CHAPTER

3 Objects and Classes

THIS chapter begins the discussion of *object-oriented programming*. A fundamental component of object-oriented programming is the specification, implementation, and use of objects. In Chapter 2, we saw several examples of objects, including strings and files, that are part of the mandatory Java library. We also saw that these objects have an internal state that can be manipulated by applying the dot operator to select a method. In Java, the state and functionality of an object is given by defining a *class*. An object is then an instance of a class.

In this chapter, we will see:

- How Java uses the class to achieve *encapsulation* and *information hiding*
- How classes are implemented and automatically documented
- How classes are grouped into *packages*

3.1 What Is Object-oriented Programming?

Object-oriented programming emerged as the dominant paradigm of the mid-1990's. In this section we discuss some of the things that Java provides in the way

of object-oriented support and mention some of the principles of object-oriented programming.

Objects are entities that have structure and state. Each object defines operations that may access or manipulate that state.

At the heart of object-oriented programming is the *object*. An object is a data type that has structure and state. Each object defines operations that may access or manipulate that state. As we have already seen, in Java an object is distinguished from a primitive type, but this is a particular feature of Java rather than the object-oriented paradigm. In addition to performing general operations, we can do the following:

- Create new objects, possibly with initialization.
- Copy or test for equality.
- Perform I/O on these objects.

An object is an *atomic unit*: Its parts cannot be dissected by the general users of the object.

Also, we view the object as an *atomic unit* that the user ought not to dissect. Most of us would not even think of fiddling around with the bits that represent a floating-point number, and we would find it completely ridiculous to try to increment some floating-point object by altering its internal representation ourselves.

Information hiding makes implementation details, including components of an object, inaccessible.

The atomicity principle is known as *information hiding*. The user does not get direct access to the parts of the object or their implementations; they can be accessed only indirectly by methods supplied with the object. We can view each object as coming with the warning, "Do not open—no user-serviceable parts inside." In real life, most people who try to fix things that have such a warning wind up doing more harm than good. In this respect, programming mimics the real world. The grouping of data and the operations that apply to them to form an

aggregate, while hiding implementation details of the aggregate, is known as *encapsulation*.

Encapsulation is the grouping of data and the operations that apply to them to form an aggregate, while hiding the implementation of the aggregate.

An important goal of object-oriented programming is to support code reuse. Just as engineers use components over and over in their designs, programmers should be able to reuse objects rather than repeatedly reimplementing them. When we have an implementation of the exact object that we need to use, reuse is a simple matter. The challenge is to use an existing object when the object that is needed is not an exact match but is merely very similar.

Object-oriented languages provide several mechanisms to support this goal. One is the use of *generic* code. If the implementation is identical except for the basic type of the object, there is no need to completely rewrite code: Instead, we write the code generically so that it works for any type. For instance, the logic used to sort an array of objects is independent of the types of objects being sorted, so a generic algorithm could be used.

The *inheritance* mechanism allows us to extend the functionality of an object. In other words, we can create new types with restricted (or extended) properties of the original type. Inheritance goes a long way toward our goal of code reuse.

Another important object-oriented principle is *polymorphism*. A polymorphic reference type can reference objects of several different types. When methods are applied to the polymorphic type, the operation that is appropriate to the actual referenced object is automatically selected. In Java, this is implemented as part of inheritance. Polymorphism allows us to implement classes that share common logic. As is discussed in Chapter 4, this is illustrated in the Java libraries. The use

of inheritance to create these hierarchies distinguishes object-oriented programming from the simpler *object-based programming*.

In Java, generic algorithms are implemented as part of inheritance. Chapter 4 discusses inheritance and polymorphism. In this chapter, we describe how Java uses classes to achieve encapsulation and information hiding.

A *class* in Java consists of *fields* that store data and *methods* that are applied to instances of the class.

An *object* in Java is an instance of a class. A *class* is similar to a C structure or Pascal/Ada record, except that there are two important enhancements. First, members can be both functions and data, known as *methods* and *fields*, respectively. Second, the visibility of these members can be restricted. Because methods that manipulate the object's state are members of the class, they are accessed by the dot member operator, just like the fields. In object-oriented terminology, when we make a call to a method we are passing a message to the object. Types discussed in Chapter 2, such as `String`, `ArrayList`, `StringTokenizer`, and `FileReader`, are all classes implemented in the Java library.

3.2 A Simple Example

Functionality is supplied as additional members; these *methods* manipulate the object's state.

Recall that when you are designing the class, it is important to be able to hide internal details from the class user. This is done in two ways. First, the class can define functionality as class members, called *methods*. Some of these methods describe how an instance of the structure is created and initialized, how equality tests are performed, and how output is performed. Other methods would be specific to the particular structure. The idea is that the internal data fields that represent an object's state should not be manipulated directly by the class user but

instead should be manipulated only through use of the methods. This idea can be strengthened by hiding members from the user. To do this, we can specify that they be stored in a *private* section. The compiler will enforce the rule that members in the private section are inaccessible by methods that are not in the class of the object. Generally speaking, all data members should be private.

Public members are visible to nonclass routines; private members are not.

Figure 3.1 illustrates a class declaration for an `IntCell` object.[1] The declaration consists of two parts: public and private. *Public* members represents the portion that is visible to the user of the object. Since we expect to hide data, generally only methods and constants would be placed in the public section. In our example, we have methods that read from and write to the `IntCell` object. The private section contains the data: this is invisible to the user of the object. The `storedValue` member must be accessed through the publicly visible routines `read` and `write`; it cannot be accessed directly by `main`. Another way of viewing this is shown in Figure 3.2.

1. Public classes must be placed in files of the same name. Thus `IntCell` must be in file *IntCell.java*. We will discuss the meaning of `public` at line 5 when we talk about packages.

```
 1 // IntCell class
 2 //  int read( )         -->  Returns the stored value
 3 //  void write( int x ) -->  x is stored
 4
 5 public class IntCell
 6 {
 7       // Public methods
 8     public int read( )         { return storedValue; }
 9     public void write( int x ) { storedValue = x; }
10
11       // Private internal data representation
12     private int storedValue;
13 }
```

Figure 3.1 A complete declaration of an `IntCell` class

Figure 3.2 `IntCell` members: `read` and `write` are accessible, but `storedValue` is hidden

Members that are declared private are not visible to nonclass routines.

Figure 3.3 shows how `IntCell` objects are used. Since `read` and `write` are members of the `IntCell` class, they are accessed by using the dot member operator. The `storedValue` member could also be accessed by using the dot member operator, but since it is private, the access at line 14 would be illegal if it were not commented out.

A *field* is a member that stores data; a *method* is a member that performs an action.

Here is a summary of the terminology. The class defines *members*, which may be either *fields* (data) or *methods* (functions). The methods can act on the fields and may call other methods. The visibility modifier `public` means that the member is accessible to anyone via the dot operator. The visibility modifier

private means that the member is accessible only by other methods of this class. With no visibility modifier, we have package visible access, which is discussed in Section 3.6.4. There is also a fourth modifier known as *protected*, which is discussed in Chapter 4.

```
 1 // Exercise the IntCell class
 2
 3 public class TestIntCell
 4 {
 5     public static void main( String [ ] args )
 6     {
 7         IntCell m = new IntCell( );
 8
 9         m.write( 5 );
10         System.out.println( "Cell contents: " + m.read( ) );
11
12         // The next line would be illegal if uncommented
13         // because storedValue is a private member
14     //  m.storedValue = 0;
15     }
16 }
```

Figure 3.3 A simple test routine to show how `IntCell` objects are accessed

3.3 Javadoc

When designing a class, the *class specification* represents the class design and tells us what can be done to an object. The *implementation* represents the internals of how this is accomplished. As far as the class user is concerned, these internal details are not important. In many cases, the implementation represents proprietary information that the class designer may not wish to share. However, the specification must be shared; otherwise, the class is unusable.

The *class specification* describes what can be done to an object. The *implementation* represents the internals of how the specifications are met.

In many languages, the simultaneous sharing of the specification and hiding of the implementation is accomplished by placing the specification and implementation in separate source files. For instance, C++ has the class interface, which is placed in a `.h` file and a class implementation, which is in a `.cpp` file. In the `.h` file, the class interface restates the methods (by providing method headers) that are implemented by the class.

The *javadoc* program automatically generates documentation for classes.

Java takes a different approach. It is easy to see that a list of the methods in a class, with signatures and return types, can be automatically documented from the implementation. Java uses this idea: The program *javadoc*, which comes with all Java systems, can be run to automatically generate documentation for classes. The output of *javadoc* is a set of HTML files that can be viewed or printed with a browser.

The Java implementation file can also add *javadoc* comments that begin with the token starter `/**`. Those comments are automatically added in a uniform and consistent manner to the documentation produced by *javadoc*.

javadoc tags include `@author`, `@param`, `@return`, and `@throws`. They are used in *javadoc* comments.

There also are several special tags that can be included in the *javadoc* comments. Some of these are `@author`, `@param`, `@return`, and `@throws`. Figure 3.4 illustrates the use of the *javadoc* commenting features for the `IntCell` class. At line 3, the `@author` tag is used. This tag must precede the class definition. Line 10 illustrates the use of the `@return` tag and line 19, the `@param` tag. These tags must appear prior to a method declaration. The first token that follows the `@param` tag is the parameter name. The `@throws` tag is not shown, but it has the same syntax as `@param`.

Some of the output that results from running *javadoc* is shown in Figure 3.5 (on page 105). Run *javadoc* by supplying the name (including the `.java` extension) of the source file.

The output of *javadoc* is purely commentary, except for the method headers. The compiler does not check that these comments are implemented. Nonetheless, the importance of proper documentation of classes can never be overstated. *javadoc* makes the task of generating well-formatted documentation easier.

```
 1 /**
 2  * A class for simulating an integer memory cell
 3  * @author Mark A. Weiss
 4  */
 5
 6 public class IntCell
 7 {
 8     /**
 9      * Get the stored value.
10      * @return the stored value.
11      */
12     public int read( )
13     {
14         return storedValue;
15     }
16
17     /**
18      * Store a value.
19      * @param x the number to store.
20      */
21     public void write( int x )
22     {
23         storedValue = x;
24     }
25
26     private int storedValue;
27 }
```

Figure 3.4 `IntCell` declaration with *javadoc* comments

3.4 Basic Methods

Some methods are common to all classes. This section discusses *mutators*, *accessors*, and three special methods: the constructors, `toString`, and `equals`. Also discussed is `main`.

3.4.1 Constructors

A *constructor* tells how an object is declared and initialized.

As mentioned earlier, a basic property of objects is that they can be defined, possibly with initialization. In Java, the method that controls how an object is created and initialized is the *constructor*. Because of overloading, an object may define multiple constructors.

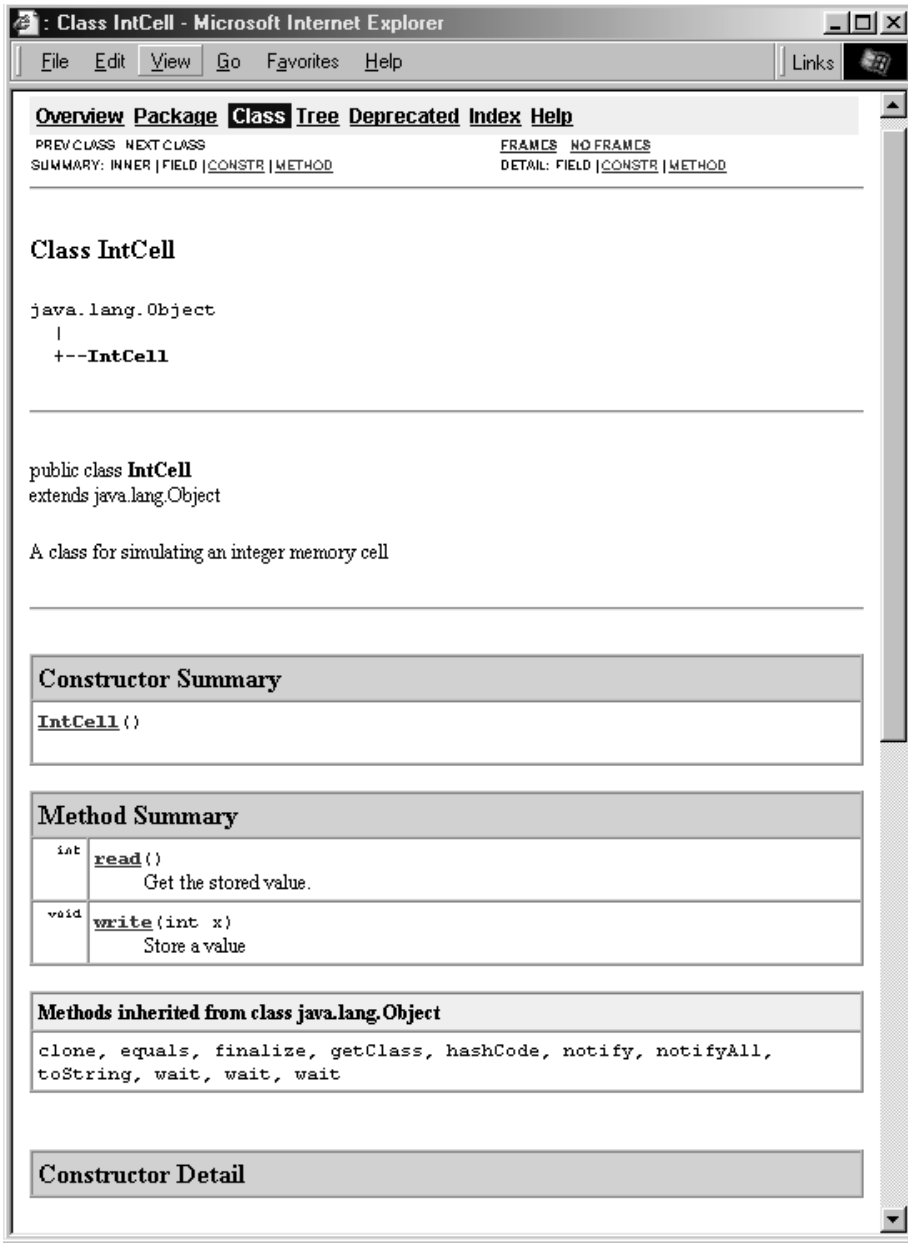

Figure 3.5 *javadoc* output for Figure 3.4 (partial output)

The default constructor is a member-by-member application of a default initialization.

If no constructor is provided, as in the case for the `IntCell` class in Figure 3.1, a default constructor is generated that initializes each data member using the normal defaults. This means that primitive fields are initialized to zero and reference fields are initialized to the null reference. (These defaults can be replaced by inline field initialization, which is executed prior to execution of constructor bodies.) Thus, in the case of `IntCell`, the `storedValue` component is 0.

To write a constructor, we provide a method that has the same name as the class and no return type. In Figure 3.6, there are two constructors: one begins at line 7 and the other at line 15. Using these constructors, we can construct `Date` objects in either of the following ways:

```
Date d1 = new Date( );
Date d2 = new Date( 4, 15, 2002 );
```

Note that once a constructor is written, a default zero-parameter constructor is no longer generated. If you want one, you have to write it. Thus the constructor at line 7 is required in order to allow construction of the object that `d1` references.

3.4.2 Mutators and Accessors

Class fields are typically declared private. Thus they cannot be directly accessed by nonclass routines. Sometimes, however, we would like to examine the value of a field. We may even want to change it.

One alternative to do this is to declare the fields public. This is typically a poor choice, however, because it violates information-hiding principles. Instead, we can provide methods to examine and change each field. A method that examines but does not change the state of an object is an *accessor*. A method that changes the state is a *mutator* (because it mutates the state of the object).

A method that examines but does not change the state of an object is an *accessor*. A method that changes the state is a *mutator*.

Special cases of accessors and mutators examine only a single field. These accessors typically have names beginning with `get`, such as `getMonth`, while these mutators typically have names beginning with `set`, such as `setMonth`.

The advantage of using a mutator is that the mutator can ensure that changes in the state of the object are consistent. Thus a mutator that changes the `day` field in a `Date` object can make sure that only legal dates result.

3.4.3 Output and `toString`

Typically, we want to output the state of an object using `print`. The way this is done is by writing the class method `toString`. This method returns a `String` suitable for output. As an example, Figure 3.6 shows a bare-bones implementation of the `toString` method for the `Date` class.

The `toString` method can be provided. It returns a `String` based on the object state.

```
 1 // Minimal Date class that illustrates some Java features
 2 // No error checks or javadoc comments
 3
 4 public class Date
 5 {
 6         // Zero-parameter constructor
 7     public Date( )
 8     {
 9         month = 1;
10         day = 1;
11         year = 2002;
12     }
13
14         // Three-parameter constructor
15     public Date( int theMonth, int theDay, int theYear )
16     {
17         month = theMonth;
18         day   = theDay;
19         year  = theYear;
20     }
21
22         // Return true if two equal values
23     public boolean equals( Object rhs )
24     {
25         if( ! ( rhs instanceof Date ) )
26             return false;
27         Date rhDate = ( Date ) rhs;
28         return rhDate.month == month && rhDate.day == day &&
29                rhDate.year == year;
30     }
31
32         // Conversion to String
33     public String toString( )
34     {
35         return month + "/" + day + "/" + year;
36     }
37
38         // Fields
39     private int month;
40     private int day;
41     private int year;
42 }
```

Figure 3.6 A minimal `Date` class that illustrates constructors and the `equals` and `toString` methods

3.4.4 equals

The `equals` method is used to test if two objects represent the same value. The signature is always

The `equals` method can be provided to test if two references are referring to the same value.

```
public boolean equals( Object rhs )
```

Notice that the parameter is of reference type `Object`, rather than the class type (the reason for this is discussed in Chapter 4). Typically, the `equals` method for class `ClassName` is implemented to return `true` only if `rhs` is an instance of `ClassName`, and after the conversion to `ClassName`, all the primitive fields are equal (via `==`) and all the reference fields are equal (via member-by-member application of `equals`).

An example of how `equals` is implemented is provided in Figure 3.6 for the `Date` class. The `instanceof` operator is discussed in Section 3.5.3.

