Chapter 5

INHERITANCE

- Classes, Superclasses, and Subclasses
- Object: The Cosmic Superclass
- Generic Array Lists
- Object Wrappers and Autoboxing
- Methods with a Variable Number of Parameters
- Enumeration Classes
- Reflection
- Design Hints for Inheritance
Chapter 5 ■ Inheritance

Chapter 4 introduced you to classes and objects. In this chapter, you learn about inheritance, another fundamental concept of object-oriented programming. The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods and fields and you add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

As with the previous chapter, if you are coming from a procedure-oriented language like C, Visual Basic, or COBOL, you will want to read this chapter carefully. For experienced C++ programmers or those coming from another object-oriented language like Smalltalk, this chapter will seem largely familiar, but there are many differences between how inheritance is implemented in Java and how it is done in C++ or in other object-oriented languages.

This chapter also covers reflection, the ability to find out more about classes and their properties in a running program. Reflection is a powerful feature, but it is undeniably complex. Because reflection is of greater interest to tool builders than to application programmers, you can probably glance over that part of the chapter upon first reading and come back to it later.

Classes, Superclasses, and Subclasses

Let’s return to the Employee class that we discussed in the previous chapter. Suppose (alas) you work for a company at which managers are treated differently from other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for receiving their salary, managers get bonuses if they actually achieve what they are supposed to do. This is the kind of situation that cries out for inheritance. Why? Well, you need to define a new class, Manager, and add functionality. But you can retain some of what you have already programmed in the Employee class, and all the fields of the original class can be preserved. More abstractly, there is an obvious “is-a” relationship between Manager and Employee. Every manager is an employee: This “is-a” relationship is the hallmark of inheritance.

Here is how you define a Manager class that inherits from the Employee class. You use the Java keyword extends to denote inheritance.

```java
class Manager extends Employee
{
    added methods and fields
}
```

C++ NOTE: Inheritance is similar in Java and C++. Java uses the extends keyword instead of the : token. All inheritance in Java is public inheritance; there is no analog to the C++ features of private and protected inheritance.

The keyword extends indicates that you are making a new class that derives from an existing class. The existing class is called the superclass, base class, or parent class. The new class is called the subclass, derived class, or child class. The terms superclass and subclass are those most commonly used by Java programmers, although some programmers prefer the parent/child analogy, which also ties in nicely with the “inheritance” theme.
The Employee class is a superclass, but not because it is superior to its subclass or contains more functionality. In fact, the opposite is true: subclasses have more functionality than their superclasses. For example, as you will see when we go over the rest of the Manager class code, the Manager class encapsulates more data and has more functionality than its superclass Employee.

NOTE: The prefixes super and sub come from the language of sets used in theoretical computer science and mathematics. The set of all employees contains the set of all managers, and this is described by saying it is a superset of the set of managers. Or, put it another way, the set of all managers is a subset of the set of all employees.

Our Manager class has a new field to store the bonus, and a new method to set it:

```java
class Manager extends Employee {
    private double bonus;

    public void setBonus(double b) {
        bonus = b;
    }

    . . .
}
```

There is nothing special about these methods and fields. If you have a Manager object, you can simply apply the setBonus method.

```java
Manager boss = . . .;
boss.setBonus(5000);
```

Of course, if you have an Employee object, you cannot apply the setBonus method—it is not among the methods that are defined in the Employee class.

However, you can use methods such as getName and getHireDay with Manager objects. Even though these methods are not explicitly defined in the Manager class, they are automatically inherited from the Employee superclass.

Similarly, the fields name, salary, and hireDay are inherited from the superclass. Every Manager object has four fields: name, salary, hireDay, and bonus.

When defining a subclass by extending its superclass, you only need to indicate the differences between the subclass and the superclass. When designing classes, you place the most general methods into the superclass and more specialized methods in the subclass. Factoring out common functionality by moving it to a superclass is common in object-oriented programming.

However, some of the superclass methods are not appropriate for the Manager subclass. In particular, the getSalary method should return the sum of the base salary and the bonus. You need to supply a new method to override the superclass method:

```java
class Manager extends Employee {
    . . .
}
```
Chapter 5 ▪ Inheritance

```java
public double getSalary()
{
    ...
    ...
}
```

How can you implement this method? At first glance, it appears to be simple—just return the sum of the salary and bonus fields:
```java
public double getSalary()
{
    return salary + bonus; // won't work
}
```

However, that won't work. The `getSalary` method of the `Manager` class has no direct access to the private fields of the superclass. This means that the `getSalary` method of the `Manager` class cannot directly access the salary field, even though every `Manager` object has a field called `salary`. Only the methods of the `Employee` class have access to the private fields. If the `Manager` methods want to access those private fields, they have to do what every other method does—use the public interface, in this case, the public `getSalary` method of the `Employee` class.

So, let’s try this again. You need to call `getSalary` instead of simply accessing the salary field.
```java
public double getSalary()
{
    double baseSalary = getSalary(); // still won't work
    return baseSalary + bonus;
}
```

The problem is that the call to `getSalary` simply calls itself, because the `Manager` class has a `getSalary` method (namely, the method we are trying to implement). The consequence is an infinite set of calls to the same method, leading to a program crash.

We need to indicate that we want to call the `getSalary` method of the `Employee` superclass, not the current class. You use the special keyword `super` for this purpose. The call
```java
super.getSalary()
```
calls the `getSalary` method of the `Employee` class. Here is the correct version of the `getSalary` method for the `Manager` class:
```java
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

**NOTE:** Some people think of `super` as being analogous to the `this` reference. However, that analogy is not quite accurate—`super` is not a reference to an object. For example, you cannot assign the value `super` to another object variable. Instead, `super` is a special keyword that directs the compiler to invoke the superclass method.

As you saw, a subclass can add fields, and it can add or override methods of the superclass. However, inheritance can never take away any fields or methods.
Classes, Superclasses, and Subclasses

C++ NOTE: Java uses the keyword super to call a superclass method. In C++, you would use the name of the superclass with the :: operator instead. For example, the getSalary method of the Manager class would call Employee::getSalary instead of super.getSalary.

Finally, let us supply a constructor.

```java
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Here, the keyword super has a different meaning. The instruction

```java
super(n, s, year, month, day);
```

is shorthand for “call the constructor of the Employee superclass with n, s, year, month, and day as parameters.”

Because the Manager constructor cannot access the private fields of the Employee class, it must initialize them through a constructor. The constructor is invoked with the special super syntax. The call using super must be the first statement in the constructor for the subclass.

If the subclass constructor does not call a superclass constructor explicitly, then the default (no-parameter) constructor of the superclass is invoked. If the superclass has no default constructor and the subclass constructor does not call another superclass constructor explicitly, then the Java compiler reports an error.

NOTE: Recall that the this keyword has two meanings: to denote a reference to the implicit parameter and to call another constructor of the same class. Likewise, the super keyword has two meanings: to invoke a superclass method and to invoke a superclass constructor. When used to invoke constructors, the this and super keywords are closely related. The constructor calls can only occur as the first statement in another constructor. The constructor parameters are either passed to another constructor of the same class (this) or a constructor of the superclass (super).

C++ NOTE: In a C++ constructor, you do not call super, but you use the initializer list syntax to construct the superclass. The Manager constructor looks like this in C++:

```cpp
Manager::Manager(String n, double s, int year, int month, int day) // C++
    : Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

Having redefined the getSalary method for Manager objects, managers will automatically have the bonus added to their salaries.
Here’s an example of this at work: we make a new manager and set the manager’s bonus:

```java
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

We make an array of three employees:

```java
Employee[] staff = new Employee[3];
```

We populate the array with a mix of managers and employees:

```java
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

We print out everyone’s salary:

```java
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
```

This loop prints the following data:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Now `staff[1]` and `staff[2]` each print their base salary because they are `Employee` objects. However, `staff[0]` is a `Manager` object and its `getSalary` method adds the bonus to the base salary.

What is remarkable is that the call `e.getSalary()` picks out the correct `getSalary` method. Note that the declared type of `e` is `Employee`, but the actual type of the object to which `e` refers can be either `Employee` or `Manager`.

When `e` refers to an `Employee` object, then the call `e.getSalary()` calls the `getSalary` method of the `Employee` class. However, when `e` refers to a `Manager` object, then the `getSalary` method of the `Manager` class is called instead. The virtual machine knows about the actual type of the object to which `e` refers, and therefore can invoke the correct method.

The fact that an object variable (such as the variable `e`) can refer to multiple actual types is called polymorphism. Automatically selecting the appropriate method at runtime is called dynamic binding. We discuss both topics in more detail in this chapter.

---

**C++ NOTE:** In Java, you do not need to declare a method as virtual. Dynamic binding is the default behavior. If you do not want a method to be virtual, you tag it as `final`. (We discuss the `final` keyword later in this chapter.)

Listing 5–1 contains a program that shows how the salary computation differs for `Employee` and `Manager` objects.
Classes, Superclasses, and Subclasses

Listing 5–1  ManagerTest.java

```java
import java.util.*;

/**
   * This program demonstrates inheritance.
   * @version 1.21 2004-02-21
   * @author Cay Horstmann
   */
public class ManagerTest
{
    public static void main(String[] args)
    {
        // construct a Manager object
        Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
        boss.setBonus(5000);

        Employee[] staff = new Employee[3];

        // fill the staff array with Manager and Employee objects
        staff[0] = boss;
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);

        // print out information about all Employee objects
        for (Employee e : staff)
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
    }
}

class Employee
{
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }
```
Chapter 5  ■  Inheritance

Listing 5–1  ManagerTest.java (continued)

