Module-3

Inheritance and Interfaces

Inheritance Basics

Inheritance in java is implemented by using the keyword **extends.**

Inheritance: is incorporating the definition (features) of one class in another class using extends keyword

Example

```
Class A
{
//members of class A
}
Class B extends A
{
// members of class B and this also inherits all the members of class A
}
```

```
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
```

Inheritance

Inheritance allows the creation of **hierarchical classifications**.

Using inheritance, you can create a **general class** that defines **characteristics common** to a set of related items.

This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

In the terminology of Java,

a class that is inherited is called a **superclass**.

The class that does the inheriting is called a **subclass.**

Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

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));
 }
}
```

Syntax of inheritance

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

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

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

  void sum() {
    total = i + j; // ERROR, j is not accessible here
  }
}

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

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Total is " + subOb.total);
  }
}
```

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) { // pass object to constructor
    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;
  }
}
```

```
// 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 m) {
   width = w;
   height = h;
   depth = d;
   weight = m;
 }
}

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);
  }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

- BoxWeight inherits all of the characteristics of Box and adds to them the weight component.
- It is not necessary for BoxWeight to re-create all of the features found in Box. It can simply extend Box to meet its own purposes.
- 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.

Using super

In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been.

For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box.

Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members.

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

Example:

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

A complete implementation of BoxWeight using super()

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

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    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;
  }
}

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

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

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

This program generates the following 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
```

A Second Use for super

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 overcome name hiding

```
// 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
```

Creating a Multilevel Hierarchy

Till now we have been using simple class hierarchies that consist of only a superclass and a subclass.

However, you can build hierarchies that contain as many layers of inheritance as you like.

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 properties found in all of its superclasses. I

In this case, **C inherits all aspects of B and A.**

Example on Multilevel Inheritance

```
// Extend BoxWeight to include shipping
costs.
// Start with Box.
class Box {
 private double width;
 private double height;
 private double depth;
// construct clone of an object
 Box(Box ob) { // pass object to constructor
 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;
 }
}
```

```
// Add weight.
class BoxWeight extends Box {
 double weight; // weight of box
 // construct clone of an object
 BoxWeight(BoxWeight ob) {
// pass object to constructor
 super(ob);
 weight = ob.weight;
 }
 // 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;
 }
}
```

```
// Add shipping costs.
class Shipment extends BoxWeight {
 double cost;
 // construct clone of an object
 Shipment(Shipment ob) {
// pass object to constructor
 super(ob);
 cost = ob.cost;
 }
 // constructor when all parameters are
//specified
 Shipment(double w, double h, double d,
 double m, double c) {
 super(w, h, d, m);
// call superclass constructor
 cost = c;
 }
```

```
// default constructor
 Shipment() {
 super();
 cost = -1;
 }
 // constructor used when cube is created
 Shipment(double len, double m, double
c) {
 super(len, m);
 cost = c;
 }
}
```

```
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);
 }
}
```

The output of this program is shown here:

```
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 and in what order Constructors Are Executed in multilevel inheritance

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.

Example:

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

The output from this program is shown here:

```
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.

Example on 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
  }
}
```

The output produced by this program is shown here:

```
k: 3
```

Using the super to invoke the base class method

If you wish 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
```

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. For example, consider this modified version of the preceding example:

```
// 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
```

```
// 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
```

```
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;
  }
}
```

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.

Applying Method Overriding

A more practical example that uses method overriding

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

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
```

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

That is, sometimes you will want to create a 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.

One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

To declare an abstract method in an abstract class, use this general form:

abstract type name(parameter-list);

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.

Example-1

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

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

Example-2

```
// 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;
  }
}
```

Using final with Inheritance

The keyword final has three uses.

First, it can be used to create the constant.

The other two uses of final apply to inheritance. Both are examined here

- Using final to Prevent Overriding
- Using final to Prevent Inheritance

Using final to Prevent Overriding

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

Using final to Prevent Inheritance

```
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**.

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. 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.

Interfaces

Interface

using interface, you can specify what a class must do, but not how it does it.

Interfaces are syntactically similar to classes, but they **lack instance variables, and, as a general rule, their methods are declared without any body.**

Once it is defined, any number of classes can implement an interface. Also, **one class can implement any number of interfaces.**

interfaces can declare **constants (public static final fields),** but they cannot have instance variables

Defining an Interface

An interface is defined much like a class. This is a simplified 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;
}
```

Notice that the **methods that are declared have no bodies.** They end with a semicolon after the parameter list.

