

## 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 commercial 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.

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.

*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.

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.

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.

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 maintenance and versioning. Suppose we decide in the second version that it is better to store names in last name, first name format,

```
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** The Person class stores name, age, address, and phone number.

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.

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 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** The Student class stores name, age, address, phone number, and gpa via copy-and-paste.

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.

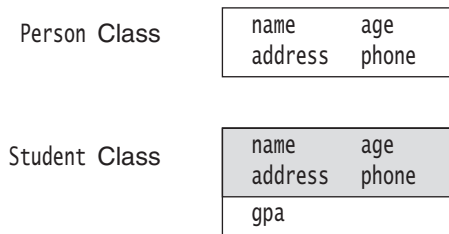
Figure 4.3 shows the Student class. The data layout for the Person and Student classes is shown in Figure 4.4. It illustrates that the memory footprint of any Student object includes all fields that would be contained in a Person object. However, because those fields are declared private by Person, they 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

```

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



**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 the Student class indicates fields that are not accessible in the Student class, but are nonetheless present.

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.

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

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.

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

Let us summarize the syntax 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.

```

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 members
12    // Additional data fields (generally private)
13    // Additional private methods
14 }
```

Figure 4.5 The general layout of inheritance

- Any base class method that is defined in the derived class's public section is overridden. The new definition will be applied to objects of the derived class.
- 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.

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

## 4.1.2 Type Compatibility

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.

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( ) );
```

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

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 compatibility 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( getTodayDay( ).equals( "Tuesday" ) )
    p = s;
else
    p = e;
System.out.println( "Person is " + p.toString( ) );
```

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.

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 `toString` method that is used will be the one appropriate for the dynamic type of the controlling object reference.

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.

#### 4.1.4 Inheritance Hierarchies

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.

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

#### 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 package visible access (if the base and derived classes are in the same package), as appropriate. However, this allows access by other classes in addition to derived classes.

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.<sup>1</sup> 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. 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.

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

## 4.1.6 The Constructor and super

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

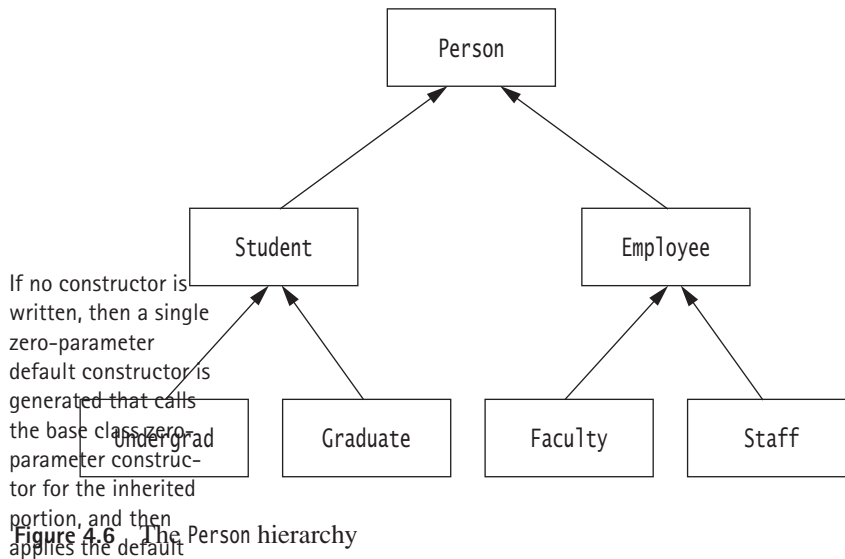


Figure 4.6 The Person hierarchy

1. The rule for protected visibility is quite complex. A protected member of class B is visible to all methods in any classes 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.

```

1 class Demo extends java.io.FilterInputStream
2 {
3     // FilterInputStream has protected data field named in
4     public void foo( )
5     {
6         java.io.FilterInputStream b = this; // legal
7         System.out.println( in );          // legal
8         System.out.println( this.in );     // legal
9         System.out.println( b.in );       // illegal
10    }
11 }