```java
50.  public Date getHireDay()
51.  {
52.    return hireDay;
53.  }
54.
55.  public void raiseSalary(double byPercent)
56.  {
57.    double raise = salary * byPercent / 100;
58.    salary += raise;
59.  }
60.
61.  private String name;
62.  private double salary;
63.  private Date hireDay;
64. }
65.
66. class Manager extends Employee
67. {
68.   /**
69.    * @param n the employee's name
70.    * @param s the salary
71.    * @param year the hire year
72.    * @param month the hire month
73.    * @param day the hire day
74.   */
75.   public Manager(String n, double s, int year, int month, int day)
76.   {
77.     super(n, s, year, month, day);
78.     bonus = 0;
79.   }
80.
81.   public double getSalary()
82.   {
83.     double baseSalary = super.getSalary();
84.     return baseSalary + bonus;
85.   }
86.
87.   public void setBonus(double b)
88.   {
89.     bonus = b;
90.   }
91.
92.   private double bonus;
93. }
```
**Inheritance Hierarchies**

Inheritance need not stop at deriving one layer of classes. We could have an `Executive` class that extends `Manager`, for example. The collection of all classes extending from a common superclass is called an *inheritance hierarchy*, as shown in Figure 5–1. The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain*.

There is usually more than one chain of descent from a distant ancestor class. You could form a subclass `Programmer` or `Secretary` that extends `Employee`, and they would have nothing to do with the `Manager` class (or with each other). This process can continue as long as is necessary.

*C++ NOTE: Java does not support multiple inheritance. (For ways to recover much of the functionality of multiple inheritance, see the section on Interfaces in the next chapter.)*

![Figure 5-1 Employee inheritance hierarchy](image)

**Polymorphism**

A simple rule enables you to know whether or not inheritance is the right design for your data. The “is-a” rule states that every object of the subclass is an object of the superclass. For example, every manager is an employee. Thus, it makes sense for the `Manager` class to be a subclass of the `Employee` class. Naturally, the opposite is not true—not every employee is a manager.

Another way of formulating the “is-a” rule is the *substitution principle*. That principle states that you can use a subclass object whenever the program expects a superclass object.
For example, you can assign a subclass object to a superclass variable.

Employee e;
e = new Employee(. . .);  // Employee object expected
e = new Manager(. . .);  // OK, Manager can be used as well

In the Java programming language, object variables are polymorphic. A variable of type Employee can refer to an object of type Employee or to an object of any subclass of the Employee class (such as Manager, Executive, Secretary, and so on).

We took advantage of this principle in Listing 5–1:

Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;

In this case, the variables staff[0] and boss refer to the same object. However, staff[0] is considered to be only an Employee object by the compiler. That means, you can call

boss.setBonus(5000); // OK

but you can’t call

staff[0].setBonus(5000); // ERROR

The declared type of staff[0] is Employee, and the setBonus method is not a method of the Employee class.

However, you cannot assign a superclass reference to a subclass variable. For example, it is not legal to make the assignment

Manager m = staff[i]; // ERROR

The reason is clear: Not all employees are managers. If this assignment were to succeed and m were to refer to an Employee object that is not a manager, then it would later be possible to call m.setBonus(...) and a runtime error would occur.

CAUTION: In Java, arrays of subclass references can be converted to arrays of superclass references without a cast. For example, consider this array of managers:

Manager[] managers = new Manager[10];

It is legal to convert this array to an Employee[] array:

Employee[] staff = managers; // OK

Sure, why not, you may think. After all, if manager[i] is a Manager, it is also an Employee. But actually, something surprising is going on. Keep in mind that managers and staff are references to the same array. Now consider the statement

staff[0] = new Employee("Harry Hacker", ...);

The compiler will cheerfully allow this assignment. But staff[0] and manager[0] are the same reference, so it looks as if we managed to smuggle a mere employee into the management ranks. That would be very bad—calling managers[0].setBonus(1000) would try to access a nonexistent instance field and would corrupt neighboring memory.

To make sure no such corruption can occur, all arrays remember the element type with which they were created, and they monitor that only compatible references are stored into them. For example, the array created as new Manager[10] remembers that it is an array of managers. Attempting to store an Employee reference causes an ArrayStoreException.
Dynamic Binding

It is important to understand what happens when a method call is applied to an object. Here are the details:

1. The compiler looks at the declared type of the object and the method name. Let’s say we call x.f(param), and the implicit parameter x is declared to be an object of class C. Note that there may be multiple methods, all with the same name, f, but with different parameter types. For example, there may be a method f(int) and a method f(String). The compiler enumerates all methods called f in the class C and all public methods called f in the superclasses of C. Now the compiler knows all possible candidates for the method to be called.

2. Next, the compiler determines the types of the parameters that are supplied in the method call. If among all the methods called f there is a unique method whose parameter types are a best match for the supplied parameters, then that method is chosen to be called. This process is called overloading resolution. For example, in a call x.f("Hello"), the compiler picks f(String) and not f(int). The situation can get complex because of type conversions (int to double, Manager to Employee, and so on). If the compiler cannot find any method with matching parameter types or if multiple methods all match after applying conversions, then the compiler reports an error. Now the compiler knows the name and parameter types of the method that needs to be called.

   NOTE: Recall that the name and parameter type list for a method is called the method’s signature. For example, f(int) and f(String) are two methods with the same name but different signatures. If you define a method in a subclass that has the same signature as a superclass method, then you override that method. The return type is not part of the signature. However, when you override a method, you need to keep the return type compatible. Prior to Java SE 5.0, the return types had to be identical. However, it is now legal for the subclass to change the return type of an overridden method to a subtype of the original type. For example, suppose that the Employee class has a

   public Employee getBuddy() { ... }

   Then the Manager subclass can override this method as

   public Manager getBuddy() { ... } // OK in Java SE 5.0

   We say that the two getBuddy methods have covariant return types.

3. If the method is private, static, final, or a constructor, then the compiler knows exactly which method to call. (The final modifier is explained in the next section.) This is called static binding. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtime. In our example, the compiler would generate an instruction to call f(String) with dynamic binding.

4. When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the actual type of the object to which x refers. Let’s say the actual type is D, a subclass of C. If the class D
defines a method f(String), that method is called. If not, D's superclass is searched for a method f(String), and so on. It would be time consuming to carry out this search every time a method is called. Therefore, the virtual machine precomputes for each class a method table that lists all method signatures and the actual methods to be called. When a method is actually called, the virtual machine simply makes a table lookup. In our example, the virtual machine consults the method table for the class D and looks up the method to call for f(String). That method may be D.f(String) or X.f(String), where X is some superclass of D. There is one twist to this scenario. If the call is super.f(param), then the compiler consults the method table of the superclass of the implicit parameter.

Let's look at this process in detail in the call e.getSalary() in Listing 5–1. The declared type of e is Employee. The Employee class has a single method, called getSalary, with no method parameters. Therefore, in this case, we don't worry about overloading resolution. Because the getSalary method is not private, static, or final, it is dynamically bound. The virtual machine produces method tables for the Employee and Manager classes. The Employee table shows that all methods are defined in the Employee class itself:

Employee:
- getName() -> Employee.getName()
- getSalary() -> Employee.getSalary()
- getHireDay() -> Employee.getHireDay()
- raiseSalary(double) -> Employee.raiseSalary(double)

Actually, that isn't the whole story—as you will see later in this chapter, the Employee class has a superclass Object from which it inherits a number of methods. We ignore the Object methods for now.

The Manager method table is slightly different. Three methods are inherited, one method is redefined, and one method is added.

Manager:
- getName() -> Employee.getName()
- getSalary() -> Manager.getSalary()
- getHireDay() -> Employee.getHireDay()
- raiseSalary(double) -> Employee.raiseSalary(double)
- setBonus(double) -> Manager.setBonus(double)

At runtime, the call e.getSalary() is resolved as follows:
1. First, the virtual machine fetches the method table for the actual type of e. That may be the table for Employee, Manager, or another subclass of Employee.
2. Then, the virtual machine looks up the defining class for the getSalary() signature. Now it knows which method to call.
3. Finally, the virtual machine calls the method.

Dynamic binding has a very important property: it makes programs extensible without the need for modifying existing code. Suppose a new class Executive is added and there is the possibility that the variable e refers to an object of that class. The code containing the call e.getSalary() need not be recompiled. The Executive.getSalary() method is called automatically if e happens to refer to an object of type Executive.
CAUTION: When you override a method, the subclass method must be at least as visible as the superclass method. In particular, if the superclass method is public, then the subclass method must also be declared as public. It is a common error to accidentally omit the public specifier for the subclass method. The compiler then complains that you try to supply a weaker access privilege.

Preventing Inheritance: Final Classes and Methods

Occasionally, you want to prevent someone from forming a subclass from one of your classes. Classes that cannot be extended are called final classes, and you use the final modifier in the definition of the class to indicate this. For example, let us suppose we want to prevent others from subclassing the Executive class. Then, we simply declare the class by using the final modifier as follows:

```java
final class Executive extends Manager
{
    ...
}
```

You can also make a specific method in a class final. If you do this, then no subclass can override that method. (All methods in a final class are automatically final.) For example:

```java
class Employee
{
    ...
    public final String getName()
    {
        return name;
    }
    ...
}
```

NOTE: Recall that fields can also be declared as final. A final field cannot be changed after the object has been constructed. However, if a class is declared as final, only the methods, not the fields, are automatically final.

There is only one good reason to make a method or class final: to make sure that the semantics cannot be changed in a subclass. For example, the `getTime` and `setTime` methods of the Calendar class are final. This indicates that the designers of the Calendar class have taken over responsibility for the conversion between the Date class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the String class is a final class. That means nobody can define a subclass of String. In other words, if you have a String reference, then you know it refers to a String and nothing but a String.

Some programmers believe that you should declare all methods as final unless you have a good reason that you want polymorphism. In fact, in C++ and C#, methods do not use polymorphism unless you specifically request it. That may be a bit extreme, but we agree that it is a good idea to think carefully about final methods and classes when you design a class hierarchy.
In the early days of Java, some programmers used the `final` keyword in the hope of avoiding the overhead of dynamic binding. If a method is not overridden, and it is short, then a compiler can optimize the method call away—a process called *inlining*. For example, inlining the call `e.getName()` replaces it with the field access `e.name`. This is a worthwhile improvement—CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one. However, if `getName` can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the overriding code may do.

Fortunately, the just-in-time compiler in the virtual machine can do a better job than a traditional compiler. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called, and not actually overridden, the just-in-time compiler can inline the method. What happens if the virtual machine loads another subclass that overrides an inlined method? Then the optimizer must undo the inlining. That's slow, but it happens rarely.

*C++ NOTE:* In C++, a method is not dynamically bound by default, and you can tag it as `inline` to have method calls replaced with the method source code. However, there is no mechanism that would prevent a subclass from overriding a superclass method. In C++, you can write classes from which no other class can derive, but doing so requires an obscure trick, and there are few reasons to write such a class. (The obscure trick is left as an exercise to the reader. Hint: Use a virtual base class.)

### Casting

Recall from Chapter 3 that the process of forcing a conversion from one type to another is called casting. The Java programming language has a special notation for casts. For example,

```java
    double x = 3.405;
    int nx = (int)x;
```

converts the value of the expression `x` into an integer, discarding the fractional part.

Just as you occasionally need to convert a floating-point number to an integer, you also need to convert an object reference from one class to another. To actually make a cast of an object reference, you use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast. For example:

```java
    Manager boss = (Manager)staff[0];
```

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the `ManagerTest` class, the `staff` array had to be an array of `Employee` objects because some of its entries were regular employees. We would need to cast the managerial elements of the array back to `Manager` to access any of its new variables. (Note that in the sample code for the first section, we made a special effort to avoid the cast. We initialized the `boss` variable with a `Manager` object before storing it in the array. We needed the correct type to set the bonus of the manager.)
As you know, in Java every object variable has a type. The type describes the kind of object the variable refers to and what it can do. For example, \texttt{staff[i]} refers to an \texttt{Employee} object (so it can also refer to a \texttt{Manager} object).

The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass reference to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass reference to a subclass variable, you are promising more. Then you must use a cast so that your promise can be checked at runtime.

What happens if you try to cast down an inheritance chain and you are “lying” about what an object contains?

\begin{verbatim}
Manager boss = (Manager) staff[1]; // ERROR
\end{verbatim}

When the program runs, the Java runtime system notices the broken promise and generates a \texttt{ClassCastException}. If you do not catch the exception, your program terminates. Thus, it is good programming practice to find out whether a cast will succeed before attempting it. Simply use the \texttt{instanceof} operator. For example:

\begin{verbatim}
if (staff[1] instanceof Manager)
{
  boss = (Manager) staff[1];
  . . .
}
\end{verbatim}

Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed. For example, the cast

\begin{verbatim}
Date c = (Date) staff[1];
\end{verbatim}

is a compile-time error because \texttt{Date} is not a subclass of \texttt{Employee}.

To sum up:

- You can cast only within an inheritance hierarchy.
- Use \texttt{instanceof} to check before casting from a superclass to a subclass.

\begin{tabular}{|c|}
\hline
\textbf{NOTE: The test} & \\
\hline
\texttt{x instanceof C} & \\
\hline
\end{tabular}

\texttt{x instanceof C} does not generate an exception if \texttt{x} is \texttt{null}. It simply returns \texttt{false}. That makes sense. Because \texttt{null} refers to no object, it certainly doesn’t refer to an object of type \texttt{C}.

Actually, converting the type of an object by performing a cast is not usually a good idea. In our example, we do not need to cast an \texttt{Employee} object to a \texttt{Manager} object for most purposes. The \texttt{getSalary} method will work correctly on both objects of both classes. The dynamic binding that makes polymorphism work locates the correct method automatically.

The only reason to make the cast is to use a method that is unique to managers, such as \texttt{setBonus}. If for some reason you find yourself wanting to call \texttt{setBonus} on \texttt{Employee} objects, ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a \texttt{setBonus} method. Remember, it takes only one uncaught \texttt{ClassCastException} to terminate your program. In general, it is best to minimize the use of casts and the \texttt{instanceof} operator.
C++ NOTE: Java uses the cast syntax from the "bad old days" of C, but it works like the safe `dynamic_cast` operation of C++. For example,

```cpp
Manager boss = (Manager) staff[1]; // Java
```

is the same as

```cpp
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

with one important difference. If the cast fails, it does not yield a null object but throws an exception. In this sense, it is like a C++ cast of `references`. This is a pain in the neck. In C++, you can take care of the type test and type conversion in one operation.

```cpp
if (boss != NULL) . . .
```

In Java, you use a combination of the `instanceof` operator and a cast.

```java
if (staff[1] instanceof Manager)
{
    Manager boss = (Manager) staff[1];
    . . .
}
```

**Abstract Classes**

As you move up the inheritance hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes so general that you think of it more as a basis for other classes than as a class with specific instances you want to use. Consider, for example, an extension of our Employee class hierarchy. An employee is a person, and so is a student. Let us extend our class hierarchy to include classes Person and Student. Figure 5–2 shows the inheritance relationships between these classes.

---

**Figure 5–2 Inheritance diagram for Person and its subclasses**

Why bother with so high a level of abstraction? There are some attributes that make sense for every person, such as the name. Both students and employees have names,
and introducing a common superclass lets us factor out the `getName` method to a higher level in the inheritance hierarchy.

Now let's add another method, `getDescription`, whose purpose is to return a brief description of the person, such as

- an employee with a salary of $50,000.00
- a student majoring in computer science

It is easy to implement this method for the `Employee` and `Student` classes. But what information can you provide in the `Person` class? The `Person` class knows nothing about the person except the name. Of course, you could implement `Person.getDescription()` to return an empty string. But there is a better way. If you use the `abstract` keyword, you do not need to implement the method at all.

```java
public abstract String getDescription();
// no implementation required
```

For added clarity, a class with one or more abstract methods must itself be declared abstract.

```java
abstract class Person
{
  // ...  
  public abstract String getDescription();
}
```

In addition to abstract methods, abstract classes can have fields and concrete methods.

```java
abstract class Person
{
  public Person(String n)
  {
    name = n;
  }

  public abstract String getDescription();

  public String getName()
  {
    return name;
  }

  private String name;
}
```

**TIP:** Some programmers don’t realize that abstract classes can have concrete methods. You should always move common fields and methods (whether abstract or not) to the superclass (whether abstract or not).

Abstract methods act as placeholders for methods that are implemented in the subclasses. When you extend an abstract class, you have two choices. You can leave some or all of the abstract methods undefined. Then you must tag the subclass as abstract as well. Or you can define all methods. Then the subclass is no longer abstract.
For example, we will define a Student class that extends the abstract Person class and implements the getDescription method. Because none of the methods of the Student class are abstract, it does not need to be declared as an abstract class.

A class can even be declared as abstract even though it has no abstract methods. Abstract classes cannot be instantiated. That is, if a class is declared as abstract, no objects of that class can be created. For example, the expression

```java
new Person("Vince Vu")
```

is an error. However, you can create objects of concrete subclasses.

Note that you can still create object variables of an abstract class, but such a variable must refer to an object of a nonabstract subclass. For example:

```java
Person p = new Student("Vince Vu", "Economics");
```

Here `p` is a variable of the abstract type `Person` that refers to an instance of the nonabstract subclass `Student`.

