Chapter 10

Saving Time and Money: Reusing Existing Code

In This Chapter

- Tweaking your code
- Adding new life to old code
- Making changes without spending a fortune

Wouldn't it be nice if every piece of software did just what you wanted it to do? In an ideal world, you could simply buy a program, make it work right away, plug it seamlessly into new situations, and update it easily whenever your needs changed. Unfortunately, software of this kind doesn't exist. (*Nothing* of this kind exists.) The truth is that no matter what you want to do, you can find software that does some of it, but not all of it.

This is one reason that object-oriented programming has been successful. For years, companies were buying prewritten code only to discover that the code didn't do what they wanted it to do. So the companies began messing with the code. Their programmers dug deep into the program files, changed variable names, moved subprograms around, reworked formulas, and generally made the code worse. The reality was that if a program didn't already do what you wanted (even if it did something ever so close to it), you could never improve the situation by mucking around inside the code. The best option was to chuck the whole program (expensive as that was) and start over. What a sad state of affairs!

Object-oriented programming has brought about a big change. An object-oriented program is, at its heart, designed to be modified. Using correctly written software, you can take advantage of features that are already built in, add new features of your own, and override features that don't suit your needs. The best aspect of this situation is that the changes you make are clean — no clawing and digging into other people's brittle program code. Instead, you make nice, orderly additions and modifications without touching the existing code's internal logic. It's the ideal solution.

The Last Word on Employees — Or Is It?

When you write an object-oriented program, you start by considering the data. You're writing about accounts. So what's an account? You're writing code to handle button clicks. So what's a button? You're writing a program to send payroll checks to employees. What's an employee?

In this chapter's first example, an employee is someone with a name and a job title — sure, employees have other characteristics, but for now I stick to the basics:

```
class Employee {
  String name;
  String jobTitle;
}
```

Of course, any company has different kinds of employees. For example, your company may have full-time and part-time employees. Each full-time employee has a yearly salary:

```
class FullTimeEmployee extends Employee {
  double salary;
}
```

In this example, the words `extends Employee` tell Java that the new class (the `FullTimeEmployee` class) has all the properties that any `Employee` has and, possibly, more. In other words, every `FullTimeEmployee` object is an `Employee` object (an employee of a certain kind, perhaps). Like any `Employee`, a `FullTimeEmployee` has a `name` and a `jobTitle`. But a `FullTimeEmployee` also has a salary. That's what the words `extends Employee` do for you.

A part-time employee has no fixed yearly salary. Instead, every part-time employee has an hourly pay rate and a certain number of hours worked in a week:

```
class PartTimeEmployee extends Employee {
  double hourlyPay;
  int hoursWorked;
}
```

So far, a `PartTimeEmployee` has four characteristics: `name`, `jobTitle`, `hourlyPay`, and number of `hoursWorked`.

Then you have to consider the big shots — the executives. Every executive is a full-time employee. But in addition to earning a salary, every executive receives a bonus (even if the company goes belly-up and needs to be bailed out):

```
class Executive extends FullTimeEmployee {
  double bonus;
}
```

Java's `extends` keyword is cool because, by extending a class, you inherit all the complicated code that's already in the other class. The class you extend can be a class that you have (or another developer has) already written. One way or another, you're able to reuse existing code and to add ingredients to the existing code.

Here's another example: The creators of Android wrote the `Activity` class, with its 5,000 lines of code. You get to use all those lines of code for free by simply typing `extends Activity`:

```
public class MainActivity extends Activity {
```

With the two words `extends Activity`, your new `MainActivity` class can do all the things that a typical Android activity can do — start running, find items in the app's `res` directory, show a dialog box, respond to a low-memory condition, start another activity, return an answer to an activity, finish running, and much more.

Extending a class

So useful is Java's `extends` keyword that developers have several different names to describe this language feature:

- **Superclass/subclass:** The `Employee` class (see the earlier section "The Last Word on Employees — Or Is It?") is the *superclass* of the `FullTimeEmployee` class. The `FullTimeEmployee` class is a *subclass* of the `Employee` class.
- **Parent/child:** The `Employee` class is the *parent* of the `FullTimeEmployee` class. The `FullTimeEmployee` class is a *child* of the `Employee` class.

  In fact, the `Executive` class extends the `FullTimeEmployee` class, which in turn extends the `Employee` class. So `Executive` is a *descendent* of `Employee`, and `Employee` is an *ancestor* of `Executive`. The Unified Modeling Language (UML) diagram in Figure 10-1 illustrates this point.

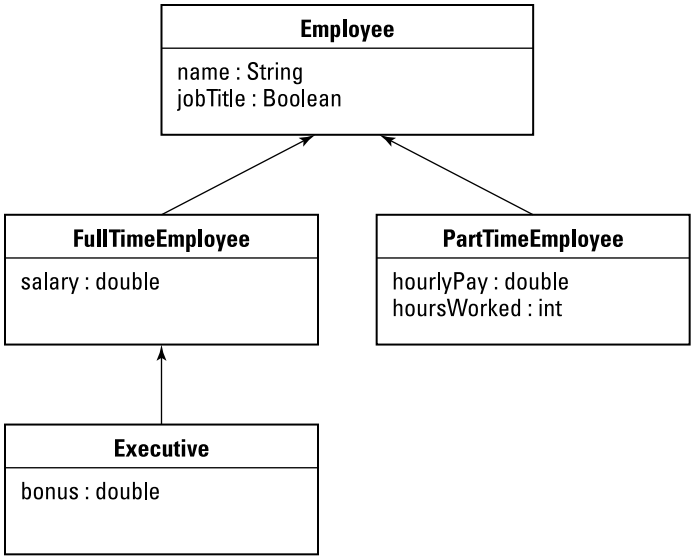


Figure 10-1: A class, two child classes, and a grandchild class.

- **Inheritance:** The `FullTimeEmployee` class *inherits* the `Employee` class's members. (If any of the `Employee` class's members were declared to be `private`, the `FullTimeEmployee` class wouldn't inherit those members.)

  The `Employee` class has a `name` field, so the `FullTimeEmployee` class has a `name` field, and the `Executive` class has a `name` field. In other words, with the declarations of `Employee`, `FullTimeEmployee`, and `Executive` at the start of this section, the code in Listing 10-1 is legal.

  All descendants of the `Employee` class have `name` fields, even though a `name` field is explicitly declared only in the `Employee` class itself.

Listing 10-1: Using the `Employee` Class and Its Subclasses

```
public class Main {

  public static void main(String[] args) {
    Employee employee = new Employee();
    employee.name = "Sam";

    FullTimeEmployee ftEmployee = new FullTimeEmployee();
    ftEmployee.name = "Jennie";

    Executive executive = new Executive();
    executive.name = "Harriet";
  }
}
```

Almost every Java class extends another Java class. I write *almost* because one (and only one) class doesn't extend any other class. Java's built-in `Object` class doesn't extend anything. The `Object` class is at the top of Java's class hierarchy. Any class whose header has no `extends` clause automatically extends Java's `Object` class. So every other Java class is, directly or indirectly, a descendent of the `Object` class, as shown in Figure 10-2.

The notion of extending a class is one pillar of object-oriented programming. In the 1970s, computer scientists were noticing that programmers tended to reinvent the wheel. If you needed code to balance an account, for example, you started writing code from scratch to balance an account. Never mind that other people had written their own account-balancing code. Integrating other peoples' code with yours, and adapting other peoples' code to your own needs, was a big headache. All things considered, it was easier to start from scratch.

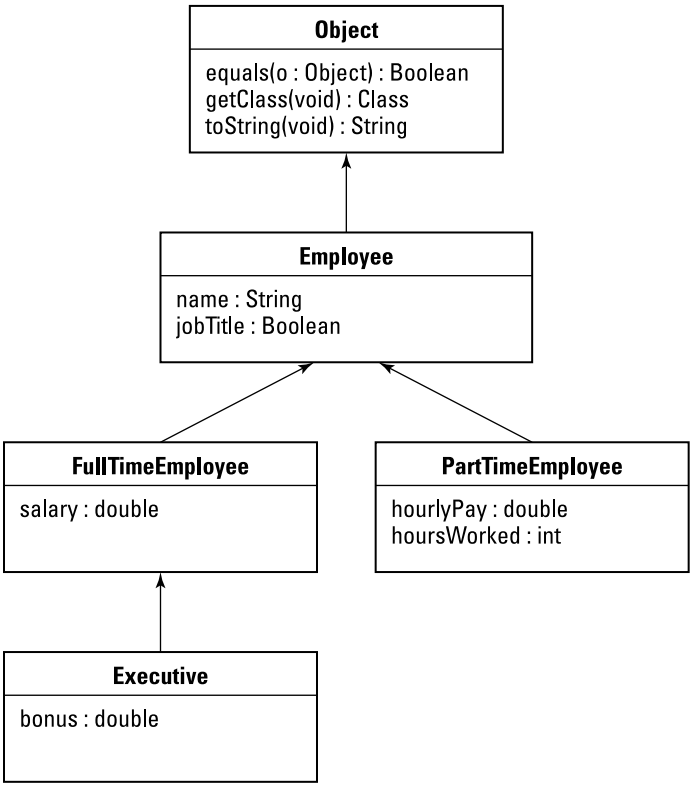


Figure 10-2: Everything comes from Java's `Object` class.

Then, in the 1980s, object-oriented programming became popular. The notion of classes and subclasses provided a clean way to connect existing code (such as Android's `Activity` class code) with new code (such as your new `MainActivity` class code). By extending an existing class, you hook into the class's functionality, and you reuse features that have already been programmed.

By reusing code, you avoid the work of reinventing the wheel. But you also make life easier for the end user. When you extend Android's `Activity` class, your new activity behaves like other peoples' activities because both your activity and the other peoples' activities inherit the same behavior from Android's `Activity` class. With so many apps behaving the same way, the user learns familiar patterns. It's a win-win situation.

Overriding methods

In this section, I expand on all the employee code snippets from the start of this chapter. From these snippets, I can present a fully baked program example. The example, as laid out in Listings 10-2 through 10-6, illustrates some important ideas about classes and subclasses.

Listing 10-2: What Is an Employee?

```
package org.allyourcode.company;

import javax.swing.JOptionPane;

public class Employee {
  String name;
  String jobTitle;

  public Employee() {
  }

  public Employee(String name, String jobTitle) {
    this.name = name;
    this.jobTitle = jobTitle;
  }

  public void showPay() {
    JOptionPane.showMessageDialog(null, name +
        ", Pay not known");
  }
}
```

Listing 10-3: Full-Time Employees Have Salaries

```
package org.allyourcode.company;

import java.text.NumberFormat;
import java.util.Locale;

import javax.swing.JOptionPane;

public class FullTimeEmployee extends Employee {
  double salary;

  static NumberFormat currency =
      NumberFormat.getCurrencyInstance(Locale.US);

  public FullTimeEmployee() {
  }

  public FullTimeEmployee(String name,
                          String jobTitle,
                          double salary) {
    this.name = name;
    this.jobTitle = jobTitle;
    this.salary = salary;
  }

  public double pay() {
    return salary;
  }

  @Override
  public void showPay() {
    JOptionPane.showMessageDialog(null, name + ", " +
        currency.format(pay()));
  }
}
```

Listing 10-4: Executives Get Bonuses

```
package org.allyourcode.company;

public class Executive extends FullTimeEmployee {
  double bonus;

  public Executive() {
  }

  public Executive(String name, String jobTitle,
                   double salary, double bonus) {
    this.name = name;
```

(continued)

Listing 10-4 *(continued)*

```
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.bonus = bonus;
  }

  @Override
  public double pay() {
    return salary + bonus;
  }
}
```

Listing 10-5: Part-Time Employees Are Paid by the Hour

```
package org.allyourcode.company;

import java.text.NumberFormat;
import java.util.Locale;

import javax.swing.JOptionPane;

public class PartTimeEmployee extends Employee {
  double hourlyPay;
  int hoursWorked;

  static NumberFormat currency =
      NumberFormat.getCurrencyInstance(Locale.US);

  public PartTimeEmployee() {
  }

  public PartTimeEmployee(String name,
                          String jobTitle,
                          double hourlyPay,
                          int hoursWorked) {
    this.name = name;
    this.jobTitle = jobTitle;
    this.hourlyPay = hourlyPay;
    this.hoursWorked = hoursWorked;
  }

  public double pay() {
    return hourlyPay * hoursWorked;
  }

  @Override
  public void showPay() {
    JOptionPane.showMessageDialog(null, name + ", " +
        currency.format(pay()));
  }
}
```

Listing 10-6: Putting Your Employee Classes to the Test

```
package org.allyourcode.company;

public class Main {

  public static void main(String[] args) {
    Employee employee =
        new Employee("Barry", "Author");

    FullTimeEmployee ftEmployee =
        new FullTimeEmployee("Ed", "Manager", 10000.00);

    PartTimeEmployee ptEmployee =
        new PartTimeEmployee("Joe", "Intern", 8.00, 20);

    Executive executive =
        new Executive("Jane", "CEO", 20000.00, 5000.00);

    employee.showPay();
    ftEmployee.showPay();
    ptEmployee.showPay();
    executive.showPay();
  }

}
```

Figure 10-3 shows a run of the code in Listings 10-2 through 10-6, and Figure 10-4 contains a UML diagram for the classes in these listings. (In Figure 10-4, I ignore the `Main` class from Listing 10-6. The `Main` class isn't interesting, because it's not part of the `Employee` class hierarchy. The `Main` class is simply a subclass of Java's `Object` class.)

In Figure 10-4, I use strikethrough text and simulated handwriting to represent overridden methods. These typographical tricks are my own inventions. Neither the strikethrough nor the simulated handwriting is part of the UML standard. In fact, the UML standard has all kinds of rules that I ignore in this book. My main purpose in showing you the rough UML diagrams is to help you visualize the hierarchies of classes and their subclasses.

Consider the role of the `showPay` method in Figure 10-4 and in Listings 10-2 through 10-6. In the figure, `showPay` appears in all except the `Executive` class; in the listings, I define `showPay` in all except the `Executive` class.

The `showPay` method appears for the first time in the `Employee` class (refer to Listing 10-2), where it serves as a placeholder for not knowing the employee's pay. The `FullTimeEmployee` class (refer to Listing 10-3) would inherit this vacuous `showPay` class except that the `FullTimeEmployee` class declares its own version of `showPay`. In the terminology from Chapter 5, the `showPay` method in `FullTimeEmployee` *overrides* the `showPay` method in `Employee`.

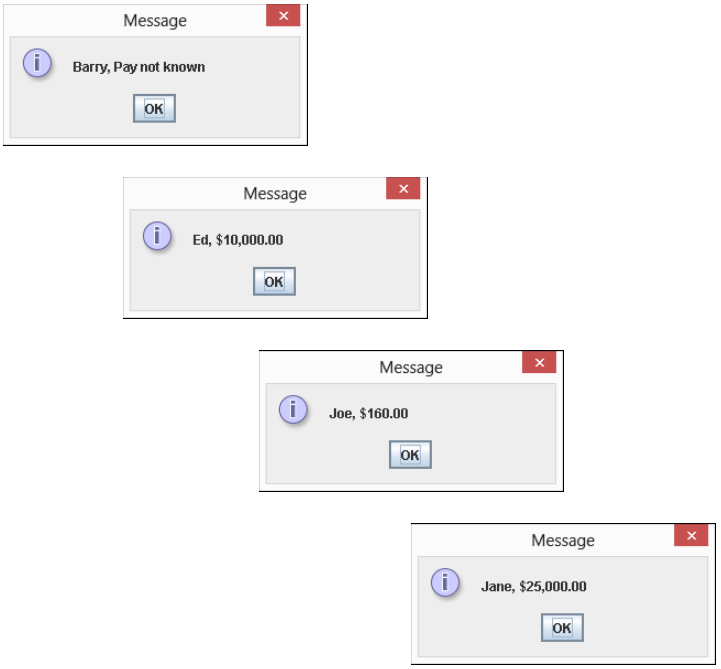


Figure 10-3: Running the code in Listings 10-2 through 10-6.

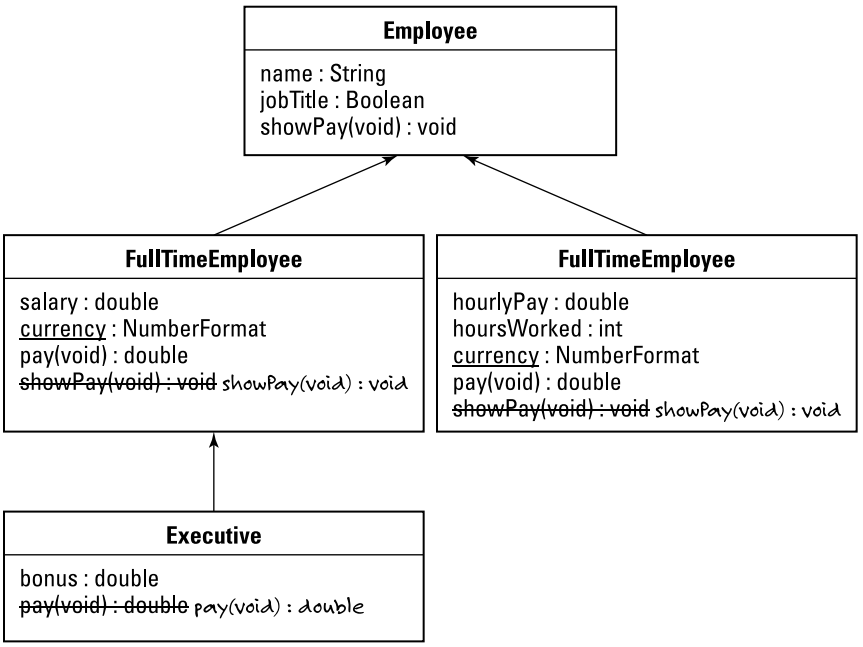


Figure 10-4: Classes and sub-classes with fields and methods.

Listing 10-6 contains a call to a full-time employee's `showPay` method:

```
FullTimeEmployee ftEmployee = ... Etc.
ftEmployee.showPay();
```

And in Figure 10-3, the call to `ftEmployee.showPay()` gives you the `FullTimeEmployee` class's version of `showPay`, not the `Employee` class's clueless version of `showPay`. (If `ftEmployee.showPay()` called the `Employee` class's version of `showPay`, you'd see `Ed, Pay not known` in Figure 10-3.) Overriding a method declaration means taking precedence over that existing version of the method.

Of course, overriding a method isn't the same as obliterating a method. In Listing 10-6, the snippet

```
Employee employee = ... Etc.
employee.showPay();
```

conjures up the `Employee` class's noncommittal version of `showPay`. It happens because an object declared with the `Employee` constructor has no `salary` field, no `hourlyPay` field, and no `showPay` method other than the method declared in the `Employee` class. The `Employee` class, and any objects declared using the `Employee` constructor, could do their work even if the other classes (`FullTimeEmployee`, `PartTimeEmployee`, and so on) didn't exist.

The only way to override a method is to declare a method with the same name and the same parameters inside a subclass. By *same parameters,* I mean the same number of parameters, each with the same type. For example, `calculate(int count, double amount)` overrides `calculate(int x, double y)` because both declarations have two parameters: The first parameter in each declaration is of type `int`, and the second parameter in each declaration is of type `double`. But `calculate(int count, String amount)` doesn't override `calculate(int count, double amount)`. In one declaration, the second parameter has type `double`, and in the other declaration, the second parameter has type `String`. If you call `calculate(42, 2.71828)`, you get the `calculate(int x, double y)` method, and if you call `calculate(42, "Euler")` you get the `calculate(int count, String amount)` method.

Listings 10-2 through 10-5 have other examples of overriding methods. For example, the `Executive` class in Listing 10-4 overrides its parent class's `pay` method, but not the parent class's `showPay` method. Calculating an executive's pay is different from calculating an ordinary full-time employee's pay. But after you know the two peoples' pay amounts, showing an executive's pay is no different from showing an ordinary full-time employee's pay.

When I created this section's examples, I considered giving the `Employee` class a `pay` method (returning `0` on each call). This strategy would make it unnecessary for me to create identical `showPay` methods for the `FullTimeEmployee` and `PartTimeEmployee` classes. For various reasons, (none of them interesting), I decided against doing it that way.

Overriding works well in situations in which you want to tweak an existing class's features. Imagine having a news ticker that does everything you want except scroll sideways. (I'm staring at one on my computer right now! As one news item disappears toward the top, the next news item scrolls in from below. The program's options don't allow me to change this setting.) After studying the code's documentation, you can subclass the program's `Ticker` class and override the `Ticker` class's `scroll` method. In your new `scroll` method, the user has the option to move text upward, downward, sideways, or inside out (whatever that means).

Java annotations

In Java, elements that start with an at-sign (`@`) are *annotations.* Java didn't have annotations until Java 5.0, so if you try to use the `@Override` annotation with Java 1.4.2, for example, you'll see some nasty-looking error messages. That's okay because Android requires Java 5.0 or Java 6. You can't use earlier versions of Java to create Android apps.

In Listings 10-3, 10-4, and 10-5, each `@Override` annotation reminds Java that the method immediately below the annotation has the same name and the same parameter types as a method in the parent class. The use of the `@Override` annotation is optional. If you remove all `@Override` lines from Listings 10-3, 10-4, and 10-5, the code works the same way.

So why use the `@Override` annotation? Imagine leaving it off and mistakenly putting the following method in Listing 10-4:

```
public void showPay(double salary) {
  JOptionPane.showMessageDialog(null, name + ", " +
      currency.format(salary));
}
```

You might think that you've overridden the parent class's `showPay` method, but you haven't! The `Employee` class's `showPay` method has no parameters, and your new `FullTimeEmployee` class's `showPay` method has a parameter. Eclipse looks at this stuff in the editor and says, "Okay, I guess the developer is inheriting the `Employee` class's `showPay` method and declaring an additional version of `showPay`. Both `showPay` methods are available in the

FullTimeEmployee class." (By the way, when Eclipse speaks, you can't see my lips moving.)

Everything goes fine until you run the code and see the message Pay not known when you call ftEmployee.showPay(). The Java virtual machine is calling the parameterless version of showPay, which the FullTimeEmployee class inherits from its parent.

The problem in this hypothetical example isn't so much that you commit a coding error — everybody makes mistakes like this one. (Yes, even I do. I make lots of them.) The problem is that, without an @Override annotation, you don't catch the error until you're running the program. That is, you don't see the error message as soon as you compose the code in the Eclipse editor. Waiting until runtime can be as painless as saying, "Aha! I know why this program didn't run correctly." But waiting until runtime can also be quite painful — as painful as saying, "My app was rated 1 on a scale of 5 because of this error that I didn't see until a user called my bad showPay method."

Ideally, Eclipse is aware of your intention to override an existing method, and it can complain to you while you're staring at the editor. If you use the @Override annotation in conjunction with the bad showPay method, you see the blotches shown in Figure 10-5. That's good because you can fix the problem long before the problem shows up in a run of your code.

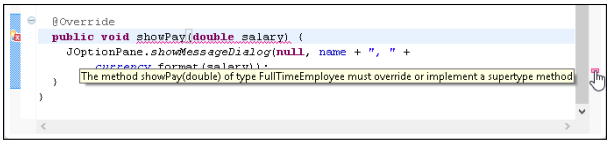


Figure 10-5: The showPay method doesn't override the parent class's showPay method.

More about Java's Modifiers

I start the conversation about Java's modifiers in Chapters 6 and 9. Chapter 6 describes the keyword final as it applies to variables, and Chapter 9 deals with the keywords public and private. In this section, I add a few more fun facts about Java modifiers.

The word `final` has many uses in Java programs. In addition to having final variables, you can have these elements:

- **Final class:** If you declare a class to be `final`, no one (not even you) can extend it.
- **Final method:** If you declare a method to be `final`, no one (not even you) can override it.

Figures 10-6 and 10-7 put these rules into perspective. In Figure 10-6, I can't extend the `Stuff` class, because the `Stuff` class is `final`. And in Figure 10-7, I can't override the `Stuff` class's `increment` method because that `increment` method is `final`.

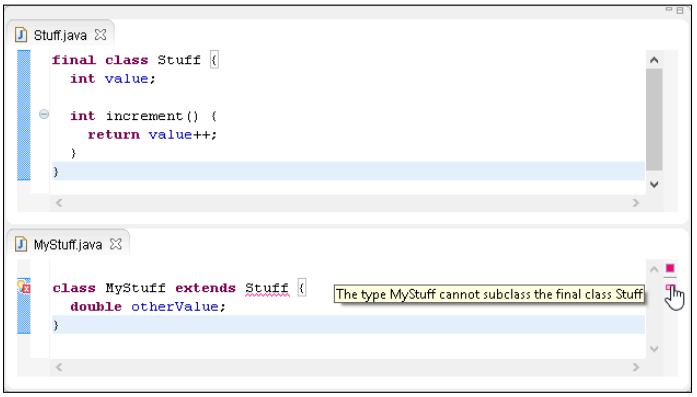


Figure 10-6: Trying to extend a final class.

Figure 10-7: Trying to override a final method.

You can apply Java's `protected` keyword to a class's members. This `protected` keyword has always seemed a bit strange to me. In common English usage, when my possessions are "protected," my possessions aren't as available as they'd normally be. But in Java, when you preface a field or a method with the `protected` keyword, you make that field or method a bit more available than it would be by default, as shown in Figure 10-8.

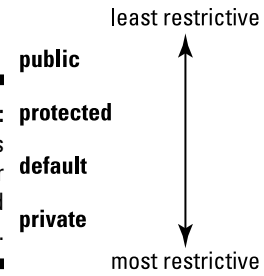


Figure 10-8: Access modes for fields and methods.

Here's what I say in Chapter 9 about members with default access:

> *A default member of a class (a member whose declaration doesn't contain the words `public`, `private`, or `protected`) can be used by any code inside the same package as that class.*

The same thing is true about a `protected` class member. But in addition, a `protected` member is inherited outside the class's package by any subclass of the class containing that protected member.

Huh? What does that last sentence mean, about `protected` members? To make things concrete, Figure 10-9 shows you the carefree existence in which two classes are in the same package. With both `Stuff` and `MyStuff` in the same package, the `MyStuff` class inherits the `Stuff` class's default `value` variable and the `Stuff` class's default `increment` method.

If you move the `Stuff` class to a different package, `MyStuff` no longer inherits the `Stuff` class's default `value` variable or the `Stuff` class's default `increment` method, as shown in Figure 10-10.

But if you turn `value` into a `protected` variable and you turn `increment` into a `protected` method, the `MyStuff` class again inherits its parent class's `value` variable and `increment` method, as shown in Figure 10-11.

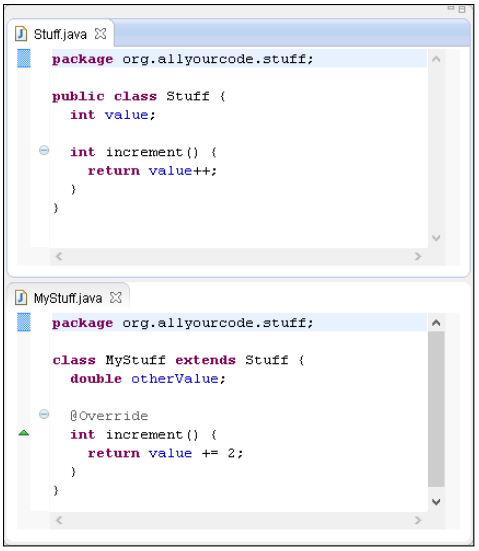


Figure 10-9: Two classes in the same package.

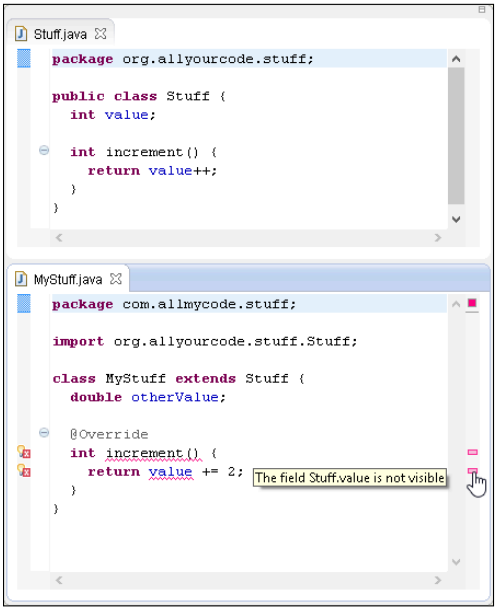


Figure 10-10: Classes in different packages.

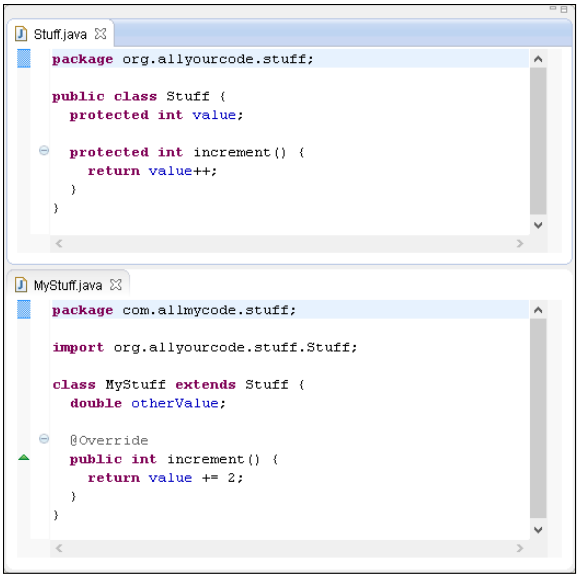


Figure 10-11: Using the protected modifier.

Notice one more detail in Figure 10-11. I change the `MyStuff` class's `increment` method from default to `public`. I do this to avoid seeing an interesting little error message. You can't override a method with another method whose access is more restrictive than the original method. In other words, you can't override a public method with a private method. You can't even override a public method with a default method.

Java's default access is more restrictive than protected access (see Figure 10-8). So you can't override a protected method with a default method. In Figure 10-11, I avoid the whole issue by making public the `MyStuff` class's `increment` method. That way, I override the `increment` method with the least restrictive kind of access.

Keeping Things Simple

Most computer programs operate entirely in the virtual realm. They have no bricks, nails, or girders. So you can type a fairly complicated computer program in minutes. Even with no muscle and no heavy equipment, you can create a structure whose complexity rivals that of many complicated physical structures. You, the developer, have the power to build intricate, virtual bridges.

One goal of computer programming is to manage complexity. A good app isn't simply useful or visually appealing — a good app's code is nicely organized, easy to understand, and easy to modify.

Certain programming languages, like C++, support *multiple inheritance,* in which a class can have more than one parent class. For example, in C++ you can create a `Book` class, a `TeachingMaterial` class, and a `Textbook` class. You can make `Textbook` extend both `Book` and `TeachingMaterial`. This feature makes class hierarchies quite flexible, but it also makes those same hierarchies extremely complicated. You need tricky rules to decide how to inherit the `move` methods of both the computer's `Mouse` class and the rodent's `Mouse` class.

To avoid all this complexity, Java doesn't support multiple inheritance. In Java, each class has one (and only one) superclass. A class can have any number of subclasses. You can (and will) create many subclasses of Android's `Activity` class. And other developers create their own subclasses of Android's `Activity` class. But classes don't have multiple personalities. A Java class can have only one parent. The `Executive` class (refer to Listing 10-4) cannot extend both the `FullTimeEmployee` class and the `PartTimeEmployee` class.

Using an interface

The relationship between a class and its subclass is one of inheritance. In many real-life families, a child inherits assets from a parent. That's the way it works.

But consider the relationship between an editor and an author. The editor says, "By signing this contract, you agree to submit a completed manuscript by the fifteenth of July." Despite any excuses that the author gives before the deadline date (and, believe me, authors make plenty of excuses), the relationship between the editor and the author is one of obligation. The author agrees to take on certain responsibilities; and, in order to continue being an author, the author must fulfill those responsibilities. (By the way, there's no subtext in this paragraph — none at all.)

Now consider Barry Burd. Who? Barry Burd — that guy who writes *Java Programming For Android Developers For Dummies and certain other For Dummies* books (all from Wiley Publishing). He's a parent, and he's also an author. You want to mirror this situation in a Java program, but Java doesn't support multiple inheritance. You can't make Barry extend both a `Father` class and an `Author` class at the same time.

Fortunately for Barry, Java has interfaces. A class can extend only one parent class, but a class can implement many interfaces. A parent class is a bunch of stuff that a class inherits. On the other hand, as with the relationship between an editor and an author, an *interface* is a bunch of stuff that a class is obliged to provide.

Here's another example. Listings 10-2 through 10-5 describe what it means to be an employee of various kinds. Though a company might hire consultants, consultants who work for the company aren't employees. Consultants are normally self-employed. They show up temporarily to help companies solve problems and then leave the companies to work elsewhere. In the United States, differentiating between an employee and a consultant is important: So serious are the U.S. tax withholding laws that labeling a consultant an "employee" of any kind would subject the company to considerable legal risk.

To include consultants with employees in your code, you need a `Consultant` class that's separate from your existing `Employee` class hierarchy. On the other hand, consultants have a lot in common with a company's regular employees. For example, every consultant has a `showPay` method. You want to represent this commonality in your code, so you create an interface. The interface obligates a class to give meaning to the method name `showPay`, as shown in Listing 10-7.

Listing 10-7: Behold! An Interface!

```
package org.allyourcode.company;

public interface Payable {

  public void showPay();

}
```

The element in Listing 10-7 isn't a class — it's a Java interface. Here's a description of the listing's code:

> As an interface, I have a header, but no body, for the `showPay` method. In this interface, the `showPay` method takes no arguments and returns `void`. A class that claims to implement me (the `Payable` interface) must provide (either directly or indirectly) a body for the `showPay` method. That is, a class that claims to implement `Payable` must, in one way or another, implement the `showPay` method.

To find out about the difference between a method declaration's header and its body, see Chapter 5.

Listings 10-8 and 10-9 implement the `Payable` interface and provide bodies for the `showPay` method.

Listing 10-8: Implementing an Interface

```
package org.allyourcode.company;

import java.text.NumberFormat;
import java.util.Locale;

import javax.swing.JOptionPane;

public class Consultant implements Payable {

  String name;
  double hourlyFee;
  int hoursWorked;

  static NumberFormat currency =
      NumberFormat.getCurrencyInstance(Locale.US);

  public Consultant() {
  }

  public Consultant(String name, String jobTitle,
                    double hourlyFee, int hoursWorked) {
    this.name = name;
    this.hourlyFee = hourlyFee;
    this.hoursWorked = hoursWorked;
  }

  public double pay() {
    return hourlyFee * hoursWorked;
  }

  @Override
  public void showPay() {
    JOptionPane.showMessageDialog(null, name + ", " +
          currency.format(pay()));
  }
}
```

Listing 10-9: Another Class Implements the Interface

```
package org.allyourcode.company;

import javax.swing.JOptionPane;

public class Employee implements Payable {
```

```
  String name;
  String jobTitle;

  public Employee() {
  }

  public Employee(String name, String jobTitle) {
    this.name = name;
    this.jobTitle = jobTitle;
  }

  @Override
  public void showPay() {
    JOptionPane.showMessageDialog(null, name +
        ", Pay not known");
  }
}
```

In Listings 10-8 and 10-9, both the `Consultant` and `Employee` classes implement the `Payable` interface — the interface that summarizes what it means to be paid by the company. Implementing this interface guarantees that these classes have bodies for the showPay method. This guarantee allows any other code to safely call `employee.showPay()` or `consultant.showPay()`.

In this section's example, two otherwise unrelated classes (`Employee` and `Consultant`) both implement the `Payable` interface. When I picture a Java interface, it's an element that cuts across levels of Java's class/subclass hierarchy, as shown in Figure 10-12.

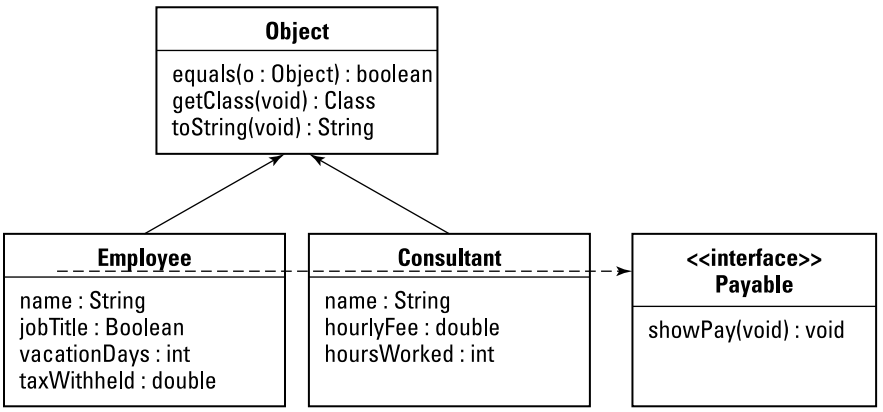


Figure 10-12: An interface cuts across the class hierarchy.

The dotted line in Figure 10-12 isn't part of standard UML. The folks who manage the standard have much better ways to represent interfaces than I use in this chapter's figures.

Creating a callback

In this chapter's (just discussed) "Using an interface" section, I reveal how an interface helps me realize the commonalities among various pay-receiving classes. The interface gives me an elegant way to mirror the connections in the real-world's data. But aside from its elegance, the interface in the "Using an interface" section doesn't make any problems easier to solve. The code with and without the interface is basically the same.

So in this section, I describe another problem that I solve using an interface. (In fact, the use of an interface plays a key role in the problem's solution.) This section's code is a bit more complicated than the code in the "Using an interface" section, but this section's code illustrates a widely used programming technique.

Many scenarios in application development involve *callbacks*. Imagine a stopwatch program. The program tells you when ten seconds have gone by. It has two statements: one to start a countdown and another to notify the user that the time is up. You can write the code this way:

```
try {
  Thread.sleep(10000);
} catch (InterruptedException e) {
  e.printStackTrace();
}
JOptionPane.showMessageDialog(null, "Time's up!");
```

Java's built-in `Thread` class has a `sleep` method that makes your app's action pause for any number of milliseconds you want. Ten thousand milliseconds is the same as ten seconds.

The `try/catch` business surrounding the `sleep` method call is part of the Java exception-handling feature. I cover it in Chapter 13.

Your code looks sensible, but it's seriously flawed. While your program puts itself to sleep for ten seconds, the user doesn't get a response from it — its buttons are frozen. Your program is sleeping, so the user can't use any other feature that your program offers. The user touches your program's widgets and presses your program's Cancel button, but the program doesn't respond. Yes, this is a great way to guarantee a 1-of-5 rating at Google Play (its app store).

To fix the problem, you take advantage of somebody's `TimerCommon` class, a general-purpose class that sleeps for a certain period on behalf of your

program. While the `TimerCommon` object sleeps, your program can remain awake, responding to the user's clicks, taps, inputs, swipes, or whatever.

(By the way, the `TimerCommon` class isn't part of the Java API. Somebody posted the `TimerCommon` class on the web along with a note permitting any developer to use the code.)

When the `TimerCommon` object wakes up, the object calls one of your program's methods. (In this section's example, your method is named `alert`.) Until the `TimerCommon` object calls your `alert` method, the method sits quietly in your program, doing nothing. Rather than execute the `alert` method, your program responds to the user's requests. Slick!

Now review the general flow of execution in the stopwatch code: First you set the `TimerCommon` object in motion. The `TimerCommon` object takes a brief nap. Finally, when the `TimerCommon` object wakes up, the `TimerCommon` object calls you back. In other words, the `TimerCommon` object issues a *callback,* as shown in Figure 10-13.

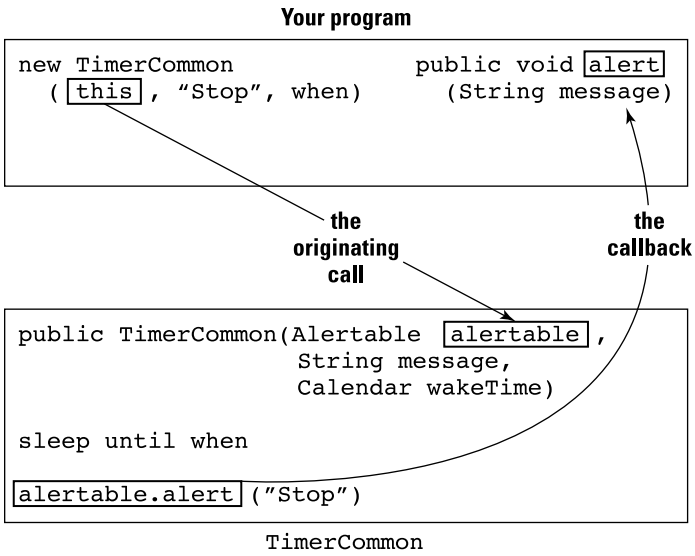


Figure 10-13: The TimerCommon object calls you back.

Listings 10-10 through 10-13 have the basic code to illustrate the callback technique.

Listing 10-10: Implementing the Alertable Interface

```
package org.allyourcode.stopwatch;

import java.util.Calendar;

import javax.swing.JOptionPane;

import com.example.timers.Alertable;
import com.example.timers.TimerCommon;

public class StopWatch implements Alertable {

  public StopWatch(int seconds) {
    Calendar wakeTime = Calendar.getInstance();
    wakeTime.add(Calendar.SECOND, seconds);
    new TimerCommon(this, "Stop", wakeTime);
  }

  @Override
  public void alert(String message) {
    JOptionPane.showMessageDialog(null, message);
  }
}
```

Listing 10-11: The Alertable Interface

```
package com.example.timers;

public interface Alertable {

  public void alert(String message);
}
```

Listing 10-12: Receiving an Alertable Parameter Value

```
package com.example.timers;

import java.util.Calendar;

public class TimerCommon {

  public TimerCommon(Alertable alertable,
                     String message,
                     Calendar wakeTime) {

    long whenMillis = wakeTime.getTimeInMillis();
```

```
    long currentMillis = System.currentTimeMillis();

    try {
      Thread.sleep(whenMillis - currentMillis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    alertable.alert(message);
  }
}
```

Listing 10-13: Everything Has to Start Somewhere!

```
package org.allyourcode.stopwatch;

public class Main {

  public static void main(String[] args) {
    new StopWatch(10);
  }
}
```

When you run the code in Listings 10-10 through 10-13, you experience a ten-second delay. Then you see the dialog box shown in Figure 10-14.

Figure 10-14: Running the code in Listings 10-10 through 10-13.

At the start of this section, I complain that without `TimerCommon`, your stopwatch code isn't responsive to new user input. Well, I must confess that the code in Listings 10-10 through **10**-13 doesn't solve the responsiveness problem. To make the program more responsive, you use the interface tricks in Listings 10-10 through 10-13, and, in addition, you put `TimerCommon` in a thread of its own. The trouble is that the separate thread business doesn't help you understand how interfaces work, so I don't bother creating an extra thread in this section's example. For a more honest multi-threading example, see Chapter 13.

One program; two Eclipse projects

To emphasize my point about the `StopWatch` and `TimerCommon` classes being developed independently, I've spread Listings 10-10 through 10-13 over two different Eclipse projects. The `StopWatch` and `Main` classes star in the 10-10 project, and `Alertable` and `TimerCommon` star in the 10-11 project. To make this multiproject code work, you have to tell Eclipse about one project's dependency on the other. Here's how:

1. **Right-click (in Windows) or Control-click (on a Mac) the 10-10 project's branch in the Package Explorer in Eclipse.**

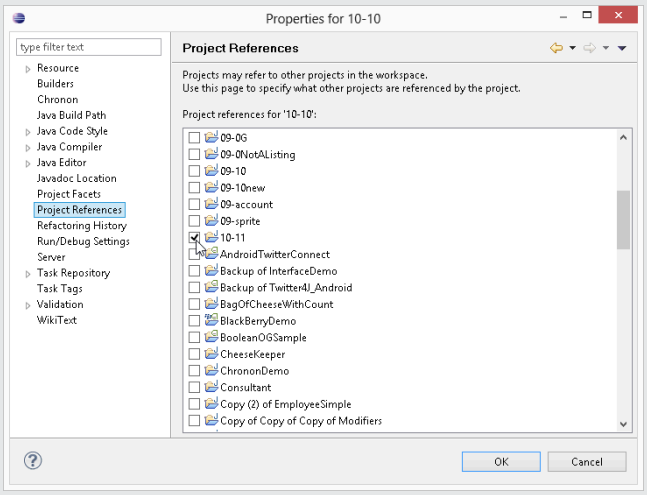
   You do this because, in this section's example, the code in the 10-10 project makes use of the code in the 10-11 project. (The `StopWatch` class creates a new `TimerCommon` instance.)

2. **From the contextual menu that appears, choose Properties.**

   The dialog box labeled Properties for 10-10 opens. On the left side, you see a list of categories.

3. **In the list of categories, click to select the Project References item.**

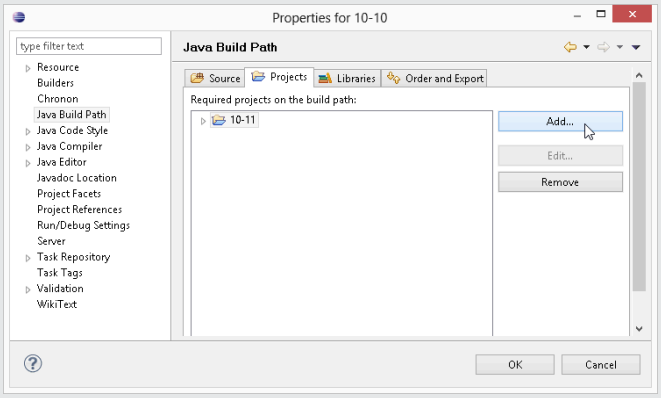
4. **In the main body of the dialog box, select the check box labeled 10-11, as shown in the first sidebar figure.**



Remember that the 10-10 project uses the constructor that's declared in the 10-11 project.

