

# Inheritance

## Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- Ex: The following program creates a superclass called A and a subclass called B

```
// 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();
    }
}
```

The output from this program is shown here:

```
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
```

- The subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.
- The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
    // body of class
}
```

## 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**.

```
// 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
    }
}
```

- This program will not compile because the reference to **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

## Constructors and Inheritance

- It is possible for both superclass and subclasses to have their own constructors.
- The constructor in the superclass constructs the superclass portion of the object and the constructor in the subclass constructs the subclass portion of the object
- When only the subclass defines a constructor, simply construct subclass object.
- The superclass portion of the object is constructed automatically using its default constructor.
- For example,

```

// Add a constructor to Triangle.

// A class for two-dimensional objects.
class TwoDShape {
    private double width; // these are
    private double height; // now private

    // Accessor methods for width and height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;

    // Constructor
    Triangle(String s, double w, double h) {
        setWidth(w);
        setHeight(h); // Initialize TwoDShape portion of object.

        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

class Shapes3 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle("filled", 4.0, 4.0);
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

- In this program, **Triangle's** constructor initializes the members of **TwoDClass**.

## Using super

- **super** has two general forms. The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

### 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);`

- Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.

```
// Add constructors to TwoDShape.
class TwoDShape {
    private double width;
    private double height;

    // Parameterized constructor.
    TwoDShape(double w, double h) { ← A constructor for TwoDShape.
        width = w;
        height = h;
    }

    // Accessor methods for width and height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;

    Triangle(String s, double w, double h) {
        super(w, h); // call superclass constructor
        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }
}
```

Use **super( )** to execute the **TwoDShape** constructor.

```

        void showStyle() {
            System.out.println("Triangle is " + style);
        }
    }

    class Shapes4 {
        public static void main(String[] args) {
            Triangle t1 = new Triangle("filled", 4.0, 4.0);
            Triangle t2 = new Triangle("outlined", 8.0, 12.0);

            System.out.println("Info for t1: ");
            t1.showStyle();
            t1.showDim();
            System.out.println("Area is " + t1.area());

            System.out.println();

            System.out.println("Info for t2: ");
            t2.showStyle();
            t2.showDim();
            System.out.println("Area is " + t2.area());
        }
    }
}

```

- In this program, Triangle() calls super() with the parameters w and h. this causes TwoDShape() constructor to be called, which then initializes width and height using these values.
- Following program includes both default and parameterized constructor

```

// Add more constructors to TwoDShape.
class TwoDShape {
    private double width;
    private double height;

    // A default constructor.
    TwoDShape() {
        width = height = 0.0;
    }

    // Parameterized constructor.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Construct object with equal width and height.
    TwoDShape(double x) {
        width = height = x;
    }
}

```

```

    // Accessor methods for width and height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

```

```

// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;

    // A default constructor.
    Triangle() {

```

```

        super();
        style = "none";
    }

    // Constructor
    Triangle(String s, double w, double h) {
        super(w, h); // call superclass constructor
        style = s;
    }

    // One argument constructor.
    Triangle(double x) {
        super(x); // call superclass constructor
        // default style to filled
        style = "filled";
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

```

Use **super()** to call the various forms of the **TwoDShape** constructor.

```

class Shapes5 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);
        Triangle t3 = new Triangle(4.0);

        t1 = t2;

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
        System.out.println();

        System.out.println("Info for t3: ");
        t3.showStyle();
        t3.showDim();
        System.out.println("Area is " + t3.area());
        System.out.println();
    }
}

```

Here is the output from this version.

```

Info for t1:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

Info for t2:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

Info for t3:
Triangle is filled
Width and height are 4.0 and 4.0
Area is 8.0

```

### Using super to Access Superclass Members

- 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.

```
// 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();
    }
}
```

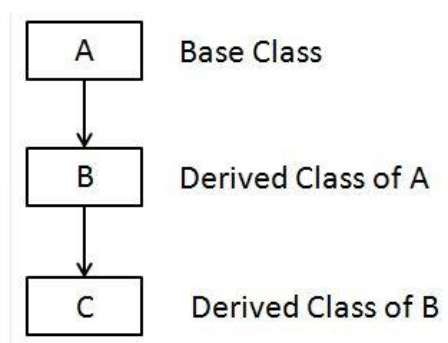
This program displays the following:

```
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

## Creating a Multilevel Hierarchy

- 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 aspects found in all of its superclasses. In this case, C inherits all aspects of B and A.