```

```

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** A constructor for new class Student; uses super

tells us that the inherited portion is a single entity, and the base class constructor tells us how to initialize this single entity.

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.

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.

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

### 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.

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.

A `final` method is invariant over the inheritance hierarchy and may 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.

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.

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.

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.<sup>2</sup> Static methods have no controlling object and thus are resolved at compile time using static binding.

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.

## 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.<sup>3</sup> 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

---

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 run time, rather than the compiler at compile time. This ensures that dependent classes do not get out of sync as a result of the optimization.

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.

the spirit of an IS-A relationship. Thus you may not override a public method with a package-visible method.

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:

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

## 4.1.9 Type Compatibility 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 each 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 );
```

```
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

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

The above code 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*.

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 run-time exception that signals a pro-

```

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** An illustration of polymorphism with arrays

gramming error. Casts are always double-checked at run time 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 the following:

```
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

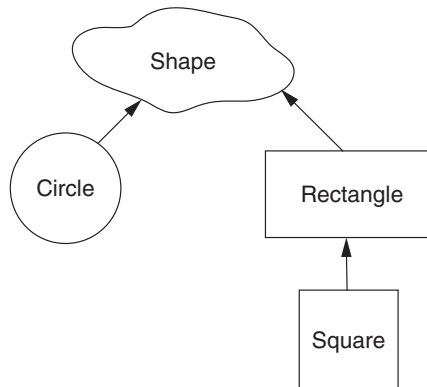


Figure 4.10 The hierarchy of shapes used in an inheritance example

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

Figure 4.11 A 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

also include a perimeter method. Observe that Square reuses code inherited from Rectangle.

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 instances of operators is a symptom 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 the following:

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

```

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 [ ] arr )
15     {
16         for( int i = 0; i < arr.length; i++ )
17             System.out.println( arr[ i ] );
18     }
19
20     public static void main( String [ ] args )
21     {
22         Shape [ ] a = { 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** A sample program that uses the shape hierarchy

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 run-time 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.<sup>4</sup>

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 because of a typing error: An Area function is written instead of area, making it difficult to track down the error at run time.

A better solution for area is to throw a run-time 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 run time. 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* 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 explicitly be declared as such. When a

Abstract methods and classes represent placeholders.

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

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

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 override 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`.

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.

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

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

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

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

1. *Final methods*. The virtual machine may choose at run time to perform inline optimization, thus avoiding dynamic binding. We use a

```
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

- 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.* Overriding 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

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*.

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

## 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.

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 usually are omitted.

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

Figure 4.15 The `Comparable` interface

## 4.4.2 Implementing an Interface

A class implements an interface by

1. Declaring that it implements the interface
2. Defining implementations for all the interface methods

An example is shown in Figure 4.16. Here, we complete 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.

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

```

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
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-compatible with *I*. If a class *C* implements interfaces *I*<sub>1</sub>, *I*<sub>2</sub>, and *I*<sub>3</sub>, then *C* IS-A *I*<sub>1</sub>, *C* IS-A *I*<sub>2</sub>, and *C* IS-A *I*<sub>3</sub>, and is type-compatible with *I*<sub>1</sub>, *I*<sub>2</sub>, and *I*<sub>3</sub>.
2. The instanceof operator can be used to determine if a reference is type-compatible 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 a 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 the 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's "hash code."

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.

### 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 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 exception 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.