5. **In the list of categories, select the Java Build Path item.**

6. **In the main body of the Properties for 10-10 dialog box, select the Projects tab, as shown in the second sidebar figure.**



7. **On the right side of the Projects tab, click the Add button.**

   The Required Project Selection dialog box opens.

8. **In the Required Project Selection dialog box, select the 10-11 option to place a check mark next to it.**

   Remember (again) that the 10-10 project uses the constructor that's declared in the 10-11 project.

9. **Click OK to dismiss the Required Project Selection dialog box.**

   As a result, the Properties for 10-10 dialog box looks like the one shown in the second sidebar figure.

10. **Click OK to save your changes and to dismiss the Properties for 10-10 dialog box.**

    Now Eclipse knows that Project 10-10 depends on some code from 10-11.

A brief explanation of this section's code

In Listing 10-10, your code calls `new TimerCommon(this, "Stop", when)`. Here's the equivalent command, translated into English:

> Create a new `TimerCommon` object; tell it to call `this` code back at the moment that I've named `when`. Have the new `TimerCommon` object deliver a `"Stop"` message back to `this` code.

A detailed explanation of this section's code

The constructor call in Listing 10-13 creates a `StopWatch` instance. To understand how Listings 10-10 through 10-12 work, you have to trace the progress of that `StopWatch` instance throughout the run of the program (you can follow along in Figure 10-13):

- **In Listing 10-10, Java's `this` keyword represents the `StopWatch` instance.**

  The word `this` appears inside a `TimerCommon` constructor. So the next bunch of code to be executed is the code inside the `TimerCommon` constructor's body.

- **In the `TimerCommon` constructor's body (refer to Listing 10-12), the `alertable` parameter becomes synonymous with the original `StopWatch` instance.**

  The `TimerCommon` instance "sleeps" for a while.

- **Finally, with `alertable` referring to the `StopWatch` instance, Listing 10-12 calls `alertable.alert(message)`.**

  In other words, Listing 10-12 calls back the original `StopWatch` instance. Listing 10-12 knows how to call the original `StopWatch` instance, because the `StopWatch` instance passed itself (the keyword `this`) in the `TimerCommon` construction call.

How do interfaces help with all this? Remember that the `TimerCommon` class isn't your own code. Someone else wrote the `TimerCommon` class and placed it in a separate `com.example.timers` package. Whoever wrote the `TimerCommon` class knew nothing about you or your `StopWatch` class (the code in Listing 10-10). In particular, the `TimerCommon` class doesn't contain the following code:

```
public TimerCommon(StopWatch yourStopWatch,
                   String message,
                   Calendar wakeTime) {

yourStopWatch.alert(message);
```

Instead, the `TimerCommon` class is written for a more general audience. The `TimerCommon` class contains the following lines:

```
public TimerCommon(Alertable alertable,
                   String message,
                   Calendar wakeTime) {

alertable.alert(message);
```

The class's constructor expects its first argument to implement the `Alertable` interface. And sure enough, the first argument in `new TimerCommon(this, "Stop", when)` in Listing 10-10 is `this`, which is your `StopWatch` instance, which (Oh, joy!) implements `Alertable`. Here's the best part: As long as your class implements the `Alertable` interface, your class is guaranteed to have an `alert` method with one `String` argument (refer to Listing 10-11). So the `TimerCommon` class can safely call your code's `alert` method.

Time doesn't pass

Java's `Calendar` class has a misleading name: An instance of the `Calendar` class is a moment in time, not an entire month or year full of times. In Listing 10-10, the line

```
wakeTime = Calendar.
    getInstance()
```

makes `wakeTime` refer to a particular moment. In fact, when you call the parameterless `Calendar.getInstance()`, you get the current moment (the precise millisecond in which the method call is executed). You can check that moment's fields (the `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR`, `MINUTE`, `SECOND` and `MILLISECOND` fields). But you can also see the moment as a number of milliseconds since midnight on January 1, 1970.

A `Calendar` object's `getTimeInMillis` method finds the exact number of milliseconds since January 1, 1970 for that object. (Nowadays, it's a huge number.) The call `add(Calendar.SECOND, seconds)` adds a certain number of seconds to a particular `Calendar` moment. And the `System` class's static `currentTimeMillis` method provides a one-step way to find out how many milliseconds have passed since that landmark date in 1970.

How versatile is this interface?

The previous section shows what an interface can do. There, an interface bridges the gap between two otherwise unrelated pieces of code. To belabor this point even further (if that's possible), consider a new app of mine — a reminder app.

Here I sit, halfway around the world from where you created your stopwatch program. I know all about the `TimerCommon` class, but I know nothing about your stopwatch app. (Okay, maybe in real life, you live 20 miles from me in New Jersey, and I know about your stopwatch app because I wrote it for this chapter and the app isn't really yours. Who cares?) Here I am, halfway around the world, knowing nothing about your stopwatch app, using the `TimerCommon` class to create a completely different program — a reminder program. The code is in Listings 10-14 through 10-16.

Listing 10-14: What Is an Appointment?

```
package com.allmycode.reminder;

import java.util.Calendar;

public class Appointment {
  String name;
```

(continued)

Listing 10-14 ***(continued)***

```
  Calendar when;

  public Appointment(String name, Calendar when) {
    this.name = name;
    this.when = when;
  }
}
```

Listing 10-15: A Reminder Is an Appointment That's Alertable

```
package com.allmycode.reminder;

import java.awt.Toolkit;
import java.util.Calendar;

import javax.swing.JOptionPane;

import com.example.timers.Alertable;
import com.example.timers.TimerCommon;

public class Reminder extends Appointment
                      implements Alertable {

  public Reminder(String name, Calendar when) {
    super(name, when);
    new TimerCommon(this, name, when);
  }

  @Override
  public void alert(String message) {
    Toolkit.getDefaultToolkit().beep();
    JOptionPane.showMessageDialog(null, message,
        "Reminder!", JOptionPane.WARNING_MESSAGE);
  }
}
```

Listing 10-16: Creating a Reminder

```
package com.allmycode.reminder;

import java.util.Calendar;

public class Main {

  public static void main(String[] args) {
    Calendar when = Calendar.getInstance();
    when.add(Calendar.SECOND, 5);
    new Reminder("Take a break!", when);
  }
}
```

The call to `beep()` in Listing 10-15 makes some sort of noise (no big surprise). But you might want to know a bit about the details. Java has a `Toolkit` class with a static `getDefaultToolkit` method. A call to `Toolkit.getDefaultToolkit()` returns a connection to the user's operating system. This connection (an instance of the `Toolkit` class) has its own `beep` method. There! Now you know.

As in your case, my class implements the `Alertable` interface and has an `alert(String message)` method. In Listing 10-15, my `Reminder` object passes itself (`this`) to a new `TimerCommon` object. Because the `TimerObject` class's code expects the first constructor parameter to be `Alertable`, everything is okay. The `TimerCommon` object sleeps until it's time to remind the user. At the appropriate time, the `TimerCommon` object calls my object's `alert` method — again, the use of an interface adds versatility to the code by cutting across class/subclass lines, as shown in Figure 10-15.