```

class A {
    int a;
    A()
    {
        a=10;
    }
    void funcA() {
        System.out.println("This is class A");
    }
}
class B extends A {
    int b;
    B(){
        super(); //calls its superclass constructor(class A)
        b=20;
    }
    void funcB() {
        System.out.println("This is class B");
    }
}
class C extends B {
    int c;
    C(){
        super(); //calls its superclass constructor(class B)
        c=30;
    }
    void funcC() {
        System.out.println("This is class C");
    }
}
public class Demo1 {
    public static void main(String args[]) {
        C obj = new C();
        obj.funcA();
        obj.funcB();
        obj.funcC();
        System.out.println("Values of a, b and c are: "+obj.a+" "+obj.b+" "+obj.c);
    }
}

```

#### OUTPUT:

```

This is class A
This is class B
This is class C
Values of a, b and c are: 10 20 30

```

- In the above program, class B inherits A, C inherits B. So, C includes all members of classes B and A.
- A class C object can be used to call methods and access variables defined by itself and its superclasses
- super() always refers to the constructor in the closest superclass.
- The **super( )** in **C** calls the constructor in **B**. The **super( )** in **B** calls the constructor in **A**.

#### When Constructors Are Called?

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass
- For example,

```

// 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();
    }
}

```

The output from this program is shown here:

```

Inside A's constructor
Inside B's constructor
Inside C's constructor

```

- The constructors are called in order of derivation

## A 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. i.e. a superclass reference can refer to a subclass object

Example:

```

class X{
    int a;
    X(int i){
        a=i;
    }
}

class Y extends X{
    int b;
    Y(int i, int j){
        super(i);
        b=j;
    }
}

```

```

public class SupSubRef {
    public static void main(String[] args) {
        X x=new X(10);
        Y y=new Y(5,6);
        X x2;
        x2 = x;
        System.out.println("value of a "+ x2.a);
        x2=y; //assign Y reference to X reference
        System.out.println("value of a"+ x2.a);
        //System.out.println("value of b "+ x2.b); //Error, X doesn't
        //have a b member
    }
}

```

- It is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, 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 why **x2** can't access **b** even when it refers to a **Y** object.

## 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 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.

Example:

```

// 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
    }
}

```

The output produced by this program is shown here:

k: 3

- When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.
- To access the superclass version of an overridden method, you can do so by using **super**.

```

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);
    }
}

```

If you substitute this version of **A** into the previous program, you will see the following output:

i and j: 1 2  
k: 3

Here, **super.show()** calls the superclass version of **show()**.

```
// 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);
    }
}

// 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;
    }

    // overload show()
    void show(String msg) {
        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
    }
}
```

The output produced by this program is shown here:

```
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. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

## Overridden Methods Support Polymorphism or Dynamic Method Dispatch

- Dynamic method dispatch (**late binding**) 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**
- 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.
- 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.

Example,

```
// 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
    }
}
```

The output from the program is shown here:

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

- This program creates one superclass called **A** and two subclasses of it, called **B** and **C**.
- Subclasses **B** and **C** override **callme()** declared in **A**. Inside the **main()** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.
- The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**.
- As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call.

## Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism
- 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.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.

## Applying Method Overriding

- The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object.
- It also defines a method called **area()** that computes the area of an object. The program derives two subclasses from **Figure**.
- The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
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);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```

        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());
    }
}

```

The output from the program is shown here:

```

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

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        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);
    }
}

```

## Using Abstract Classes

- Sometimes you will want to create a superclass 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.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method
- You can have methods that must be overridden by the subclass in order for the subclass to have any meaning. If we consider the class Triangle, it has no meaning if area( ) is not defined.
- In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the **abstract method**.
- Methods must 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
- To declare an abstract method, use this **general form**:  
         abstract type name(parameter-list);
- No method body is present
- Any class that contains one or more abstract methods must also be declared abstract.



- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator
- You cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must implement all of the abstract methods in the superclass

```
// 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() {
        System.out.println("B's implementation of callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

- No objects of class **A** are declared in the program. It is not possible to instantiate an abstract class.
- One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references
- It must be possible to create a reference to an abstract class so that it can be used to point to a subclass object

```

// 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);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    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());
    }
}

```

## Using final

- The keyword **final** has three uses.
- 1. **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.
  - The following fragment illustrates final

```

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!");
    }
}

```

- Because **meth()** is declared as **final**, it cannot be overridden in **B**.
- If you attempt to do so, a compile-time error will result

## 2. Using final to Prevent Inheritance

- You can prevent a class from being inherited by preceding the class declaration with **final**.
- 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.

```

-----
Here is an example of a final class:

final class A {
    // ...
}

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

```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

## 3. Using final with Data Members

- If a class variable's name is preceded with **final**, its value cannot be changed throughout the lifetime of the program.
- Initial value can be assigned.

```

class finalKey{
    final int i=10;
    void change(){
        i=30; //Error, as i is declared as final,
              //value of i cannot be changed.
    }
}

class name {
    public static void main(String[] args) {
        finalKey k=new finalKey();
        k.change();
    }
}

```

## 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 the following methods, which means that they are available in every object

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.

- The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others.
- The **equals( )** method compares the contents of two objects. It returns true if the objects are equivalent, and false otherwise.
- The **toString( )** method returns a string that contains a description of the object on which it is called

# Interfaces

## Interface Fundamentals

- An interface defines a set of methods that will be implemented by a class
- Interfaces are syntactically similar to classes, except that no method can include a body
- This means that an interface provides no implementation whatsoever of the methods it defines
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- To implement an interface, a class must provide bodies (implementations) for the methods declared by the interface.
- By providing the interface keyword, Java allows you to fully utilize the “**one interface, multiple methods**” aspect of polymorphism.

## Creating an Interface

- An interface is defined much like a class. This is the general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    // ...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

- When no access specifier is included, then default access results, and the 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. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface

```
interface Callback {  
    void callback(int param);  
}
```

- It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter

## Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
  - To implement an interface, include the **implements** clause in a class definition, and
  - Then create the methods defined by the interface
- The general form of a class that includes the **implements** clause looks like this:

```
class classname extends superclass implements interface{  
    // class-body  
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition

```
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {

        System.out.println("callback called with " + p);
    }
}
```

Notice that `callback()` is declared using the **public** access specifier.

### Using Interface References

- An interface declaration creates a new reference type.
- When a class implements an interface, it is adding that interface's type to its type
- As a result, an instance of a class that implements an interface is also an instance of that interface type.
- Because an interface defines a type, you can declare a reference variable of an interface type.
- In other words, you can create an interface reference variable.
- It can refer to any object that implements the interface.
- When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed.

The following example calls the **callback()** method via an interface reference variable:

```
interface Callback {
    void callback(int param);
}

class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {

        System.out.println("callback called with " + p);
    }
}

class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

The output of this program is shown here:

```
callback called with 42
```

### Implementing Multiple Interfaces

- A class can implement more than one interface.

- The class must implement all of the methods specified by each interface.
- For example

```
interface IfA{
    void doSomething();
}
interface IfB{
    void doSomethingElse();
}

//Implement both IfA and IfB
class MyClass implements IfA, IfB{
    public void doSomething(){
        System.out.println("Doing something");
    }
    public void doSomethingElse(){
        System.out.println("Doing something else");
    }
}
```

- In this example, **MyClass** specifies both **IfA** and **IfB** in its implements clause.

## Constants in Interfaces

- Interface can also include variables. Such variables are not instance variables. Instead, they are implicitly **public, final and static** and must be initialised.
- Hence, they are constants

```
// An interface that contains constants.
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Boundary Error";
}

class IConstD implements IConst {
    public static void main(String args[]) {
        int nums[] = new int[MAX];

        for(int i=MIN; i < 11; i++) {
            if(i >= MAX) System.out.println(ERRORMSG);
            else {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}
```

These are constants.

- IConst interface defines three constants. **MIN** and **MAX** are of type **int** and **ERRORMSG** is of type **String**.
- As required, they are given initial values.

## Interfaces can be Extended

- One interface can inherit another by use of the keyword **extends**
- When class implements an interface that inherits another interface. It must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B now includes meth1() and meth2() - it adds meth3().
interface B extends A {
    void meth3();
}

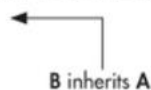
// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }

    public void meth2() {
        System.out.println("Implement meth2().");
    }

    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String args[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```



- In this example, interface A is extended by interface B
- MyClass implements B. This means that MyClass must implement all of the methods defined by both interfaces A and B.

## Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**
- An interface nested in another interface is implicitly **public**
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

```
// A nested interface example.

// This class contains a member interface.
class A {
```



```

// this is a nested interface
public interface NestedIF {
    boolean isNotNegative(int x);
}

// B implements the nested interface.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false : true;
    }
}

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

        // use a nested interface reference
        A.NestedIF nif = new B();

        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}

```

- **A** defines a member interface called **NestedIF** and that it is declared **public**.
- Next, **B** implements the nested interface by specifying  
implements **A.NestedIF**
- The name is fully qualified by the enclosing class name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**
- Since **B** implements only the nested interface **NestedIF**, it does not need to implement **doSomething( )** method.