**C++ NOTE:** In C++, an abstract method is called a *pure virtual function* and is tagged with a trailing `= 0`, such as in

```cpp
class Person // C++
{
 public:
   virtual string getDescription() = 0;
   ...
};
```

A C++ class is abstract if it has at least one pure virtual function. In C++, there is no special keyword to denote abstract classes.

Let us define a concrete subclass Student that extends the abstract Person class:

```java
class Student extends Person
{
   public Student(String n, String m)
   {
      super(n);
      major = m;
   }

   public String getDescription()
   {
      return "a student majoring in " + major;
   }

   private String major;
}
```

The Student class defines the getDescription method. Therefore, all methods in the Student class are concrete, and the class is no longer an abstract class.
The program shown in Listing 5–2 defines the abstract superclass `Person` and two concrete subclasses, `Employee` and `Student`. We fill an array of `Person` references with employee and student objects:

```java
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

We then print the names and descriptions of these objects:

```java
for (Person p : people)
    System.out.println(p.getName() + ", " + p.getDescription());
```

Some people are baffled by the call

```java
p.getDescription()
```

Isn’t this call an undefined method? Keep in mind that the variable `p` never refers to a `Person` object because it is impossible to construct an object of the abstract `Person` class. The variable `p` always refers to an object of a concrete subclass such as `Employee` or `Student`. For these objects, the `getDescription` method is defined.

Could you have omitted the abstract method altogether from the `Person` superclass and simply defined the `getDescription` methods in the `Employee` and `Student` subclasses? If you did that, then you wouldn’t have been able to invoke the `getDescription` method on the variable `p`. The compiler ensures that you invoke only methods that are declared in the class.

Abstract methods are an important concept in the Java programming language. You will encounter them most commonly inside `interfaces`. For more information about interfaces, turn to Chapter 6.

### Listing 5–2  PersonTest.java

```java
import java.util.*;

/**<n
 * This program demonstrates abstract classes.
 * @version 1.01 2004-02-21
 * @author Cay Horstmann
 */

public class PersonTest
{
    public static void main(String[] args)
    {
        Person[] people = new Person[2];

        // fill the people array with Student and Employee objects
        people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        people[1] = new Student("Maria Morris", "computer science");

        // print out names and descriptions of all Person objects
        for (Person p : people)
            System.out.println(p.getName() + ", " + p.getDescription());
    }
}
```
Listing 5–2  PersonTest.java (continued)

22. }
23. abstract class Person
24. {
25.   public Person(String n)
26.   {
27.     name = n;
28.   }
29.   public abstract String getDescription();
30.   public String getName()
31.   {
32.     return name;
33.   }
34.   private String name;
35. }
36. class Employee extends Person
37. {
38.   public Employee(String n, double s, int year, int month, int day)
39.   {
40.     super(n);
41.     salary = s;
42.     GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
43.     hireDay = calendar.getTime();
44.   }
45.   public double getSalary()
46.   {
47.     return salary;
48.   }
49.   public Date getHireDay()
50.   {
51.     return hireDay;
52.   }
53.   public String getDescription()
54.   {
55.     return String.format("an employee with a salary of $%.2f", salary);
56.   }
57.   public void raiseSalary(double byPercent)
58.   {
59.     double raise = salary * byPercent / 100;
60.     salary += raise;
61.   }
62. }
Protected Access

As you know, fields in a class are best tagged as private, and methods are usually tagged as public. Any features declared private won’t be visible to other classes. As we said at the beginning of this chapter, this is also true for subclasses: a subclass cannot access the private fields of its superclass.

There are times, however, when you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access a superclass field. In that case, you declare a class feature as protected. For example, if the superclass Employee declares the hireDay field as protected instead of private, then the Manager methods can access it directly. However, the Manager class methods can peek inside the hireDay field of Manager objects only, not of other Employee objects. This restriction is made so that you can’t abuse the protected mechanism and form subclasses just to gain access to the protected fields.

In practice, use protected fields with caution. Suppose your class is used by other programmers and you designed it with protected fields. Unknown to you, other programmers may inherit classes from your class and then start accessing your protected fields. In this case, you can no longer change the implementation of your class without upsetting the other programmers. That is against the spirit of OOP, which encourages data encapsulation.
Protected methods make more sense. A class may declare a method as protected if it is tricky to use. This indicates that the subclasses (which, presumably, know their ancestors well) can be trusted to use the method correctly, but other classes cannot.

A good example of this kind of method is the clone method of the Object class—see Chapter 6 for more details.

*C++ NOTE:* As it happens, protected features in Java are visible to all subclasses as well as to all other classes in the same package. This is slightly different from the C++ meaning of protected, and it makes the notion of protected in Java even less safe than in C++.

Here is a summary of the four access modifiers in Java that control visibility:
1. Visible to the class only (private).
2. Visible to the world (public).
3. Visible to the package and all subclasses (protected).
4. Visible to the package—the (unfortunate) default. No modifiers are needed.

**Object: The Cosmic Superclass**

The Object class is the ultimate ancestor—every class in Java extends Object. However, you never have to write

```java
class Employee extends Object
```

The ultimate superclass Object is taken for granted if no superclass is explicitly mentioned. Because every class in Java extends Object, it is important to be familiar with the services provided by the Object class. We go over the basic ones in this chapter and refer you to later chapters or to the on-line documentation for what is not covered here. (Several methods of Object come up only when dealing with threads—see Volume II for more on threads.)

You can use a variable of type Object to refer to objects of any type:

```java
Object obj = new Employee("Harry Hacker", 35000);
```

Of course, a variable of type Object is only useful as a generic holder for arbitrary values. To do anything specific with the value, you need to have some knowledge about the original type and then apply a cast:

```java
Employee e = (Employee) obj;
```

In Java, only the primitive types (numbers, characters, and boolean values) are not objects. All array types, no matter whether they are arrays of objects or arrays of primitive types, are class types that extend the Object class.

```java
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

*C++ NOTE:* In C++, there is no cosmic root class. However, every pointer can be converted to a void* pointer.
The equals Method

The equals method in the Object class tests whether one object is considered equal to another. The equals method, as implemented in the Object class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two PrintStream objects for equality. However, you will often want to implement state-based equality testing, in which two objects are considered equal when they have the same state.

For example, let us consider two employees equal if they have the same name, salary, and hire date. (In an actual employee database, it would be more sensible to compare IDs instead. We use this example to demonstrate the mechanics of implementing the equals method.)

```java
class Employee
{
    . . .

    public boolean equals(Object otherObject)
    {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;

        // must return false if the explicit parameter is null
        if (otherObject == null) return false;

        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass()) return false;

        // now we know otherObject is a non-null Employee
        Employee other = (Employee) otherObject;

        // test whether the fields have identical values
        return name.equals(other.name) 
            && salary == other.salary 
            && hireDay.equals(other.hireDay);
    }
}
```

The getClass method returns the class of an object—we discuss this method in detail later in this chapter. In our test, two objects can only be equal when they belong to the same class.

When you define the equals method for a subclass, first call equals on the superclass. If that test doesn’t pass, then the objects can’t be equal. If the superclass fields are equal, then you are ready to compare the instance fields of the subclass.

```java
class Manager extends Employee
{
    . . .

    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;

        // . . .
    }
}
```
Equality Testing and Inheritance

How should the equals method behave if the implicit and explicit parameters don’t belong to the same class? This has been an area of some controversy. In the preceding example, the equals method returns false if the classes don’t match exactly. But many programmers use an instanceof test instead:

```java
if (!(otherObject instanceof Employee)) return false;
```

This leaves open the possibility that otherObject can belong to a subclass. However, this approach can get you into trouble. Here is why. The Java Language Specification requires that the equals method has the following properties:

1. It is reflexive: For any non-null reference x, x.equals(x) should return true.
2. It is symmetric: For any references x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
3. It is transitive: For any references x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
4. It is consistent: If the objects to which x and y refer haven’t changed, then repeated calls to x.equals(y) return the same value.
5. For any non-null reference x, x.equals(null) should return false.

These rules are certainly reasonable. You wouldn’t want a library implementor to ponder whether to call x.equals(y) or y.equals(x) when locating an element in a data structure.

However, the symmetry rule has subtle consequences when the parameters belong to different classes. Consider a call

```java
e.equals(m)
```

where e is an Employee object and m is a Manager object, both of which happen to have the same name, salary, and hire date. If Employee.equals uses an instanceof test, the call returns true. But that means that the reverse call

```java
m.equals(e)
```

also needs to return true—the symmetry rule does not allow it to return false or to throw an exception.

That leaves the Manager class in a bind. Its equals method must be willing to compare itself to any Employee, without taking manager-specific information into account! All of a sudden, the instanceof test looks less attractive!

Some authors have gone on record that the getClass test is wrong because it violates the substitution principle. A commonly cited example is the equals method in the AbstractSet class that tests whether two sets have the same elements. The AbstractSet class has two concrete subclasses, TreeSet and HashSet, that use different algorithms for locating set elements. You really want to be able to compare any two sets, no matter how they are implemented.
However, the set example is rather specialized. It would make sense to declare `AbstractSet.equals` as `final`, because nobody should redefine the semantics of set equality. (The method is not actually `final`. This allows a subclass to implement a more efficient algorithm for the equality test.)

The way we see it, there are two distinct scenarios:

- If subclasses can have their own notion of equality, then the symmetry requirement forces you to use the `getClass` test.
- If the notion of equality is fixed in the superclass, then you can use the `instanceof` test and allow objects of different subclasses to be equal to another.

In the example of the employees and managers, we consider two objects to be equal when they have matching fields. If we have two `Manager` objects with the same name, salary, and hire date, but with different bonuses, we want them to be different. Therefore, we used the `getClass` test.

But suppose we used an employee ID for equality testing. This notion of equality makes sense for all subclasses. Then we could use the `instanceof` test, and we should declare `Employee.equals` as `final`.

NOTE: The standard Java library contains over 150 implementations of `equals` methods, with a mishmash of using `instanceof`, calling `getClass`, catching a `ClassCastException`, or doing nothing at all.

Here is a recipe for writing the perfect `equals` method:

1. Name the explicit parameter `otherObject`—later, you need to cast it to another variable that you should call `other`.

2. Test whether this happens to be identical to `otherObject`:
   ```java
   if (this == otherObject) return true;
   ```
   This statement is just an optimization. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields.

3. Test whether `otherObject` is `null` and return `false` if it is. This test is required.
   ```java
   if (otherObject == null) return false;
   ```

4. Compare the classes of `this` and `otherObject`. If the semantics of `equals` can change in subclasses, use the `getClass` test:
   ```java
   if (getClass() != otherObject.getClass()) return false;
   ```
   If the same semantics holds for all subclasses, you can use an `instanceof` test:
   ```java
   if (!(otherObject instanceof ClassName)) return false;
   ```

5. Cast `otherObject` to a variable of your class type:
   ```java
   ClassName other = (ClassName) otherObject
   ```

6. Now compare the fields, as required by your notion of equality. Use `==` for primitive type fields, `equals` for object fields. Return `true` if all fields match, `false` otherwise.
   ```java
   return field1 == other.field1
         && field2.equals(other.field2)
         && . . .;
   ```
   If you redefine `equals` in a subclass, include a call to `super.equals(other)`.
TIP: If you have fields of array type, you can use the static `Arrays.equals` method to check that corresponding array elements are equal.

CAUTION: Here is a common mistake when implementing the `equals` method. Can you spot the problem?

```java
public class Employee
{
    public boolean equals(Employee other)
    {
        return name.equals(other.name)
        && salary == other.salary
        && hireDay.equals(other.hireDay);
    }
    ...
}
```

This method declares the explicit parameter type as `Employee`. As a result, it does not override the `equals` method of the `Object` class but defines a completely unrelated method.

Starting with Java SE 5.0, you can protect yourself against this type of error by tagging methods that are intended to override superclass methods with `@Override`:

```java
@Override public boolean equals(Object other)
```

If you made a mistake and you are defining a new method, the compiler reports an error. For example, suppose you add the following declaration to the `Employee` class:

```java
@Override public boolean equals(Employee other)
```

An error is reported because this method doesn't override any method from the `Object` superclass.

---

### Java.util.Arrays 1.2

- `static boolean equals(type[] a, type[] b) 5.0`

  returns true if the arrays have equal lengths and equal elements in corresponding positions. The arrays can have component types `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

---

**The `hashCode` Method**

A hash code is an integer that is derived from an object. Hash codes should be scrambled—if x and y are two distinct objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. Table 5–1 lists a few examples of hash codes that result from the `hashCode` method of the `String` class.

The `String` class uses the following algorithm to compute the hash code:

```java
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```
The `hashCode` method is defined in the `Object` class. Therefore, every object has a default hash code. That hash code is derived from the object’s memory address. Consider this example:

```java
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Table 5–2 shows the result.

Note that the strings `s` and `t` have the same hash code because, for strings, the hash codes are derived from their contents. The string builders `sb` and `tb` have different hash codes because no `hashCode` method has been defined for the `StringBuilder` class, and the default `hashCode` method in the `Object` class derives the hash code from the object’s memory address.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method for objects that users might insert into a hash table. (We discuss hash tables in Chapter 2 of Volume II.)

The `hashCode` method should return an integer (which can be negative). Just combine the hash codes of the instance fields so that the hash codes for different objects are likely to be widely scattered.