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

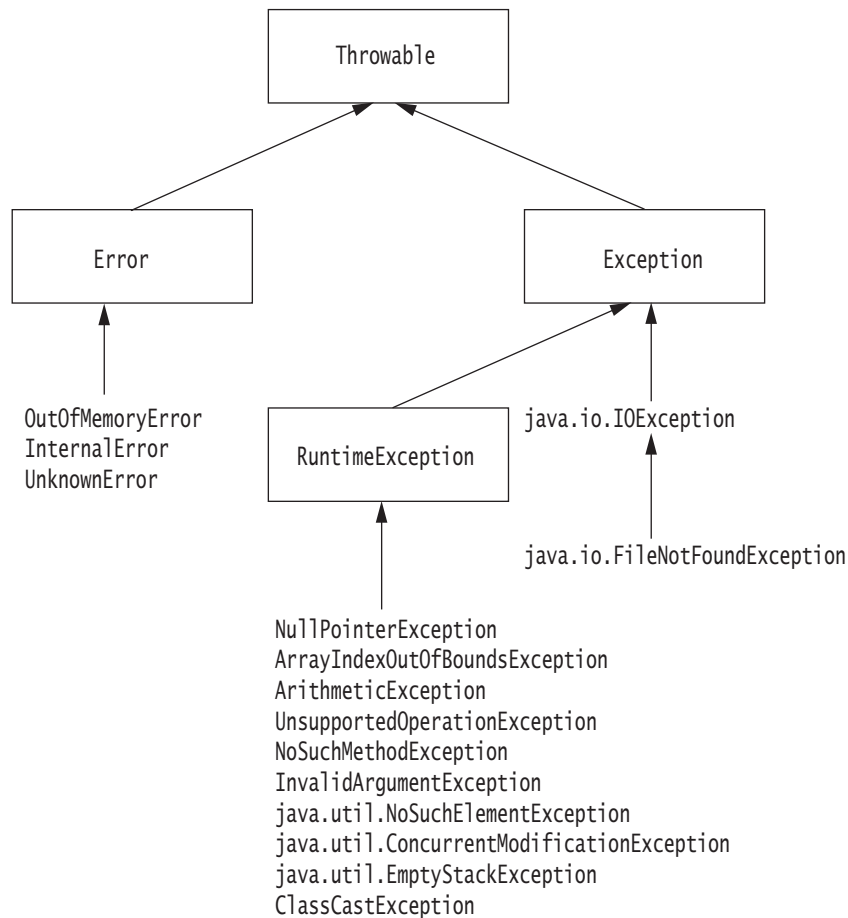


Figure 4.17 The hierarchy of exceptions (partial list)

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`.)

```

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 message.
17     */
18    public NoSuchElementException( String msg )
19    {
20        super( msg );
21    }
22 }

```

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

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 classes.

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 and Writer 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 );

```

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

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

**Figure 4.19** The 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 }
```

**Figure 4.20** A program that demonstrates the wrapping of streams and readers

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, which we will see again in Section 4.6.2.

Similar to the `BufferedReader` 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 4 bytes that represent a 32-bit integer. Writing data that 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.<sup>5</sup> 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:

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

---

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 integrity 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.

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 birth date, 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.<sup>6</sup>

---

6. Direct support for generic methods and classes has been announced by Sun in June 2001 as a future language addition, perhaps as early as Java 1.5. JDK 1.4 will contain a beta version of the likely implementation, known as *Generic Java*. A Generic Java Compiler supports generics and generates standard Java. The approach described in this section is the one most widely used now, and until generics are fully implemented in Java.

## 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.)

```

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 A generic `MemoryCell` class

```

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

```

Figure 4.22 Using the generic `MemoryCell` class

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

`MemoryCell` is a fairly small example. 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.

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.

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, embellishing 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.

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.

## 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.

```
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** A simplified ArrayList, with add, get, and size

```
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** An illustration of the Integer wrapper class

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.

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.

```
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    private MemoryCell m = new MemoryCell( );
11 }
```

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

## 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.

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 `Comparables`. The code to do this is shown in Figure 4.26.

```

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** A generic `findMax` routine, with demo using shapes and strings

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.

Third, as before, primitives cannot be passed as `Comparables`, 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, because it might be impossible to declare that a class implements a needed interface. For instance, the class

```
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** A simplified version of the `Integer` class in `java.lang`. It omits static methods and a `hashCode` method.

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 section 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.

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,

```
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** The `SimpleRectangle` class, which does not implement the `Comparable` interface

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 values and references. So we appear not to have a way of passing a function.