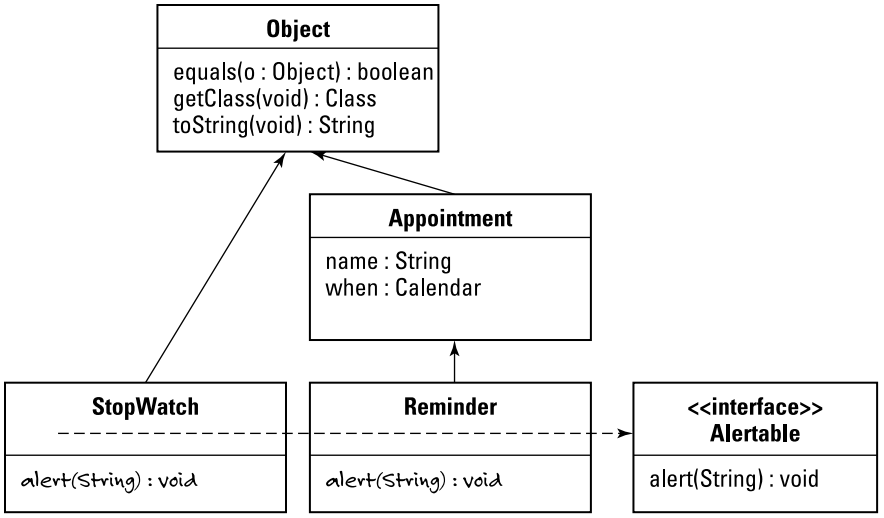


Figure 10-15: Cutting across class lines.

Java's super keyword

Here's an excerpt from Listing 10-15:

```
public class Reminder extends Appointment
                      implements Alertable {

  public Reminder(String name, Calendar when) {
    super(name, when);
    new TimerCommon(this, name, when);
  }
```

In Listing 10-15, the word `super` stands for *the superclass's constructor*. In particular, the call `super(name, when)` tells Java to find the superclass of the current class, to call that superclass's constructor, and to feed the parameter values `name` and `when` to the superclass constructor.

My `Reminder` class extends the `Appointment` class (refer to Listing 10-14). So in Listing 10-15, the call `super(name, when)` invokes one of the `Appointment` class's constructors.

Of course, the `Appointment` class had better have a constructor whose types match the `super` call's parameter types (`String` for `name` and `Calendar` for `when`). Otherwise, the Eclipse editor displays lots of red marks. Fortunately, the `Appointment` class in Listing 10-14 has the appropriate two-parameter constructor.

```
public Appointment(String name, Calendar when) {
  this.name = name;
  this.when = when;
}
```

What Does This Have to Do with Android?

Employees and consultants make good examples of classes and subclasses. But at this point in the book, you might be interested in a more practical programming example. How about an Android app? The first example is ruefully simple, but it's one that an Android programmer sees every day. It's an Android *Activity*.

A typical Android app displays one screen at a time, as shown in Figure 10-16. A screenful of material might present the user with a list of options and a Start button. The next screenful (after the user clicks Start, for example) shows some helpful information, such as a map, a video, or a list of items for sale. When the user touches on this information screen, the app's display changes to reveal a third screen, showing detailed information about whatever option the user selected. Eventually, the user dismisses the detail screen by clicking the Back button.

In Android terminology, each screenful of material is an *activity*. As the user progresses through the sequence of screens displayed in Figure 10-16, Android displays three activities. (It displays the middle activity twice — once after the user clicks Start and a second time after the user dismisses the detailed-info activity.)

Figure 10-16: Android displays a sequence of screens.

Android developers deal with activities all the time, so the creators of Android have created an `Activity` class. The `Activity` class is part of Android's Application Programming Interface (API), the enormous library of classes that's available to every Android developer. You download the Android API when you follow the instructions in Chapter 2.

In Chapter 4, you create a brand-new Android app. Eclipse creates some skeletal code (enough to run a simple "Hello" program). I've copied this skeletal code in Listing 10-17.

Listing 10-17: Eclipse Creates a Main Activity

```
package com.allmycode.myfirstandroidapp;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

}
```

The following list can help you relate elements from Listing 10-17 to this chapter's discussion of classes, subclasses, and interfaces:

- **Every Android app is in a package of its own.**

  The app in Listing 10-17 belongs to the package named `com.allmycode.myfirstandroidapp`.

- **If the first part of a package name is `android`, that package probably belongs to Google's Android operating system code.**

  For example, Android's `Activity` class lives in the `android.app` package. When I import `android.app.Activity`, I can refer to the `Activity` class in the rest of Listing 10-17 without repeating the class's fully qualified name.

  Java has no rule to enforce package naming conventions. You can create your own package and name it `android.app`, and you can use that package in code that has nothing to do with Google Android. But a good developer never looks for trouble. If convention dictates that the word `android` signals a package in the official Android API, don't use the word `android` in your own package names.

- **The `MainActivity` class extends the `android.app.Activity` class.**

  A `MainActivity` *is an* `Activity`. Therefore, the `MainActivity` in Listing 10-17 has all the rights and responsibilities that any `Activity` instance has. For example, the `MainActivity` has `onCreate` and `onCreateOptionsMenu` methods, which it overrides in Listing 10-17.

  In fact, the `MainActivity` class inherits about 5,000 lines of Java code from Android's `Activity` class. The inherited methods include ones such as `getCallingActivity`, `getCallingPackage`, `getParent`, `getTitle`, `getTitleColor`, `getWindow`, `onBackPressed`, `onKeyDown`, `onKeyLongPress`, `onLowMemory`, `onMenuItemSelected`, `setTitle`, `setTitleColor`, `startActivity`, `finish`, and many, many others. You inherit all this functionality by typing two simple words: `extends Activity`.

  The Android `Activity` class extends another class: Android's own `ContextThemeWrapper`. Without knowing what a `ContextThemeWrapper` is (and without caring), your app's own `MainActivity` class (refer to Listing 10-17) extends Android's `Activity` class, which in turn extends Android's `ContextThemeWrapper` class. So in the terminology of familial relationships, your `MainActivity` class is a descendant of Android's `ContextThemeWrapper`. Your `MainActivity` class is a kind of `ContextThemeWrapper`.

- **On creating an activity, you find out what was going on when the activity was last destroyed.**

  The parameter `savedInstanceState` stores information about what was going on when the activity was last destroyed. If the `savedInstanceState` contains any meaningful information, it's because the activity was destroyed in the middle of a run. Maybe the user tilted the device sideways, causing the activity to be destroyed and then re-created in Landscape mode. (See Chapter 5.)

  In Listing 10-17, you feed the information in the `savedInstanceState` parameter to the code's superclass, which is Android's `Activity` class. In turn, the `Activity` class's constructor does all kinds of useful things with `savedInstanceState`. Among other things, the `Activity` class's constructor restores much of your activity to the state it was in when your activity was last destroyed.

- **The `MainActivity` class inherits a `setContentView` method from the `Activity` class.**

  A call to the `setContentView` method's parameter is a code number (as described in a sidebar in Chapter 4). The `setContentView` method looks up that code number and finds an XML file in your project's `res\layout` directory. (In this example, the filename is `activity_main.xml`.) The method then *inflates* the XML file: That is, the method interprets the XML file's text as the description of a nice-looking arrangement of items on the user's screen. This arrangement becomes the overall look of your activity's screen.

- **The `MainActivity` class overrides the `Activity` class's `onCreateOptionsMenu` method.**

  At some point during the display of `MainActivity` on the screen, Android creates the activity's Options menu. (Normally, the user opens the Options menu by touching an icon containing a few dots or dashes.) In Listing 10-17, the call to `inflate` once again turns the text from an XML file (`res\menu\main.xml`) into a bunch of menu items and menu actions.

  The `onCreateOptionsMenu` method returns `true`, which means, "Yes, I've done all that I have to do in setting up the activity's Options menu." (A `false` value would indicate that other code must do the follow-up work to help set up the Options menu.)

So much for Eclipse's autogenerated app. In the next few chapters, I introduce more Java features, and I show you how to build more functionality on top of the autogenerated Android app.