The parameter to `equals` is of type `Object`.

3.4.5 main

When the *java* command is issued to start the interpreter, the `main` method in the class file referenced by the *java* command is called. Thus each class can have its own `main` function, without problem. This makes it easy to test the basic functionality of individual classes. However, although functionality can be tested, placing `main` in the class gives `main` more visibility than would be allowed in general. Thus calls from `main` to nonpublic methods will succeed in the test, even though they will be illegal in a more general setting.

3.4.6 static Fields and Methods

A *static method* is a method that does not need a controlling object.

A *static method* is a method that does not need a controlling object, and thus is typically called by supplying a class name instead of the controlling object. The most common static method is `main`. Other static methods are found in the `Integer` and `Math` classes. Examples are the methods `Integer.parseInt`, `Math.sin`, and `Math.max`. Access to a static method uses the same visibility rules as do static fields. These methods mimic global functions found in non-object-oriented languages.

Recall from Chapter 1 that some fields of the class use the modifier `static`. Specifically, in conjunction with the keyword `final`, we have constants. Without the word `final`, we have *static fields*, which have another meaning, and another use of static methods, both of which are discussed in Section 3.5.5.

3.5 Additional Constructs

Three additional keywords are `this`, `instanceof`, and `static`. `this` has several uses in Java; two are discussed in this section. `instanceof` also has several general uses; it is used here to ensure that a type-conversion can succeed. Likewise, `static` has several uses. We have already discussed *static methods*. This section covers the *static field* and *static initializer.*

3.5.1 The `this` Reference

`this` is a reference to the current object. It can be used to send the current object, as a unit, to some other method.

The first use of `this` is as a reference to the current object. Think of the `this` reference as a homing device that, at any instant in time, tells you where you are. An important use of the `this` reference is in handling the special case of self-assignment. An example of this is a program that copies one file to another. A normal algorithm begins by truncating the target file to zero length. If no check is performed to make sure the source and target file are indeed different, then the source file will be truncated — hardly a desirable feature. When dealing with two objects, one of which is written and one of which is read, we first should check for this special case, which is known as *aliasing*.

Aliasing is a special case that occurs when the same object appears in more than one role.

For a second example, suppose we have a class `Account` that has a method `finalTransfer`. This method moves all the money from one account into another. In principle, this is an easy routine to write:

```
// Transfer all money from rhs to current account
public void finalTransfer( Account rhs )
{
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```

However, consider the result:

```
    Account account1;
    Account account2;
...
    account2 = account1;
    account1.finalTransfer( account2 );
```

Since we are transferring money between the same account, there should be no change in the account. However, the last statement in `finalTransfer` assures that the account will be empty. One way to avoid this is to use an alias test:

```
// Transfer all money from rhs to current account
public void finalTransfer( Account rhs )
{
    if( this == rhs )    // Alias test
        return;
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```

3.5.2 The `this` Shorthand for Constructors

`this` can be used to make a call to another constructor in the same class.

Many classes have multiple constructors that behave similarly. We can use `this` inside a constructor to call one of the other class constructors. An alternative to the zero-parameter `Date` constructor in Figure 3.6 would be

```
public Date( )
{
    this( 1, 1, 2002 ); // Call the 3-param constructor
}
```

More complicated uses are possible, but the call to `this` must be the first statement in the constructor; thereafter more statements may follow.

3.5.3 The `instanceof` Operator

The `instanceof` operator performs a run-time test. The result of

```
exp instanceof ClassName
```

is `true` if `exp` is an instance of `ClassName`, and `false` otherwise. If `exp` is `null`, the result is always `false`. The `instanceof` operator is typically used prior to performing a type conversion and is `true` if the type conversion can succeed.

The `instanceof` operator is used to test if an expression is an instance of some class.

3.5.4 Instance Members vs. Static Members

Fields and methods declared with the keyword `static` are *static members*. If they are declared without the keyword `static`, we will refer to them as *instance members*. The next subsection explains the distinction between instance and static members.

Instance members are fields or methods declared without the `static` modifier.

3.5.5 Static Fields and Methods

Static fields are used when we have a variable that all the members of some class need to share. Typically, this is a symbolic constant, but it need not be. When a class variable is declared `static`, only one instance of the variable is ever cre-

Static fields are essentially global variables with class scope.

ated. It is not part of any instance of the class. Instead, it behaves like a single global variable but with the scope of the class. In other words, in the declaration

```
public class Sample
{
    private int x;
    private static int y;
}
```

each `Sample` object stores its own `x`, but there is only one shared `y`.

A common use of a static field is as a constant. For instance, the class `Integer` defines the field `MAX_VALUE` as

```
public static final int MAX_VALUE = 2147483647;
```

If this constant was not a static field, then each instance of an `Integer` would have a data field named `MAX_VALUE`, thus wasting space and initialization time. Instead, there is only a single variable named `MAX_VALUE`. It can be accessed by any of the `Integer` methods by using the identifier `MAX_VALUE`. It can also be accessed via an `Integer` object `obj` using `obj.MAX_VALUE`, as would any field. Note that this is allowed only because `MAX_VALUE` is public. Finally, `MAX_VALUE` can be accessed by using the class name as `Integer.MAX_VALUE` (again allowable because it is public). This would not be allowed for a nonstatic field. The last form is preferable, because it communicates to the reader that the field is indeed a static field. Another example of a static field is the constant `Math.PI`.

Even without the `final` qualifier, static fields are still useful. Figure 3.7 illustrates a typical example. Here we want to construct `Ticket` objects, giving

each ticket a unique serial number. In order to do this, we have to have some way of keeping track of all the previously used serial numbers; this is clearly shared data, and not part of any one `Ticket` object.

A *static field* is shared by all (possibly zero) instances of the class.

Each `Ticket` object will have its instance member `serialNumber`; this is instance data because each instance of `Ticket` has its own `serialNumber` field. All `Ticket` objects will share the variable `ticketCount`, which denotes the number of `Ticket` objects that have been created. This variable is part of the class, rather than object-specific, so it is declared `static`. There is only one `ticketCount`, whether there is 1 `Ticket`, 10 `Ticket`s, or even no `Ticket` objects. The last point — that the static data exists even before any instances of the class are created — is important, because it means the static data cannot be initialized in constructors. One way of doing the initialization is inline, when the field is declared. More complex initialization is described in Section 3.5.6.

```
 1 class Ticket
 2 {
 3     public Ticket( )
 4     {
 5         System.out.println( "Calling constructor" );
 6         serialNumber = ++ticketCount;
 7     }
 8
 9     public int getSerial( )
10     {
11         return serialNumber;
12     }
13
14     public String toString( )
15     {
16         return "Ticket #" + getSerial( );
17     }
18
19     public static int getTicketCount( )
20     {
21         return ticketCount;
22     }
23
24     private int serialNumber;
25     private static int ticketCount = 0;
26 }
27
28 class TestTicket
29 {
30     public static void main( String[] args )
31     {
32         Ticket t1;
33         Ticket t2;
34
35         System.out.println( "Ticket count is " +
36                             Ticket.getTicketCount( ) );
37         t1 = new Ticket( );
38         t2 = new Ticket( );
39
40         System.out.println( "Ticket count is " +
41                              Ticket.getTicketCount( ) );
42
43         System.out.println( t1.getSerial( ) );
44         System.out.println( t2.getSerial( ) );
45     }
46 }
```

Figure 3.7 Ticket class: an example of static fields and methods

In Figure 3.7, we can now see that construction of `Ticket` objects is done by using `ticketCount` as the serial number, and incrementing `ticketCount`. We also provide a static method, `getTicketCount`, that returns the number of tickets. Because it is static, it can be invoked without providing an object reference, as shown on lines 36 and 41. The call on line 41 could have been made using either `t1` or `t2`, though many argue that invoking a static method using an object reference is poor style, and we would never do so in this text. However, it is significant that the call on line 36 clearly could not be made through an object reference, since at this point there are no valid `Ticket` objects. This is why it is important for `getTicketCount` to be declared as a static method; if it was declared as an instance method, it could only be called through an object reference.

A `static` method that has no implicit `this` reference, and can be invoked without an object reference.

When a method is declared as a static method, there is no implicit `this` reference. As such, it cannot access instance data or call instance methods, without providing an object reference. In other words, from inside `getTicketCount`, unqualified access of `serialNumber` would imply `this.serialNumber`, but since there is no `this`, the compiler will issue an error message. Thus, a non-static field, which is part of each instance of the class, can be accessed by a static class method only if a controlling object is provided.

3.5.6 Static Initializers

A *static initializer* is a block of code that is used to initialize static fields.

Static fields are initialized when the class is loaded. Occasionally, we need a complex initialization. For instance, suppose we need a static array that stores the square roots of the first 100 integers. It would be best to have these values computed automatically. One possibility is to provide a static method and require the programmer to call it prior to using the array.

An alternative is the *static initializer.* An example is shown in Figure 3.8. There, the static initializer extends from lines 5 to 9. The simplest use of the static initializer places initialization code for the static fields in a block that is preceded by the keyword `static`. The static initializer must follow the declaration of the static member.

```
 1 public class Squares
 2 {
 3     private static double squareRoots[ ] = new double[ 100 ];
 4
 5     static
 6     {
 7         for( int i = 0; i < squareRoots.length; i++ )
 8             squareRoots[ i ] = Math.sqrt( ( double ) i );
 9     }
10     // Rest of class
11 }
```

Figure 3.8 Example of a static initializer

```
1  /**
2   * The StringArrayList implements a growable array of String.
3   * Insertions are always done at the end.
4   */
5  public class StringArrayList
6  {
7      /**
8       * Returns the number of items in this collection.
9       * @return the number of items in this collection.
10      */
11     public int size( )
12     {
13         return theSize;
14     }
15
16     /**
17      * Returns the item at position idx.
18      * @param idx the index to search in.
19      * @throws ArrayIndexOutOfBoundsException if index is bad.
20      */
21     public String get( int idx )
22     {
23         if( idx < 0 || idx >= size( ) )
24             throw new ArrayIndexOutOfBoundsException( );
25         return theItems[ idx ];
26     }
27
28     /**
29      * Adds an item to this collection, at the end.
30      * @param x any object.
31      * @return true (as per java.util.ArrayList).
32      */
33     public boolean add( String x )
34     {
35         if( theItems.length == size( ) )
36         {
37             String [ ] old = theItems;
38             theItems = new String[ theItems.length * 2 + 1 ];
39             for( int i = 0; i < size( ); i++ )
40                 theItems[ i ] = old[ i ];
41         }
42
43         theItems[ theSize++ ] = x;
44         return true;
45     }
46
47     private static final int INIT_CAPACITY = 10;
48
49     private int         theSize = 0;
50     private String [ ] theItems = new String[ INIT_CAPACITY ];
51 }
```

Figure 3.9 Simplified `StringArrayList`, with `add`, `get`, and `size`.

3.6 Packages

A *package* is used to organize a collection of classes.

Packages are used to organize similar classes. Each package consists of a set of classes. Two classes in the same package have slightly fewer visibility restrictions among themselves than they would if they were in different packages.

Java provides several predefined packages, including `java.applet`, `java.awt`, `java.io`, `java.lang`, and `java.util`. The `java.lang` package includes the classes `Integer`, `Math`, `String`, and `System`, among others. Some of the classes in the `java.util` package are `Date`, `Random`, and `StringTokenizer`. `java.io` is used for I/O and includes the various stream classes seen in Section 2.6.

Class `C` in package `p` is specified as `p.C`. For instance, we can have a `Date` object constructed with the current time and date as an initial state using

```
java.util.Date today = new java.util.Date( );
```

Note that by including a package name, we avoid conflicts with identically named classes in other packages (such as our own `Date` class). Also, observe the typical naming convention: class names are capitalized and package names are not.

3.6.1 The `import` Directive

The *`import` directive* is used to provide a shorthand for a fully qualified class name.

Using a full package and class name can be burdensome. To avoid this, use the *`import` directive*. There are two forms of the `import` directive:

```
import packageName.ClassName;
import packageName.*;
```

In the first form, `ClassName` may be used as a shorthand for a fully qualified class name. In the second, all classes in a package may be abbreviated with the corresponding class name.

For example, with these `import` directives,

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

we may use

```
Date today = new Date( );
FileReader theFile = new FileReader( name );
```

Careless use of the `import` directive can introduce naming conflicts.

Using the `import` directive saves typing. And since the most typing is saved by using the second form, you will see that form used often. There are two disadvantages to `import` directives. First, the shorthand makes it hard to tell, by reading the code, which class is being used when there are a host of `import` directives. Also, the second form may allow shorthands for unintended classes and introduce naming conflicts that will need to be resolved by fully qualified class names.

Suppose we use

```
import java.util.*;   // Library package
import weiss.util.*;  // User-defined package
```

with the intention of importing the `java.util.Random` class and a package that we have written ourselves. Then, if we have our own `Random` class in `weiss.util`, the `import` directive will generate a conflict with `weiss.util.Random` and will need to be fully qualified. Furthermore, if we

are using a class in one of these packages, by reading the code we will not know whether it originated from the library package or our own package. We would have avoided these problems if we had used the first form:

```
import java.util.Random;
```

and for this reason, we use the first form only in the text, and avoid "wild card" import directives.

`java.lang.*` is automatically imported.

The `import` directives must appear prior to the beginning of a class declaration. We saw an example of this in Figure 2.16. Also, the entire package `java.lang` is automatically imported. This is why we may use shorthands such as `Math.max`, `Integer.parseInt`, `System.out`, and so on.

3.6.2 The `package` Statement

The *`package` statement* indicates that a class is part of a package. It must precede the class definition.

To indicate that a class is part of a package, we must do two things. First, we must include the *`package` statement* as the first line, prior to the class definition. Second, we must place the code in an appropriate subdirectory.

In this text, we use the two packages shown in Figure 3.10. Other programs, including test programs and the application programs in Part III of the text, are stand-alone classes and not part of a package.

An example of how the `package` statement is used is shown in Figure 3.11. Here, we have the static method `longPause` that simply sleeps for a billion milliseconds (approximately two weeks). This method is useful because when some integrated environments run console applications from inside their environments,

they close the output console as soon as the program terminates. This can make it hard to see the output. `longPause` keeps the console from closing in this situation.

Package	Use
`weiss.util`	A reimplementation of a subset of the `java.util` package containing various data structures.
`weiss.nonstandard`	Various data structures, in a simplified form, using nonstandard conventions that are different from `java.util`.

Figure 3.10 Packages defined in this text

```
 1 package weiss.nonstandard;
 2
 3 public class Exiting
 4 {
 5     // Suspend current program for a long time
 6     public static void longPause( )
 7     {
 8         try
 9           { Thread.sleep( 1000000000 ); }
10         catch( InterruptedException e ) { }
11     }
12 }
```

Figure 3.11 A class `Exiting` with a single static method, which is part of the package `weiss.nonstandard`

3.6.3 The CLASSPATH Environment Variable

The CLASSPATH variable specifies files and directories that should be searched to find classes.

Packages are searched for in locations that are named in the CLASSPATH variable. What does this mean? Here are possible settings for CLASSPATH, first for a Windows 95 system and second for a Unix system:

```
SET CLASSPATH=.;C:\bookcode\
setenv CLASSPATH .:$HOME/bookcode/
```

In both cases, the CLASSPATH variable lists directories (or jar files[2]) that contain the package's class files. For instance, if your CLASSPATH is corrupted, you will not be able to run even the most trivial program because the current directory will not be found.

A class in package p must be in a directory p that will be found by searching through the CLASSPATH list.

A class in package p must be in a directory p that will be found by searching through the CLASSPATH list; each . in the package name represents a subdirectory. Starting with Java 1.2, the current directory (directory .) is always scanned if CLASSPATH is not set at all, so if you are working from a single main directory, you can simply create subdirectories in it and not set CLASSPATH. Most likely, however, you'll want to create a separate Java subdirectory and then create package subdirectories in there. You would then augment the CLASSPATH variable to include . and the Java subdirectory. This was done in the previous Unix declaration when we added $HOME/bookcode/ to the CLASSPATH. Inside the bookcode directory, you create a subdirectory named weiss, and in that subdi-

2. A jar file is basically a compressed archive (like a zip file) with extra files containing Java specific information. The *jar* tool, supplied with the JDK can be used to create and expand jar files.

rectory, `util` and `nonstandard`. In the `nonstandard` subdirectory, you place the code for the `Exiting` class.

An application, written in any directory at all, can then use the `longPause` method either by issuing

```
weiss.nonstandard.Exiting.longPause( );
```

or, simply using `Exiting.longPause`, if an appropriate `import` directive is provided.

3.6.4 Package Visibility Rules

Fields with no visibility modifiers are *package visibile*, meaning that they are visible only to other classes in the same package.

Packages have several important visibility rules. First, if no visibility modifier is specified for a field, then the field is *package visible*. This means that it is visible only to other classes in the same package. This is more visible than private (which is invisible even to other classes in the same package) but less visible than public (which is visible to nonpackage classes, too).

Non-public classes are visible only to other classes in the same package.

Second, only public classes of a package may be used outside the package. That is why we have often used the `public` qualifier prior to `class`. Classes may not be declared `private`.[3] Package visible access extends to classes, too. If a class is not declared `public`, then it may be accessed by other classes in the same package only; this is a *package visible class*. In Part IV, we will see that package visible classes can be used without violating information-hiding princi-

3. This applies to top-level classes shown so far; later we will see nested and inner classes, which may be declared private.

ples. Thus there are some cases in which package visible classes can be very useful.

All classes that are not part of a package but are reachable through the `CLASSPATH` variable are considered part of the same default package. As a result, package visible applies between all of them. This is why visibility is not affected if the `public` modifier is omitted from nonpackage classes. However, this is poor use of package visible member access. We use it only to place several classes in one file, because that tends to make examining and printing the examples easier. Since a public class must be in a file of the same name, there can be only one public class per file.

3.7 A Design Pattern: Composite (Pair)

A *design pattern* describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts.

