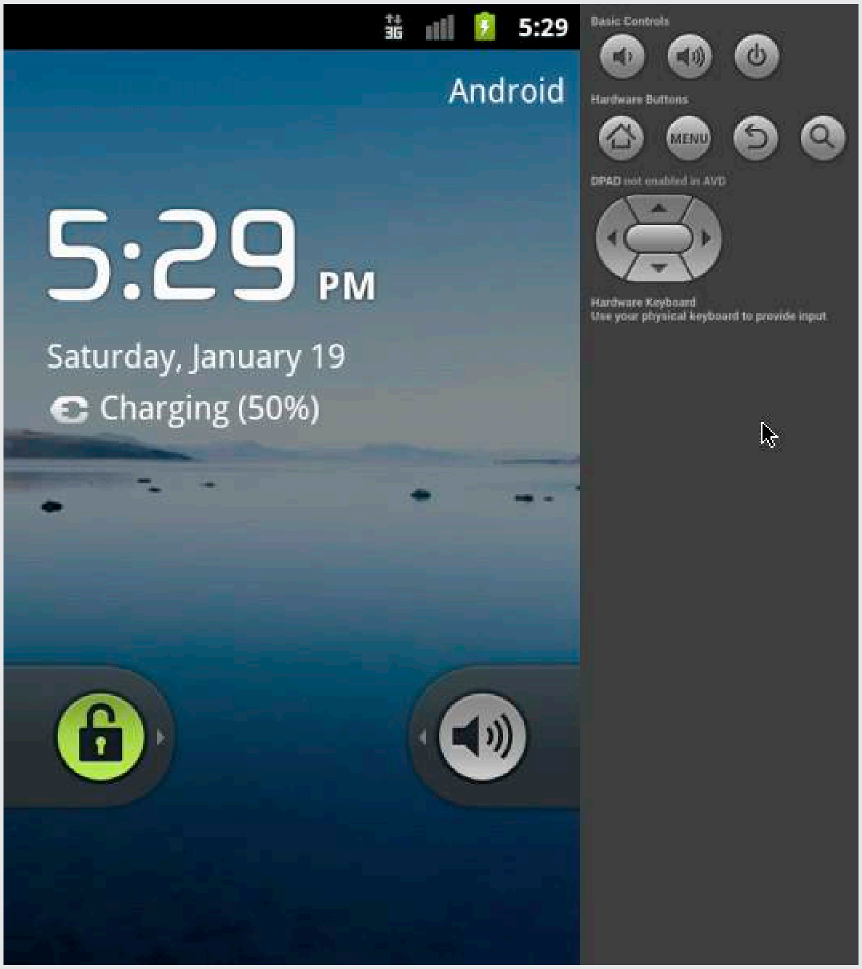
Part III

Working with the Big Picture: Object-Oriented Programming



Check out the article "Classes and Objects" (and more) online at `www.dummies.com/extras/javaprogrammingforandroiddevelopers`.

In this part . . .

- ✓ Understanding object-oriented programming (at last!)
- ✓ Reusing code
- ✓ Establishing lines of communication among the parts of your app

Chapter 9

Why Object-Oriented Programming Is Like Selling Cheese

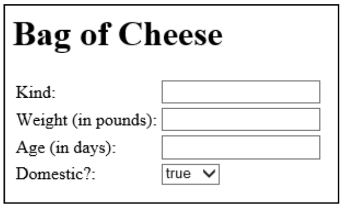
In This Chapter

- The truth about object-oriented programming
- Why a class is actually a Java type
- An end to the mystery surrounding words like `static`

Andy's Cheese and Java Emporium carries fine cheeses and freshly brewed java from around the world (especially from Java in Indonesia). The Emporium is in Cheesetown, Pennsylvania, a neighborhood along the Edenville–Cheesetown Road in Franklin County.

The emporium sells cheese by the bag, each containing a certain variety, such as Cheddar, Swiss, Munster, or Limburger. Bags are labeled by weight and by the number of days the cheese was aged (admittedly, an approximation). Bags also carry the label *Domestic* or *Imported,* depending on the cheese's country of origin.

Before starting up the emporium, Andy had lots of possessions — material and otherwise. He had a family, a cat, a house, an abandoned restaurant property, a bunch of restaurant equipment, a checkered past, and a mountain of debt. But for the purpose of this narrative, Andy had only one thing: a form. Yes, Andy had developed a form for keeping track of his emporium's inventory. The form is shown in Figure 9-1.

