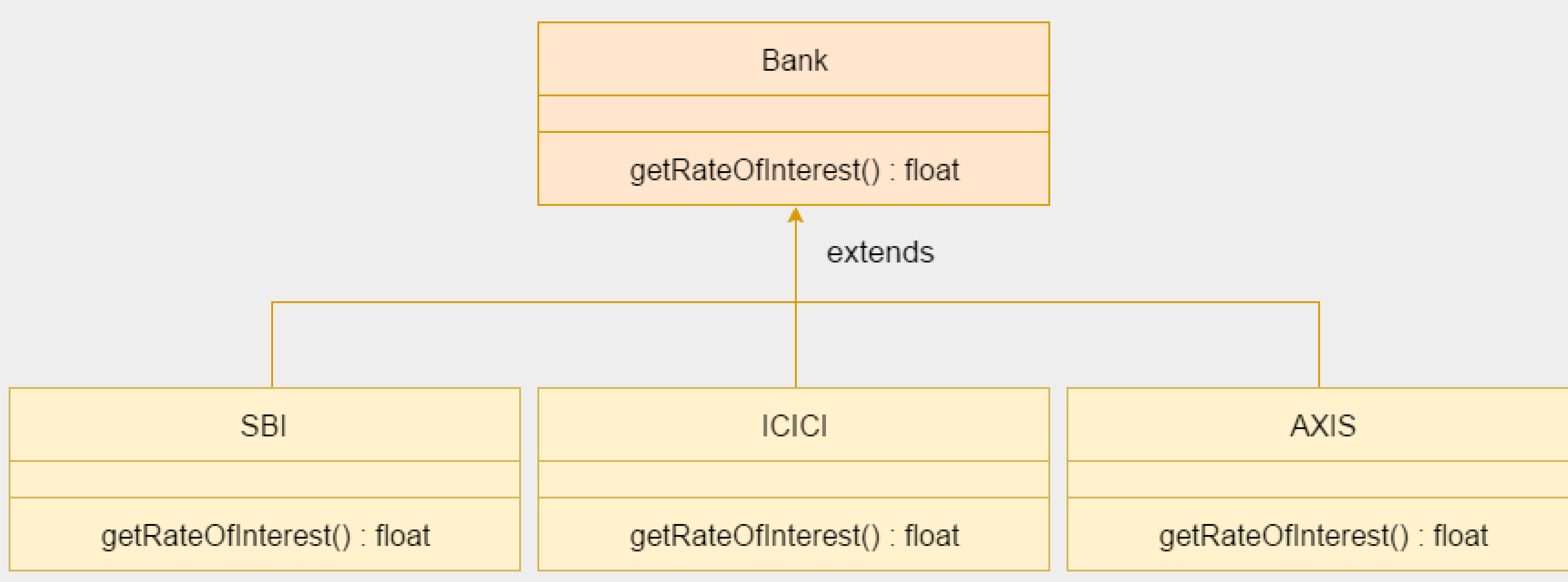
```
This is k: 3
i and j: 1 2
```

1. 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.
2. Therefore, no overriding (or name hiding) takes place – so show( ) is overloaded here.

Method Overriding - Summary

[Source: (3)]

Method Overriding - Summary



[Source: (3)]

Method Overriding - Summary

```
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
class Bank{
        int getRateOfInterest( ){
                return 0;
        }
}
class SBI extends Bank{
        int getRateOfInterest( ){
                return 8;
        }
}
class ICICI extends Bank{
        int getRateOfInterest( ){
                return 7;
        }
}
class AXIS extends Bank{
        int getRateOfInterest( ){
                return 9;
        }
}
```

```
//Test class to create objects and call the methods
class Test{
        public static void main(String args[ ]){
                SBI s=new SBI();
                ICICI i=new ICICI();
                AXIS a=new AXIS();
                System.out.println("SBI Rate of Interest: “
                                        +s.getRateOfInterest( ));
                System.out.println("ICICI Rate of Interest: "
                                        +i.getRateOfInterest( ));
                System.out.println("AXIS Rate of Interest: "
                                        +a.getRateOfInterest( ));
        }
}
```

OUTPUT

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

[Source: (3)]

Method Overriding vs Overloading

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

[Source: (3)]

Method Overriding vs Overloading

```
//Method Overloading example
class OverloadingExample{

        static int add(int a,int b){
                return a+b;
        }

        static int add(int a,int b,int c){
                return a+b+c;
        }
}
```

```
//Method Overriding example
class Animal{

        void eat( ){
                System.out.println("eating...");
        }
}

class Dog extends Animal{

        void eat( ){
                System.out.println("eating bread...");
        }
}
```

[Source: (3)]

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

- As aready discussed, a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.

- When an overridden method is called through a superclass 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.