Although software design and programming are often difficult challenges, many experienced software engineers will argue that software engineering really has only a relatively small set of basic problems. Perhaps this is an understatement, but it is true that many basic problems are seen over and over in software projects. Software engineers who are familiar with these problems, and in particular, the efforts of other programmers in solving these problems, have the advantage of not needing to "reinvent the wheel."

The idea of a design pattern is to document a problem and its solution so that others can take advantage of the collective experience of the entire software engineering community. Writing a pattern is much like writing a recipe for a cook-

book; many common patterns have been written and rather than expending energy reinventing the wheel, these patterns can be used to write better programs. Thus a *design pattern* describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts.

Throughout the text we will discuss several problems that often arise in a design, and a typical solution that is employed to solve the problem. We start with the following simple problem.

A common design pattern is to return two objects as a *pair*.

In most languages, a function can return only a single object. What do we do if we need to return two or more things? The easiest way to do this is to combine the objects into a single object using either an array or a class. The most common situation in which multiple objects need to be returned is the case of two objects. So a common design pattern is to return the two objects as a *pair*. This is the *Composite pattern*.

Pairs are useful for implementing key-value pairs in maps and dictionaries.

In addition to the situation described above, pairs are useful for implementing maps and dictionaries. In both these abstractions, we maintain key-value pairs: the pairs are added into the map or dictionary, and then we search for a key, returning its value. One common way to implement a map as to use a set. In a set, we have a collection of items, and search for a match. If the items are pairs, and the match criterion is based exclusively on the key component of the pair, then it is easy to write an adapter class that constructs a map on the basis of a set. We will see this idea explored in more detail in Chapter 19.

Summary

This chapter described the Java class and package constructs. The class is the Java mechanism that is used to create new reference types; the package is used to group related classes. For each class, we can

- define the construction of objects,
- provide for information hiding and atomicity, and
- define methods to manipulate the objects.

The class consists of two parts: the specification and the implementation. The specification tells the user of the class what the class does; the implementation does it. The implementation frequently contains proprietary code and in some cases is distributed only as a `.class` file. The specification, however, is public knowledge. In Java, a specification that lists the class methods can be generated from the implementation by using *javadoc*.

Information hiding can be enforced by using the `private` directive. Initialization of objects is controlled by the constructors, and the components of the object can be examined and changed by accessor and mutator methods, respectively. Figure 3.9 illustrates many of these concepts, as applied to simplified version of `ArrayList`. This class, `StringArrayList`, supports `add`, `get`, and `size`. A more complete version that includes `set`, `remove`, and `clear`, is in the online code.

The features discussed in this chapter implement the fundamental aspects of object-based programming. The next chapter discusses inheritance, which is central to object-oriented programming.

Objects of the Game

accessor A method that examines an object but does not change its state. (107)

aliasing A special case that occurs when the same object appears in more than one role. (111)

atomic unit In reference to an object, its parts cannot be dissected by the general users of the object. (96)

class Consists of fields and methods that are applied to instances of the class. (98)

class specification Describes the functionality, but not the implementation. (101)

`CLASSPATH` variable Specifies directories and files that should be searched to find classes. (124)

Composite The pattern in which we store two or more objects in one entity. (127)

constructor Tells how an object is declared and initialized. The default constructor is a member-by-member default initialization, with primitive fields initialized to zero and reference fields initialized to `null`. (104)

design pattern Describes a problem that occurs over and over in software engineering, and then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts. (126)

encapsulation The grouping of data and the operations that apply to them to form an aggregate while hiding the implementation of the aggregate. (97)

`equals` method Can be implemented to test if two objects represent the same value. The formal parameter is always of type `Object`. (109)

field A class member that stores data. (100)

implementation Represents the internals of how the specifications are met. As far as the class user is concerned, the implementation is not important. (101)

import directive Used to provide a shorthand for a fully qualified class name. (120)

information hiding Makes implementation details, including components of an object, inaccessible. (96)

instance members Members declared without the static modifier. (113)

`instanceof` operator Tests if an expression is an instance of a class. (113)

javadoc Automatically generates documentation for classes. (102)

***javadoc* tag** Includes `@author`, `@param`, `@return`, and `@exception`. Used inside of *javadoc* comments. (102)

method A function supplied as a member that, if not static, operates on an instance of the class. (98)

mutator A method that changes the state of the object. (107)

object An entity that has structure and state and defines operations that may access or manipulate that state. An instance of a class. (96)

object-based programming Uses the encapsulation and information hiding features of objects but does not use inheritance. (97)

object-oriented programming Distinguished from object-based programming by the use of inheritance to form hierarchies of classes. (97)

package Used to organize a collection of classes. (120)

package statement Indicates that a class is a member of a package. Must precede the class definition. (122)

package visible access Members that have no visibility modifiers are only accessible to methods in classes in the same package. (125)

package visible class A class that is not public and is accessible only to other classes in the same package. (125)

Pair The composite pattern with two objects. (127)

private A member that is not visible to nonclass methods. (100)

public A member that is visible to nonclass methods. (100)

static field A field that is shared by all instances of a class. (117)

static initializer A block of code that is used to initialize static fields. (118)

static method A method that has no implicit `this` reference and thus can be invoked without a controlling object reference. (117)

`this` constructor call Used to make a call to another constructor in the same class. (112)

`this` reference A reference to the current object. It can be used to send the current object, as a unit, to some other method. (111)

`toString` method Returns a `String` based on the object state. (107)

Common Errors

1. Private members cannot be accessed outside of the class. Remember that, by default, class members are package visible: They are visible only within the package.
2. Use `public class` instead of `class` unless you are writing a throw-away helper class.
3. The formal parameter to `equals` must be of type `Object`. Otherwise, although the program will compile, there are cases in which a default `equals` (that always returns `false`) will be used instead.
4. Static methods cannot access nonstatic members without a controlling object.
5. Classes that are part of a package must be placed in an identically named directory that is reachable from the `CLASSPATH`.
6. `this` is a final reference and may not be altered.

On the Internet

Following are the files that are available:

TestIntCell.java	Contains a `main` that tests `IntCell`, shown in Figure 3.3.
IntCell.java	Contains the `IntCell` class, shown in Figure 3.4. The output of *javadoc* can also be found as **IntCell.html**.
Date.java	Contains the `Date` class, shown in Figure 3.6.
Exiting.java	Contains the `longPause` method, shown in Figure 3.11. Found in package `weiss.util`.
Ticket.java	Contains the `Ticket` static member example in Figure 3.7.
Squares.java	Contains the static initializer sample code in Figure 3.8.
StringArrayList.java	Contains a more complete version of `StringArrayList` code in Figure 3.9.

ReadStringsWithStringArrayList.java Contains a test program for `StringArrayList`.

Exercises

In Short

3.1. What is information hiding? What is encapsulation? How does Java support these concepts?

3.2. Explain the public and private sections of the class.

3.3. Describe the role of the constructor.

3.4. If a class provides no constructor, what is the result?

3.5. Explain the uses of `this` in Java.

3.6. What is package visible access?

3.7. For a class `ClassName`, how is output performed?

3.8. Give the two types of import directive forms that allow `longPause` to be used without providing the `weiss.util` package name.

3.9. What is a design pattern?

3.10. For the code in Figure 3.12, which resides entirely in one file,

a. Line 17 is illegal, even though line 18 is legal. Explain why.

b. Which of lines 20 to 24 are legal and which are not? Explain why.

```
 1 class Person
 2 {
 3     public static final int NO_SSN = -1;
 4
 5     private int SSN = 0;
 6     String name = null;
 7 }
 8
 9 class TestPerson
10 {
11     private Person p = new Person( );
12
13     public static void main( String [] args )
14     {
15         Person q = new Person( );
16
17         System.out.println( p );             // illegal
18         System.out.println( q );             // legal
19
20         System.out.println( q.NO_SSN );      // ?
21         System.out.println( q.SSN );         // ?
22         System.out.println( q.name );        // ?
23         System.out.println( Person.NO_SSN ); // ?
24         System.out.println( Person.SSN );    // ?
25     }
26 }
```

Figure 3.12 Code for Exercise 3.10

In Theory

3.11. Aclass provides a single private constructor. Why would this be useful?

3.12. Suppose that the `main` method in Figure 3.3 was part of the `IntCell` class.

a. Would the program still work?

b. Could the commented-out line in `main` be uncommented without generating an error?

In Practice

3.13. A *combination lock* has the following basic properties: the combination (a sequence of three numbers) is hidden; the lock can be opened by providing

the combination; and the combination can be changed, but only by someone who knows the current combination. Design a class with public methods `open` and `changeCombo` and private data fields that store the combination. The combination should be set in the constructor. Disable copying of combination locks.

3.14. Wild card import directives are dangerous because ambiguities and other surprises can be introduced. Recall that both `java.awt.List` and `java.util.List` are classes. Starting with the code in Figure 3.13:

a. Compile the code; you should get an ambiguity.

b. Add an import directive to explicitly use `java.awt.List`. The code should now compile and run.

c. Uncomment the local `List` class; and remove the import directive you just added. The code should compile and run.

d. Recomment the local `List`, reverting back to the situation at the start. Recompile to see the surprising result. What happens if you add the explicit import directive from step (b)?

```
 1 import java.util.*;
 2 import java.awt.*;
 3
 4 class List  // COMMENT OUT THIS CLASS TO START EXPERIMENT
 5 {
 6     public String toString( ) { return "My List!!"; }
 7 }
 8
 9 class WildCardIsBad
10 {
11     public static void main( String [] args )
12     {
13         System.out.println( new List( ) );
14     }
15 }
```

Figure 3.13 Code for Exercise 3.14 illustrates why wildcard imports are bad

Programming Projects

3.15. Write a class that supports rational numbers. The fields should be two `long` variables, one each that stores the numerator and denominator. Store the rational number in reduced form, with the denominator always nonnegative. Provide a reasonable set of constructors; the methods `add`, `subtract`, `multiply`, and `divide`; as well as `toString`, `equals`, and `compareTo` (that behaves like the one in the `String` class). Make sure that `toString` correctly handles the case in which the denominator is zero.

3.16. Implement a simple `Date` class. You should be able to represent any date from January 1, 1800, to December 31, 2500; subtract two dates; increment a date by a number of days; and compare two dates using both `equals` and `compareTo`. A `Date` is represented internally as the number of days since some starting time, which, here, is the start of 1800. This makes all methods except for construction and `toString` trivial.

The rule for leap years is a year is a leap year if it is divisible by 4 and not divisible by 100 unless it is also divisible by 400. Thus 1800, 1900, and 2100 are not leap years, but 2000 is. The constructor must check the validity of the date, as must `toString`. The `Date` could be bad if an increment or subtraction operator caused it to go out of range.

Once you have decided on the specifications, you can do an implementation. The difficult part is converting between the internal and external representations of a date. What follows is a possible algorithm.

Set up two arrays that are static fields. The first array, `daysTillFirstOfMonth`, will contain the number of days until the first of each month in a nonleap year. Thus it contains 0, 31, 59, 90, and so on. The second array, `daysTillJan1`, will contain the number of days until the first of each year, starting with `firstYear`. Thus it contains 0, 365, 730, 1095, 1460, 1826, and so on because 1800 is not a leap year, but 1804 is. You should have your program initialize this array once using a static initializer. You can then use the array to convert from the internal representation to the external representation.

3.17. Implement a `Complex` number class. Recall that a complex number consists of a real part and an imaginary part. Support the same operations as the `Rational` class, when meaningful (for instance, `compareTo` is not meaningful). Add accessor methods to extract the real and imaginary parts.

3.18. Implement a complete `IntType` class that supports a reasonable set of constructors, `add`, `subtract`, `multiply`, `divide`, `equals`, `compareTo`, and `toString`. Maintain an `IntType` as a sufficiently large array. For this class, the difficult operation is division, followed closely by multiplication.

References

More information only classes can be found in the references at the end of Chapter 1. The classic reference on design patterns is [1]. This book describes 23 standard patterns, some of which we will discuss later.

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass, 1995.

CHAPTER

4 Inheritance

AS mentioned in Chapter 3, an important goal of object-oriented programming is code reuse. Just as engineers use components over and over in their designs, programmers should be able to reuse objects rather than repeatedly reimplement them. In an object-oriented programming language, the fundamental mechanism for code reuse is *inheritance*. Inheritance allows us to extend the functionality of an object. In other words, we can create new types with restricted (or extended) properties of the original type, in effect forming a hierarchy of classes.

Inheritance is more than simply code reuse, however. By using inheritance correctly, it enables the programmer to more easily maintain and update code, both of which are essential in large commerical applications. Understanding of the use of inheritance is essential in order to write significant Java programs, and it is also used by Java to implement generic methods and classes.

In this chapter, we will see:

- General principles of inheritance, including *polymorphism*
- How inheritance is implemented in Java
- How a collection of classes can be derived from a single abstract class

- The *interface*, which is a special kind of a class
- How Java implements generic programming using inheritance

4.1 What Is Inheritance?

In an *IS-A relationship*, we say the derived class *is a* (variation of the) base class.

Inheritance is the fundamental object-oriented principle that is used to reuse code among related classes. Inheritance models the *IS-A relationship*. In an IS-A relationship, we say the derived class *is a* (variation of the) base class. For example, a Circle IS-A Shape and a Car IS-A Vehicle. However, an Ellipse IS-NOT-A Circle. Inheritance relationships form *hierarchies*. For instance, we can extend Car to other classes, since a ForeignCar IS-A Car (and pays tariffs) and a DomesticCar IS-A Car (and does not pay tariffs), and so on.

```
1  class Person
2  {
3      public Person( String n, int ag, String ad, String p )
4        { name = n; age = ag; address = ad; phone = p; }
5
6      public String toString( )
7        { return getName( ) + " " + getAge( ) + " "
8                            + getPhoneNumber( ); }
9
10     public String getName( )
11       { return name; }
12
13     public int getAge( )
14       { return age; }
15
16     public String getAddress( )
17       { return address; }
18
19     public String getPhoneNumber( )
20       { return phone; }
21
22     public void setAddress( String newAddress )
23       { address = newAddress; }
24
25     public void setPhoneNumber( String newPhone )
26       { phone = newPhone; }
27
28     private String name;
29     private int    age;
30     private String address;
31     private String phone;
32 }
```

Figure 4.1 `Person` class: stores name, age, address, and phone number

Another type of relationship is a *HAS-A* (or IS-COMPOSED-OF) *relationship*. This type of relationship does not possess the properties that would be natural in an inheritance hierarchy. An example of a HAS-A relationship is that a car HAS-A steering wheel. HAS-A relationships should not be modeled by inheritance. Instead, they should use the technique of *composition*, in which the components are simply made private data fields.

In a *HAS-A relationship*, we say the derived class *has a* (instance of the) base class. *Composition* is used to model HAS-A relationships.

As we will see in forthcoming chapters, the Java language itself makes extensive use of inheritance in implementing its class libraries.

4.1.1 Creating New Classes

Our inheritance discussion will center around an example. Figure 4.1 shows a typical class. The `Person` class is used to store information about a person; in our case we have private data that includes the name, age, address, and phone number, along with some public methods that can access and perhaps change this information. We can imagine that in reality, this class is significantly more complex, storing perhaps 30 data fields with 100 methods.

```
1  class Student
2  {
3      public Student( String n, int ag, String ad, String p,
4                      double g )
5        { name = n; age = ag; address = ad; phone = p; gpa = g; }
6
7      public String toString( )
8        { return getName( ) + " " + getAge( ) + " "
9               + getPhoneNumber( ) + " " + getGPA( ); }
10
11     public String getName( )
12       { return name; }
13
14     public int getAge( )
15       { return age; }
16
17     public String getAddress( )
18       { return address; }
19
20     public String getPhoneNumber( )
21       { return phone; }
22
23     public void setAddress( String newAddress )
24       { address = newAddress; }
25
26     public void setPhoneNumber( String newPhone )
27       { phone = newPhone; }
28
29     public double getGPA( )
30       { return gpa; }
31
32     private String name;
33     private int    age;
34     private String address;
35     private String phone;
36     private double gpa
37 }
```

Figure 4.2 `Student` class: stores name, age, address, phone number, gpa via copy-and-paste

```
 1 class Student extends Person
 2 {
 3     public Student( String n, int ag, String ad, String p,
 4                     double g )
 5       {
 6         /* OOPS! Need some syntax; see Section 4.1.6 */
 7          gpa = g; }
 8
 9     public String toString( )
10       { return getName( ) + " " + getAge( ) + " "
11              + getPhoneNumber( ) + " " + getGPA( ); }
12
13     public double getGPA( )
14       { return gpa; }
15
16     private double gpa;
17 }
```

Figure 4.3 Inheritance used to create `Student` class

Now suppose we want to have a `Student` class, or an `Employee` class, or both. Imagine that a `Student` is similar to a `Person`, with the addition of only a few extra data members and methods. In our simple example, imagine that the difference is that a `Student` adds a `gpa` field and a `getGPA` accessor. Similarly, imagine that the `Employee` has all of the same components as a `Person`, but also has a `salary` field and methods to manipulate the salary.

One option in designing these classes is the classic *copy-and-paste*: we copy the `Person` class, change the name of the class and constructors, and then add the new stuff. This strategy is illustrated in Figure 4.2.

Copy-and-paste is a weak design option, wrought with significant liabilities. First, there is the problem that if you copy garbage, you wind up with more garbage. This makes it very hard to fix programming errors that are detected, especially when they are detected late.

Second, is the related issue of maintainence and versioning. Suppose we decide in the second version that it is better to store names in last name, first name format, rather than as a single field. Or perhaps it is better to store addresses using a special `Address` class. In order to maintain consistency, these should be done for all classes. Using copy-and-paste, these design changes have to be done in numerous places.

Third, and more subtle, is the fact that using copy-and-paste, `Person`, `Student`, and `Employee` are three separate entities with zero relationship between each other, in spite of their similarities. So, for instance, if we have a routine that accepted a `Person` as a parameter, we could not send in a `Student`. We would thus have to copy and paste all of those routines to make them work for these new types.