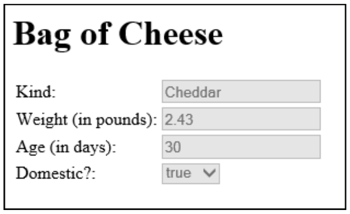


Bag of Cheese

Kind:	
Weight (in pounds):	
Age (in days):	
Domestic?:	true

Figure 9-1: An online form.

Exactly one week before the emporium's grand opening, Andy's supplier delivered one bag of cheese. Andy entered the bag's information into the inventory form. The result is shown in Figure 9-2.

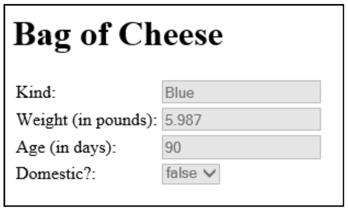


Bag of Cheese

Kind:	Cheddar
Weight (in pounds):	2.43
Age (in days):	30
Domestic?:	true

Figure 9-2: A virtual bag of cheese.

Andy had only a form and a bag of cheese (which isn't much to show for all his hard work), but the next day the supplier delivered five more bags of cheese. Andy's second entry looked like the one shown in Figure 9-3, and the next several entries looked similar.



Bag of Cheese

Kind:	Blue
Weight (in pounds):	5.987
Age (in days):	90
Domestic?:	false

Figure 9-3: Another virtual bag of cheese.

At the end of the week, Andy was giddy: He had exactly one inventory form and six bags of cheese.

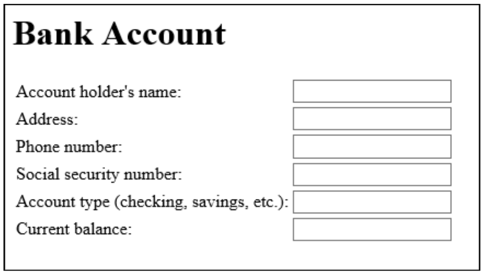
The story doesn't end here. As the grand opening approached, Andy's supplier brought many more bags so that, eventually, Andy had his inventory form and several hundred bags of cheese. The business even became an icon on Interstate Highway 81 in Cheesetown, Pennsylvania. But as far as you're concerned, the business had, has, and always will have only one form and any number of cheese bags.

That's the essence of object-oriented programming!

Classes and Objects

Java is an object-oriented programming language. A program that you create in Java consists of at least one class.

A class is like Andy's blank form, described in this chapter's introduction. That is, a class is a general description of some kind of thing. In the introduction to this chapter, the class (the form) describes the characteristics that any bag of cheese possesses. But imagine other classes. For example, Figure 9-4 illustrates a bank account class:

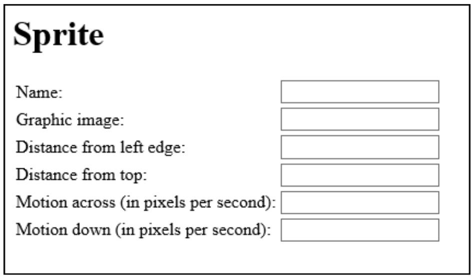


Bank Account

Account holder's name:	
Address:	
Phone number:	
Social security number:	
Account type (checking, savings, etc.):	
Current balance:	

Figure 9-4: A bank account class.

Figure 9-5 illustrates a sprite class, which is a class for a character in a computer game:



Sprite

Name:	
Graphic image:	
Distance from left edge:	
Distance from top:	
Motion across (in pixels per second):	
Motion down (in pixels per second):	

Figure 9-5: A sprite class.

What is a class, really?

In practice, a class doesn't look like any of the forms in Figures 9-1 through 9-5. In fact, a class doesn't look like anything. Instead, a Java class is a bunch of text describing the kinds of things that I refer to as "blanks to be filled in." Listing 9-1 contains a real Java class — the kind of class you write when you program in Java.

Listing 9-1: A Class in the Java Programming Language

```
package com.allmycode.andy;

public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;
}
```

As a developer, your primary job is to create classes. You don't develop attractive online forms like the form in Figure 9-1. Instead, you write Java language code — code containing descriptions, like the one in Listing 9-1.

Compare Figure 9-1 with Listing 9-1. In what ways are they the same, and in what ways are they different? What does one have that the other doesn't have?

- **The form in Figure 9-1 appears on a user's screen. The code in Listing 9-1 does not.**

  A Java class isn't necessarily tied to a particular display. Yes, you can display a bank account on a user's screen. But the bank account isn't a bunch of items on a computer screen — it's a bunch of information in the bank's computers.

  In fact, some Java classes are difficult to visualize. Android's `SQLiteOpenHelper` class assists developers in the creation of databases. An `SQLiteOpenHelper` doesn't look like anything in particular, and certainly not an online form or a bag of cheese.

- **Online forms appear in some contexts but not in others. In contrast, classes affect every part of every Java program's code.**

  Forms show up on web pages, in dialog boxes, and in other situations. But when you use a word processing program to type a document, you deal primarily with free-form input. I didn't write this paragraph by filling in some blanks. (Heaven knows! I wish I could!)

The paragraphs I've written started out as part of a document in an Android word processing application. In the document, every paragraph has its own alignment, borders, indents, line spacing, styles, and many other characteristics. As a Java class, a list of paragraph characteristics might look something like this:

```
class Paragraph {
  int alignment;
  int borders;
  double leftIndent;
  double lineSpacing;
  int style;
}
```

When I create a paragraph, I don't fill in a form. Instead, I type words, and the underlying word processing app deals silently with its `Paragraph` class.

- **The form shown in Figure 9-1 contains several fields, and so does the code in Listing 9-1.**

  In an online form, a field is a blank space — a place that's eventually filled with specific information. In Java, a *field* is any characteristic that you (the developer) attribute to a class. The `BagOfCheese` class in Listing 9-1 has four fields, and each of the four fields has a name: `kind`, `weight`, `daysAged`, or `isDomestic`.

  Like an online form, a Java class describes items by listing the characteristics that each of the items has. Both the form in Figure 9-1 and the code in Listing 9-1 say essentially the same thing: Each bag of cheese has a certain kind of cheese, a certain weight, a number of days that the cheese was aged, and a domestic-or-imported characteristic.

- **The code in Listing 9-1 describes exactly the kind of information that belongs in each blank space. The form in Figure 9-1 is much more permissive.**

  Nothing in Figure 9-1 indicates what kinds of input are permitted in the Weight field. The weight in pounds can be a whole number (0, 1, 2, and so on) or a decimal number (such as 3.14159, the weight of a big piece of "pie"). What happens if the user types the words *three pounds* into the form in Figure 9-1? Does the form accept this input, or does the computer freeze up? A developer can add extra code to test for valid input in a form, but, on its own, a form cares little about the kind of input that the user enters.

  In contrast, the code in Listing 9-1 contains this line:

  ```
  double weight;
  ```

This line tells Java that every bag of cheese has a characteristic named `weight` and that a bag's weight must be of type `double`. Similarly, each bag's `daysAged` value is an `int`, each bag's `isDomestic` value is `boolean`, and each bag's `kind` value has the type `String`.

The unfortunate pun in the previous paragraph makes life more difficult for me, the author! A Java `String` has nothing to do with the kind of cheese that peels into strips. A Java `String` is a sequence of characters, like the sequence `"Cheddar"` or the sequence `"qwoiehasljsal"` or the sequence `"Go2theMoon!"`. So the `String kind` line in Listing 9-1 indicates that a bag of cheese might contain `"Cheddar"`, but it might also contain `"qwoiehasljsal"` cheese or `"Go2theMoon!"` cheese. Well, that's what happens when Andy starts a business from scratch.

What is an object?

At the start of this chapter's detailed Cheese Emporium exposé, Andy had nothing to his name except an online form — the form in Figure 9-1. Life was simple for Andy and his dog Fido. But eventually the suppliers delivered bags of cheese. Suddenly, Andy had more than just an online form —he had things whose characteristics matched the fields in the form. One bag had the characteristics shown in Figure 9-2; another bag had the characteristics shown in Figure 9-3.

In the terminology of object-oriented programming, each bag of cheese is an *object*, and each bag of cheese is an *instance* of the class in Listing 9-1.

You can also think of classes and objects as part of a hierarchy. The `BagOfCheese` class is at the top of the hierarchy, and each instance of the class is attached to the class itself. See Figures 9-6 and 9-7.

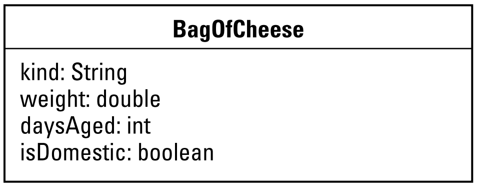


Figure 9-6: First, Andy has a class.