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*.

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.

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.

*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.

```

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 and rewritten for the weiss.util package

```

1 public class Utils
2 {
3     // Generic findMax, with a function object.
4     // Precondition: a.length > 0.
5     public static Object findMax( Object [ ] a, Comparator cmp )
6     {
7         int maxIndex = 0;
8
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** The generic findMax algorithm, using a function object

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 OrderRectBy-Width, 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`.<sup>7</sup> 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 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.

```
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 CompareTest
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** An example of a function object

---

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

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.

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.

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 in this text (in Chapter 15).

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.

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 a part of the outer class and can be declared with a visibility specifier. All outer class members are visible to the nested class's methods.

## 4.7.2 Local Classes

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.

```

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 the `OrderRectByWidth` class declaration

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

An advantage of declaring a class inside of a method is that the class's 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.

```

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 the `OrderRectByWidth` class declaration further

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.

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.

Anonymous classes introduce significant language complications.

Anonymous classes are often used to implement function objects.

## 4.8 DYNAMIC BINDING DETAILS

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

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

```

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

- Static methods, regardless of how the method is invoked
- Final methods
- Private methods (since they are invoked only from inside the class and are thus implicitly 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 run-time 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 )
```

```
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 An illustration of static binding for parameters

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

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*.

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

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

The only remaining issue is which class's version of that method is used. This is done by having the virtual machine deduce the run-time type of this object. Once the run-time 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`.<sup>8</sup> This second step is called *dynamic binding*.

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.

```

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 ) ); // true
28         System.out.println( obj1.sameVal( obj2 ) ); // false
29     }
30 }

```

**Figure 4.36** An illustration of overloading `equals` instead of overriding `equals`. Here, the call to the `sameVal` returns false!

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

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 a parameter, and the `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 overriding `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`.

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 run time, 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.

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

METHOD	OVERLOADING	COMMENTS
final	Potentially inlined	Invariant over the inheritance hierarchy (method is never redefined).
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

type conversion operators. Interfaces are also widely used for generic components, and to implement the function object pattern.

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. (108)

**abstract method** A method that has no meaningful definition and is thus always defined in the derived class. (107)

**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. (120)

**anonymous class** A class that has no name and is useful for implementing short function objects. (130)

**base class** The class on which the inheritance is based. (94)

**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). (90)

**decorator pattern** The pattern that involves the combining of several wrappers in order to add functionality. (118)

**derived class** A completely new class that nonetheless has some compatibility with the class from which it was derived. (94)

**dynamic binding** A run-time decision to apply the method corresponding to the actual referenced object. (96)

- extends clause** A clause used to declare that a new class is a subclass of another class. (94)
- final class** A class that may not be extended. (100)
- final method** A method that may not be overridden and is invariant over the inheritance hierarchy. Static binding is used for final methods. (99)
- function object** An object passed to a generic function with the intention of having its single method used by the generic function. (126)
- functor** A function object. (126)
- generic programming** Used to implement type-independent logic. (118)
- HAS-A relationship** A relationship in which the derived class has a (instance of the) base class. (90)
- implements clause** A clause used to declare that a class implements the methods of an interface. (111)
- 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 `Throwable` and `InputStream`. (94)
- interface** A special kind of abstract class that contains no implementation details. (110)
- IS-A relationship** A relationship in which the derived class is a (variation of the) base class. (90)
- leaf class** A final class. (100)
- local class** A class inside a method, declared with no visibility modifier. (130)
- 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. (109)
- nested class** A class inside a class, declared with the `static` modifier. (129)
- partial overriding** The act of augmenting a base class method to perform additional, but not entirely different, tasks. (100)
- 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. (96)
- protected class member** Accessible by the derived class and classes in the same package. (97)
- static binding** The decision on which class's version of a method to use is made at compile time. Is only used for static, final, or private methods. (100)
- 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. (133)