They are, **essentially, abstract methods.** Each class that includes such an interface must implement all of the methods.

Prior to JDK 8, an i**nterface could not define any implementation**r. Thus, prior to JDK 8, an interface could **define only "what," but not "how."**

JDK 8 changes this. Beginning with JDK 8, it is possible to **add a default implementation to an interface method.**

Thus, it is now possible for **interface to specify some behavior.**

As a general rule, you will **still often create and use interfaces in which no default method body exist**. For this reason, we will begin by discussing the interface in its traditional form.

The default method is described at the end of this chapter.

variables can be declared inside of interface declarations. They are **implicitly final and static,** meaning they cannot be changed by the implementing class.

They must also be initialized. All methods and variables are implicitly public.

Example:

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

Implementing Interfaces

After 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 required by the interface.

class classname [extends superclass] [implements interface [,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, t**he type signature of the implementing method must match exactly the type signature specified in the interface definition.**

Example

class Client implements Callback {

// Implement Callback's interface

public void callback(int p) {

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

}

}

It is both permissible and common for classes that implement interfaces to define additional members of their own.

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Accessing Implementations Through Interface References

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

```
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
```

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable has knowledge only of the methods declared by its interface declaration. Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

Another Example

```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
```

Now, try the following class:

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as abstract. For example:

```
abstract class Incomplete implements Callback {
 int a, b;
 void show() {
 System.out.println(a + " " + 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**. This differs from a **top-level interface**, which must either be declared as **public or use the default access** level, as previously described.

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.**

Thus, outside of the class or interface in which a nested interface is declared, **its name must be fully qualified.**

Nested Interface Example

```
// 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");
  }
}
```

Applying Interfaces -Stack program (Fixed Stack)

```
// Define an integer stack interface.
interface IntStack {
 void push(int item); // store an item
 int pop(); // retrieve an item
}

// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
 private int stck[];
 private int tos;
 // allocate and initialize stack
 FixedStack(int size) {
 stck = new int[size];
 tos = -1;
 }
```

```
// Push an item onto the stack
 public void push(int item) {
 if(tos==stck.length-1) // use length member
 System.out.println("Stack is full.");
 else
 stck[++tos] = item;
 }


// Pop an item from the stack
 public int pop() {
 if(tos < 0) {
 System.out.println("Stack underflow.");
 return 0;
 }
 else
 return stck[tos--];
 }
}
```

```
class IFTest {
 public static void main(String args[]) {
 FixedStack mystack1 = new FixedStack(5);
 FixedStack mystack2 = new FixedStack(8);
 // push some numbers onto the stack
 for(int i=0; i<5; i++) mystack1.push(i);
 for(int i=0; i<8; i++) mystack2.push(i);
 // pop those numbers off the stack
 System.out.println("Stack in mystack1:");
 for(int i=0; i<5; i++)
 System.out.println(mystack1.pop());
 System.out.println("Stack in mystack2:");
 for(int i=0; i<8; i++)
 System.out.println(mystack2.pop());
 }
}
```

// Implement a "growable" stack.

```
class DynStack implements IntStack {
 private int stck[];
 private int tos;
 // allocate and initialize stack
 DynStack(int size) {
 stck = new int[size];
 tos = -1;
 }
 // Push an item onto the stack
 public void push(int item) {
 // if stack is full, allocate a larger stack
 if(tos==stck.length-1) {
 int temp[] = new int[stck.length * 2]; // double size
 for(int i=0; i<stck.length; i++) temp[i] = stck[i];
```

```
stck = temp;
 stck[++tos] = item;
 }
 else
 stck[++tos] = item;
 }
 // Pop an item from the stack
 public int pop() {
 if(tos < 0) {
 System.out.println("Stack underflow.");
 return 0;
 }
 else
 return stck[tos--];
 }
}
```

```
class IFTest2 {
 public static void main(String args[]) {
 DynStack mystack1 = new DynStack(5);
 DynStack mystack2 = new DynStack(8);
 // these loops cause each stack to grow
 for(int i=0; i<12; i++) mystack1.push(i);
 for(int i=0; i<20; i++) mystack2.push(i);
 System.out.println("Stack in mystack1:");
 for(int i=0; i<12; i++)
 System.out.println(mystack1.pop());
 System.out.println("Stack in mystack2:");
 for(int i=0; i<20; i++)
 System.out.println(mystack2.pop());
 }
}
```

/* Create an interface variable and access stacks through it. */