For example, here is a `hashCode` method for the `Employee` class:

```java
class Employee {
    public int hashCode() {
    }
```

---

**Table 5–1 Hash Codes Resulting from the `hashCode` Function**

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello</td>
<td>69609650</td>
</tr>
<tr>
<td>Harry</td>
<td>69496448</td>
</tr>
<tr>
<td>Hacker</td>
<td>-2141031506</td>
</tr>
</tbody>
</table>

**Table 5–2 Hash Codes of Strings and String Builders**

<table>
<thead>
<tr>
<th>Object</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>2556</td>
</tr>
<tr>
<td>sb</td>
<td>20526976</td>
</tr>
<tr>
<td>t</td>
<td>2556</td>
</tr>
<tr>
<td>tb</td>
<td>20527144</td>
</tr>
</tbody>
</table>

Note that the strings `s` and `t` have the same hash code because, for strings, the hash codes are derived from their contents. The string builders `sb` and `tb` have different hash codes because no `hashCode` method has been defined for the `StringBuilder` class, and the default `hashCode` method in the `Object` class derives the hash code from the object’s memory address.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method for objects that users might insert into a hash table. (We discuss hash tables in Chapter 2 of Volume II.)

The `hashCode` method should return an integer (which can be negative). Just combine the hash codes of the instance fields so that the hash codes for different objects are likely to be widely scattered.

For example, here is a `hashCode` method for the `Employee` class:
return 7 * name.hashCode()
    + 11 * new Double(salary).hashCode()
    + 13 * hireDay.hashCode();
}

Your definitions of equals and hashCode must be compatible: if \( x \).equals\( (y) \) is true, then \( x\).hashCode() must be the same value as \( y\).hashCode(). For example, if you define Employee.equals to compare employee IDs, then the hashCode method needs to hash the IDs, not employee names or memory addresses.

---

**TIP:** If you have fields of array type, you can use the static Arrays.hashCode method to compute a hash code that is composed of the hash codes of the array elements.

---

### API

**java.lang.Object 1.0**

- int hashCode()
  
  returns a hash code for this object. A hash code can be any integer, positive or negative. Equal objects need to return identical hash codes.

---

**java.util.Arrays 1.2**

- static int hashCode(type[] a) 5.0
  
  computes the hash code of the array a, which can have component type Object, int, long, short, char, byte, boolean, float, or double.

### The toString Method

Another important method in Object is the toString method that returns a string representing the value of this object. Here is a typical example. The toString method of the Point class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Most (but not all) toString methods follow this format: the name of the class, followed by the field values enclosed in square brackets. Here is an implementation of the toString method for the Employee class:

```java
public String toString()
{
    return "Employee[name=" + name
    + ",salary=" + salary
    + ",hireDay=" + hireDay
    + "]";
}
```

Actually, you can do a little better. Rather than hardwiring the class name into the toString method, call getClass().getName() to obtain a string with the class name.

```java
public String toString()
{
    return getClass().getName();
}
```
The toString method then also works for subclasses. Of course, the subclass programmer should define its own toString method and add the subclass fields. If the superclass uses getClass().getName(), then the subclass can simply call super.toString(). For example, here is a toString method for the Manager class:

```java
class Manager extends Employee {
    ...
    public String toString() {
        return super.toString() + 
            "[bonus=" + bonus + 
            "]";
    }
}
```

Now a Manager object is printed as

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

The toString method is ubiquitous for an important reason: whenever an object is concatenated with a string by the "+" operator, the Java compiler automatically invokes the toString method to obtain a string representation of the object. For example:

```java
Point p = new Point(10, 20);
String message = "The current position is " + p;
// automatically invokes p.toString()
```

TIP: Instead of writing `x.toString()`, you can write `"" + x`. This statement concatenates the empty string with the string representation of `x` that is exactly `x.toString()`. Unlike toString, this statement even works if `x` is of primitive type.

If `x` is any object and you call

```
System.out.println(x);
```
then the println method simply calls `x.toString()` and prints the resulting string.

The Object class defines the toString method to print the class name and the hash code of the object. For example, the call

```
System.out.println(System.out)
```
produces an output that looks like this:

```
java.io.PrintStream@2f6684
```

The reason is that the implementor of the PrintStream class didn’t bother to override the toString method.
CAUTION: Annoyingly, arrays inherit the toString method from Object, with the added twist that the array type is printed in an archaic format. For example,

```java
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

yields the string "[I@1a46e30". (The prefix [I denotes an array of integers.) The remedy is to call the static Arrays.toString method instead. The code

```java
String s = Arrays.toString(luckyNumbers);
```

yields the string "[2, 3, 5, 7, 11, 13]".

To correctly print multidimensional arrays (that is, arrays of arrays), use Arrays.deepToString.

The toString method is a great tool for logging. Many classes in the standard class library define the toString method so that you can get useful information about the state of an object. This is particularly useful in logging messages like this:

```java
System.out.println("Current position = " + position);
```

As we explain in Chapter 11, an even better solution is

```java
Logger.global.info("Current position = " + position);
```

TIP: We strongly recommend that you add a toString method to each class that you write. You, as well as other programmers who use your classes, will be grateful for the logging support.

The program in Listing 5–3 implements the equals, hashCode, and toString methods for the Employee and Manager classes.

```
Listing 5–3  EqualsTest.java

1. import java.util.*;
2. 
3. /**
4. * This program demonstrates the equals method.
5. * @version 1.11 2004-02-21
6. * @author Cay Horstmann
7. */
8. public class EqualsTest 
9. {
10.     public static void main(String[] args) 
11.     {
12.         Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
13.         Employee alice2 = alice1;
14.         Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15.         Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16.         System.out.println("alice1 == alice2: " + (alice1 == alice2));
17. ```
System.out.println("alice1 == alice3: " + (alice1 == alice3));
System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));
System.out.println("alice1.equals(bob): " + alice1.equals(bob));
System.out.println("bob.toString(): " + bob);
Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
System.out.println("boss.toString(): " + boss);
System.out.println("carl.equals(boss): " + carl.equals(boss));
System.out.println("alice1.hashCode(): " + alice1.hashCode());
System.out.println("alice3.hashCode(): " + alice3.hashCode());
System.out.println("bob.hashCode(): " + bob.hashCode());
System.out.println("carl.hashCode(): " + carl.hashCode());
}
}

class Employee
{
public Employee(String n, double s, int year, int month, int day)
{
    name = n;
salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
hireDay = calendar.getTime();
}

public String getName()
{
    return name;
}

public double getSalary()
{
    return salary;
}

public Date getHireDay()
{
    return hireDay;
}

public void raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
salary += raise;
}
public boolean equals(Object otherObject)
{
    // a quick test to see if the objects are identical
    if (this == otherObject) return true;

    // must return false if the explicit parameter is null
    if (otherObject == null) return false;

    // if the classes don't match, they can't be equal
    if (getClass() != otherObject.getClass()) return false;

    // now we know otherObject is a non-null Employee
    Employee other = (Employee) otherObject;

    // test whether the fields have identical values
    return name.equals(other.name) && salary == other.salary && hireDay.equals(other.hireDay);
}

public int hashCode()
{
    return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * hireDay.hashCode();
}

public String toString()
{
    return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
}

private String name;
private double salary;
private Date hireDay;

class Manager extends Employee
{
    public Manager(String n, double s, int year, int month, int day)
    {
        super(n, s, year, month, day);
        bonus = 0;
    }

    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

Listing 5–3  EqualsTest.java (continued)
Object: The Cosmic Superclass

Listing 5–3  EqualsTest.java (continued)

```java
public void setBonus(double b) {
    bonus = b;
}

public boolean equals(Object otherObject) {
    if (!super.equals(otherObject)) return false;
    Manager other = (Manager) otherObject;
    // super.equals checked that this and other belong to the same class
    return bonus == other.bonus;
}

public int hashCode() {
    return super.hashCode() + 17 * new Double(bonus).hashCode();
}

public String toString() {
    return super.toString() + "[bonus=\" + bonus + "]";
}

private double bonus;
```

API java.lang.Object 1.0

- getClass() returns a class object that contains information about the object. As you see later in this chapter, Java has a runtime representation for classes that is encapsulated in the Class class.
- equals(Object otherObject) compares two objects for equality; returns true if the objects point to the same area of memory, and false otherwise. You should override this method in your own classes.
- toString() returns a string that represents the value of this object. You should override this method in your own classes.
- clone() creates a clone of the object. The Java runtime system allocates memory for the new instance and copies the memory allocated for the current object.
NOTE: Cloning an object is important, but it also turns out to be a fairly subtle process filled with potential pitfalls for the unwary. We will have a lot more to say about the clone method in Chapter 6.

**API** java.lang.Class 1.0

- String getName()
  returns the name of this class.
- Class getSuperclass()
  returns the superclass of this class as a Class object.

**Generic Array Lists**

In many programming languages—in particular, in C—you have to fix the sizes of all arrays at compile time. Programmers hate this because it forces them into uncomfortable trade-offs. How many employees will be in a department? Surely no more than 100. What if there is a humongous department with 150 employees? Do we want to waste 90 entries for every department with just 10 employees?

In Java, the situation is much better. You can set the size of an array at runtime.

```java
int actualSize = ...;
Employee[] staff = new Employee[actualSize];
```

Of course, this code does not completely solve the problem of dynamically modifying arrays at runtime. Once you set the array size, you cannot change it easily. Instead, the easiest way in Java to deal with this common situation is to use another Java class, called ArrayList. The ArrayList class is similar to an array, but it automatically adjusts its capacity as you add and remove elements, without your needing to write any code.

As of Java SE 5.0, ArrayList is a **generic class** with a type parameter. To specify the type of the element objects that the array list holds, you append a class name enclosed in angle brackets, such as ArrayList<Employee>. You will see in Chapter 13 how to define your own generic class, but you don’t need to know any of those technicalities to use the ArrayList type.

Here we declare and construct an array list that holds Employee objects:

```java
ArrayList<Employee> staff = new ArrayList<Employee>();
```

NOTE: Before Java SE 5.0, there were no generic classes. Instead, there was a single ArrayList class, a “one size fits all” collection that holds elements of type Object. If you must use an older version of Java, simply drop all <...> suffixes. You can still use ArrayList without a <...> suffix in Java SE 5.0 and beyond. It is considered a “raw” type, with the type parameter erased.

NOTE: In even older versions of the Java programming language, programmers used the Vector class for dynamic arrays. However, the ArrayList class is more efficient, and there is no longer any good reason to use the Vector class.
You use the add method to add new elements to an array list. For example, here is how you populate an array list with employee objects:

```java
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

The array list manages an internal array of object references. Eventually, that array will run out of space. This is where array lists work their magic: If you call add and the internal array is full, the array list automatically creates a bigger array and copies all the objects from the smaller to the bigger array.

If you already know, or have a good guess, how many elements you want to store, then call the ensureCapacity method before filling the array list:

```java
staff.ensureCapacity(100);
```

That call allocates an internal array of 100 objects. Then, the first 100 calls to add do not involve any costly reallocation.

You can also pass an initial capacity to the ArrayList constructor:

```java
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

---

**CAUTION:** Allocating an array list as

```java
new ArrayList<Employee>(100) // capacity is 100
```

is *not* the same as allocating a new array as

```java
new Employee[100] // size is 100
```

There is an important distinction between the capacity of an array list and the size of an array. If you allocate an array with 100 entries, then the array has 100 slots, ready for use. An array list with a capacity of 100 elements has the potential of holding 100 elements (and, in fact, more than 100, at the cost of additional reallocations); but at the beginning, even after its initial construction, an array list holds no elements at all.

---

The size method returns the actual number of elements in the array list. For example,

```java
staff.size()
```

returns the current number of elements in the staff array list. This is the equivalent of

```java
a.length
```

for an array a.

Once you are reasonably sure that the array list is at its permanent size, you can call the trimToSize method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory.

Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use trimToSize when you are sure you won’t add any more elements to the array list.
Chapter 5  ■ Inheritance

C++ NOTE: The ArrayList class is similar to the C++ vector template. Both ArrayList and vector are generic types. But the C++ vector template overloads the [] operator for convenient element access. Because Java does not have operator overloading, it must use explicit method calls instead. Moreover, C++ vectors are copied by value. If a and b are two vectors, then the assignment a = b makes a into a new vector with the same length as b, and all elements are copied from b to a. The same assignment in Java makes both a and b refer to the same array list.

API  java.util.ArrayList<T>  1.2

- ArrayList<T>()
  constructs an empty array list.
- ArrayList<T>(int initialCapacity)
  constructs an empty array list with the specified capacity.
  Parameters: initialCapacity the initial storage capacity of the array list
- boolean add(T obj)
  appends an element at the end of the array list. Always returns true.
  Parameters: obj the element to be added
- int size()
  returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)
- void ensureCapacity(int capacity)
  ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.
  Parameters: capacity the desired storage capacity
- void trimToSize()
  reduces the storage capacity of the array list to its current size.

Accessing Array List Elements

Unfortunately, nothing comes for free. The automatic growth convenience that array lists give requires a more complicated syntax for accessing the elements. The reason is that the ArrayList class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.

Instead of using the pleasant [] syntax to access or change the element of an array, you use the get and set methods.

For example, to set the i-th element, you use
   staff.set(i, harry);
This is equivalent to
   a[i] = harry;
for an array a. (As with arrays, the index values are zero-based.)
CAUTION: Do not call `list.set(i, x)` until the `size` of the array list is larger than `i`. For example, the following code is wrong:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```

Use the `add` method instead of `set` to fill up an array, and use `set` only to replace a previously added element.

To get an array list element, use

```
Employee e = staff.get(i);
```

This is equivalent to

```
Employee e = a[i];
```

NOTE: Before Java SE 5.0, there were no generic classes, and the get method of the raw `ArrayList` class had no choice but to return an `Object`. Consequently, callers of `get` had to cast the returned value to the desired type:

```
Employee e = (Employee) staff.get(i);
```

The raw `ArrayList` is also a bit dangerous. Its `add` and `set` methods accept objects of any type. A call

```
staff.set(i, new Date());
```

compiles without so much as a warning, and you run into grief only when you retrieve the object and try to cast it. If you use an `ArrayList<Employee>` instead, the compiler will detect this error.

You can sometimes get the best of both worlds—flexible growth and convenient element access—with the following trick. First, make an array list and add all the elements:

```
ArrayList<X> list = new ArrayList<X>();
while ( . . . ) {
    x = . . . ;
    list.add(x);
}
```

When you are done, use the `toArray` method to copy the elements into an array:

```
X[] a = new X[list.size()];
list.toArray(a);
```

Sometimes, you need to add elements in the middle of an array list. Use the `add` method with an index parameter:

```
int n = staff.size() / 2;
staff.add(n, e);
```

The elements at locations `n` and above are shifted up to make room for the new entry. If the new size of the array list after the insertion exceeds the capacity, then the array list reallocates its storage array.
Similarly, you can remove an element from the middle of an array list:

   Employee e = staff.remove(n);

The elements located above it are copied down, and the size of the array is reduced by one.

Inserting and removing elements is not terribly efficient. It is probably not worth worrying about for small array lists. But if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead. We explain how to program with linked lists in Chapter 13.

As of Java SE 5.0, you can use the “for each” loop to traverse the contents of an array list:

```java
for (Employee e : staff)
    do something with e
```

This loop has the same effect as

```java
for (int i = 0; i < staff.size(); i++)
    {  
    Employee e = staff.get(i);  
    do something with e  
    }
```

Listing 5–4 is a modification of the EmployeeTest program of Chapter 4. The Employee[] array is replaced by an ArrayList<Employee>. Note the following changes:

- You don’t have to specify the array size.
- You use add to add as many elements as you like.
- You use size() instead of length to count the number of elements.
- You use a.get(i) instead of a[i] to access an element.

Listing 5–4  ArrayListTest.java

```java
1. import java.util.*;
2. 
3. // This program demonstrates the ArrayList class.
4. // Version 1.1 2004-02-21
5. // @author Cay Horstmann
6. 
7. public class ArrayListTest
8. {
9.    public static void main(String[] args)
10.    {
11.       // fill the staff array list with three Employee objects
12.       ArrayList<Employee> staff = new ArrayList<Employee>();
13.       staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
14.       staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
15.       staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
16.    }
17. }
```
Listing 5–4  ArrayListTest.java (continued)

```java
19.  // raise everyone's salary by 5%
20.  for (Employee e : staff)
21.     e.raiseSalary(5);
22.
23.  // print out information about all Employee objects
24.  for (Employee e : staff)
25.     System.out.println("name=" + e.getName() + ", salary=" + e.getSalary() + ", hireDay=
26.          + e.getHireDay());
27. }
28. }
29.
30. class Employee
31. {  
32.     public Employee(String n, double s, int year, int month, int day)
33.     {  
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         hireDay = calendar.getTime();
38.     }
39.     
40.     public String getName()
41.     {  
42.         return name;
43.     }
44.     
45.     public double getSalary()
46.     {  
47.         return salary;
48.     }
49.     
50.     public Date getHireDay()
51.     {  
52.         return hireDay;
53.     }
54.     
55.     public void raiseSalary(double byPercent)
56.     {  
57.         double raise = salary * byPercent / 100;
58.         salary += raise;
59.     }
60.     
61.     private String name;
62.     private double salary;
63.     private Date hireDay;
64. }
```
Chapter 5 ■ Inheritance

### API

```java
java.util.ArrayList<T> 1.2
```

- **void set(int index, T obj)**
  puts a value in the array list at the specified index, overwriting the previous contents.
  
  **Parameters:**
  - `index` the position (must be between 0 and size() - 1)
  - `obj` the new value

- **T get(int index)**
  gets the value stored at a specified index.
  
  **Parameters:**
  - `index` the index of the element to get (must be between 0 and size() - 1)

- **void add(int index, T obj)**
  shifts up elements to insert an element.
  
  **Parameters:**
  - `index` the insertion position (must be between 0 and size())
  - `obj` the new element

- **T remove(int index)**
  removes an element and shifts down all elements above it. The removed element is returned.
  
  **Parameters:**
  - `index` the position of the element to be removed (must be between 0 and size() - 1)

### Compatibility between Typed and Raw Array Lists

When you write new code with Java SE 5.0 and beyond, you should use type parameters, such as `ArrayList<Employee>`, for array lists. However, you may need to interoperate with existing code that uses the raw `ArrayList` type.

Suppose that you have the following legacy class:

```java
public class EmployeeDB
{
    public void update(ArrayList list) { ... }
    public ArrayList find(String query) { ... }
}
```

You can pass a typed array list to the `update` method without any casts.

```java
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

The staff object is simply passed to the `update` method.

⚠️ **CAUTION:** Even though you get no error or warning from the compiler, this call is not completely safe. The `update` method might add elements into the array list that are not of type `Employee`. When these elements are retrieved, an exception occurs. This sounds scary, but if you think about it, the behavior is simply as it was before Java SE 5.0. The integrity of the virtual machine is never jeopardized. In this situation, you do not lose security, but you also do not benefit from the compile-time checks.
Conversely, when you assign a raw ArrayList to a typed one, you get a warning.

```java
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```

**NOTE:** To see the text of the warning, compile with the option `-Xlint:unchecked`.

Using a cast does not make the warning go away.

```java
ArrayList<Employee> result = (ArrayList<Employee>)
    employeeDB.find(query); // yields another warning
```

Instead, you get a different warning, telling you that the cast is misleading.

This is the consequence of a somewhat unfortunate limitation of generic types in Java. For compatibility, the compiler translates all typed array lists into raw `ArrayList` objects after checking that the type rules were not violated. In a running program, all array lists are the same—there are no type parameters in the virtual machine. Thus, the casts `(ArrayList)` and `(ArrayList<Employee>)` carry out identical runtime checks.

There isn’t much you can do about that situation. When you interact with legacy code, study the compiler warnings and satisfy yourself that the warnings are not serious.

### Object Wrappers and Autoboxing

Occasionally, you need to convert a primitive type like `int` to an object. All primitive types have class counterparts. For example, a class `Integer` corresponds to the primitive type `int`. These kinds of classes are usually called **wrappers**. The wrapper classes have obvious names: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void`, and `Boolean`. (The first six inherit from the common superclass `Number`. The wrapper classes are immutable—you cannot change a wrapped value after the wrapper has been constructed. They are also final, so you cannot subclass them.

Suppose we want an array list of integers. Unfortunately, the type parameter inside the angle brackets cannot be a primitive type. It is not possible to form an `ArrayList<int>`.

Here, the `Integer` wrapper class comes in. It is ok to declare an array list of `Integer` objects.

```java
ArrayList<Integer> list = new ArrayList<Integer>();
```

**CAUTION:** An `ArrayList<Integer>` is far less efficient than an `int[]` array because each value is separately wrapped inside an object. You would only want to use this construct for small collections when programmer convenience is more important than efficiency.

Another Java SE 5.0 innovation makes it easy to add and get array elements. The call

```java
list.add(3);
```

is automatically translated to

```java
list.add(new Integer(3));
```

This conversion is called **autoboxing**.

**NOTE:** You might think that **autowrapping** would be more consistent, but the “boxing” metaphor was taken from C#.
Conversely, when you assign an `Integer` object to an `int` value, it is automatically unboxed. That is, the compiler translates

```java
int n = list.get(i);
```

into

```java
int n = list.get(i).intValue();
```

Automatic boxing and unboxing even works with arithmetic expressions. For example, you can apply the increment operator to a wrapper reference:

```java
Integer n = 3;
n++;
```

The compiler automatically inserts instructions to unbox the object, increment the resulting value, and box it back.

In most cases, you get the illusion that the primitive types and their wrappers are one and the same. There is just one point in which they differ considerably: identity. As you know, the `==` operator, applied to wrapper objects, only tests whether the objects have identical memory locations. The following comparison would therefore probably fail:

```java
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

However, a Java implementation may, if it chooses, wrap commonly occurring values into identical objects, and thus the comparison might succeed. This ambiguity is not what you want. The remedy is to call the `equals` method when comparing wrapper objects.

NOTE: The autoboxing specification requires that `boolean`, `byte`, `char` (127, and `short` and `int` between –128 and 127) are wrapped into fixed objects. For example, if `a` and `b` had been initialized with 100 in the preceding example, then the comparison would have had to succeed.

Finally, let us emphasize that boxing and unboxing is a courtesy of the compiler, not the virtual machine. The compiler inserts the necessary calls when it generates the bytecodes of a class. The virtual machine simply executes those bytecodes.

You will often see the number wrappers for another reason. The designers of Java found the wrappers a convenient place to put certain basic methods, like the ones for converting strings of digits to numbers.

To convert a string to an integer, you use the following statement:

```java
int x = Integer.parseInt(s);
```

This has nothing to do with `Integer` objects—`parseInt` is a static method. But the `Integer` class was a good place to put it.

The API notes show some of the more important methods of the `Integer` class. The other number classes implement corresponding methods.
CAUTION: Some people think that the wrapper classes can be used to implement methods that can modify numeric parameters. However, that is not correct. Recall from Chapter 4 that it is impossible to write a Java method that increments an integer parameter because parameters to Java methods are always passed by value.

```java
public static void triple(int x) // won't work
{
    x = 3 * x; // modifies local variable
}
```

Could we overcome this by using an `Integer` instead of an `int`?

```java
public static void triple(Integer x) // won't work
{
    ...
}
```

The problem is that `Integer` objects are immutable: the information contained inside the wrapper can't change. You cannot use these wrapper classes to create a method that modifies numeric parameters.

If you do want to write a method to change numeric parameters, you can use one of the `holder` types defined in the `org.omg.CORBA` package. There are types `IntHolder`, `BooleanHolder`, and so on. Each holder type has a public (!) field `value` through which you can access the stored value.

```java
public static void triple(IntHolder x)
{
    x.value = 3 * x.value;
}
```

---

**API**

```java
java.lang.Integer 1.0
```

- `int intValue()`
  - returns the value of this `Integer` object as an `int` (overrides the `intValue` method in the `Number` class).

- `static String toString(int i)`
  - returns a new `String` object representing the number `i` in base 10.

- `static String toString(int i, int radix)`
  - lets you return a representation of the number `i` in the base specified by the `radix` parameter.

- `static int parseInt(String s)`

- `static int parseInt(String s, int radix)`
  - returns the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).

- `static Integer valueOf(String s)`

- `static Integer valueOf(String s, int radix)`
  - returns a new `Integer` object initialized to the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).
Methods with a Variable Number of Parameters

Before Java SE 5.0, every Java method had a fixed number of parameters. However, it is now possible to provide methods that can be called with a variable number of parameters. (These are sometimes called “varargs” methods.)

You have already seen such a method: printf. For example, the calls

```java
System.out.printf("%d", n);
```

and

```java
System.out.printf("%d %s", n, "widgets");
```

both call the same method, even though one call has two parameters and the other has three.

The printf method is defined like this:

```java
public class PrintStream
{
    public PrintStream printf(String fmt, Object... args) { return format(fmt, args); }
}
```

Here, the ellipsis ... is a part of the Java code. It denotes that the method can receive an arbitrary number of objects (in addition to the fmt parameter).

The printf method actually receives two parameters, the format string, and an Object[] array that holds all other parameters. (If the caller supplies integers or other primitive type values, autoboxing turns them into objects.) It now has the unenviable task of scanning the fmt string and matching up the i\text{th} format specifier with the value args[i].

In other words, for the implementor of printf, the Object... parameter type is exactly the same as Object[].

The compiler needs to transform each call to printf, bundling the parameters into an array and autoboxing as necessary:

```java
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" });
```

You can define your own methods with variable parameters, and you can specify any type for the parameters, even a primitive type. Here is a simple example: a function that computes the maximum of a variable number of values.

```java
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

Simply call the function like this:

```java
double m = max(3.1, 40.4, -5);
```

The compiler passes a new double[] { 3.1, 40.4, -5 } to the max function.
NOTE: It is legal to pass an array as the last parameter of a method with variable parameters. For example:

```java
System.out.printf("%d %s", new Object[ ] { new Integer(1), "widgets" } );
```

Therefore, you can redefine an existing function whose last parameter is an array to a method with variable parameters, without breaking any existing code. For example, `MessageFormat.format` was enhanced in this way in Java SE 5.0. If you like, you can even declare the main method as

```java
public static void main(String... args)
```

### Enumeration Classes

You saw in Chapter 3 how to define enumerated types in Java SE 5.0 and beyond. Here is a typical example:

```java
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE ; } 
```

The type defined by this declaration is actually a class. The class has exactly four instances—it is not possible to construct new objects. Therefore, you never need to use `equals` for values of enumerated types. Simply use `==` to compare them.