**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. (90)

**super constructor call** A call to the base class constructor. (99)

**super object** An object used in partial overloading to apply a base class method. (100)

**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. (120)



## 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.

---

<b>PersonDemo.java</b>	The Person hierarchy and test program.
<b>Shape.java</b>	The abstract Shape class.
<b>Circle.java</b>	The Circle class.
<b>Square.java</b>	The Square class.
<b>Rectangle.java</b>	The Rectangle class.
<b>ShapeDemo.java</b>	A test program for the Shape example.
<b>NoSuchElementException.java</b>	The exception class in Figure 4.18. This is part of <code>weiss.util</code> . <b>ConcurrentModificationException.java</b> and <b>EmptyStackException.java</b> are also online.
<b>DecoratorDemo.java</b>	An illustration of the decorator pattern, including buffering, compression, and serialization.
<b>MemoryCell.java</b>	The <code>MemoryCell</code> class in Figure 4.21.
<b>TestMemoryCell.java</b>	The test program for the memory cell class shown in Figure 4.22.
<b>SimpleArrayList.java</b>	The generic simplified <code>ArrayList</code> class in Figure 4.23, with some additional methods. A test program is provided in <b>ReadStrings WithSimpleArrayList.java</b> .
<b>PrimitiveWrapperDemo.java</b>	Demonstrates the use of the <code>Integer</code> class, as shown in Figure 4.24.
<b>StorageCellDemo.java</b>	The <code>StorageCell</code> adapter as shown in Figure 4.25, and a test program.
<b>FindMaxDemo.java</b>	The <code>findMax</code> generic algorithm in Figure 4.26.
<b>SimpleRectangle.java</b>	Contains the <code>SimpleRectangle</code> class in Figure 4.28.
<b>Comparator.java</b>	The <code>Comparator</code> interface in Figure 4.29.
<b>CompareTest.java</b>	Illustrates the function object, with no nested classes, as shown in Figure 4.31.
<b>CompareTestInner1.java</b>	Illustrates the function object, with a nested class, as shown in Figure 4.32.
<b>CompareTestInner2.java</b>	Illustrates the function object, with a local class, as shown in Figure 4.33.
<b>CompareTestInner3.java</b>	Illustrates the function object, with an anonymous class, as shown in Figure 4.34.
<b>StaticParamsDemo.java</b>	The demonstration of static overloading and dynamic binding shown in Figure 4.35.
<b>BadEqualsDemo.java</b>	Illustrates the consequences of overloading instead of overriding <code>equals</code> , as shown in 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?

```

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** A program to test visibility

- e. Write a three-parameter constructor for `Base`. Then write a five-parameter constructor for `Derived`.
  - 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 insist that these variables are marked `final`.)
- 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:
- 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?

```
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** The classes for Exercise 4.18

- 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 methods `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 method `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 `Comparables` 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 `IllegalArgumentException` 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 an indication 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 similar to `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). Implement a separate `SortedArrayList` class that supports `add`, `get`, `remove`, and `size`.
- 4.28 This exercise asks you to write a generic `countMatches` method. Your method will take two parameters. The first parameter is an array of `int`. The second parameter is a function object that returns a `Boolean`.
- Give a declaration for an interface that expresses the requisite function object.
  - `countMatches` returns the number of array items for which the function object returns `true`. Implement `countMatches`.
  - 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. Reuse the interface in Exercise 4.28(a).
- Write a function object `EqualsK`. `EqualsK` contains one data member (`k`). `EqualsK` is constructed with a single parameter (the default is zero) that is used to initialize `k`. Its method returns `true` if the parameter is equal to `k`.
  - 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 GregorianCalendar.
- 4.33 Implement a tax payer 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, CA., 1994.
2. T. Budd, *Understanding Object-Oriented Programming With Java*, Addison-Wesley, Reading, MA., 2001.
3. D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, MA., 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, MA., 1992.
5. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, N.J., 1988.