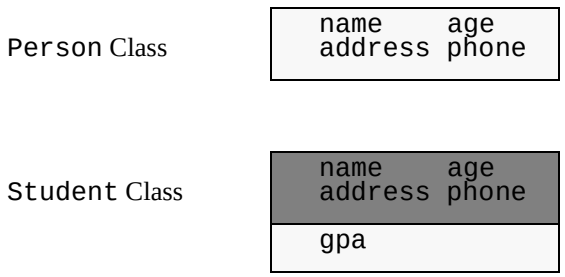

Figure 4.4 Memory layout with inheritance. Light shading indicates fields that are private, and accessible only by methods of the class. Dark shading in `Student` class indicates fields that are not accessible in the `Student` class, but are nonetheless present.

Inheritance solves all three of these problems. Using inheritance, we would say that a `Student` *IS-A* `Person`. We would then specify the changes that a `Student` has relative to `Person`. There are only three types of changes that are allowed:

1. `Student` can add new fields (e.g. `gpa`).
2. `Student` can add new methods (e.g. `getGPA`).
3. `Student` can override existing methods (e.g. `toString`).

Two changes are specifically not allowed, because they would violate the notion of an IS-A relationship:

1. `Student` cannot remove fields.
2. `Student` cannot remove methods.

Finally, the new class must specify its own constructors; this is likely to involve some syntax that we will discuss in Section 4.1.6.

Figure 4.3 shows the `Student` class. The data layout for the `Person` and `Student` classes is shown in Figure 4.4. It illustrates that the memory footprint of any `Student` object includes all fields that would be contained in a `Person` object. However, because those fields are declared private by `Person`, they are not accessible by `Student` class methods. That is why the constructor is problematic at this point: we cannot touch the data fields in any `Student` method, and instead can only manipulate the inherited private fields by using public `Person` methods. Of course, we could make the inherited fields public, but that would generally be a terrible design decision. It would embolden the implementors of the `Student` and `Employee` classes to access the inherited fields directly. If that was done, and modifications such as a change in the `Person`'s

data representation of the name or address were made to the `Person` class, we would now have to track down all of the dependencies, which would bring us back to the copy-and-paste liabilities.

As we can see, except for the constructors, the code is relatively simple. We have added one data field, added one new method, and overridden an existing method. Internally, we have memory for all of the inherited fields, and we also have implementations of all original methods that have not been overridden. The amount of new code we have to write for `Student` would be roughly the same, regardless of how small or large the `Person` class was, and we have the benefit of *direct code reuse* and easy maintainence. Observe also, that we have done so without disturbing the implementation of the existing class.

Inheritance allows us to derive classes from a *base class* without disturbing the implementation of the base class.

```
 1 public class Derived extends Base
 2 {
 3     // Any members that are not listed are inherited unchanged
 4     // except for constructor
 5
 6         // public members
 7     // Constructor(s) if default is not acceptable
 8     // Base methods whose definitions are to change in Derived
 9     // Additional public methods
10
11         // private member
12     // Additional data fields (generally private)
13     // Additional private methods
14 }
```

Figure 4.5 General layout of public inheritance

The *extends clause* is used to declare that a class is derived from another class.

Let us summarize the synax so far. A derived class inherits all the properties of a base class. It can then add data members, override methods, and add new methods. Each derived class is a completely new class. A typical layout for inheritance is shown in Figure 4.5 and uses an *extends clause*. An `extends` clause declares that a class is derived from another class. A derived class *extends* a base class. Here is a brief description of a derived class:

A derived class inherits all data members from the base class and may add more data members.

- Generally all data is private, so we add additional data fields in the derived class by specifying them in the private section.
- Any base class methods that are not specified in the derived class are inherited unchanged, with the exception of the constructor. The special case of the constructor is discussed in Section 4.1.6.
- Any base class method that is defined in the derived class' public section is overridden. The new definition will be applied to objects of the derived class.

The derived class inherits all methods from the base class. It may accept or redefine them. It also can define new methods.

- Public base class methods may not be redefined in the private section of the derived class, because that would be tantamount to removing methods and would violate the IS-A relationship.
- Additional methods can be added in the derived class.

4.1.2 Type Compatability

The direct code reuse described in the preceding paragraph is a significant gain. However, the more significant gain is *indirect code reuse*. This gain comes from the fact that a `Student` IS-A `Person` and an `Employee` IS-A `Person`.

Each *derived class* is a completely new class that nonetheless has some compatibility with the class from which it was derived.

Because a `Student` IS-A `Person`, a `Student` object can be accessed by a `Person` reference. The following code is thus legal:

```
Student s = new Student( "Joe", 26, "1 Main St",
                         "202-555-1212", 4.0 );
Person p = s;
System.out.println( "Age is " + p.age( ) );
```

This is legal because the static-type (i.e. compile-time type) of `p` is `Person`. Thus `p` may reference any object that IS-A `Person`, and any method that we invoke through the `p` reference is guaranteed to make sense, since once a method is defined for `Person`, it cannot be removed by a derived class.

You might ask why this is a big deal. The reason is that this applies not only to assignment, but also to parameter passing. A method whose formal parameter is a `Person` can receive anything that IS-A `Person`, including `Student` and `Employee`.

So consider the following code written in *any class*:

```
public static boolean isOlder( Person p1, Person p2 )
{
    return p1.getAge( ) > p2.getAge( );
}
```

Consider the following declarations, in which constructor arguments are missing to save space:

```
Person   p = new Person( ... );
Student  s = new Student( ... );
Employee e = new Employee( ... );
```

The single `isOlder` routine can be used for all of the following calls: `isOlder(p,p)`, `isOlder(s,s)`, `isOlder(e,e)`, `isOlder(p,e)`, `isOlder(p,s)`, `isOlder(s,p)`, `isOlder(s,e)`, `isOlder(e,p)`, `isOlder(e,s)`.

All in all, we now have leveraged one non-class routine to work for nine different cases. In fact there is no limit to the amount of reuse this gets us. As soon as we use inheritance to add a fourth class into the hierarchy, we now have 4 times 4, or 16 different methods, without changing `isOlder` at all! The reuse is even more significant if a method were to take three `Person` references as parameters. And imagine the huge code reuse if a method takes an array of `Person` references.

Thus, for many people, the type compatability of derived classes with their base classes is the most important thing about inheritance because it leads to massive *indirect code reuse*. And as `isOlder` illustrates, it also makes it very easy to add in new types that automatically work with existing methods.

4.1.3 Dynamic Binding and Polymorphism

There is the issue of overriding methods: if the type of the reference and the class of the object being referenced (in the example above, these are `Person` and `Student`, respectively) disagree, and they have different implementations, whose implementation is to be used?

As an example, consider the following fragment:

```
Student s = new Student( "Joe", 26, "1 Main St",
                         "202-555-1212", 4.0 );
Employee e = new Employee( "Boss", 42, "4 Main St.",
                     "203-555-1212", 100000.0 );
Person p = null;
if( getTodaysDay( ).equals( "Tuesday" ) )
    p = s;
else
    p = e;
System.out.println( "Person is " + p.toString( ) );
```

Here the static type of `p` is `Person`. When we run the program, the dynamic type (i.e. the type of the object actually being referenced) will be either `Student` or `Employee`. It is impossible to deduce the dynamic type until the program runs. Naturally, however, we would want the dynamic type to be used, and that is what happens in Java. When this code fragment is run, the method that is used will be the one appropriate for the dynamic type of the controlling object reference.

A *polymorphic* variable can reference objects of several different types. When operations are applied to the polymorphic variable, the operation appropriate to the referenced object is automatically selected.

This is an important object-oriented principle known as *polymorphism*. A reference variable that is polymorphic can reference objects of several different types. When operations are applied to the reference, the operation that is appropriate to the actual referenced object is automatically selected. All reference types are polymorphic in Java. This is also known as *dynamic binding* or *late binding*.

A derived class is type compatible with its base class, meaning that a reference variable of the base class type may reference an object of the derived class, but not vice versa. Sibling classes (that is, classes derived from a common class) are not type compatible.

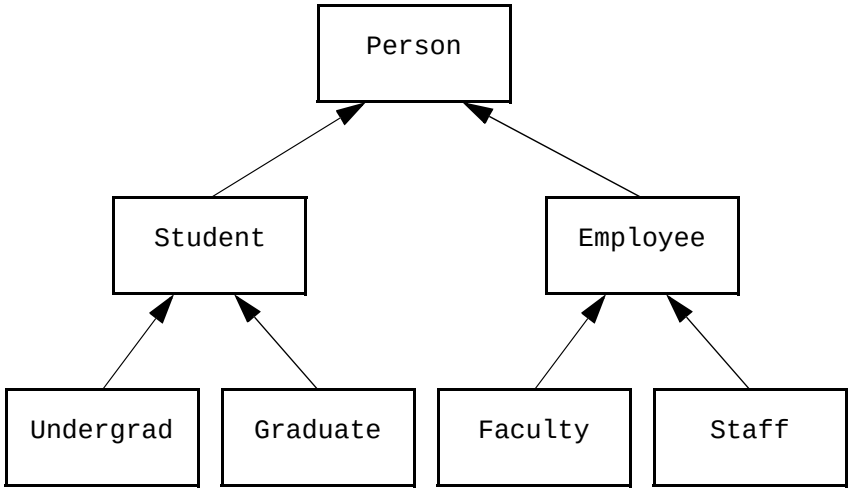

Figure 4.6 The `Person` hierarchy

4.1.4 Inheritance Hierarchies

If *X* IS-A *Y*, then *X* is a *subclass* of *Y* and *Y* is a *superclass* of *X*. These relationships are transitive.

As mentioned earlier, the use of inheritance typically generates a hierarchy of classes. Figure 4.6 illustrates a possible `Person` hierarchy. Notice that `Faculty` is indirectly, rather than directly, derived from `Person` — so faculty are people too! This fact is transparent to the user of the classes because IS-A relationships are transitive. In other words, if *X* IS-A *Y* and *Y* IS-A *Z*, then *X* IS-A *Z*. The `Person` hierarchy illustrates the typical design issues of factoring out commonalities into base classes and then specializing in the derived classes. In this hierarchy, we say that the derived class is a *subclass* of the base class and the base class is a *superclass* of the derived class. These relationships are transitive, and furthermore, the `instanceof` operator works with subclasses. Thus if

`obj` is of type `Undergrad` (and not `null`), then `obj instanceof Person` is `true`.

4.1.5 Visibility Rules

We know that any member that is declared with private visibility is accessible only to methods of the class. Thus as we have seen, any private members in the base class are not accessible to the derived class.

Occasionally we want the derived class to have access to the base class members. There are two basic options. The first is to use either public or friendly access, as appropriate. However, this allows access to other classes in addition to derived classes.

A *protected class member* is visible to the derived class and also classes in the same package.

If we want to restrict access to only derived classes, we can make members protected. A *protected class member* is visible to methods in a derived class and also methods in classes in the same package, but not to anyone else.[1] Declaring data members as protected or public violates the spirit of encapsulation and information hiding and is generally done only as a matter of programming expediency. Typically, a better alternative is to write accessor and mutator methods or to use friendly access. However, if a protected declaration allows you to avoid convoluted code, then it is not unreasonable to use it. In this text, protected data members are used for precisely this reason. Protected methods are also used in this text. This allows a derived class to inherit an internal method without making it accessible outside the class hierarchy. Notice that in toy code, in which all classes are in the default unnamed package, protected members are visible.

4.1.6 The Constructor and super

If no constructor is written, then a single zero-parameter default constructor is generated that calls the base class zero-parameter constructor for the inherited portion, and then applies the default initialization for any additional data fields.

Each derived class should define its constructors. If no constructor is written, then a single zero-parameter default constructor is generated. This constructor will call the base class zero-parameter constructor for the inherited portion and then apply the default initialization for any additional data fields (meaning 0 for primitive types, and `null` for reference types).

Constructing a derived class object by first constructing the inherited portion is standard practice. In fact, it is done by default, even if an explicit derived class constructor is given. This is natural because the encapsulation viewpoint tells us that the inherited portion is a single entity, and the base class constructor tells us how to initialize this single entity.

---

1. The rule for protected visibility is quite complex. A protected member of class B is visible to all methods in class that are in the same package as B. It is also visible to methods in any class D that is in a different package than B if D extends B, but only if accessed through a reference that is type-compatible with D (including an implicit or explicit `this`). Specifically, it is NOT VISIBLE in class D if accessed through a reference of type B. The following example illustrates this.

```
class Demo extends java.io.FilterInputStream
{        // FilterInputStream has protected data field named in
    public void foo( )
    {
         java.io.FilterInputStream b = this;  // legal
         System.out.println( in );            // legal
         System.out.println( this.in );       // legal
         System.out.println( b.in );          // illegal
    }
}
```

Base class constructors can be explicitly called by using the method `super`.

Thus the default constructor for a derived class is in reality

```
public Derived( )
{
    super( );
}
```

The `super` method can be called with parameters that match a base class constructor. As an example, Figure 4.7 illustrates the implementation of the `Student` constructor.

`super` is used to call the base class constructor.

The `super` method can be used only as the first line of a constructor. If it is not provided, then an automatic call to `super` with no parameters is generated.

```
1  class Student extends Person
2  {
3      public Student( String n, int ag, String ad, String p,
4                      double g )
5        { super( n, ag, ad, p );  gpa = g; }
6
7      // toString and getAge omitted
8
9      private double gpa;
10 }
```

Figure 4.7 Constructor for new class `Student`; uses `super`

4.1.7 `final` Methods and Classes

As described earlier, the derived class either overrides or accepts the base class methods. In many cases, it is clear that a particular base class method should be invariant over the hierarchy, meaning that a derived class should not override it. In this case, we can declare that the method is *`final`* and cannot be overridden.

A *final method* is invariant over the inheritance hierarchy and may not be overridden.

Declaring invariant methods `final` is not only good programming practice. It also can lead to more efficient code. It is good programming practice because in addition to declaring your intentions to the reader of the program and documentation, you prevent the accidental overriding of a method that should not be overridden.

To see why using `final` may make for more efficient code, suppose base class `Base` declares a final method `f` and suppose `Derived` extends `Base`. Consider the routine

```
void doIt( Base obj )
{
    obj.f( );
}
```

Static binding could be used when the method is invariant over the inheritance hierarchy.

Since `f` is a final method, it does not matter whether `obj` actually references a `Base` or `Derived` object; the definition of `f` is invariant, so we know what `f` does. As a result, a compile-time decision, rather than a run-time decision, could be made to resolve the method call. This is known as *static binding*. Because binding is done during compilation rather than at run time, the program should run faster. Whether this is noticeable would depend on how many times we avoid making the run-time decision while executing the program.

Static methods have no controlling object and thus are resolved at compile time using static binding.

A corollary to this observation is that if `f` is a trivial method, such as a single field accessor, and is declared `final`, the compiler could replace the call to `f` with its inline definition. Thus the method call would be replaced by a single line that accesses a data field, thereby saving time. If `f` is not declared `final`, then this is impossible, since `obj` could be referencing a derived class object, for

which the definition of `f` could be different.[2] Static methods have no controlling object and thus are resolved at compile time using static binding.

A *final class* may not be extended. A *leaf class* is a final class.

Similar to the final method is the *final class*. A final class cannot be extended. As a result, all of its methods are automatically final methods. As an example, the `String` class is a final class. Notice that the fact that a class has only final methods does not imply that it is a final class. Final classes are also known as *leaf classes* because in the inheritance hierarchy, which resembles a tree, final classes are at the fringes, like leaves.

In the `Person` class, the trivial accessors and mutators (those starting with `get` and `set`) are good candidates for final methods, and they are declared as such in the online code.

2. In the preceding two paragraphs, we says that static binding and inline optimizations "could be" done because although compile-time decisions would appear to make sense, Section 8.4.3.3 of the language specification makes clear that inline optimizations for trivial final methods can be done, but this optimization must be done by the virtual machine at runtime, rather than the compiler at compile time. This ensures that dependent classes do not get out of sync as a result of the optimization.

4.1.8 Overriding a Method

The derived class method must have the same return type and signature and may not add exceptions to the throws list.

Methods in the base class are overridden in the derived class by simply providing a derived class method with the same signature.[3] The derived class method must have the same return type and may not add exceptions to the throws list. The derived class may not reduce visibility, as that would violate the spirit of an IS-A relationship. Thus you may not override a public method with a package visible method.

Partial overriding involves calling a base class method by using `super`.

Sometimes the derived class method wants to invoke the base class method. Typically, this is known as *partial overriding*. That is, we want to do what the base class does, plus a little more, rather than doing something entirely different. Calls to a base class method can be accomplished by using `super`. Here is an example:

3. If a different signature is used, you simply have overloaded the method, and now there are two methods with different signatures available for the compiler to choose from.

```
 1 class Student extends Person
 2 {
 3     public Student( String n, int ag, String ad, String p,
 4                     double g )
 5       { super( n, ag, ad, p ); gpa = g; }
 6
 7     public String toString( )
 8       { return super.toString( ) + getGPA( ); }
 9
10     public double getGPA( )
11       { return gpa; }
12
13     private double gpa;
14 }
15
16 class Employee extends Person
17 {
18     public Employee( String n, int ag, String ad,
19                      String p, double s )
20       { super( n, ag, ad, p ); salary = s; }
21
22     public String toString( )
23       { return super.toString( ) + " " + getSalary( ); }
24
25     public double getSalary( )
26       { return salary; }
27
28     public void raise( double percentRaise )
29       { salary *= ( 1 + percentRaise ); }
30
31     private double salary;
32 }
```

Figure 4.8 The complete `Student` and `Employee` classes, using both forms of `super`

```
public class Workaholic extends Worker
{
    public void doWork( )
    {
        super.doWork( );   // Work like a Worker
        drinkCoffee( );    // Take a break
        super.doWork( );   // Work like a Worker some more
    }
}
```

A more typical example is the overriding of standard methods, such as `toString`. Figure 4.8 illustrates this use in the `Student` and `Employee` classes.

```
 1 class PersonDemo
 2 {
 3     public static void printAll( Person[ ] arr )
 4     {
 5         for( int i = 0; i < arr.length; i++ )
 6         {
 7             if( arr[ i ] != null )
 8             {
 9                 System.out.print( "[" + i + "] " );
10                 System.out.println( arr[ i ].toString( ) );
11             }
12         }
13     }
14
15     public static void main( String [ ] args )
16     {
17         Person [ ] p = new Person[ 4 ];
18
19         p[0] = new Person( "joe", 25, "New York",
20                            "212-555-1212" );
21         p[1] = new Student( "becky", 27, "Chicago",
22                             "312-555-1212", 4.0 );
23         p[3] = new Employee( "bob", 29, "Boston",
24                              "617-555-1212", 100000.0 );
25
26         printAll( p );
27     }
28 }
```

Figure 4.9 Illustration of polymorphism with arrays

4.1.9 Type Compatability Revisited

Figure 4.9 illustrates the typical use of polymorphism with arrays. At line 17, we create an array of four `Person` references, which will be initialized to `null`. The values of these references can be set at lines 19 to 24, and we know that all

the assignments are legal because of the ability of a base type reference to refer to objects of a derived type.

The `printAll` routine simply steps through the array and calls the `toString` method, using dynamic binding. The test at line 7 is important because, as we have seen, some of the array references could be `null`.

In the example, suppose that prior to completing the printing, we want to give `p[3]` – which we know is an employee – a raise? Since `p[3]` is an `Employee`, it might seem that

```
p[3].raise( 0.04 );
```

would be legal. But it is not. The problem is that the static type of `p[3]` is a `Person`, and `raise` is not defined for `Person`. At compile time, only (visible) members of the *static type* of the reference can appear to the right of the dot operator.

We can change the static type by using a cast:

```
((Employee) p[3]).raise( 0.04 );
```

makes the static type of the reference to the left of the dot operator an `Employee`. If this is impossible (for instance `p[3]` is in a completely different hierarchy), the compiler will complain. If it is possible for the cast to make sense, the program will compile, and so the above code will successfully give a 4% raise to `p[3]`. This construct, in which we change the static type of an expression from a base class to a class farther down in the inheritance hierarchy is known as a *downcast*.

A *downcast* is a cast down the inheritance hierarchy. Casts are always verified at runtime by the virtual machine.

What if `p[3]` was not an `Employee`? For instance, what if we used the following?

```
((Employee) p[1]).raise( 0.04 ); // p[1] is a Student
```

In that case the program would compile, but the virtual machine would throw a `ClassCastException`, which is a runtime exception that signals a programming error. Cast are always double-checked at runtime to ensure that the programmer (or a malicious hacker) is not trying to subvert Java's strong typing system. The safe way of doing these types of calls is to use `instanceof` first:

```
if( p[3] instanceof Employee )
    ((Employee) p[3]).raise( 0.04 );
```

4.2 Designing Hierarchies

Suppose we have a `Circle` class, and for any non-null `Circle c`, `c.area()` returns the area of `Circle c`. Additionally, suppose we have a `Rectangle` class, and for any non-null `Rectangle r`, `r.area()` returns the area of `Rectangle r`. Possibly we have other classes such as `Ellipse`, `Triangle`, and `Square`, all with `area` methods. Suppose we have an array that contains references to these objects and we want to compute the total area of all the objects. Since they all have an `area` method for all classes, polymorphism is an attractive option, yielding code such as:

```
public static totalArea( WhatType [ ] arr )
{
    double total = 0.0;
    for( int i = 0; i < arr.length; i++ )
        if( arr[ i ] != null )
            total += arr[ i ].area( );

     return total;
}
```

For this code to work, we need to decide the type declaration for *WhatType*. None of `Circle`, `Rectangle`, etc. will work, since there is no IS-A relationship. Thus we need to define a type, say `Shape`, such that `Circle` IS-A `Shape`, `Rectangle` IS-A `Shape`, etc. A possible hierarchy is illustrated in Figure 4.10. Additionally, in order for `arr[i].area()` to make sense, `area` must be a method available for `Shape`.

This suggests a class for `Shape`, as shown in Figure 4.11. Once we have the `Shape` class, we can provide others, as shown in Figure 4.12. These classes also include a `perimeter` method. Observe that `Square` reuses code inherited from `Rectangle`.

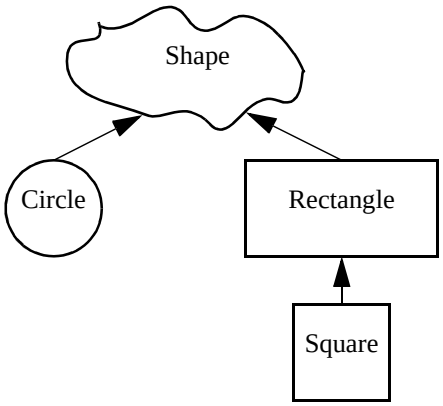

Figure 4.10 The hierarchy of shapes used in an inheritance example

```
1 public class Shape
2 {
3     public double area( )
4     {
5         return -1;
6     }
7 }
```

Figure 4.11 Possible `Shape` class

```
1  public class Circle extends Shape
2  {
3      public Circle( double rad )
4        { radius = rad; }
5
6      public double area( )
7        { return Math.PI * radius * radius; }
8
9      public double perimeter( )
10       { return 2 * Math.PI * radius; }
11
12     public String toString( )
13       { return "Circle: " + radius; }
14
15     private double radius;
16 }
17
18 public class Rectangle extends Shape
19 {
20     public Rectangle( double len, double wid )
21       { length = len; width = wid; }
22
23     public double area( )
24       { return length * width; }
25
26     public double perimeter( )
27       { return 2 * ( length + width ); }
28
29     public String toString( )
30       { return "Rectangle: " + length + " " + width; }
31
32     public double getLength( )
33       { return length; }
34
35     public double getWidth( )
36       { return width; }
37
38     private double length;
39     private double width;
40 }
41
42 public class Square extends Rectangle
43 {
44     public Square( double side )
45       { super( side, side ); }
46
47     public String toString( )
48       { return "Square: " + getLength( ); }
49 }
```

Figure 4.12 `Circle`, `Rectangle`, and `Square` classes

```
1  class ShapeDemo
2  {
3      public static double totalArea( Shape [ ] arr )
4      {
5          double total = 0;
6
7          for( int i = 0; i < arr.length; i++ )
8              if( arr[ i ] != null )
9                  total += arr[ i ].area( );
10
11         return total;
12     }
13
14     public static void printAll( Shape [ ] a )
15     {
16         for( int i = 0; i < a.length; i++ )
17             System.out.println( a[ i ] );
18     }
19
20     public static void main( String [] args )
21     {
22         Shape [ ] a = new Shape[ ] {  new Circle( 2.0 ),
23                           new Rectangle( 1.0, 3.0 ),
24                           null, new Square( 2.0 ) };
25
26         System.out.println( "Total area = " + totalArea( a ) );
27         printAll( a );
28     }
29 }
```

Figure 4.13 Sample program that uses the shape hierarchy

The code in Figure 4.12, with classes that extend the simple `Shape` class in Figure 4.11 that returns -1 for `area`, can now be used polymorphically, as shown in Figure 4.13.

Too many `instanceof` operators is a sympton of poor object-oriented design.

A huge benefit of this design is that we can add a new class to the hierarchy without disturbing implementations. For instance, suppose we want to add triangles into the mix. All we need to do is have `Triangle` extend `Shape`, override `area` appropriately, and now `Triangle` objects can be included in any `Shape[]` object. Observe that this involves:

- NO CHANGES to the `Shape` class
- NO CHANGES to the `Circle`, `Rectangle`, or `Square` classes
- NO CHANGES to the `totalArea` method

making it difficult to break existing code in the process of adding new code. Notice also the lack of any `instanceof` tests, which is typical of good polymorphic code.

4.2.1 Abstract Methods and Classes

Although the code in the previous example works, improvements are possible in the `Shape` class written in Figure 4.11. Notice that the `Shape` class itself, and the `area` method in particular are *placeholders*: the `Shape`'s `area` method is never intended to be called directly. It is there so that the compiler and runtime system can conspire to use dynamic binding and call an appropriate `area` method. In fact, examining `main`, we see that `Shape` objects themselves are not supposed to be created either. The class exists simply as a common superclass for the others.[4]

The programmer has attempted to signal that calling `Shape`'s area is an error by returning -1, which is an obviously impossible area. But this is a value that might be ignored. Furthermore, this is a value that will be returned if when extending `Shape`, `area` is not overridden. This failure to override could occur

4. Declaring a private `Shape` constructor DOES NOT solve the second problem: the constructor is needed by the subclasses.

because of a typing error: an `Area` function is written instead of `area`, making it difficult to track down the error at runtime.

Abstract methods and classes represent placeholders.

A better solution for `area` is to throw a runtime exception (`UnsupportedOperationException` is a good one) in the `Shape` class. This is preferable to returning -1 because the exception will not be ignored.

However, even that solution resolves the problem at runtime. It would be better to have syntax that explicitly states that `area` is a placeholder and does not need any implementation at all, and that further, `Shape` is a placeholder class and cannot be constructed, even though it may declare constructors, and will have a default constructor if none are declared. If this syntax were available, then the compiler could, at compile-time, declare as illegal any attempts to construct a `Shape` instance. It could also declare as illegal any classes, such as `Triangle`, for which there are attempts to construct instances, even though `area` has not been overridden. This exactly describes abstract methods and abstract classes.

An *abstract method* has no meaningful definition and is thus always defined in the derived class.

An *abstract method* is a method that declares functionality that all derived class objects must eventually implement. In other words, it says what these objects can do. However, it does not provide a default implementation. Instead, each object must provide its own implementation.

A class that has at least one abstract method is an *abstract class*. Java requires that all abstract classes be declared as such. When a derived class fails to override an abstract method with an implementation, the method remains abstract in the derived class. As a result, if a class that is not intended to be abstract fails to over-

ride an abstract method, the compiler will detect the inconsistency and report an error.

An example of how we can make `Shape` abstract is shown in Figure 4.14. No changes are required to any of the other code in Figures 4.12 and 4.13. Observe that an abstract class can have methods that are not abstract, as is the case with `semiperimeter`.

```
1 public abstract class Shape
2 {
3     public abstract double area( );
4     public abstract double perimeter( );
5
6     public double semiperimeter( )
7       { return perimeter( ) / 2; }
8 }
```

Figure 4.14 An abstract `Shape` class, Figures 4.12 and 4.13 are unchanged

An abstract class can also declare both static and instance fields. Like nonabstract classes, these fields would typically be private, and the instance fields would be initialized by constructors. Although abstract classes cannot be created, these constructors will be called when the derived classes use `super`. In a more extensive example, the `Shape` class could include the coordinates of the object's extremities, which would be set by constructors, and it could provide implementation of methods, such as `positionOf`, that are independent of the actual type of object; `positionOf` would be a final method.

A class with at least one abstract method must be an *abstract class*.

As mentioned earlier, the existence of at least one abstract method makes the base class abstract and disallows creation of it. Thus a `Shape` object cannot itself be created; only the derived objects can. However, as usual, a `Shape` variable can reference any concrete derived object, such as a `Circle` or `Rectangle`. Thus

```
Shape a, b;
a = new Circle( 3.0 );      // Legal
b = new Shape( "circle" );  // Illegal
```

Before continuing, let us summarize the four types of class methods:

1. *Final methods*. The virtual machine may choose at runtime to perform inline optimization, thus avoiding dynamic binding. We use a final method only when the method is invariant over the inheritance hierarchy (that is, when the method is never redefined).
2. *Abstract methods*. Overriding is resolved at run time. The base class provides no implementation and is abstract. The absence of a default requires either that the derived classes provide an implementation or that the classes themselves be abstract.
3. *Static methods*. Overridding is resolved at compile time because there is no controlling object.
4. *Other methods*. Overriding is resolved at run time. The base class provides a default implementation that may be either overridden by the derived classes or accepted unchanged by the derived classes.

4.3 Multiple Inheritance

Multiple inheritance is used to derive a class from several base classes. Java does not allow multiple inheritance.

All the inheritance examples seen so far derived one class from a single base class. In *multiple inheritance*, a class may be derived from more than one base class. For instance, we may have a `Student` class and an `Employee` class. A `StudentEmployee` could then be derived from both classes.

Although multiple inheritance sounds attractive, and some languages (including C++) support it, it is wrought with subtleties that make design difficult. For instance, the two base classes may contain two methods that have the same signature but different implementations. Alternately, they may have two identically named fields. Which one should be used?

For example, suppose that in the previous `StudentEmployee` example `Person` is a class with data field `name` and method `toString`. Suppose, too, that `Student` extends `Person` and overrides `toString` to include the year of graduation. Further, suppose that `Employee` extends `Person` but does not override `toString`; instead, it declares that it is `final`.

1. Since `StudentEmployee` inherits the data members from both `Student` and `Employee`, do we get two copies of `name`?
2. If `StudentEmployee` does not override `toString`, which `toString` method should be used?

When many classes are involved, the problems are even larger. It appears however, that the typical multiple inheritance problems can be traced to conflicting implementations or conflicting data fields. As a result, Java does not allow multiple inheritance. Instead, it provides an alternative known as the *interface*.

4.4 The Interface

The *interface* is an abstract class that contains no implementation details.

The *interface* in Java is the ultimate abstract class. It consists of public abstract methods and public static final fields, only.

A class is said to *implement* the interface if it provides definitions for all of the abstract methods in the interface. A class that implements the interface behaves as if it had extended an abstract class specified by the interface.

```
1 package java.lang;
2
3 public interface Comparable
4 {
5     int compareTo( Object other );
6 }
```

Figure 4.15 `Comparable` interface

In principle, the main difference between an interface and an abstract class is that although both provide a specification of what the subclasses must do, the interface is not allowed to provide any implementation details either in the form of data fields or implemented methods. The practical effect of this is that multiple interfaces do not suffer the same potential problems as multiple inheritance because we cannot have conflicting implementations. Thus, while a class may extend only one other class, it may implement more than one interface.

4.4.1 Specifying an Interface

Syntactically, virtually nothing is easier than specifying an interface. The interface looks like a class declaration, except that it uses the keyword `interface`.

It consists of a listing of the methods that must be implemented. An example is the `Comparable` interface, shown in Figure 4.15, which is part of the standard `java.lang` package, starting with Java 1.2.

The `Comparable` interface specifies one method that every subclass must implement: `compareTo`, which behaves like the `String compareTo` method. In fact, `String` implements precisely this interface. Note that we do not have to specify that these methods are `public` and `abstract`. Since these modifiers are required for interface methods, they can and should be omitted.

4.4.2 Implementing an Interface

A class implements an interface by

1. declaring that it implements the interface, and
2. defining implementations for all the interface methods.

The *`implements` clause* is used to declare that a class implements an interface. The class must implement all interface methods or it remains abstract.

An example is shown in Figure 4.16. Here, we finalize the `Shape` class, which we used in Section 4.2.

Line 1 shows that when implementing an interface, we use *`implements`* instead of `extends`. `Shape` is abstract because it has abstract methods; if it did not, it would not need to be declared abstract. We can provide any methods that we want, but we must provide at least those listed in the interface. The interface is implemented at lines 6 to 17. Notice that we must implement the *exact method* specified in the interface. Thus these methods take `Object` as a parameter, instead of `Shape` or `Comparable`.

A class that implements an interface can be extended if it is not final. The extended class automatically implements the interface. Thus, `Circle` automatically implements `Comparable`, and it has inherited the `compareTo` method from `Shape`.

A class that implements an interface may still extend one other class. The `extends` clause must precede the `implements` clause.

```
 1 public abstract class Shape implements Comparable
 2 {
 3     public abstract double area( );
 4     public abstract double perimeter( );
 5
 6     public int compareTo( Object rhs )
 7     {
 8         Shape other = (Shape) rhs;
 9         double diff = area( ) - other.area( );
10
11         if( diff == 0 )
12             return 0;
13         else if( diff < 0 )
14             return -1;
15         else
16             return 1;
17     }
18
19     public double semiperimeter( )
20       { return perimeter( ) / 2; }
21 }
```

Figure 4.16 The `Shape` class (final version), which implements the `Comparable` interface

4.4.3 Multiple Interfaces

As we mentioned earlier, a class may implement multiple interfaces. The syntax for doing so is simple. A class implements multiple interfaces by

1. listing the interfaces (comma separated) that it implements, and
2. defining implementations for all of the interface methods.

The interface is the ultimate in abstract classes and represents an elegant solution to the multiple inheritance problem.

4.4.4 Interfaces are Abstract Classes

Because an interface is an abstract class, all the rules of inheritance apply. Specifically:

1. The IS-A relationship holds. If class *C* implements interface *I*, then *C* IS-A *I* and is type-compatable with *I*. If a class *C* implements interfaces I_1, I_2, and I_3, then *C* IS-A I_1, *C* IS-A I_2, and *C* IS-A I_3, and is type compatable with I_1, I_2, and I_3.
2. The `instanceof` operator can be used to determine if a reference is type-compatable with an interface.
3. When a class implements an interface method, it may not reduce visibility. Since all interface methods are public, all implementations must be public.
4. When a class implements an interface method, it may not add checked exceptions to the throws list. If a class implements multiple interfaces in which the same method occurs with different throws list, the throws list of the implementation may list only checked exceptions that are in the intersection of the throws lists of the interface methods.
5. When a class implements an interface method, it must implement the exact signature (not including throws list); otherwise, it inherits an abstract version of the interface method, and has provided a non-abstract overloaded, but different method.
6. A class may not implement two interfaces that contain a method with the same signature and different return types, since it would be impossible to provide both methods in one class.
7. If a class fails to implement any methods in an interface, it must be declared abstract.
8. Interfaces can extend other interfaces (including multiple interfaces).

4.5 Fundamental Inheritance in Java

Two important places where inheritance is used in Java are the `Object` class and the hierarchy of exceptions.

4.5.1 The `Object` Class

Java specifies that if a class does not extend another class, then it implicitly extends the class `Object` (defined in `java.lang`). As a result, every class is either a direct or indirect subclass of `Object`.