```
class IFTest3 {
 public static void main(String args[]) {
 IntStack mystack; // create an interface reference variable
 DynStack ds = new DynStack(5);
 FixedStack fs = new FixedStack(8);
 mystack = ds; // load dynamic stack
 // push some numbers onto the stack
 for(int i=0; i<12; i++) mystack.push(i);
```

```
mystack = fs; // load fixed stack
 for(int i=0; i<8; i++) mystack.push(i);
 mystack = ds;
 System.out.println("Values in dynamic stack:");
 for(int i=0; i<12; i++)
 System.out.println(mystack.pop());
 mystack = fs;
 System.out.println("Values in fixed stack:");
 for(int i=0; i<8; i++)
 System.out.println(mystack.pop());
 }
}
```

Variables in Interfaces

You can use interfaces to **import shared constants into multiple classes by simply declaring an interface** that contains variables that are initialized to the desired values.

When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined constants or const declarations.)**

```
        return NO;              // 30%
      else if (prob < 60)
        return YES;             // 30%
      else if (prob < 75)
        return LATER;           // 15%
      else if (prob < 98)
        return SOON;            // 13%

      else
        return NEVER;           // 2%
    }
  }
```

```
  public static void main(String args[]) {
    Question q = new Question();

    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Example

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
```

```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.

When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Example

```
// 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 arg[]) {
    MyClass ob = new MyClass();

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

try removing the implementation for meth1( ) in MyClass. This will cause a compile-time error.

As stated earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Default Interface Methods

prior to JDK 8, an interface could not define any implementation. This meant that for all previous versions of Java, t**he methods specified by an interface were abstract,** containing no body.

This is the traditional form of an interface and is the type of interface that the preceding discussions have used.

The release of JDK 8 has changed this by adding a **new capability to interface called the default method.** A default method lets you define a **default implementation for an interface method.**

In other words, by use of a default method, i**t is now possible for an interface method to provide a body, rather than being abstract.**

During its development, the default method was also referred to as an **extension method**, and you will likely see both terms used.

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a class.

The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
}
```

```
// Implement MyIF.
class MyIFImp implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {

  public static void main(String args[]) {

    MyIFImp obj = new MyIFImp();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
  }
}
```

The output is shown here:

```
100
Default String
```

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**:

```
class MyIFImp2 implements MyIF {
  // Here, implementations for both getNumber( ) and getString( ) are provided.
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "This is a different string.";
  }
}
```

Now, when **getString( )** is called, a different string is returned.

default methods do offer a bit of what one would normally associate with the concept of multiple inheritance.

For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both.

Thus, to a limited extent, default methods do support multiple inheritance of behavior.

As you might guess, in such a situation, it is possible that a name conflict will occur.

First, in all cases,

A class implementation takes priority over an interface default implementation. Thus, if MyClass provides an override of the reset( ) default method, MyClass' version is used. This is the case even if MyClass implements both Alpha and Beta. In this case, both defaults are overridden by MyClass' implementation.

Multiple Inheritance Issues

Java does not support the multiple inheritance of classes.

Now that an interface can include default methods, **you might be wondering if an interface can provide a way around this restriction.**

The answer is, **essentially, no**. Recall that there is still a key difference between a class and an interface: **a class can maintain state information (especially through the use of instance variables), but an interface cannot.**

example

assume that two interfaces called **Alpha and Beta** are implemented by a class called **MyClass.**

What happens if both **Alpha and Beta provide a method called reset( )** for which both declare a **default implementation**? Is the version by **Alpha or the version by Beta used by MyClass**? Or, consider a situation in which **Beta extends Alpha**.

Which version of the default method is used? Or, **what if MyClass provides its own implementation of method?**

To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

Second,

In cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, t**hen an error will result**.

Continuing with the example, if MyClass implements both Alpha and Beta, but does not override reset( ), then an error will occur

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset( )** will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**. Its general form is shown here:

InterfaceName.super.*methodName( )*

For example, if **Beta** wants to refer to **Alpha**'s default for **reset( )**, it can use this statement:

```
Alpha.super.reset();
```

Example

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }

  // This is a static interface method.
  static int getDefaultNumber() {
    return 0;
  }
}
```

One last point:
static interface methods are not inherited by either an implementing class or a subinterface.

The **getDefaultNumber( )** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

Use static Methods in an Interface

JDK 8 added another new capability to interface: the ability to define one or more static methods.

Like static methods in a class, a static method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.

Instead, a static method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName