

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

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. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```
class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

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

        // assign BoxWeight reference to Box reference
        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. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that 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 **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

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. 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). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. 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. Each use is examined here.

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.

To see how **super()** is used, consider this improved version of the **BoxWeight** class:

```
// 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;
    }
}
```

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. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super()** was called with three arguments. Since constructors can be overloaded, **super()** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that

a box can be constructed. In each case, **super()** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// 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
```

Pay special attention to this constructor in **BoxWeight**:

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    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.

Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.

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. Consider this simple class hierarchy:

```
// 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. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

Up to this point, 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. 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**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```

// 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

```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

NOTE In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

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. The following program illustrates 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();
    }
}
```

The output from this program is shown here:

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

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

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. Consider the following:

```
// 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**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass' version. This allows all instance variables to be displayed.

```

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()**.

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
```

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**.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. 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.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a 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.

Here is an example that illustrates 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
}
}

```

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. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme()** method.

NOTE Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

Why Overridden Methods?

As stated earlier, 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. 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

Let's look at a more practical example that uses 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;
    }
}

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);
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

```

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. In this case, if an object is derived from **Figure**, then its area can be obtained by calling `area()`. The interface to this operation is the same no matter what type of figure is being used.

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. 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. This is the case with the class **Figure** used in the preceding example. The definition of `area()` is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. 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*.

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);
```

As you can see, 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. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

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

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

```
// 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());
    }
}

```

As the comment inside `main()` indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override `area()`. To prove this to yourself, try creating a subclass that does not override `area()`. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable `figref` is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. 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.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, 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**.

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 <i>object</i>)	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 <i>milliseconds</i>) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i>)	Waits on another thread of execution.

The methods `getClass()`, `notify()`, `notifyAll()`, and `wait()` are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: `equals()` and `toString()`. The `equals()` method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The `toString()` method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using `println()`. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for `getClass()`. This relates to Java's *generics* feature, which is described in Chapter 14.

CHAPTER

9

Packages and Interfaces

This chapter examines two of Java's most innovative features: packages and interfaces. *Packages* are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract an interface from its implementation. Using **interface**, you can specify a set of methods that can be implemented by one or more classes. In its traditional form, the **interface**, itself, does not actually define any implementation. Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Packages

In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name "Foobar" as a class name. Or, imagine the entire Internet community arguing over who first named a class "Espresso.") Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

```
package MyPack
```

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

```
C:\MyPrograms\Java\MyPack
```

then the class path to **MyPack** is

```
C:\MyPrograms\Java
```

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if (bal < 0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
    }
}
```



```

        for(int i=0; i<3; i++) current[i].show();
    }
}

```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1 Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
    }
}
```

```

        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **Derived.java**:

```

package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

// class only
// System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **SamePackage.java**:

```

package p1;

class SamePackage {
    SamePackage() {

        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}

```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```

package p2;

class Protection2 extends p1.Protection {
    Protection2() {

```

```

        System.out.println("derived other package constructor");

// class or package only
// System.out.println("n = " + n);

// class only
// System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **OtherPackage.java**:

```

package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");

// class or package only
// System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}

```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```

// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String args[]) {
        Protection obl = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}

```

The test file for **p2** is shown next:

```

// Demo package p2.
package p2;

```

```
// Instantiate the various classes in p2.
public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}
```

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1 [.pkg2].(classname | *);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

As shown in Table 9-1, when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if (bal < 0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

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

```

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}

```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, 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. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. 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 the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

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;
}

```

When no access modifier 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. *name* is the name of the interface, and can be any valid identifier. 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.

Before continuing an important point needs to be made. JDK 8 added a feature to **interface** that makes a significant change to its capabilities. Prior to JDK 8, an interface could not define any implementation whatsoever. This is the type of interface that the preceding simplified form shows, in which no method declaration supplies a body. 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. However, default methods constitute what is, in essence, a special-use feature, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no default methods 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.

As the general form shows, 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**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

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

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 required by the interface. The general form of a class that includes the **implements** clause looks like this:

```
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, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier:

```
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 modifier.

REMEMBER When you implement an interface method, it must be declared as **public**.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback()** and adds the method **nonIfaceMeth()**:

```

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

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar to using a superclass reference to access a subclass object, as described in Chapter 8.

CAUTION Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

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 **nonInterfaceMeth()** since it is defined by **Client** but not **Callback**.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// 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
```

As you can see, the version of **callback()** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

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);
    }
    //...
}

```

Here, the class **Incomplete** does not implement **callback()** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback()** or be declared **abstract** itself.

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.

Here is an example that demonstrates a nested interface:

```

// 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");
    }
}

```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

```
implements A.NestedIF
```

Notice that 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**, this is legal.

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push()** and **pop()** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

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

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// 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());
    }
}

```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```

// 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());
    }
}

```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push()** and **pop()** are resolved at run time rather than at compile time.

```

/* 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());
}
}

```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push()** and **pop()** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push()** and **pop()** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

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.) If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```

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)

```

```

        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%
    }
}

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;
        }
    }
}

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

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

```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble()** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are

defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

NOTE The technique of using an interface to define shared constants, as just described, is controversial. It is described here for completeness.

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. Following is an 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();  
    }  
}
```

As an experiment, you might want to 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

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the 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, it 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.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove()**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove()** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove()**, even though it was not needed. Today, a default implementation for **remove()** can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**. Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it

is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

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";
    }
}
```

MyIF declares two methods. The first, **getNumber()**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString()**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString()** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString()** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// 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

```

As you can see, the default implementation of **getString()** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString()**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString()** for some purpose beyond that supported by its default.)

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.

A More Practical Example

Although the preceding shows the mechanics of using default methods, it doesn't illustrate their usefulness in a more practical setting. To do this, let's once again return to the **IntStack** interface shown earlier in this chapter. For the sake of discussion, assume that **IntStack** is widely used and many programs rely on it. Further assume that we now want to add a method to **IntStack** that clears the stack, enabling the stack to be re-used. Thus, we want to evolve the **IntStack** interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion

of default methods, it is now easy to do. For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item

    // Because clear( ) has a default, it need not be
    // implemented by a preexisting class that uses IntStack.
    default void clear() {
        System.out.println("clear() not implemented.");
    }
}
```

Here, the default behavior of **clear()** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear()** because it was not defined by the earlier version of **IntStack**. However, **clear()** can be implemented by a new class that implements **IntStack**. Furthermore, **clear()** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

One other point: In real-world code, **clear()** would have thrown an exception, rather than displaying an error message. Exceptions are described in the next chapter. After working through that material, you might want to try modifying **clear()** so that its default implementation throws an **UnsupportedOperationException**.

Multiple Inheritance Issues

As explained earlier in this book, 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.

The preceding notwithstanding, 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.

For 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 the

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

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.

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, then 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();
```

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

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber()**. It returns zero.

```
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;  
}  
}
```

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

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

As mentioned, no implementation or instance of **MyIF** is required to call `getDefaultNumber()` because it is **static**.

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

Final Thoughts on Packages and Interfaces

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

CHAPTER

10

Exception Handling

This chapter examines Java's exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
    // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

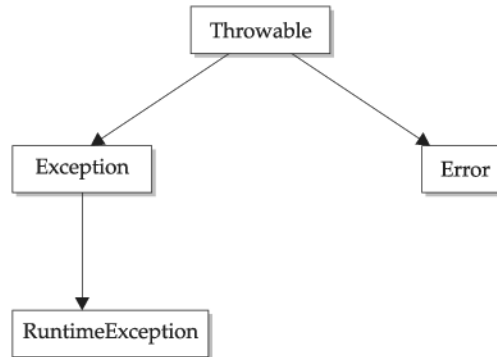
NOTE Beginning with JDK 7, there is another form of the **try** statement that supports *automatic resource management*. This form of **try**, called **try-with-resources**, is described in Chapter 13 in the context of managing files because files are some of the most commonly used resources.

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:



Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```

class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
  
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```

java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
  
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.