The diagrams in Figures 9-6 and 9-7 are part of the standardized Unified Modeling Language (UML). For more info about UML, visit `www.omg.org/spec/UML/`.

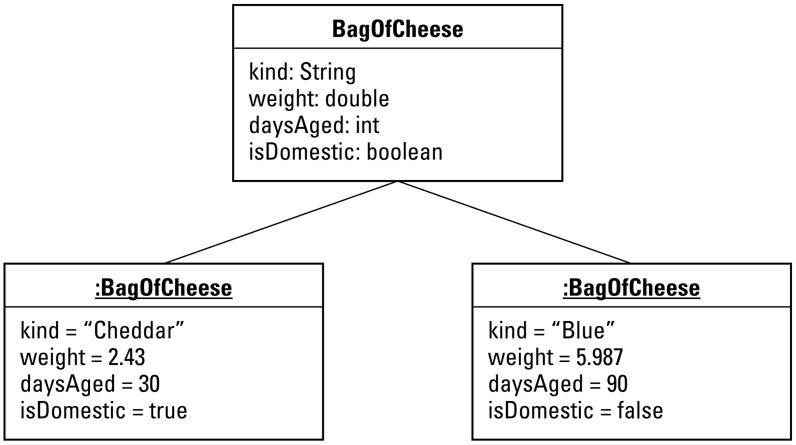


Figure 9-7: Later, Andy has a class and two objects.

REMEMBER

An object is a particular thing. (For Andy, an object is a particular bag of cheese.) A class is a description with blanks to be filled in. (For Andy, a class is a form with four blank fields: a field for the kind of cheese, another field for the cheese's weight, a third field for the number of days aged, and a fourth field for the Domestic-or-Imported designation.)

And don't forget: Your primary job is to create classes. You don't develop attractive online forms like the form in Figure 9-1. Instead, you write Java language code — code containing descriptions, like the one in Listing 9-1.

Creating objects

Listing 9-2 contains real-life Java code to create two objects — two instances of the class in Listing 9-1.

Listing 9-2: Creating Two Objects

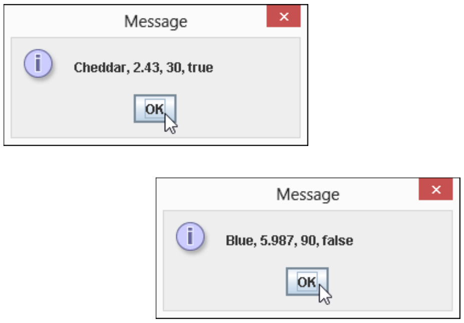
```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag1 = new BagOfCheese();
    bag1.kind = "Cheddar";
    bag1.weight = 2.43;
    bag1.daysAged = 30;
```

(continued)

Listing 9-2 *(continued)*

```
        bag1.isDomestic = true;

        BagOfCheese bag2 = new BagOfCheese();
        bag2.kind = "Blue";
        bag2.weight = 5.987;
        bag2.daysAged = 90;
        bag2.isDomestic = false;

        JOptionPane.showMessageDialog(null,
            bag1.kind + ", " +
            bag1.weight + ", " +
            bag1.daysAged + ", " +
            bag1.isDomestic);

        JOptionPane.showMessageDialog(null,
            bag2.kind + ", " +
            bag2.weight + ", " +
            bag2.daysAged + ", " +
            bag2.isDomestic);
    }
}
```

A run of the code in Listing 9-2 is shown in Figure 9-8.



```
Cheddar, 2.43, 30, true
```

```
Blue, 5.987, 90, false
```

Figure 9-8: Running the code from Listing 9-2.

To vary the terminology, I might say that the code in Listing 9-2 creates “two `BagOfCheese` objects” or “two `BagOfCheese` instances,” or I might say that the `new BagOfCheese()` statements in Listing 9-2 *instantiate* the `BagOfCheese` class. One way or another, Listing 9-1 declares the existence of one class, and Listing 9-2 declares the existence of two objects.

In Listing 9-2, each use of the words `new BagOfCheese()` is a *constructor call*. For details, see the “Calling a constructor” section later in this chapter.

To run the code in Listing 9-2, you put two Java files (`BagOfCheese.java` from Listing 9-1 and `CreateBags.java` from Listing 9-2) in the same Eclipse project.