Dynamic Method Dispatch

- 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.
- In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Dynamic Method Dispatch

```
// Dynamic Method Dispatch
class A {
	void callme( ) {
		System.out.println("Inside A's callme method");
	}
}

class B extends A {
	// override callme( )
	void callme( ) {
		System.out.println("Inside B's callme method");
	}
}

class C extends A {
	// override callme( )
	void callme( ) {
		System.out.println("Inside C's callme method");
	}
}
```

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

		A a = new A( );   // object of type A
		B b = new B( );   // object of type B
		C c = new C( );   // object of type C

		A r;            // obtain a reference of type A

		r = a;          // r refers to an A object
		r.callme( ); // calls A's version of callme

		r = b;          // r refers to a B object
		r.callme( ); // calls B's version of callme

		r = c;                  // r refers to a C object
		r.callme( );// calls C's version of callme
	}
}
```

Dynamic Method Dispatch

OUTPUT

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

NOTE: the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

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

                A a = new A( );   // object of type A
                B b = new B( );   // object of type B
                C c = new C( );   // object of type C

                A r;              // obtain a reference of type A

                r = a;            // r refers to an A object
                r.callme( ); // calls A's version of callme

                r = b;            // r refers to a B object
                r.callme( ); // calls B's version of callme

                r = c;            // r refers to a C object
                r.callme( );// calls C's version of callme
        }
}
```

Why Overridden Methods?

- The overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization.
- Used correctly, the superclass provides all elements that a subclass can use directly.
- It also defines those methods that the derived class must implement on its own.

Why Overridden Methods?

- This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.
- Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

```
// Using run-time polymorphism (a more practical example)
class Figure {
	double dim1;
	double dim2;
	Figure(double a, double b) {
		dim1 = a;
		dim2 = b;
	}
	double area( ) {
		System.out.println("Area for Figure is undefined.");
		return 0;
	}
}
class Rectangle extends Figure {
	Rectangle(double a, double b) {
		super(a, b);
	}
	double area( ) {  // override area for rectangle
		System.out.println("Inside Area for Rectangle.");
		return dim1 * dim2;
	}
}
```

Applying Method Overriding

```
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        double area( ) {     // override area for right triangle
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class FindAreas {
        public static void main(String args[ ]) {
                Figure f = new Figure(10, 10);
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                Figure figref;
                figref = r;
                System.out.println("Area is " + figref.area());
                figref = t;
                System.out.println("Area is " + figref.area());
                figref = f;
                System.out.println("Area is " + figref.area());
        }
}
```

OUTPUT

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

1. Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects.
2. In this case, if an object is derived from Figure, then its area can be obtained by calling area( ).
3. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classess

- It is used to achieve abstraction which is one of the pillar of Object Oriented Programming(OOP).
- Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.
- A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- To declare a class abstract, use this general form :

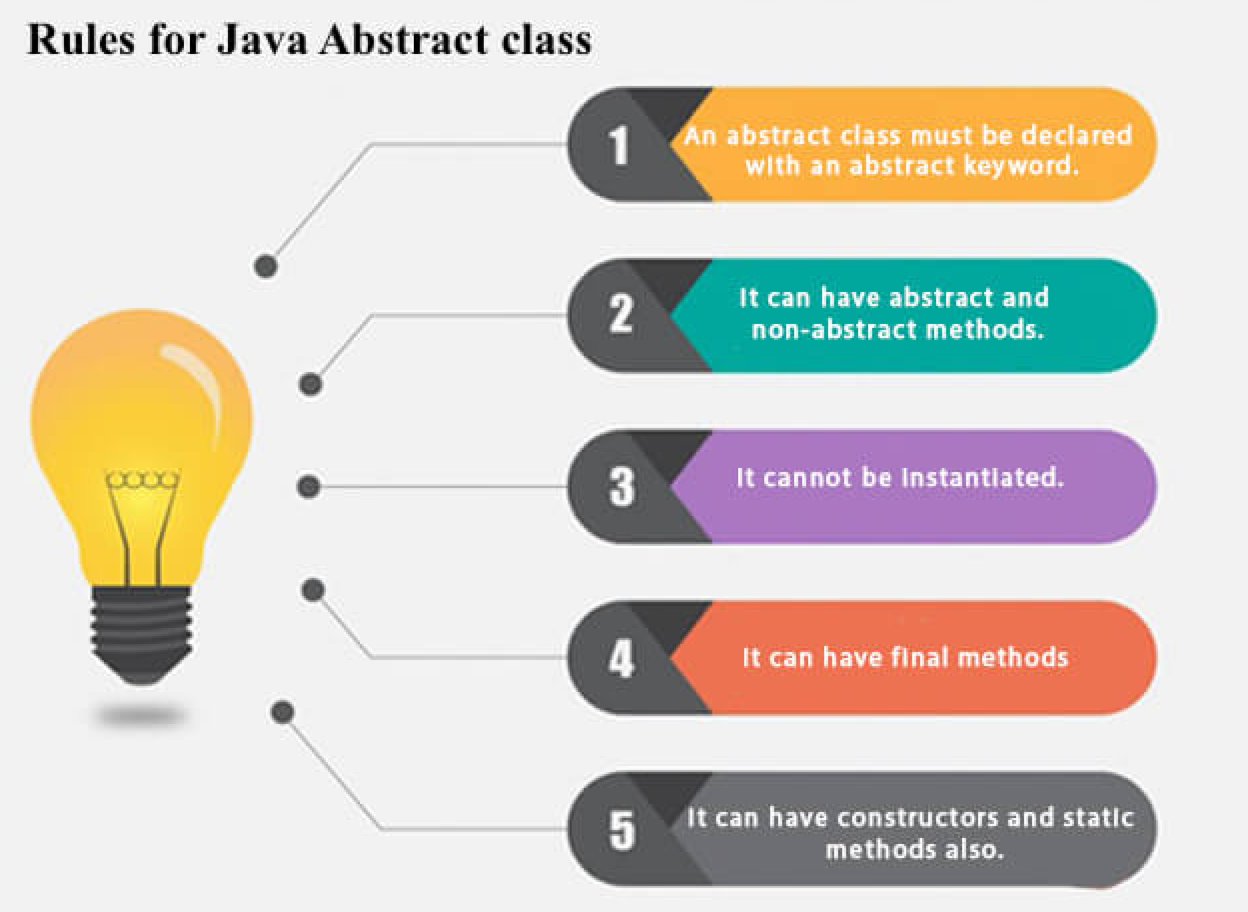
```
abstract class class-name{
        //body of class
}
```

Using Abstract Classess

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:

```
abstract type name(parameter-list);  // no method body is present.
```

Using Abstract Classess



[Source: (3)]

Using Abstract Classess

```
// A Simple demonstration of abstract.
abstract class A {
        abstract void callme( );
        // concrete methods are still allowed in abstract classes
        void callmetoo( ) {
                System.out.println("This is a concrete method.");
        }
}
class B extends A {
        void callme( ) { // must override*
                System.out.println("B's implementation of callme.");
        }
}
class AbstractDemo {
        public static void main(String args[ ]) {
                B b = new B( );
                b.callme( );
                b.callmetoo( );
        }
}
```

1. Notice that no objects of class A are declared in the program
2. class A implements a concrete method (non-abstract) called callmetoo( ).
3. Abstract classes can include as much implementation as they see fit.
4. Abstract classes can not be instantiated, but can create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

* ohterwise **Compile Time** error will occur

Using Abstract Classess

```
// Improving th Figure class shown earlier
// Using abstract methods and classes
abstract class Figure{
        double dim1;
        double dim2;
        Figure(double a, double b) {
                dim1 = a;
                dim2 = b;
        }
        // area( ) is now an abstract method
        abstract double area( );
}
class Rectangle extends Figure {
        Rectangle(double a, double b) {
                super(a, b);
        }
        // Must override area( )*
        double area( ) {
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
}
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        // Must override area( )*
        double area( ) {
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class AbstractAreas {
        public static void main(String args[ ]) {
            // Figure f = new Figure(10, 10);    // illegal now
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                Figure figref; // this is OK, no object is created
                figref = r;
                System.out.println("Area is " + figref.area( ));
                figref = t;
                System.out.println("Area is " + figref.area( ));
        }
}
```

* ohterwise **Compile Time** error will occur

Using final with Inheritance

- The keyword final has three uses:
  1. Create the equivalent of a named constant (already discussed).
  2. Using final to prevent Overriding
     - To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
     - Methods declared as final cannot be overridden
  3. Using final to prevent Inheritance
     - To prevent a class from being inherited, precede the class declaration with final.
     - Declaring a class as final implicitly declares all of its methods as final, too.
- Can we declare a class as both abstract and final ?

Using final to Prevent Overridding

```
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!");
        }
}
```

* **Compile Time** error will occur

Using final to Prevent Inheritance

```
final class A {
	//...
}

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

NOTE: A **final class** can not have **abstract methods** and an **abstract class** can not be declared **final**.

The Object Class

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object. That is, Object is a superclass of all other classes.
- This means that a reference variable of type Object can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.
- Object defines some methods, which means that they are available in every object.

The Object Class

	Method	Purpose
	Object clone( )	Creates a new object that is the same as the object being cloned.
	boolean equals(Object *object*)	Determines whether one object is equal to another.
	void finalize( )	Called before an unused object is recycled.
*	Class<?> getClass( )	Obtains the class of an object at run time.
	int hashCode( )	Returns the hash code associated with the invoking object.
*	void notify( )	Resumes execution of a thread waiting on the invoking object.
*	void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
	String toString( )	Returns a string that describes the object.
*	void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

References

R Reference for this topic

1. **Book-** Java: The Complete Reference, Tenth Edition: Herbert Schildt
2. **Web-** https://www.tutorialspoint.com/java/index.htm
3. **Web-** https://www.javatpoint.com/inheritance-in-java
4. **Web-** https://docs.oracle.com/javase/tutorial/java/IandI/index.html