You can, if you like, add constructors, methods, and fields to an enumerated type. Of course, the constructors are only invoked when the enumerated constants are constructed. Here is an example.

```java
enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }

    private String abbreviation;
}
```

All enumerated types are subclasses of the class `Enum`. They inherit a number of methods from that class. The most useful one is `toString`, which returns the name of the enumerated constant. For example, `Size.SMALL.toString()` returns the string "SMALL".

The converse of `toString` is the static `valueOf` method. For example, the statement

```java
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

sets `s` to `Size.SMALL`.

Each enumerated type has a static `values` method that returns an array of all values of the enumeration. For example, the call

```java
Size[] values = Size.values();
```

returns the array with elements `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE`, and `Size.EXTRA_LARGE`.

The `ordinal` method yields the position of an enumerated constant in the `enum` declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns 1.

The short program in Listing 5–5 demonstrates how to work with enumerated types.
NOTE: The Enum class has a type parameter that we have ignored for simplicity. For example, the enumerated type Size actually extends Enum<Size>. The type parameter is used in the compareTo method. (We discuss the compareTo method in Chapter 6 and type parameters in Chapter 12.)

Listing 5–5 EnumTest.java

```java
import java.util.*;

public class EnumTest {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
        String input = in.next().toUpperCase();
        Size size = Enum.valueOf(Size.class, input);
        System.out.println("size=");
        System.out.println("abbreviation=");
        if (size == Size.EXTRA_LARGE) {
            System.out.println("Good job--you paid attention to the _.");
        }
    }
}

eenum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
    private String abbreviation;
}
```

API java.lang.Enum<E> 5.0

- static Enum valueOf(Class enumClass, String name)
  returns the enumerated constant of the given class with the given name.
- String toString()
  returns the name of this enumerated constant.
- int ordinal()
  returns the zero-based position of this enumerated constant in the enum declaration.
• int compareTo(E other)
  returns a negative integer if this enumerated constant comes before other, zero if
  this == other, and a positive integer otherwise. The ordering of the constants is
  given by the enum declaration.

### Reflection

The reflection library gives you a very rich and elaborate toolset to write programs
that manipulate Java code dynamically. This feature is heavily used in JavaBeans, the
component architecture for Java (see Volume II for more on JavaBeans). Using reflection, Java can support tools like the ones to which users of Visual Basic have grown accustomed. In particular, when new classes are added at design or runtime, rapid application development tools can dynamically inquire about the capabilities of the classes that were added.

A program that can analyze the capabilities of classes is called reflective. The reflection mechanism is extremely powerful. As the next sections show, you can use it to
• Analyze the capabilities of classes at runtime;
• Inspect objects at runtime, for example, to write a single toString method that works
  for all classes;
• Implement generic array manipulation code; and
• Take advantage of Method objects that work just like function pointers in languages
  such as C++.

Reflection is a powerful and complex mechanism; however, it is of interest mainly to
tool builders, not application programmers. If you are interested in programming applica-
tions rather than tools for other Java programmers, you can safely skip the remainder
of this chapter and return to it later.

### The Class Class

While your program is running, the Java runtime system always maintains what is
called runtime type identification on all objects. This information keeps track of the class
to which each object belongs. Runtime type information is used by the virtual machine
to select the correct methods to execute.

However, you can also access this information by working with a special Java class. The
class that holds this information is called, somewhat confusingly, Class. The getClass()
method in the Object class returns an instance of Class type.

```java
Employee e;
    ...
Class cl = e.getClass();
```

Just like an Employee object describes the properties of a particular employee, a Class object
describes the properties of a particular class. Probably the most commonly used method
of Class is getName. This returns the name of the class. For example, the statement
```
System.out.println(e.getClass().getName() + " " + e.getName());
```
prints
```
Employee Harry Hacker
```
if e is an employee, or
```
Manager Harry Hacker
```
if e is a manager.

If the class is in a package, the package name is part of the class name:

```java
Date d = new Date();
Class cl = d.getClass();
String name = cl.getName(); // name is set to "java.util.Date"
```

You can obtain a Class object corresponding to a class name by using the static `forName` method.

```java
String className = "java.util.Date";
Class cl = Class.forName(className);
```

You would use this method if the class name is stored in a string that varies at runtime. This works if `className` is the name of a class or interface. Otherwise, the `forName` method throws a checked exception. See the section “A Primer on Catching Exceptions” on page 219 to see how to supply an exception handler whenever you use this method.

**TIP:** At startup, the class containing your `main` method is loaded. It loads all classes that it needs. Each of those loaded classes loads the classes that it needs, and so on. That can take a long time for a big application, frustrating the user. You can give users of your program the illusion of a faster start with the following trick. Make sure that the class containing the `main` method does not explicitly refer to other classes. First display a splash screen. Then manually force the loading of other classes by calling `Class.forName`.

A third method for obtaining an object of type `Class` is a convenient shorthand. If `T` is any Java type, then `T.class` is the matching class object. For example:

```java
Class cl1 = Date.class; // if you import java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Note that a `Class` object really describes a type, which may or may not be a class. For example, `int` is not a class, but `int.class` is nevertheless an object of type `Class`.

**NOTE:** As of Java SE 5.0, the `Class` class is parameterized. For example, `Employee.class` is of type `Class<Employee>`. We are not dwelling on this issue because it would further complicate an already abstract concept. For most practical purposes, you can ignore the type parameter and work with the raw `Class` type. See Chapter 13 for more information on this issue.

**CAUTION:** For historical reasons, the `getName` method returns somewhat strange names for array types:

- `Double[].class.getName()` returns `"[Ljava.lang.Double;"
- `int[].class.getName()` returns `"[I"

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the `==` operator to compare class objects. For example:

```java
if (e.getClass() == Employee.class) . . .
```
Another example of a useful method is one that lets you create an instance of a class on the fly. This method is called, naturally enough, `newInstance()`. For example,

```java
e.getClass().newInstance();
```

creates a new instance of the same class type as `e`. The `newInstance` method calls the default constructor (the one that takes no parameters) to initialize the newly created object. An exception is thrown if the class has no default constructor.

Using a combination of `forName` and `newInstance` lets you create an object from a class name stored in a string:

```java
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```

**NOTE:** If you need to provide parameters for the constructor of a class you want to create by name in this manner, then you can't use statements like the preceding. Instead, you must use the `newInstance` method in the `Constructor` class.

**C++ NOTE:** The `newInstance` method corresponds to the idiom of a virtual constructor in C++. However, virtual constructors in C++ are not a language feature but just an idiom that needs to be supported by a specialized library. The `Class` class is similar to the `type_info` class in C++, and the `getClass` method is equivalent to the `typeid` operator. The Java `Class` is quite a bit more versatile than `type_info`, though. The C++ `type_info` can only reveal a string with the name of the type, not create new objects of that type.

---

**A Primer on Catching Exceptions**

We cover exception handling fully in Chapter 11, but in the meantime you will occasionally encounter methods that threaten to throw exceptions.

When an error occurs at runtime, a program can “throw an exception.” Throwing an exception is more flexible than terminating the program because you can provide a handler that “catches” the exception and deals with it.

If you don’t provide a handler, the program still terminates and prints a message to the console, giving the type of the exception. You may already have seen exception reports when you accidentally used a null reference or overstepped the bounds of an array.

There are two kinds of exceptions: unchecked exceptions and checked exceptions. With checked exceptions, the compiler checks that you provide a handler. However, many common exceptions, such as accessing a null reference, are unchecked. The compiler does not check whether you provide a handler for these errors—after all, you should spend your mental energy on avoiding these mistakes rather than coding handlers for them.

But not all errors are avoidable. If an exception can occur despite your best efforts, then the compiler insists that you provide a handler. The `Class.forName` method is an example of a method that throws a checked exception. In Chapter 11, you will see several exception handling strategies. For now, we just show you the simplest handler implementation.
Place one or more statements that might throw checked exceptions inside a try block. Then provide the handler code in the catch clause.

```java
try {
    statements that might throw exceptions
} catch(Exception e) {
    handler action
}
```

Here is an example:

```java
try {
    String name = ...; // get class name
    Class cl = Class.forName(name); // might throw exception
    ... // do something with cl
} catch(Exception e) {
    e.printStackTrace();
}
```

If the class name doesn’t exist, the remainder of the code in the try block is skipped and the program enters the catch clause. (Here, we print a stack trace by using the `printStackTrace()` method of the `Throwable` class. `Throwable` is the superclass of the `Exception` class.) If none of the methods in the try block throws an exception, the handler code in the catch clause is skipped.

You only need to supply an exception handler for checked exceptions. It is easy to find out which methods throw checked exceptions—the compiler will complain whenever you call a method that threatens to throw a checked exception and you don’t supply a handler.

---

**API**

**java.lang.Class 1.0**

- `static Class forName(String className)`
  returns the `Class` object representing the class with name `(className).

- `Object newInstance()`
  returns a new instance of this class.

**API**

**java.lang.reflect.Constructor 1.1**

- `Object newInstance(Object[] args)`
  constructs a new instance of the constructor’s declaring class.

  **Parameters:** `args` the parameters supplied to the constructor. See the section on reflection for more information on how to supply parameters.
void printStackTrace()

prints the Throwable object and the stack trace to the standard error stream.

Using Reflection to Analyze the Capabilities of Classes

Here is a brief overview of the most important parts of the reflection mechanism for letting you examine the structure of a class.

The three classes Field, Method, and Constructor in the java.lang.reflect package describe the fields, methods, and constructors of a class, respectively. All three classes have a method called getName that returns the name of the item. The Field class has a method getType that returns an object, again of type Class, that describes the field type. The Method and Constructor classes have methods to report the types of the parameters, and the Method class also reports the return type. All three of these classes also have a method called getModifiers that returns an integer, with various bits turned on and off, that describes the modifiers used, such as public and static. You can then use the static methods in the Modifier class in the java.lang.reflect package to analyze the integer that getModifiers returns. Use methods like isPublic, isPrivate, or isFinal in the Modifier class to tell whether a method or constructor was public, private, or final. All you have to do is have the appropriate method in the Modifier class work on the integer that getModifiers returns. You can also use the Modifier.toString method to print the modifiers.

The getFields, getMethods, and getConstructors methods of the Class class return arrays of the public fields, methods, and constructors that the class supports. This includes public members of superclasses. The getDeclaredFields, getDeclaredMethods, and getDeclaredConstructors methods of the Class class return arrays consisting of all fields, methods, and constructors that are declared in the class. This includes private and protected members, but not members of superclasses.

Listing 5–6 shows you how to print out all information about a class. The program prompts you for the name of a class and then writes out the signatures of all methods and constructors as well as the names of all data fields of a class. For example, if you enter

java.lang.Double

the program prints

public class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
    public static boolean isInfinite(double);
    public boolean isInfinite();

    public static java.lang.Number valueOf(double value);
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
public static double parseDouble(java.lang.String);
public static native long doubleToLongBits(double);
public static native long doubleToRawLongBits(double);
public static native double longBitsToDouble(long);

public static final double POSITIVE_INFINITY;
public static final double NEGATIVE_INFINITY;
public static final double MAX_VALUE;
public static final double MIN_VALUE;
public static final java.lang.Class TYPE;
private double value;
private static final long serialVersionUID;
}

What is remarkable about this program is that it can analyze any class that the Java interpreter can load, not just the classes that were available when the program was compiled. We use this program in the next chapter to peek inside the inner classes that the Java compiler generates automatically.

Listing 5–6  ReflectionTest.java

```java
import java.util.*;
import java.lang.reflect.*;

public class ReflectionTest {

    public static void main(String[] args) {
        // read class name from command line args or user input
        String name;
        if (args.length > 0) name = args[0];
        else {
            Scanner in = new Scanner(System.in);
            System.out.println("Enter class name (e.g. java.util.Date): ");
            name = in.next();
        }
        try {
            //...
// print class name and superclass name (if != Object)
Class cl = Class.forName(name);
Class supercl = cl.getSuperclass();
String modifiers = Modifier.toString(cl.getModifiers());
if (modifiers.length() > 0) System.out.print(modifiers + " ");
System.out.print("class " + name);
if (supercl != null && supercl != Object.class) System.out.print(" extends "
  + supercl.getName());

System.out.println("\n\n");
printConstructors(cl);
System.out.println();
printMethods(cl);
System.out.println();
printFields(cl);
System.out.println("} ");
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
System.exit(0);

<--5-6>
/*
 * Prints all constructors of a class
 * @param cl a class
 */
public static void printConstructors(Class cl)
{
  Constructor[] constructors = cl.getDeclaredConstructors();
  for (Constructor c : constructors)
  {
    String name = c.getName();
    System.out.print(" ");
    String modifiers = Modifier.toString(c.getModifiers());
    if (modifiers.length() > 0) System.out.print(modifiers + " ");
    System.out.print(name + "\n");
    // print parameter types
    Class[] paramTypes = c.getParameterTypes();
    for (int j = 0; j < paramTypes.length; j++)
    {
      if (j > 0) System.out.print(" , ");
      System.out.print(paramTypes[j].getName());
    }
    System.out.println("\n");
  }
}
74. } 
75. } 
76. /** 
77. * Prints all methods of a class 
78. * @param cl a class 
79. */ 
80. public static void printMethods(Class cl) 
81. { 
82. Method[] methods = cl.getDeclaredMethods(); 
83. 
84. for (Method m : methods) 
85. { 
86. Class retType = m.getReturnType(); 
87. String name = m.getName(); 
88. 
89. System.out.print(" "); 
90. // print modifiers, return type, and method name 
91. String modifiers = Modifier.toString(m.getModifiers()); 
92. if (modifiers.length() > 0) System.out.print(modifiers + " "); 
93. System.out.print(retType.getName() + " " + name + "("); 
94. 
95. // print parameter types 
96. Class[] paramTypes = m.getParameterTypes(); 
97. for (int j = 0; j < paramTypes.length; j++) 
98. { 
99. if (j > 0) System.out.print(", "); 
100. System.out.print(paramTypes[j].getName()); 
101. } 
102. System.out.println(";"); 
103. 
104. } 
105. 
106. /** 
107. * Prints all fields of a class 
108. * @param cl a class 
109. */ 
110. public static void printFields(Class cl) 
111. { 
112. Field[] fields = cl.getDeclaredFields(); 
113. 
114. for (Field f : fields) 
115. { 
116. Class type = f.getType(); 
117. String name = f.getName(); 
118. System.out.print(" "); 
119. String modifiers = Modifier.toString(f.getModifiers()); 
120. if (modifiers.length() > 0) System.out.print(modifiers + " "); 
121. System.out.println(type.getName() + " " + name + ");"); 
122. 
123. } 
124. }
Reflection

```java
java.lang.Class 1.0
```

- `Field[] getFields() 1.1`  
  `Field[] getDeclaredFields() 1.1`  
  getFields returns an array containing Field objects for the public fields of this class or its superclasses; getDeclaredFields returns an array of Field objects for all fields of this class. The methods return an array of length 0 if there are no such fields or if the Class object represents a primitive or array type.