The `Object` class contains several methods, and since it is not abstract, all have implementations. The most commonly-used method is `toString`, which we have already seen. If `toString` is not written for a class, an implementation is provided that concatenates the name of the class, an **@**, and the class' "hashCode".

Other important methods are `equals` and the `hashCode`, which we will discuss in more detail in Chapter 6, and a set of somewhat tricky methods that advanced Java programmers need to know about.

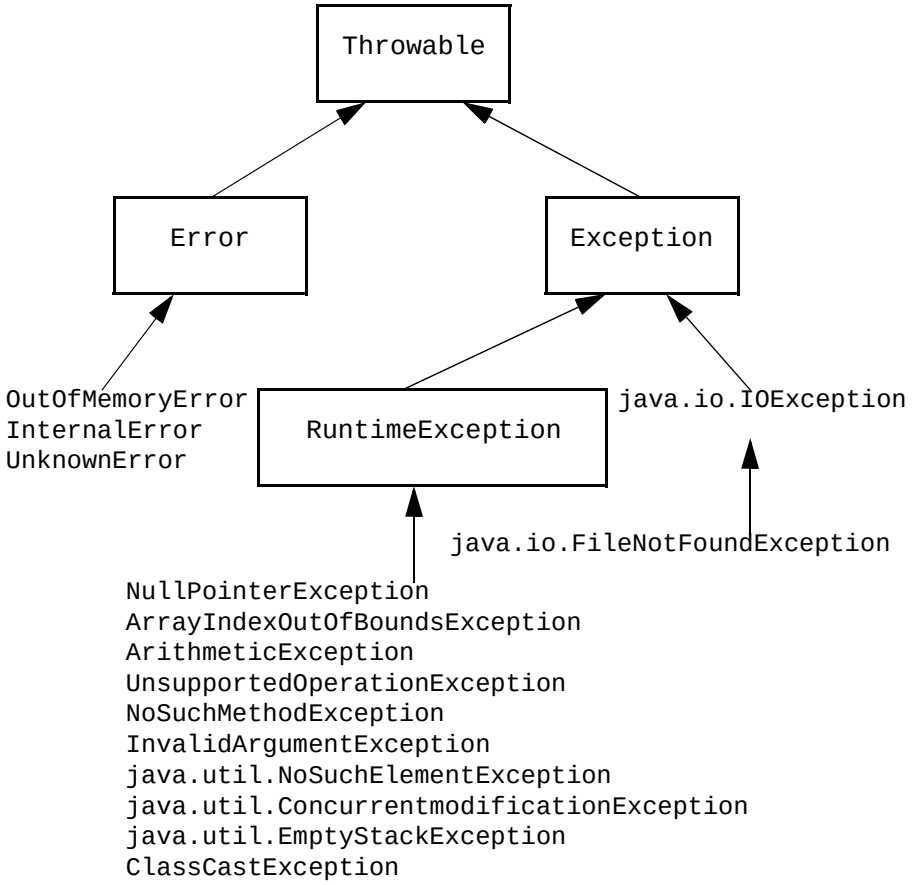

Figure 4.17 The hierarchy of exceptions (partial list)

4.5.2 The Hierarchy of Exceptions

As described in Section 2.5, there are several types of exceptions. The root of the hierarchy, a portion of which is shown in Figure 4.17, is `Throwable`, which defines defines a set of `printStackTrace` methods, provides a `toString` implementation, a pair of constructors, and little else. The hierarchy is split off into `Error`, `RuntimeException`, and checked exceptions. A checked excep-

tion is any `Exception` that is not a `RuntimeException`. For the most part, each new class extends another exception class, providing only a pair of constructors. It is possible to provide more, but none of the standard exceptions bother to do so. In `weiss.util`, we implement three of the standard `java.util` exceptions. One such implementation, which shows that new exception classes typically provide little more than constructors, is shown in Figure 4.18.

```
 1 package weiss.util;
 2
 3 public class NoSuchElementException extends RuntimeException
 4 {
 5     /**
 6      * Constructs a NoSuchElementException with
 7      * no detail message.
 8      */
 9     public NoSuchElementException( )
10     {
11     }
12
13     /*
14      * Constructs a NoSuchElementException with
15      * a detail message.
16      * @param msg the detail mesage.
17      */
18     public NoSuchElementException( String msg )
19     {
20         super( msg );
21     }
22 }
```

Figure 4.18 `NoSuchElementException`, implemented in `weiss.util`.

4.5.3 I/O: The Decorator Pattern

I/O in Java looks fairly complex to use but works nicely for doing I/O with different sources, such as the terminal, files, and Internet sockets. Because it is designed to be extensible, there are lots of classes — over 50 in all. It is cumber-

some to use for trivial tasks; for instance reading a number from the terminal requires substantial work.

Input is done through the use of stream classes. Because Java was designed for Internet programming, most of the I/O centers around byte-oriented reading and writing.

Byte-oriented I/O is done with stream classes that extend `InputStream` or `OutputStream`. `InputStream` and `OutputStream` are abstract classes and not interfaces, so there is no such thing as a stream open for both input and output. These classes declare an abstract `read` and `write` method for single-byte I/O, respectively and also a small set of concrete methods such as `close` and block I/O (which can be implemented in terms of calls to single-byte I/O). Examples of these classes include `FileInputStream` and `FileOutputStream`, as well as the hidden `SocketInputStream` and `SocketOutputStream`. (The socket streams are produced by methods that return an object statically typed as `InputStream` or `OutputStream`).

Character-oriented I/O is done with classes that extend the abstract classes `Reader` and `Writer`. These also contain `read` and `write` methods. There are not as many `Reader` and `Writer` classes as `InputStream` and `OutputStream`.

`InputStreamReader` and `OutputStreamWriter` classes are *bridges* that allows the programmer to cross over from the `Stream` to `Reader` hierarchies.

However, this is not a problem, because of the `InputStreamReader` and `OutputStreamWriter` classes. These are called *bridges* because they cross over from the `Stream` to `Reader` hierarchies. An `InputStreamReader` is

constructed with any `InputStream`, and creates an object that IS-A `Reader`. For instance, we can create a `Reader` for files using:

```
InputStream fis = new FileInputStream( "foo.txt" );
Reader fin = new InputStreamReader( fis );
```

It happens that there is a `FileReader` convenience class that does this already; Figure 4.19 provides a plausible implementation.

From a `Reader`, we can do limited I/O; the `read` method returns one character. If we want to read one line instead, we need a class called `BufferedReader`. Like other `Reader` objects, a `BufferedReader` is constructed from any other `Reader`, but it provides both buffering and a `readLine` method. Thus, continuing the previous example,

```
BufferedReader bin = new BufferedReader( fin );
```

Wrapping an `InputStream` inside an `InputStreamReader` inside a `BufferedReader` works for any `InputStream`, including `System.in` or sockets. Figure 4.20, which duplicates Figure 2.15, illustrates the use of this pattern to read two numbers from the standard input.

The wrapping idea is an example of a commonly-used Java design pattern, that we will see again in Section 4.6.2.

```
1 class FileReader extends InputStreamReader
2 {
3     public FileReader( String name )
4                         throws FileNotFoundException
5       { super( new FileInputStream( name ) ); }
6 }
```

Figure 4.19 `FileReader` convenience class

```
 1 import java.io.InputStreamReader;
 2 import java.io.BufferedReader;
 3 import java.io.IOException;
 4
 5 import java.util.StringTokenizer;
 6
 7 public class MaxTest
 8 {
 9     public static void main( String args[ ] )
10     {
11         BufferedReader in = new BufferedReader( new
12                                 InputStreamReader( System.in ) );
13
14         String oneLine;
15         StringTokenizer str;
16         int x;
17         int y;
18
19         System.out.println( "Enter 2 ints on one line: " );
20         try
21         {
22             oneLine = in.readLine( );
23             if( oneLine == null )
24                 return;
25
26             str = new StringTokenizer( oneLine );
27             if( str.countTokens( ) != 2 )
28             {
29                 System.out.println( "Error: need two ints" );
30                 return;
31             }
32             x = Integer.parseInt( str.nextToken( ) );
33             y = Integer.parseInt( str.nextToken( ) );
34             System.out.println( "Max: " + Math.max( x, y ) );
35         }
36         catch( IOException e )
37           { System.err.println( "Unexpected IO error" ); }
38         catch( NumberFormatException e )
39           { System.err.println( "Error: need two ints" ); }
40
41     }
42 }
```

Figure 4.20 Program that demonstrates the wrapping of streams and readers

Similar to the `BufferReader` is the `PrintWriter`, which allows us to do `println` operations.

The `OutputStream` hierarchy includes several wrappers, such as `DataOutputStream`, `ObjectOutputStream`, and `GZIPOutputStream`.

`DataOutputStream` allows us to write primitives in binary form (rather than human-readable text form); for instance a call to `writeInt` writes the four bytes that represent a 32-bit integer. Writing data the way avoids conversions to text form, resulting in time and (sometimes) space savings. `ObjectOutputStream` allows us to write an entire object including all its components, its component's components, etc., to a stream. The object and all its components must implement the `Serializable` interface. There are no methods in the interface; one must simply declare that a class is serializable.[5] The `GZIPOutputStream` wraps an `OutputStream` and compresses the writes prior to sending it to the `OutputStream`. In addition, there is a `BufferedOutputStream` class. Similar wrappers are found on the `InputStream` side. As an example, suppose we have an array of serializable `Person` objects. We can write the objects, as a unit, compressed as follows:

5. The reason for this is that serialization, by default, is insecure. When an object is written out in an `ObjectOutputStream`, the format is well-known, so its private members can be read by a malicious user. Similarly, when an object is read back in, the data on the input stream is not checked for correctness, so it is possible to read a corrupt object. There are advanced techniques that can be used to ensure security and integriy when serialization is used, but that is beyond the scope of this text. The designers of the serialization library felt that serialization should not be the default because correct use requires knowledge of these issues, and so they placed a small roadblock in the way.

```
Person [] p = getPersons( );   // populate the array
FileOutputStream fout = new FileOutputStream("people.gzip");
BufferedOutputStream bout = new BufferedOutputStream( fout);
GZIPOutputStream gout = new GZIPOutputStream( bout );
ObjectOutputStream oout = new ObjectOutputStream( gout );
oout.writeObject( p );
oout.close( );
```

Later on, we could read everything back:

```
FileInputStream fin = new FileInputStream( "people.gzip" );
BufferedInputStream bin = new BufferedInputStream( fin );
GZIPInputStream gin = new GZIPInputStream( bin );
ObjectInputStream oin = new ObjectInputStream( gin );
Person [] p = (Person[]) oin.readObject( );
oin.close( );
```

The online code expands this example by having each `Person` store a name, a birthdate, and the two `Person` objects that represent the parents.

The idea of nesting wrappers in order to add functionality is known as the *decorator pattern*.

The idea of nesting wrappers in order to add functionality is known as the *decorator pattern*. By doing this, we have numerous small classes that are combined to provide a powerful interface. Without this pattern, each different I/O source would have to have functionality for compression, serialization, character, and byte I/O, etc. With the pattern each source is only responsible for minimal basic I/O, and then the extra features are added on by the decorators.

4.6 Implementing Generic Components

Generic programming allows us to implement type-independent logic.

Recall that an important goal of object-oriented programming is the support of code reuse. An important mechanism that supports this goal is the *generic* mechanism: If the implementation is identical except for the basic type of the object, a *generic implementation* can be used to describe the basic functionality. For

instance, a method can be written to sort an array of items; the *logic* is independent of the types of objects being sorted, so a generic method could be used.

In Java, genericity is obtained by using inheritance.

Unlike many of the newer languages (such as C++, which uses templates to implement generic programming), Java does not support generic implementations directly. This is because generic programming can be implemented using the basic concepts of inheritance. This section describes how generic methods and classes can be implemented in Java using the basic principles of inheritance.[6]

```
1  // MemoryCell class
2  //  Object read( )         -->  Returns the stored value
3  //  void write( Object x ) -->  x is stored
4
5  public class MemoryCell
6  {
7          // Public methods
8      public Object read( )         { return storedValue; }
9      public void write( Object x ) { storedValue = x; }
10
11         // Private internal data representation
12     private Object storedValue;
13 }
```

Figure 4.21 Generic `MemoryCell` class

6. Direct support for generic methods and classes is under strong consideration as a possible language addition. Currently, the approach described in this section is the one most widely used.

```
 1 public class TestMemoryCell
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         MemoryCell m = new MemoryCell( );
 6
 7         m.write( new String( "37" ) );
 8         String val = (String) m.read( );
 9         System.out.println( "Contents are: " + val );
10     }
11 }
```

Figure 4.22 Using the generic `MemoryCell` class

```
1  /**
2   * The SimpleArrayList implements a growable array of Object.
3   * Insertions are always done at the end.
4   */
5  public class SimpleArrayList
6  {
7      /**
8       * Returns the number of items in this collection.
9       * @return the number of items in this collection.
10      */
11     public int size( )
12     {
13         return theSize;
14     }
15
16     /**
17      * Returns the item at position idx.
18      * @param idx the index to search in.
19      * @throws ArrayIndexOutOfBoundsException if index is bad.
20      */
21     public Object get( int idx )
22     {
23         if( idx < 0 || idx >= size( ) )
24             throw new ArrayIndexOutOfBoundsException( );
25         return theItems[ idx ];
26     }
27
28     /**
29      * Adds an item to this collection, at the end.
30      * @param x any object.
31      * @return true (as per java.util.ArrayList).
32      */
33     public boolean add( Object x )
34     {
35         if( theItems.length == size( ) )
36         {
37             Object [ ] old = theItems;
38             theItems = new Object[ theItems.length * 2 + 1 ];
39             for( int i = 0; i < size( ); i++ )
40                 theItems[ i ] = old[ i ];
41         }
42
43         theItems[ theSize++ ] = x;
44         return true;
45     }
46
47     private static final int INIT_CAPACITY = 10;
48
49     private int         theSize = 0;
50     private Object [ ] theItems = new Object[ INIT_CAPACITY ];
51 }
```

Figure 4.23 Simplified `ArrayList`, with `add`, `get`, and `size`.

4.6.1 Using `Object` for Genericity

The basic idea in Java is that we can implement a generic class by using an appropriate superclass, such as `Object`.

Consider the `IntCell` class shown in Figure 3.2. Recall that the `IntCell` supports the `read` and `write` methods. We can, in principle, make this a generic `MemoryCell` class that stores any type of `Object` by replacing instances of `int` with `Object`. The resulting `MemoryCell` class is shown in Figure 4.21.

There are two details that must be considered when we use this strategy. The first is illustrated in Figure 4.22, which depicts a `main` that writes a `"37"` to a `MemoryCell` object and then reads from the `MemoryCell` object. To access a specific method of the object we must downcast to the correct type. (Of course in this example, we do not need the downcast, since we are simply invoking the `toString` method at line 9, and this can be done for any object.

A second important detail is that primitive types cannot be used. Only reference types are compatable with `Object`. A standard workaround to this problem is discussed momentarily.

`MemoryCell` is a fairly small example. To see a larger example that is typical of generic code reuse, Figure 4.23 shows a simplified generic `ArrayList` class; the online code fills in some additional methods.

4.6.2 Wrappers for Primitive Types

When we implement algorithms, often we run into a language typing problem: we have an object of one type, but the language syntax requires an object of a different type.

This technique illustrates the basic theme of a *wrapper class*. One typical use is to store a primitive type, and add operations that the primitive type either does not support or does not support correctly. A second example was seen in the I/O system, in which a wrapper stores a reference to an object and forwards requests to the object, embelleshing the result somehow (for instance, with buffering or compression). A similar concept is an *adapter class* (in fact wrapper and adapter are often used interchangeably). An adapter class is typically used when the interface of a class is not exactly what is needed, and provides a wrapping effect, while changing the interface.

A *wrapper class* stores an entity (the wrapee) and adds operations that the original type does not support correctly. An *adapter class* is used when the interface of a class is not exactly what is needed.

In Java, we have already seen that although every reference type is compatible with `Object`, the eight primitive types are not. As a result, Java provides a wrapper class for each of the eight primitive types. For instance, the wrapper for the `int` type is `Integer`. Each wrapper object is immutable (meaning its state can never change), stores one primitive value that is set when the object is constructed, and provides a method to retrieve the value. The wrapper classes also contain a host of static utility methods.

As an example, Figure 4.24 shows how we can use the `ArrayList` to store integers.

```
 1 import java.util.ArrayList;
 2
 3 class WrapperDemo
 4 {
 5     public static void main( String [] args )
 6     {
 7         ArrayList arr = new ArrayList( );
 8
 9         arr.add( new Integer( 46 ) );
10         Integer wrapperVal = (Integer) arr.get( 0 );
11         int val = wrapperVal.intValue( );
12         System.out.println( "Position 0: " + val );
13     }
14 }
```

Figure 4.24 Illustration of the `Integer` wrapper class.

4.6.3 Adapters: Changing an Interface

The *adapter pattern* is used to change the interface of an existing class to conform to another.

The *adapter pattern* is used to change the interface of an existing class to conform to another. Sometimes it is used to provide a simpler interface, either with fewer methods, or easier-to-use methods. Other times it is used simply to change some method names. In either case, the implementation technique is similar.

We have already seen one example of an adapter: the bridge classes `InputStreamReader` and `OutputStreamReader` that convert byte-oriented streams into character-oriented streams.

As another example, our `MemoryCell` class in Section 4.6.1 uses `read` and `write`. But what if we wanted the interface to use `get` and `put` instead? There are two reasonable alternatives. One is to cut and paste a completely new class. The other is to use composition, in which we design a new class, that wraps the behavior of an existing class.

```
1  // A class for simulating a memory cell.
2  public class StorageCell
3  {
4      public Object get( )
5        { return m.read( ); }
6
7      public void put( Object x )
8        { m.write( x ); }
9
10     MemoryCell m = new MemoryCell( );
11 }
```

Figure 4.25 An adapter class that changes the `MemoryCell` interface to use `get` and `put`.

We use this technique to implement the new class, `StorageCell`, in Figure 4.25. Its methods are implemented by calls to the wrapped `MemoryCell`. It is tempting to use inheritance instead of composition, but inheritance supplements the interface (i.e. it adds additional methods, but leaves the originals.) If that is the appropriate behavior, then indeed inheritance may be preferable to composition.

4.6.4 Using Interface Types for Genericity

Using `Object` as a generic type works only if the operations that are being performed can be expressed using only methods available in the `Object` class.

```
1  class FindMaxDemo
2  {
3      /**
4       * Return max item in a.
5       * Precondition: a.length > 0
6       */
7      public static Comparable findMax( Comparable [] a )
8      {
9          int maxIndex = 0;
10
11         for( int i = 1; i < a.length; i++ )
12             if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
13                 maxIndex = i;
14
15         return a[ maxIndex ];
16     }
17
18     /**
19      * Test findMax on Shape and String objects.
20      */
21     public static void main( String [] args )
22     {
23         Shape [] sh1 = { new Circle(  2.0 ),
24                          new Square(  3.0 ),
25                          new Rectangle( 3.0, 4.0 ) };
26
27         String [] st1 = { "Joe", "Bob", "Bill", "Zeke" };
28
29         System.out.println( findMax( sh1 ) );
30         System.out.println( findMax( st1 ) );
31     }
32 }
```

Figure 4.26 Generic `findMax` routine, with demo using shapes and strings

Consider, for example, the problem of finding the maximum item in an array of items. The basic code is type-independent, but it does require the ability to compare any two objects and decide which is larger and which is smaller. Thus we cannot simply find the maximum of an array of `Object` — we need more information. The simplest idea would be to find the maximum of an array of `Comparable`. To determine order, we can use the `compareTo` method that we

know must be available for all `Comparable`s. The code to do this is shown in Figure 4.26.

It is important to mention a few caveats. First, only objects that implement the `Comparable` interface can be passed as elements of the `Comparable` array. Objects that have a `compareTo` method but do not declare that they implement `Comparable` are not `Comparable`, and do not have the requisite IS-A relationship.

Second, if the `Comparable` array were to have two objects that are incompatible (e.g. a `String` and a `Shape`), the `compareTo` method would throw a `ClassCastException`. This is the expected (indeed, required) behavior.

```
 1 /**
 2  * Simplified version of the Integer class in java.lang.
 3  */
 4 public final class Integer implements Comparable
 5 {
 6     public Integer( )
 7       { this( 0 ); }
 8
 9     public Integer( int x )
10       { value = x; }
11
12     public int intValue( )
13       { return value; }
14
15     public String toString( )
16       { return "" + value; }
17
18     public int compareTo( Object rhs )
19       { return value < ((Integer)rhs).value ? -1 :
20                value == ((Integer)rhs).value ? 0 : 1; }
21
22     public boolean equals( Object rhs )
23       { return rhs instanceof Integer &&
24                value == ((Integer)rhs).value; }
25
26     private int value;
27 }
```

Figure 4.27 Simplified version of the `Integer` class in `java.lang`. Omits static methods and a `hashCode` method.

Third, as before, primitives cannot be passed as `Comparable`s, but the wrappers work because they implement the `Comparable` interface. Figure 4.27 illustrates how the `Integer` class could be implemented by the Java library. This version is missing the static utility methods and also does not include a `hashCode` method that is described in Chapter 6.

Fourth, it is not required that the interface be a standard library interface.

Finally, this solution does not always work, becuase it might be impossible to declare that a class implements a needed interface. For instance, the class might be a library class, while the interface is a user-defined interface. And if the class is

final, we can't even create a new class. The next sections offers another solution for this problem, which is the *function object*. The function object uses interfaces also, and is perhaps one of the central themes encountered in the Java library.

4.7 The Functor (Function Objects)

In Section 4.6, we saw how interfaces can be used to write generic algorithms. As an example, the method in Figure 4.26 can be used to find the maximum item in an array.

However, the `findMax` method has an important limitation. That is, it works only for objects that implement the `Comparable` interface and are able to provide `compareTo` as the basis for all comparison decisions. There are many situations in which this is not feasible. As an example, consider the `SimpleRectangle` class in Figure 4.28.

```
 1 // A simple rectangle class.
 2 public class SimpleRectangle
 3 {
 4     public SimpleRectangle( int l, int w )
 5       { length = l; width = w; }
 6
 7     public int getLength( )
 8       { return length; }
 9
10     public int getWidth( )
11       { return width; }
12
13     public String toString( )
14       { return "Rectangle " + getLength( ) + " by "
15                             + getWidth( ); }
16
17     private int length;
18     private int width;
19 }
```

Figure 4.28 `SimpleRectangle` class that does not implement the `Comparable` interface

The `SimpleRectangle` class does not have a `compareTo` function, and consequently cannot implement the `Comparable` interface. The main reason for this is that because there are many plausible alternatives, it is difficult to decide on a good meaning for `compareTo`. We could base the comparison on area, perimeter, length, width, and so on. Once we write `compareTo`, we are stuck with it. What if we want to have `findMax` work with several different comparison alternatives?

The solution to the problem is to pass the comparison function as a second parameter to `findMax`, and have `findMax` use the comparison function instead of assuming the existence of `compareTo`. Thus `findMax` will now have two parameters: an array of `Object` (which need not have `compareTo` defined), and a comparison function.

The main issue left is how to pass the comparison function. Some languages allow parameters to be functions (actually they are pointers to functions). However, this solution often has efficiency problems and is not available in all object-oriented languages. Java does not allow functions to be passed as parameters; we can only pass primitive value and references. So we appear not to have a way of passing a function.

```
1  package weiss.util;
2
3  /**
4   * Comparator function object interface.
5   */
6  public interface Comparator
7  {
8      /**
9       * Return the result of comparing lhs and rhs.
10      * @param lhs first object.
11      * @param rhs second object.
12      * @return < 0 if lhs is less than rhs,
13      *           0 if lhs is equal to rhs,
14      *         > 0 if lhs is greater than rhs.
15      * @throws ClassCastException if objects
16      *         cannot be compared.
17      */
18     int compare( Object lhs, Object rhs );
19 }
```

Figure 4.29 The `Comparator` interface, originally defined in `java.util` rewritten for the `weiss.util` package.

However, recall that an object consists of data and functions. So we can embed the function in an object, and pass a reference to it. Indeed, this idea works in all object-oriented languages. The object is called a *function object*, and is sometimes also called a *functor*.

Functor is another name for a function object.

The function object class contains a method specified by the generic algorithm. An instance of the class is passed to the algorithm.

The function object often contains no data. The class simply contains a single method, with a given name, that is specified by the generic algorithm (in this case `findMax`). An instance of the class is then passed to the algorithm, which in turn calls the single method of the function object. We can design different comparison functions by simply declaring new classes. Each new class contains a different implementation of the agreed-upon single method.

In Java, to implement this idiom we use inheritance, and specifically we make use of interfaces. The interface is used to declare the signature of the agreed-upon function. As an example, Figure 4.29 shows the `Comparator` interface, which is part of the standard `java.util` package. Recall that to illustrate how the Java library is implemented, we will reimplement a portion of `java.util` as `weiss.util`.

The interface says that any (non-abstract) class that claims to be a `Comparator` must provide an implementation of the `compare` method; thus any object that is an instance of such a class has a `compare` method that it can call.

```
 1 public class Utils
 2 {
 3     // Generic findMax, with a function object.
 4     // Precondition: a.length > 0.
 5     public static Object findMax( Object [ ] a,
 6                                   Comparator cmp )
 7     {
 8         int maxIndex = 0;
 9         for( int i = 1; i < a.length; i++ )
10             if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
11                 maxIndex = i;
12
13         return a[ maxIndex ];
14     }
15 }
```

Figure 4.30 Generic `findMax` algorithm, using a function object

```
 1 class OrderRectByWidth implements Comparator
 2 {
 3     public int compare( Object obj1, Object obj2 )
 4     {
 5         SimpleRectangle r1 = (SimpleRectangle) obj1;
 6         SimpleRectangle r2 = (SimpleRectangle) obj2;
 7
 8         return( r1.getWidth() - r2.getWidth() );
 9     }
10 }
11
12 public class ConpareTest
13 {
14     public static void main( String [ ] args )
15     {
16         Object [ ] rects = new Object[ 4 ];
17         rects[ 0 ] = new SimpleRectangle( 1, 10 );
18         rects[ 1 ] = new SimpleRectangle( 20, 1 );
19         rects[ 2 ] = new SimpleRectangle( 4, 6 );
20         rects[ 3 ] = new SimpleRectangle( 5, 5 );
21
22         System.out.println( "MAX WIDTH: " +
23              Utils.findMax( rects, new OrderRectByWidth( ) ) );
24     }
25 }
```

Figure 4.31 Example of a function object

Using this interface, we can now pass a `Comparator` as the second parameter to `findMax`. If this `Comparator` is `cmp`, we can safely make the call

`cmp.compare(o1,o2)` to compare any two objects as needed. It is up to the caller of `findMax` to pass an appropriately implemented instance of `Comparator` as the actual argument.

An example is shown in Figure 4.30. `findMax` now takes two parameters. The second parameter is the function object. As shown on line 10, `findMax` expects that the function object implements a method named `compare`, and it must do so, since it implements the `Comparator` interface.

Once `findMax` is written, it can be called in `main`. To do so, we need to pass to `findMax` an array of `SimpleRectangle` objects and a function object that implements the `Comparator` interface. We implement a new class `OrderRectByWidth`, which contains the required `compare` method. `compare` returns a integer indicating if the first rectangle is less than, equal to, or greater than the second rectangle on the basis of widths. `main` simply passes an instance of `OrderRectByWidth` to `findMax`.[7] Both `main` and `OrderRectByWidth` are shown in Figure 4.31. Observe that the `OrderRectByWidth` object has no data members. This is usually true of function objects.

The function object technique is an illustration of a pattern that we see over and over again, not just in Java, but in any language that has objects. In Java, this

7. The trick of implementing `compare` by subtracting works for `int`s as long as both are the same sign. Otherwise there is a possibility of overflow.

pattern is used over and over and over again, and represents perhaps the single dominant idiomatic use of interfaces.

4.7.1 Nested Classes

Generally speaking, when we write a class, we expect, or at least hope, for it to be useful in many contexts, not just the particular application that is being worked on.

An annoying feature of the function object pattern, especially in Java, is the fact that because it is used so often, it results in the creation of numerous small classes, that each contain one method, that are used perhaps only once in a program, and that have limited applicability outside of the current application.

This is annoying for at least two reasons. First, we might have dozens of function object classes. If they are public, by rule they are scattered in separate files. If they are package visible, they might all be in the same file, but we still have to scroll up and down to find their definitions, which is likely to be far removed from the one or perhaps two places in the entire program where they are instantiated as function objects. It would be preferable if each function object class could be declared as close as possible to its instantiation. Second, once a name is used, it cannot be reused in the package without possibilities of name collisions. Although packages solve some namespace problems, they do not solve them all, especially when the same class name is used twice in the default package.

A *nested class* is a class declaration that is placed inside another class declaration – the outer class – using the keyword `static`.

With a nested class, we can solve some of these problems. A *nested class* is a class declaration that is placed inside another class declaration – the outer class – using the keyword `static`. A nested class is considered a member of the outer class. As a result, it can be public, private, package visible, or protected, and depending on the visibility, may or may not be accessible by methods that are not part of the outer class. Typically, it is private, and thus inaccessible from outside the outer class. Also, because a nested class is a member of the outer class, its methods can access private static members of the outer class, and can access private instance members when given a reference to an outer object.

A nested class is a part of the outer-class and can be declared with a visiblity specifier. All outer class members are visible to the nested class' methods.

Figure 4.32 illustrates the use of a nested class in conjunction with the function object pattern. The `static` in front of the nested class declaration of `OrderRectByWidth` is essential; without it, we have an inner class, which behaves differently and is discussed later.

Occasionally, a nested class is public. In Figure 4.32, if `OrderRectByWidth` was declared public, the class `CompareTestInner1.OrderRectByWidth` could be used from outside of the `CompareTestInner1` class.

```
1  class CompareTestInner1
2  {
3      private static class OrderRectByWidth implements Comparator
4      {
5          public int compare( Object obj1, Object obj2 )
6          {
7              SimpleRectangle r1 = (SimpleRectangle) obj1;
8              SimpleRectangle r2 = (SimpleRectangle) obj2;
9
10             return( r1.getWidth() - r2.getWidth() );
11         }
12     }
13
14     public static void main( String [ ] args )
15     {
16         Object [ ] rects = new Object[ 4 ];
17         rects[ 0 ] = new SimpleRectangle( 1, 10 );
18         rects[ 1 ] = new SimpleRectangle( 20, 1 );
19         rects[ 2 ] = new SimpleRectangle( 4, 6 );
20         rects[ 3 ] = new SimpleRectangle( 5, 5 );
21
22         System.out.println( "MAX WIDTH: " +
23              Utils.findMax( rects, new OrderRectByWidth( ) ) );
24     }
25 }
```

Figure 4.32 Using a nested class to hide `OrderRectByWidth` class declaration

```
 1 class CompareTestInner2
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         Object [ ] rects = new Object[ 4 ];
 6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
 7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
 8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
 9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11         class OrderRectByWidth implements Comparator
12         {
13             public int compare( Object obj1, Object obj2 )
14             {
15                 SimpleRectangle r1 = (SimpleRectangle) obj1;
16                 SimpleRectangle r2 = (SimpleRectangle) obj2;
17
18                 return( r1.getWidth() - r2.getWidth() );
19             }
20         }
21
22         System.out.println( "MAX WIDTH: " +
23              Utils.findMax( rects, new OrderRectByWidth( ) ) );
24     }
25 }
```

Figure 4.33 Using a method class to hide `OrderRectByWidth` class declaration further

4.7.2 Local Classes

Java also allows class declarations inside of methods. Such classes are known as *local classes* and may not be declared with a visiblity modifier, and may not be declared using the static modifier.

In addition to allowing class declarations inside of classes, Java also allows class declarations inside of methods. These classes are called *local classes*. This is illustrated in Figure 4.33.

Note that when a class is declared inside a method, it cannot be declared `private` or `static`. However, the class is visible only inside of the method in which it was declared. This makes it easier to write the class right before its first (perhaps only) use and avoid pollution of namespaces.

An advantage of declaring a class inside of a method is that the class' methods (in this case `compare`) has access to local variables of the function that are

declared prior to the class. This can be important in some applications. There is a technical rule: in order to access local variables, the variables must be declared `final`. We will not be using these types of classes in the text.

4.7.3 Anonymous Classes

One might suspect that by placing a class immediately before the line of code in which it is used, we have declared the class as close as possible to its use. However, in Java, we can do even better.

An *anonymous class* is a class that has no name.

Figure 4.34 illustrates the anonymous inner class. An *anonymous class* is a class that has no name. The syntax is that instead of writing `new Inner()`, and providing the implementation of `Inner` as a named class, we write `new Interface()`, and then provide the implementation of the interface (everything from the opening to closing brace) immediately after the `new` expression. Instead of implementing an interface anonymously, it is also possible to extend a class anonymously, providing only the overridden methods.

Anonymous classes introduce significant language complications.

The syntax looks very daunting, but after a while, one gets used to it. It complicates the language significantly, because the anonymous class is a class. As an example of the complications that are introduced, since the name of a constructor is the name of a class, how does one define a constructor for an anonymous class? The answer is that you cannot do so.

```
 1 class CompareTestInner3
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         Object [ ] rects = new Object[ 4 ];
 6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
 7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
 8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
 9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11         System.out.println( "MAX WIDTH: " +
12         Utils.findMax( rects, new Comparator( )
13             {
14                 public int compare( Object obj1, Object obj2 )
15                 {
16                    SimpleRectangle r1 = (SimpleRectangle) obj1;
17                    SimpleRectangle r2 = (SimpleRectangle) obj2;
18
19                    return( r1.getWidth() - r2.getWidth() );
20                 }
21             }
22         ) );
23     }
24 }
```

Figure 4.34 Using an anonymous class to implement the function object

Anonymous classes are often used to implement function objects.

The anonymous class is in practice very useful, and its use is often seen as part of the function object pattern in conjunction with event-handling in user interfaces. In event handling, the programmer is required to specify, in a function, what happens when certain events occur.

4.8 Dynamic Binding Details

Dynamic binding is not important for static, final, or private methods.

A common myth is that all methods and all parameters are bound at runtime. This is not true. First, there are some cases in which dynamic binding is never used or is not an issue:

- static methods, regardless of how the method is invoked
- final methods
- private methods (since they are only invoked from inside the class, and are thus impliclty final)

In other scenarios, dynamic binding is meaningfully used. But what exactly does dynamic binding mean?

In Java, the parameters to a method are always deduced statically, at compile time.

Dynamic binding means that the method that is appropriate for the object being operated on is the one that is used. However, it does not mean that the absolute best match is performed for all parameters. Specifically, in Java, the parameters to a method are always deduced statically, at compile time.

For a concrete example, consider the code in Figure 4.35. In the `whichFoo` method, a call is made to `foo`. But which `foo` is called? We expect the answer to depend on the runtime types of `arg1` and `arg2`.

Because parameters are always matched at compile time, it does not matter what type `arg2` is actually referencing. The `foo` that is matched will be

```
public void foo( Base x ) { /* */ }
```

The only issue is whether the `Base` or `Derived` version is used That is the decision that is made at runtime, when the object that `arg1` references is known.

Static overloading means that the parameters to a method are always deduced statically, at compile time.

The precise methodology used is that the compiler deduces, at compile time, the best signature, based on the static types of the parameters and the methods that are available for the static type of the controlling reference. At that point, the signature of the method is set. This step is called *static overloading*. The only remaining issue is which class' version of that method is used. This is done by having the virtual machine deduce the runtime type of this object. Once the runtime type is known, the virtual machine walks up the inheritance hierarchy, looking for the last overridden version of the method; this is the first method of the appropriate signature that the virtual machine finds as it walks up toward `Object`.[8] This second step is called *dynamic binding*.

Dynamic binding means that once the signature of an instance method is ascertained, the class of the method can be determined at runtime based on the dynamic type of the invoking object.

Static overloading can lead to subtle errors when a method that is supposed to be overridden is instead overloaded. Figure 4.36 illustrates a common programming error that occurs when implementing the `equals` method.

8. If no such method is found, perhaps because only part of the program was recompiled, then the virtual machine throws a `NoSuchMethodException`.

```
1  class Base
2  {
3      public void foo( Base x )
4        { System.out.println( "Base.Base" ); }
5
6      public void foo( Derived x )
7        { System.out.println( "Base.Derived" ); }
8  }
9
10 class Derived extends Base
11 {
12     public void foo( Base x )
13       { System.out.println( "Derived.Base" ); }
14
15     public void foo( Derived x )
16       { System.out.println( "Derived.Derived" ); }
17 }
18
19 class StaticParamsDemo
20 {
21     public static void whichFoo( Base arg1, Base arg2 )
22     {
23         // It is guaranteed that we will call foo( Base )
24         // Only issue is which class's version of foo( Base )
25         // is called; the dynamic type of arg1 is used
26         // to decide.
27         arg1.foo( arg2 );
28     }
29
30     public static void main( String [] args )
31     {
32         Base b = new Base( );
33         Derived d = new Derived( );
34
35         whichFoo( b, b );
36         whichFoo( b, d );
37         whichFoo( d, b );
38         whichFoo( d, d );
39     }
40 }
```

Figure 4.35 Illustration of static binding for parameters

The `equals` method is defined in class `Object` and is intended to return true if two objects have identical states. It takes an `Object` as parameter, and `Object` provides a default implementation that returns true only if the two

objects are the same. In other words, in class `Object`, the implementation of `equals` is roughly

```
public boolean equals( Object other )
  { return this == other; }
```

When overridding `equals`, the parameter must be of type `Object`; otherwise overloading is being done. In Figure 4.36, `equals` is not overridden; instead it is (unintentionally) overloaded. As a result, the call to `sameVal` will return false, which appears surprising, since the call to `equals` returns true and `sameVal` calls `equals`.

```
1  final class SomeClass
2  {
3      public SomeClass( int i )
4        { id = i; }
5
6      public boolean sameVal( Object other )
7        { return other instanceof SomeClass && equals( other ); }
8
9      /**
10      * This is a bad implementation!
11      * other has the wrong type, so this does
12      * not override Object's equals.
13      */
14     public boolean equals( SomeClass other )
15       { return other != null && id == other.id; }
16
17     private int id;
18 }
19
20 class BadEqualsDemo
21 {
22     public static void main( String [ ] args )
23     {
24         SomeClass obj1 = new SomeClass( 4 );
25         SomeClass obj2 = new SomeClass( 4 );
26
27         System.out.println( obj1.equals( obj2 ) );
28         System.out.println( obj1.sameVal( obj2 ) );
29     }
30 }
```

Figure 4.36 Illustration of overloading `equals` instead of overriding `equals`. Here, the call the `sameVal` returns false!

The problem is that the call in `sameVal` is, `this.equals(other)`. The static type of `this` is `SomeClass`. In `SomeClass` there are two versions of `equals`: the listed `equals` that takes a `SomeClass` as a parameter, and the inherited `equals` that takes an `Object`. The static type of the parameter (other) is `Object`, so the best match is the `equals` that takes an `Object`. At runtime the virtual machine searches for that `equals`, and finds the one in class

`Object`. And since `this` and `other` are different objects, the `equals` method in class `Object` returns false.

Thus, `equals` must be written to take an `Object` as a parameter, and typically a downcast will be required after a verification that the type is appropriate. One way of doing that is to use an `instanceof` test, but that is safe only for final classes. Overriding `equals` is actually fairly tricky in the presence of inheritance, and is discussed in Section 6.7.

Summary

Inheritance is a powerful feature that is an essential part of object-oriented programming and Java. It allows us to abstract functionality into abstract base classes and have derived classes implement and expand on that functionality. Several types of methods can be specified in the base class, as illustrated in Figure 4.37.

The most abstract class, in which no implementation is allowed, is the *interface*. The interface lists methods that must be implemented by a derived class. The derived class must both implement all of these methods (or itself be abstract) and specify, via the *`implements`* clause, that it is implementing the interface. Multiple interfaces may be implemented by a class, thus providing a simpler alternative to multiple inheritance.

Method	Overloading	Comments
`final`	Potentially inlined	Invariant over the inheritance hierarchy (method is never redefined).

Figure 4.37 Four types of class methods

Method	Overloading	Comments
abstract	Run time	Base class provides no implementation and is abstract. Derived class must provide an implementation.
static	Compile time	No controlling object.
Other	Run time	Base class provides a default implementation that may be either overridden by the derived classes or accepted unchanged by the derived classes.

Figure 4.37 Four types of class methods

Finally, inheritance allows us to easily write generic methods and classes that work for a wide range of generic types. This will typically involve using type conversion operators. Interfaces are also widely used for generic components, and to implement the function object pattern.

This chapter concludes a discussion that provided an overview of Java and object-oriented programming. This chapter concludes the first part of the text, which provided an overview of Java and object-oriented programming. We will now go on to look at algorithms and the building blocks of problem-solving.

Objects of the Game

abstract class A class that cannot be constructed but serves to specify functionality of derived classes. (172)

abstract method A method that has no meaningful definition and is thus always defined in the derived class. (170)

Adapter A class that is typically used when the interface of another class is not exactly what is needed. The adapter provides a wrapping effect, while changing the interface. (191)

anonymous class A class that has no name and is useful for implemented short function objects. (207)

base class The class on which the inheritance is based. (149)

composition Preferred mechanism to inheritance when an IS-A relationship does not hold. Instead, we say that an object of class *B* is composed of an object of class *A* (and other objects). (143)

decorator pattern The pattern that involves the combining of several wrappers in order to add functionality. (186)

derived class A completely new class that nonetheless has some compatibility with the class from which it was derived. (149)

dynamic binding A run-time decision to apply the method corresponding to the actual referenced object. (153)

`extends` clause A clause used to declare that a new class is a subclass of another class. (150)

final class A class that may not be extended. (159)

final method A method that may not be overridden and is invariant over the inheritance hierarchy. Static binding is used for final methods. (157)

function object An object passed to a generic function with the intention of having its single method used by the generic function. (200)

Functor A function object. (199)

generic programming Used to implement type-independent logic. (186)

HAS-A relationship A relationship in which the derived class has a (instance of the) base class. (143)

`implements` clause A clause used to declare that a class implements the methods of an interface. (175)

inheritance The process whereby we may derive a class from a base class without disturbing the implementation of the base class. Also allows the design of class hierarchies, such as `Exception` and `InputStream`. (149)

interface A special kind of abstract class that contains no implementation details. (174)

IS-A relationship A relationship in which the derived class is a (variation of the) base class. (142)

leaf class A final class. (159)

local class A class inside a method, declared with no visibility modifier. (206)

multiple inheritance The process of deriving a class from several base classes. Multiple inheritance is not allowed in Java. However, the alternative, multiple interfaces, is allowed. (173)

nested class A class inside a class, declared with the static modifier. (204)

partial overriding The act of augmenting a base class method to perform additional, but not entirely different, tasks. (160)

polymorphism The ability of a reference variable to reference objects of several different types. When operations are applied to the variable, the operation that is appropriate to the actual referenced object is automatically selected. (153)

protected class member Accessible by the derived class and classes in the same package. (155)

static binding The decision on which class' version of a method to use is made at compile time. Is only used for static, final, or private methods. (158)

static overloading The first step for deducing the method that will be used. In this step, the static types of the parameters are used to deduce the signature of the method that will be invoked. Static overloading is always used. (210)

subclass/superclass relationships If *X* IS-A *Y*, then *X* is a subclass of *Y* and *Y* is a superclass of *X*. These relationships are transitive. (142)

`super` constructor call A call to the base class constructor. (157)

`super` object An object used in partial overloading to apply a base class method. (160)

Wrapper A class that is used to store another type, and add operations that the primitive type either does not support or does not support correctly. (191)

Common Errors

1. Private members of a base class are not visible in the derived class.
2. Objects of an abstract class cannot be constructed.

3. If the derived class fails to implement any inherited abstract method, then the derived class becomes abstract. If this was not intended, a compiler error will result.
4. Final methods may not be overridden. Final classes may not be extended.
5. Static methods use static binding, even if they are overridden in a derived class.
6. Java uses static overloading and always selects the signature of an overloaded method at compile time.
7. In a derived class, the inherited base class members should only be initialized as an aggregate by using the `super` method. If these members are public or protected, they may later be read or assigned to individually.
8. When you send a function object as a parameter, you must send a constructed object, and not simply the name of the class.
9. Overusing anonymous classes is a common error.
10. The throws list for a method in a derived class cannot be redefined to throw an exception not thrown in the base class. Return types must also match.
11. When a method is overridden it is illegal to reduce its visibility. This is also true when implementing interface methods, which by definition are always `public`.
12. If a generic method returns a generic reference, then typically a type conversion must be used to obtain the actual returned object.

On the Internet

All of the chapter code is available online. Some of the code was presented in stages; for those classes, only one finalized version is provided.

PersonDemo.java	The `Person` hierarchy and test program.
Shape.java	The abstract `Shape` class.
Circle.java	The `Circle` class.
Square.java	The `Square` class.
Rectangle.java	The `Rectangle` class.
ShapeDemo.java	A test program for the `Shape` example.
NoSuchElementException.java	The exception class in Figure 4.18. This is part of `weiss.util`. Also online is **ConcurrentModificationException.java** and **EmptyStackException.java**.
DecoratorDemo.java	An illustration of the decorator pattern, including buffering, compression, and serialization.
MemoryCell.java	The `MemoryCell` class in Figure 4.21.
TestMemoryCell.java	The test program for the memory cell class shown in Figure 4.22.
SimpleArrayList.java	The generic simplified `ArrayList` class in Figure 4.23, with some additional methods. A test program is provided in **ReadStringsWithSimpleArrayList.java**.

PrimitiveWrapperDemo.java Demonstrates the use of the `Integer` class, as shown in Figure 4.24.

StorageCellDemo.java The `StorageCell` adapter as shown in Figure 4.25, and a test program.

FindMaxDemo.java The `findMax` generic algorithm in Figure 4.26.

SimpleRectangle.java Contains the `SimpleRectangle` class Figure 4.28.

Comparator.java The `Comparator` interface in Figure 4.29.

CompareTest.java Illustrates the function object, with no nested classes, as shown in Figure 4.31.

CompareTestInner1.java Illustrates the function object, with a nested class, as shown in Figure 4.32.

CompareTestInner2.java Illustrates the function object, with a nested class inside a method, as shown in Figure 4.33.

CompareTestInner3.java Illustrates the function object, with an anonymous class, as shown in Figure 4.34.

StaticParamsDemo.java The demonstration of static overloading and dynamic binding shown in Figure 4.35.

BadEqualsDemo.java Illustrates the consequences of overloading instead of overriding `equals`, as shown Figure 4.36.

Exercises

In Short

4.1. What members of an inherited class can be used in the derived class? What members become public for users of the derived class?

4.2. What is composition?

4.3. Explain polymorphism.

4.4. Explain dynamic binding. When is dynamic binding not used?

4.5. What is a final method?

4.6. Consider the program to test visibility in Figure 4.38.

a. Which accesses are illegal?

b. Make `main` a method in `Base`. Which accesses are illegal?

c. Make `main` a method in `Derived`. Which accesses are illegal?

d. How do these answers change if `protected` is removed from line 4?

e. Write a three-parameter constructor for `Base`. Then write a five-parameter constructor for `Derived`.

```
 1 public class Base
 2 {
 3     public    int bPublic;
 4     protected int bProtect;
 5     private   int bPrivate;
 6     // Public methods omitted
 7 }
 8
 9 public class Derived extends Base
10 {
11     public    int dPublic;
12     private   int dPrivate;
13     // Public methods omitted
14 }
15
16 public class Tester
17 {
18     public static void main( String [ ] args )
19     {
20         Base b    = new Base( );
21         Derived d = new Derived( );
22
23         System.out.println( b.bPublic + " " + b.bProtect + " "
24                             + b.bPrivate + " " + d.dPublic + " "
25                             + d.dPrivate );
26     }
27 }
```

Figure 4.38 Program to test visibility

f. The class `Derived` consists of five integers. Which are accessible to the class `Derived`?

g. A method in the class `Derived` is passed a `Base` object. Which of the `Base` object members can the `Derived` class access?

4.7. What is the difference between a final class and other classes? Why are final classes used?

4.8. What is an abstract method?

4.9. What is an abstract class?

4.10. What is an interface? How does the interface differ from an abstract class? What members may be in an interface?

4.11. Explain the design of the Java I/O library. Include a class hierarchy picture for all the classes described in Section 4.5.3.

4.12. How are generic algorithms implemented in Java?

4.13. Explain the Adapter and Wrapper patterns. How do they differ?

4.14. What are two common ways to implement adapters? What are the trade-offs between these implementation methods?Describe how function objects are implemented in Java.

4.15. What is a local class?

4.16. What is an anonymous class?

In Theory

4.17. A local class can access local variables that are declared in that method (prior to the class). Show that if this is allowed, it is possible for an instance of the local class to access the value of the local variable, even after the method has terminated. (For this reason, the compiler will allocate these local variables from a different source than usual, and as a consequence of that, the compiler will insist that these variables are immutable.)

4.18. This exercise explores how Java performs dynamic binding, and also why trivial final methods may not be inlined at compile time. Place each of the classes in Figure 4.39 in its own file:

```
 1 public class Class1
 2 {
 3     public static int x = 5;
 4
 5     public final String getX( )
 6       { return "" + x + 12; }
 7 }
 8
 9 public class Class2
10 {
11     public static void main (String[] args)
12     {
13         Class1 obj = new Class1( );
14         System.out.println( obj.getX( ) );
15     }
16 }
```

Figure 4.39 Classes for Exercise 4.18

a. Compile `Class2` and run the program. What is the output?

b. What is the exact signature (including return type) of the `getX` method that is deduced at compile time at line 14?

c. Change the `getX` routine at line 5 to return an `int`; remove the `""` from the body at line 6, and recompile `Class2`. What is the output?

d. What is the exact signature (including return type) of the `getX` method that is now deduced at compile time at line 14?

e. Change `Class1` back to its original, but recompile `Class1` only. What is the result of running the program?

f. What would the result have been had the compiler been allowed to perform inline optimization?

In Practice

4.19. Write a generic `find` routine that searches an array of `Object` for an `Object x`, returning the first item that matches (as declared by `equals` returning true).

4.20. Write generic method `min` and `max`, each of which accepts two `Comparable` parameters and returns the smaller and larger, respectively. Then use those methods on the `String` type.

4.21. Write generic methods `min`, which accepts an array of `Comparable`, and returns the smallest item. Then use the method on the `String` type.

4.22. Write generic method `max2`, which accepts an array of `Comparable` and returns an array of two `Comparable`s representing the two largest items in the array. The input array should be unchanged. Then use those methods on the `String` type.

4.23. Write generic method `sort`, which accepts an array of `Comparable` and rearranges the array in nondecreasing sorted order. Test your method on both `String` and `Shape`.

4.24. For the `Shape` example, modify the constructors in the hierarchy to throw an `InvalidArgumentException` when the parameters are negative.

4.25. Modify the `Person` class so that it can use `findMax` to obtain the alphabetically last person.

4.26. A `SingleBuffer` supports `get` and `put`: The `SingleBuffer` stores a single item and a data member that indicates whether the `SingleBuffer` is logically empty. A `put` may be applied only to an

empty buffer, and it inserts an item into the buffer. A `get` may be applied only to a nonempty buffer, and it deletes and returns the contents of the buffer. Write a generic class to implement `SingleBuffer`. Define an exception to signal errors.

4.27. A `SortedArrayList` stores a collection of `Comparable`. It is like `ArrayList`, except that `add` will place the item in the correct sorted order instead of at the end; however, at this point it will be difficult for you to use inheritance. Instead, implement a separate `SortedArrayList` that supports `add`, `get`, `remove`, and `size`.

4.28. This exercise asks you to write a generic `countMatches` function. Your function will take two parameters. The first parameter is an array of `int`. The second parameter is a function object that returns a Boolean.

a. Give a declaration for an interface that expresses the requisite function object.

b. `countMatches` returns the number of array items for which the function object returns true. Implement `countMatches`.

c. Test `countMatches` by writing a function object, `EqualsZero`, that implements your interface to accept one parameter and returns true if the parameter is equal to zero. Use an `EqualsZero` function object to test `countMatches`.

4.29. Although the function objects we have looked at store no data, this is not a requirement.

a. Give a declaration for an interface that expresses the requisite function object.

b. Write a function object `EqualsK`. `EqualsK` contains one data member (`k`). `EqualsK` is constructed with a single parameter (default is zero) that is used to initialize `k`. Its method returns true if the parameter is equal to `k`.

c. Use `EqualsK` to test `countMatches` in Exercise 4.28 (c).

Programming Projects

4.30. Rewrite the `Shape` hierarchy to store the area as a data member and have it computed by the `Shape` constructor. The constructors in the derived classes should compute an area and pass the result to the `super` method. Make `area` a final method that returns only the value of this data member.

4.31. Add the concept of a position to the `Shape` hierarchy by including coordinates as data members. Then add a `distance` method.

4.32. Write an abstract class for `Date` and its derived class `GregorianDate`.

4.33. Implement a taxpayer hierarchy that consists of a `TaxPayer` interface and the classes `SinglePayer` and `MarriedPayer` that implement the interface.

4.34. Implement a *gzip* and *gunzip* program that performs compression and uncompression of files.

References

The following books describe the general principles of object-oriented software development:

1. G. Booch, *Object-Oriented Design and Analysis with Applications (Second Edition)*, Benjamin/Cummings, Redwood City, Calif., 1994.

2. T. Budd, *Understanding Object-Oriented Programming With Java*, Addison-Wesley, Reading, Mass., 2001.

3. D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, Mass., 1993.

4. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach* (revised fourth printing), Addison-Wesley, Reading, Mass., 1992.

5. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, N.J., 1988.