- `Method[] getMethods() 1.1`  
  `Method[] getDeclaredMethods() 1.1`  
  returns an array containing Method objects: getMethods returns public methods and includes inherited methods; getDeclaredMethods returns all methods of this class or interface but does not include inherited methods.

- `Constructor[] getConstructors() 1.1`  
  ` Constructor[] getDeclaredConstructors() 1.1`  
  returns an array containing Constructor objects that give you all the public constructors (for getConstructors) or all constructors (for getDeclaredConstructors) of the class represented by this Class object.

```java
java.lang.reflect.Field 1.1
```

```java
java.lang.reflect.Method 1.1
```

```java
java.lang.reflect.Constructor 1.1
```

- `Class getDeclaringClass()`  
  returns the Class object for the class that defines this constructor, method, or field.

- `Class[] getExceptionTypes() (in Constructor and Method classes)`  
  returns an array of Class objects that represent the types of the exceptions thrown by the method.

- `int getModifiers()`  
  returns an integer that describes the modifiers of this constructor, method, or field. Use the methods in the Modifier class to analyze the return value.

- `String getName()`  
  returns a string that is the name of the constructor, method, or field.

- `Class[] getParameterTypes() (in Constructor and Method classes)`  
  returns an array of Class objects that represent the types of the parameters.

- `Class getReturnType()` (in Method classes)  
  returns a Class object that represents the return type.

```java
java.lang.reflect.Modifier 1.1
```

- `static String toString(int modifiers)`  
  returns a string with the modifiers that correspond to the bits set in modifiers.
Chapter 5  ■ Inheritance

- static boolean isAbstract(int modifiers)
- static boolean isFinal(int modifiers)
- static boolean isInterface(int modifiers)
- static boolean isNative(int modifiers)
- static boolean isPrivate(int modifiers)
- static boolean isProtected(int modifiers)
- static boolean isPublic(int modifiers)
- static boolean isStatic(int modifiers)
- static boolean isStrict(int modifiers)
- static boolean isSynchronized(int modifiers)
- static boolean isVolatile(int modifiers)

• tests the bit in the modifiers value that corresponds to the modifier in the method name.

Using Reflection to Analyze Objects at Runtime

In the preceding section, we saw how we can find out the names and types of the data fields of any object:

- Get the corresponding Class object.
- Call getDeclaredFields on the Class object.

In this section, we go one step further and actually look at the contents of the data fields. Of course, it is easy to look at the contents of a specific field of an object whose name and type are known when you write a program. But reflection lets you look at fields of objects that were not known at compile time.

The key method to achieve this examination is the get method in the Field class. If f is an object of type Field (for example, one obtained from getDeclaredFields) and obj is an object of the class of which f is a field, then f.get(obj) returns an object whose value is the current value of the field of obj. This is all a bit abstract, so let’s run through an example.

Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();  // the class object representing Employee
Field f = cl.getDeclaredField("name");  // the name field of the Employee class
Object v = f.get(harry);  // the value of the name field of the harry object
  // i.e., the String object "Harry Hacker"

Actually, there is a problem with this code. Because the name field is a private field, the get method will throw an IllegalArgumentException. You can only use the get method to get the values of accessible fields. The security mechanism of Java lets you find out what fields any object has, but it won’t let you read the values of those fields unless you have access permission.

The default behavior of the reflection mechanism is to respect Java access control. However, if a Java program is not controlled by a security manager that disallows it, you can override access control. To do this, invoke the setAccessible method on a Field, Method, or Constructor object. For example:

f.setAccessible(true);  // now OK to call f.get(harry);
The `AccessibleObject` class, the common superclass of the `Field`, `Method`, and `Constructor` classes. This feature is provided for debuggers, persistent storage, and similar mechanisms. We use it for a generic `toString` method later in this section.

There is another issue with the `get` method that we need to deal with. The `name` field is a `String`, and so it is not a problem to return the value as an `Object`. But suppose we want to look at the `salary` field. That is a `double`, and in Java, number types are not objects. To handle this, you can either use the `getDouble` method of the `Field` class, or you can call `get`, whereby the reflection mechanism automatically wraps the field value into the appropriate wrapper class, in this case, `Double`.

Of course, you can also set the values that you can get. The call `f.set(obj, value)` sets the field represented by `f` of the object `obj` to the new value.

Listing 5–7 shows how to write a generic `toString` method that works for any class. It uses `getDeclaredFields` to obtain all data fields. It then uses the `setAccessible` convenience method to make all fields accessible. For each field, it obtains the name and the value. Listing 5–7 turns each value into a string by recursively invoking `toString`.

```java
class ObjectAnalyzer {
    public String toString(Object obj) {
        Class cl = obj.getClass();
        ...;
        String r = cl.getName();
        // inspect the fields of this class and all superclasses
        do {
            r += "[";
            Field[] fields = cl.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);
            // get the names and values of all fields
            for (Field f : fields) {
                if (!Modifier.isStatic(f.getModifiers())) {
                    if (!r.endsWith("["))) r += ",”
                    r += f.getName() + “=";
                    try {
                        Object val = f.get(obj);
                        r += toString(val);
                    }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
            r += “]“;
            cl = cl.getSuperclass();
        } while (cl != null);
        return r;
    }
}
```
Chapter 5  ■  Inheritance

The complete code in Listing 5–7 needs to address a couple of complexities. Cycles of references could cause an infinite recursion. Therefore, the ObjectAnalyzer keeps track of objects that were already visited. Also, to peek inside arrays, you need a different approach. You’ll learn about the details in the next section.

You can use this toString method to peek inside any object. For example, the call

```java
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

yields the printout

```java
java.util.ArrayList[elementData=class java.lang.Object[]
{java.lang.Integer[value=1]
[null],
java.lang.Integer[value=4]
[null],
java.lang.Integer[value=9]
[null],
java.lang.Integer[value=16]
[null],
java.lang.Integer[value=25]
[null],
null, null, null, null}, size=5][modCount=5][]
```

You can use this generic toString method to implement the toString methods of your own classes, like this:

```java
public String toString()
{
    return new ObjectAnalyzer().toString(this);
}
```

This is a hassle-free method for supplying a toString method that you may find useful in your own programs.

**Listing 5–7  ObjectAnalyzerTest.java**

```java
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5. * This program uses reflection to spy on objects.
6. * @version 1.11 2004-02-21
7. * @author Cay Horstmann
8. */
9. public class ObjectAnalyzerTest
10. {
11.    public static void main(String[] args)
12.    {
13.        ArrayList<Integer> squares = new ArrayList<Integer>();
14.        for (int i = 1; i <= 5; i++)
15.            squares.add(i * i);
16.        System.out.println(new ObjectAnalyzer().toString(squares));
17.    }
18. }
19.
20. class ObjectAnalyzer
21. {
```
/**
 * Converts an object to a string representation that lists all fields.
 * @param obj an object
 * @return a string with the object's class name and all field names and
 * values
 */

public String toString(Object obj) {
    if (obj == null) return "null";
    if (visited.contains(obj)) return "...";
    visited.add(obj);
    Class cl = obj.getClass();
    if (cl == String.class) return (String) obj;
    if (cl.isArray()) {
        String r = cl.getComponentType() + "[";
        for (int i = 0; i < Array.getLength(obj); i++) {
            if (i > 0) r += ",";
            Object val = Array.get(obj, i);
            if (cl.getComponentType().isPrimitive()) r += val;
            else r += toString(val);
        }
        return r + "]";
    }
    String r = cl.getName();
    // inspect the fields of this class and all superclasses
    do {
        r += "$\{\";
        Field[] fields = cl.getDeclaredFields();
        AccessibleObject.setAccessible(fields, true);
        // get the names and values of all fields
        for (Field f : fields) {
            if (!Modifier.isStatic(f.getModifiers())) {
                if (!r.endsWith("\"\")) r += ",";
                r += f.getName() + ":";
                try {
                    Class t = f.getType();
                    Object val = f.get(obj);
                    if (!t.isPrimitive()) r += val;
                    else r += toString(val);
                } catch (Exception e) {
                }
            }
        }
    }
    String r = cl.getName();
    // inspect the fields of this class and all superclasses
    do {
        r += "$\{\";
        Field[] fields = cl.getDeclaredFields();
        AccessibleObject.setAccessible(fields, true);
        // get the names and values of all fields
        for (Field f : fields) {
            if (!Modifier.isStatic(f.getModifiers())) {
                if (!r.endsWith("\"\")) r += ",";
                r += f.getName() + ":";
                try {
                    Class t = f.getType();
                    Object val = f.get(obj);
                    if (!t.isPrimitive()) r += val;
                    else r += toString(val);
                } catch (Exception e) {
                }
            }
        }
    }
void setAccessible(boolean flag)
sets the accessibility flag for this reflection object. A value of true indicates that
Java language access checking is suppressed and that the private properties of the
object can be queried and set.

boolean isAccessible()
gets the value of the accessibility flag for this reflection object.

static void setAccessible(AccessibleObject[] array, boolean flag)
is a convenience method to set the accessibility flag for an array of objects.

Field getField(String name)
Field[] getFields()
gets the public field with the given name, or an array of all fields.

Field getDeclaredField(String name)
Field[] getDeclaredFields()
gets the field that is declared in this class with the given name, or an array of all
fields.

Object get(Object obj)
gets the value of the field described by this Field object in the object obj.

void set(Object obj, Object newValue)
sets the field described by this Field object in the object obj to a new value.
Using Reflection to Write Generic Array Code

The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. For example, when you use this feature with the `arraycopy` method from Chapter 3, you can dynamically expand an existing array while preserving the current contents.

The problem we want to solve is pretty typical. Suppose you have an array of some type that is full and you want to grow it. And suppose you are sick of writing the grow-and-copy code by hand. You want to write a generic method to grow an array.

```java
Employee[] a = new Employee[100];
...
// array is full
a = (Employee[]) arrayGrow(a);
```

How can we write such a generic method? It helps that an `Employee[]` array can be converted to an `Object[]` array. That sounds promising. Here is a first attempt to write a generic method. We simply grow the array by 10% + 10 elements (because the 10 percent growth is not substantial enough for small arrays).

```java
static Object[] badArrayGrow(Object[] a) // not useful
{
    int newLength = a.length * 11 / 10 + 10;
    Object[] newArray = new Object[newLength];
    System.arraycopy(a, 0, newArray, 0, a.length);
    return newArray;
}
```

However, there is a problem with actually using the resulting array. The type of array that this code returns is an array of objects (`Object[]`) because we created the array using the line of code

```java
new Object[newLength]
```

An array of objects cannot be cast to an array of `Employee[]`s. Java would generate a `ClassCastException` at runtime. The point is, as we mentioned earlier, that a Java array remembers the type of its entries, that is, the element type used in the new expression that created it. It is legal to cast an `Employee[]` temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into an `Employee[]` array. To write this kind of generic array code, we need to be able to make a new array of the same type as the original array. For this, we need the methods of the `Array` class in the `java.lang.reflect` package. The key is the static `newInstance` method of the `Array` class that constructs a new array. You must supply the type for the entries and the desired length as parameters to this method.

```java
Object newArray = Array.newInstance(componentType, newLength);
```

To actually carry this out, we need to get the length and component type of the new array.

We obtain the length by calling `Array.getLength(a)`. The static `getLength` method of the `Array` class returns the length of any array. To get the component type of the new array:

1. First, get the class object of `a`.
2. Confirm that it is indeed an array.
3. Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.
Why is `getLength` a method of `Array` but `getComponentType` a method of `Class`? We don’t know—the distribution of the reflection methods seems a bit ad hoc at times.

Here’s the code:

```java
static Object goodArrayGrow(Object a) // useful
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    int newLength = (length * 11 / 10 + 10);
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, length);
    return newArray;
}
```

Note that this `arrayGrow` method can be used to grow arrays of any type, not just arrays of objects.

```java
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

To make this possible, the parameter of `goodArrayGrow` is declared to be of type `Object`, not an `array of objects` (`Object[]`). The integer array type `int[]` can be converted to an `Object`, but not to an array of objects!

Listing 5–8 shows both array grow methods in action. Note that the cast of the return value of `badArrayGrow` will throw an exception.

---

**NOTE:** We present this program to illustrate how to work with arrays through reflection. If you just want to grow an array, use the `copyOf` method in the `Arrays` class.

```java
Employee[] a = new Employee[100];
...
// array is full
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

---

**Listing 5–8**  ArrayGrowTest.java

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

// This program demonstrates the use of reflection for manipulating arrays.
// Version 1.01 2004-02-21
// Author Cay Horstmann

public class ArrayGrowTest
{
    public static void main(String[] args)
    {
        int[] a = { 1, 2, 3 };
        a = (int[]) goodArrayGrow(a);
    }
```
14. arrayPrint(a);
15. String[] b = { "Tom", "Dick", "Harry" };
16. b = (String[]) goodArrayGrow(b);
17. arrayPrint(b);
18. System.out.println("The following call will generate an exception.");
19. b = (String[]) badArrayGrow(b);
20.
21. /\*
22. This method attempts to grow an array by allocating a new array and copying all elements.
23. @param a the array to grow
24. @return a larger array that contains all elements of a. However, the returned array has
25. type Object[], not the same type as a
26. */
27. static Object[] badArrayGrow(Object[] a)
28. {
29. int newLength = a.length * 11 / 10 + 10;
30. Object[] newArray = new Object[newLength];
31. System.arraycopy(a, 0, newArray, 0, a.length);
32. return newArray;
33. }
34.
35. /\*
36. This method grows an array by allocating a new array of the same type and
37. copying all elements.
38. @param a the array to grow. This can be an object array or a primitive
39. @return a larger array that contains all elements of a.
40. */
41. static Object goodArrayGrow(Object a)
42. {
43. Class cl = a.getClass();
44. if (!cl.isArray()) return null;
45. Class componentType = cl.getComponentType();
46. int length = Array.getLength(a);
47. int newLength = length * 11 / 10 + 10;
48. Object newArray = Array.newInstance(componentType, newLength);
49. System.arraycopy(a, 0, newArray, 0, length);
50. return newArray;
51. }
52.
53. /\*
54. A convenience method to print all elements in an array
55. @param a the array to print. It can be an object array or a primitive type array
56. */
57. static void arrayPrint(Object a)
Inheritance

Chapter 5 ■ Inheritance

Listing 5–8  ArrayGrowTest.java (continued)

```
63. {
64.   Class cl = a.getClass();
65.   if (!cl.isArray()) return;
66.   Class componentType = cl.getComponentType();
67.   int length = Array.getLength(a);
68.   System.out.print(componentType.getName() + "[" + length + "] = {");
69.   for (int i = 0; i < Array.getLength(a); i++)
70.     System.out.print(Array.get(a, i) + " ");
71.   System.out.println("}");
72. }
73. }
```

API  java.lang.reflect.Array 1.1

- static Object get(Object array, int index)
- static xxx getXxx(Object array, int index)
  (xxx is one of the primitive types boolean, byte, char, double, float, int, long, short.) These methods return the value of the given array that is stored at the given index.
- static void set(Object array, int index, Object newValue)
- static setXxx(Object array, int index, xxx newValue)
  (xxx is one of the primitive types boolean, byte, char, double, float, int, long, short.) These methods store a new value into the given array at the given index.
- static int getLength(Object array)
  returns the length of the given array.
- static Object newInstance(Class componentType, int length)
- static Object newInstance(Class componentType, int[] lengths)
  returns a new array of the given component type with the given dimensions.

Method Pointers!

On the surface, Java does not have method pointers—ways of giving the location of a method to another method so that the second method can invoke it later. In fact, the designers of Java have said that method pointers are dangerous and error prone and that Java interfaces (discussed in the next chapter) are a superior solution. However, as of Java 1.1, it turns out that Java does have method pointers, as a (perhaps accidental) by-product of the reflection package.

NOTE: Among the nonstandard language extensions that Microsoft added to its Java derivative J++ (and its successor, C#) is another method pointer type, called a delegate, that is different from the Method class that we discuss in this section. However, inner classes (which we will introduce in the next chapter) are a more useful construct than delegates.

To see method pointers at work, recall that you can inspect a field of an object with the get method of the Field class. Similarly, the Method class has an invoke method that lets you call the method that is wrapped in the current Method object. The signature for the invoke method is
Object invoke(Object obj, Object... args)

The first parameter is the implicit parameter, and the remaining objects provide the explicit parameters. (Before Java SE 5.0, you had to pass an array of objects or null if the method had no explicit parameters.)

For a static method, the first parameter is ignored—you can set it to null.

For example, if m1 represents the getName method of the Employee class, the following code shows how you can call it:

```java
String n = (String) m1.invoke(harry);
```

As with the get and set methods of the Field type, there's a problem if the parameter or return type is not a class but a primitive type. You either rely on autoboxing or, before Java SE 5.0, wrap primitive types into their corresponding wrappers.

Conversely, if the return type is a primitive type, the invoke method will return the wrapper type instead. For example, suppose that m2 represents the getSalary method of the Employee class. Then, the returned object is actually a Double, and you must cast it accordingly. As of Java SE 5.0, automatic unboxing takes care of the rest.

```java
double s = (Double) m2.invoke(harry);
```

How do you obtain a Method object? You can, of course, call getDeclaredMethods and search through the returned array of Method objects until you find the method that you want. Or, you can call the getMethod method of the Class class. This is similar to the getField method that takes a string with the field name and returns a Field object. However, there may be several methods with the same name, so you need to be careful that you get the right one. For that reason, you must also supply the parameter types of the desired method. The signature of getMethod is

```java
Method getMethod(String name, Class... parameterTypes)
```

For example, here is how you can get method pointers to the getName and raiseSalary methods of the Employee class:

```java
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

(Before Java SE 5.0, you had to package the Class objects into an array or to supply null if there were no parameters.)

Now that you have seen the rules for using Method objects, let's put them to work. Listing 5–9 is a program that prints a table of values for a mathematical function such as Math.sqrt or Math.sin. The printout looks like this:

```java
public static native double java.lang.Math.sqrt(double)
```

| 1.0000 | 1.0000 |
| 2.0000 | 1.4142 |
| 3.0000 | 1.7321 |
| 4.0000 | 2.0000 |
| 5.0000 | 2.2361 |
| 6.0000 | 2.4495 |
| 7.0000 | 2.6458 |
| 8.0000 | 2.8284 |
| 9.0000 | 3.0000 |
| 10.0000 | 3.1623 |
The code for printing a table is, of course, independent of the actual function that is being tabulated.

double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
    double y = (Double) f.invoke(null, x);
    System.out.printf("%10.4f | %10.4f%n", x, y);
}

Here, f is an object of type Method. The first parameter of invoke is null because we are calling a static method.

To tabulate the Math.sqrt function, we set f to
    Math.class.getMethod("sqrt", double.class)
That is the method of the Math class that has the name sqrt and a single parameter of type double.

Listing 5–9 shows the complete code of the generic tabulator and a couple of test runs.

Listing 5–9  MethodPointerTest.java

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

/**
 * This program shows how to invoke methods through reflection.
 * @version 1.1 2004-02-21
 * @author Cay Horstmann
 */
public class MethodPointerTest {
    public static void main(String[] args) throws Exception {
        // get method pointers to the square and sqrt methods
        Method square = MethodPointerTest.class.getMethod("square", double.class);
        Method sqrt = Math.class.getMethod("sqrt", double.class);

        // print tables of x- and y-values
        printTable(1, 10, 10, square);
        printTable(1, 10, 10, sqrt);
    }

    /**
     * Returns the square of a number
     * @param x a number
     * @return x squared
     */
    public static double square(double x) {
        return x * x;
    }
}
```
As this example shows clearly, you can do anything with Method objects that you can do with function pointers in C (or delegates in C#). Just as in C, this style of programming is usually quite inconvenient and always error prone. What happens if you invoke a method with the wrong parameters? The invoke method throws an exception.

Also, the parameters and return values of invoke are necessarily of type Object. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code. Therefore, errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than code that simply calls methods directly.

For that reason, we suggest that you use Method objects in your own programs only when absolutely necessary. Using interfaces and inner classes (the subject of the next chapter) is almost always a better idea. In particular, we echo the developers of Java and suggest not using Method objects for callback functions. Using interfaces for the callbacks (see the next chapter as well) leads to code that runs faster and is a lot more maintainable.

### Listing 5–9 MethodPointerTest.java (continued)

```java
/**
 * Prints a table with x- and y-values for a method
 * @param from the lower bound for the x-values
 * @param to the upper bound for the x-values
 * @param n the number of rows in the table
 * @param f a method with a double parameter and double return value
 */
public static void printTable(double from, double to, int n, Method f) {
    // print out the method as table header
    System.out.println(f);
    double dx = (to - from) / (n - 1);
    for (double x = from; x <= to; x += dx) {
        try {
            double y = (Double) f.invoke(null, x);
            System.out.printf("%10.4f | %10.4f\n", x, y);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As this example shows clearly, you can do anything with Method objects that you can do with function pointers in C (or delegates in C#). Just as in C, this style of programming is usually quite inconvenient and always error prone. What happens if you invoke a method with the wrong parameters? The invoke method throws an exception.

Also, the parameters and return values of invoke are necessarily of type Object. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code. Therefore, errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than code that simply calls methods directly.

For that reason, we suggest that you use Method objects in your own programs only when absolutely necessary. Using interfaces and inner classes (the subject of the next chapter) is almost always a better idea. In particular, we echo the developers of Java and suggest not using Method objects for callback functions. Using interfaces for the callbacks (see the next chapter as well) leads to code that runs faster and is a lot more maintainable.
Inheritance

public Object invoke(Object implicitParameter, Object[] explicitParameters)
invokes the method described by this object, passing the given parameters and returning the value that the method returns. For static methods, pass null as the implicit parameter. Pass primitive type values by using wrappers. Primitive type return values must be unwrapped.

Design Hints for Inheritance

We want to end this chapter with some hints that we have found useful when using inheritance.

1. Place common operations and fields in the superclass.

This is why we put the name field into the Person class rather than replicating it in the Employee and Student classes.

2. Don’t use protected fields.

Some programmers think it is a good idea to define most instance fields as protected, “just in case,” so that subclasses can access these fields if they need to. However, the protected mechanism doesn’t give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses protected instance fields, thereby breaking encapsulation. And second, in the Java programming language, all classes in the same package have access to protected fields, whether or not they are subclasses.

However, protected methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses. The clone method is a good example.

3. Use inheritance to model the “is–a” relationship.

Inheritance is a handy code-saver, and sometimes people overuse it. For example, suppose we need a Contractor class. Contractors have names and hire dates, but they do not have salaries. Instead, they are paid by the hour, and they do not stay around long enough to get a raise. There is the temptation to form a subclass Contractor from Employee and add an hourlyWage field.

```
class Contractor extends Employee
{
  . . .
  private double hourlyWage;
}
```

This is not a good idea, however, because now each contractor object has both a salary and hourly wage field. It will cause you no end of grief when you implement methods for printing paychecks or tax forms. You will end up writing more code than you would have by not inheriting in the first place.

The contractor/employee relationship fails the “is–a” test. A contractor is not a special case of an employee.

4. Don’t use inheritance unless all inherited methods make sense.

Suppose we want to write a Holiday class. Surely every holiday is a day, and days can be expressed as instances of the GregorianCalendar class, so we can use inheritance.

```
class Holiday extends GregorianCalendar { . . . }
```
Unfortunately, the set of holidays is not closed under the inherited operations. One of the public methods of GregorianCalendar is add. And add can turn holidays into nonholidays:

```java
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Therefore, inheritance is not appropriate in this example.

5. **Don’t change the expected behavior when you override a method.**

The substitution principle applies not just to syntax but, more important, to behavior. When you override a method, you should not unreasonably change its behavior. The compiler can’t help you—it cannot check whether your redefinitions make sense. For example, you can “fix” the issue of the add method in the Holiday class by redefining add, perhaps to do nothing, or to throw an exception, or to move on to the next holiday.

However, such a fix violates the substitution principle. The sequence of statements

```java
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

should have the expected behavior, no matter whether x is of type GregorianCalendar or Holiday.

Of course, therein lies the rub. Reasonable and unreasonable people can argue at length what the expected behavior is. For example, some authors argue that the substitution principle requires Manager.equals to ignore the bonus field because Employee.equals ignores it. These discussions are always pointless if they occur in a vacuum. Ultimately, what matters is that you do not circumvent the intent of the original design when you override methods in subclasses.

6. **Use polymorphism, not type information.**

Whenever you find code of the form

```java
if (x is of type 1)
   action1(x);
else if (x is of type 2)
   action2(x);
```

think polymorphism.

Do action1 and action2 represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call

```java
x.action();
```

and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.

Code using polymorphic methods or interface implementations is much easier to maintain and extend than code that uses multiple type tests.

7. **Don’t overuse reflection.**

The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at runtime. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications. Reflection is fragile—the compiler cannot help you find programming errors. Any errors are found at runtime and result in exceptions.
You have now seen how Java supports the fundamentals of object-oriented programming: classes, inheritance, and polymorphism. In the next chapter, we will tackle two advanced topics that are very important for using Java effectively: interfaces and inner classes.