In Listing 9-2, I use ten statements to create two bags of cheese. The first statement (`BagOfCheese bag1 = new BagOfCheese()`) does three things:

- With the words

  ```
  BagOfCheese bag1
  ```

  the first statement declares that the variable `bag1` refers to a bag of cheese.
- With the words

  ```
  new BagOfCheese()
  ```

  the first statement creates a bag with no particular cheese in it. (If it helps, you can think of it as an empty bag reserved for eventually storing cheese.)
- Finally, with the equal sign, the first statement makes the `bag1` variable refer to the newly created bag.

The next four statements in Listing 9-2 assign values to the fields of `bag1`:

```
bag1.kind = "Cheddar";
bag1.weight = 2.43;
bag1.daysAged = 30;
bag1.isDomestic = true;
```

To refer to one of an object's fields, follow a reference to the object with a dot and then the field's name. (For example, follow `bag1` with a dot, and then the field name `kind`.)

The next five statements in Listing 9-2 do the same for a second variable, `bag2`, and a second bag of cheese.

Reusing names

In Listing 9-2, I declare two variables — `bag1` and `bag2` — to refer to two different `BagOfCheese` objects. That's fine. But sometimes having only one variable and reusing it for the second object works just as well, as shown in Listing 9-3.

Listing 9-3: Reusing the bag Variable

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag = new BagOfCheese();
    bag.kind = "Cheddar";
    bag.weight = 2.43;
    bag.daysAged = 30;
    bag.isDomestic = true;

    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);

    bag = new BagOfCheese();
    bag.kind = "Blue";
    bag.weight = 5.987;
    bag.daysAged = 90;
    bag.isDomestic = false;

    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);
  }
}
```

In Listing 9-3, when the computer executes the second `bag = new BagOfCheese()` statement, the old object (the bag containing cheddar) has disappeared. Without `bag` (or any other variable) referring to that cheddar object, there's no way your code can do anything with the cheddar object. Fortunately, by the time you reach the second `bag = new BagOfCheese()` statement, you're finished doing everything you want to do with the original cheddar bag. In this case, reusing the `bag` variable is acceptable.

WARNING!

When you reuse a variable (like the one and only `bag` variable in Listing 9-3), you do so by using an assignment statement, not an initialization. In other words, you don't write `BagOfCheese bag` a second time in your code. If you do, you see error messages in the Eclipse editor.

To be painfully precise, you can, in fact, write `BagOfCheese bag` more than once in the same piece of code. For an example, see the use of shadowing later in this chapter, in the "Constructors with parameters" section.

In Listing 9-1, none of the `BagOfCheese` class's fields is `final`. In other words, the class's code lets you reassign values to the fields inside a `BagOfCheese` object. With this information in mind, you can shorten the code in Listing 9-3 even more, as shown in Listing 9-4.

Listing 9-4: Reusing a bag Object's Fields

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag = new BagOfCheese();
    bag.kind = "Cheddar";
    bag.weight = 2.43;
    bag.daysAged = 30;
    bag.isDomestic = true;

    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);

    // bag = new BagOfCheese();
    bag.kind = "Blue";
    bag.weight = 5.987;
    bag.daysAged = 90;
    bag.isDomestic = false;

    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);
  }
}
```

With the second constructor call in Listing 9-4 commented out, you don't make the `bag` variable refer to a new object. Instead, you economize by assigning new values to the existing object's fields.

In some situations, reusing an object's fields can be more efficient (quicker to execute) than creating a new object. But whenever I have a choice, I prefer to write code that mirrors real data. If an actual bag's content doesn't change from cheddar cheese to blue cheese, I prefer not to change a `BagOfCheese` object's `kind` field from `"Cheddar"` to `"Blue"`.

Calling a constructor

In Listing 9-2, the words `new BagOfCheese()` look like method calls, but they aren't — they're constructor calls. A *constructor call* creates a new object from an existing class. You can spot a constructor call by noticing that

- **A constructor call starts with Java's `new` keyword:**

  ```
  new BagOfCheese()
  ```

  and

- **A constructor call's name is the name of a Java class:**

  ```
  new BagOfCheese()
  ```

When the computer encounters a method call, the computer executes the statements inside a method's declaration. Similarly, when the computer encounters a constructor call, the computer executes the statements inside the constructor's declaration. When you create a new class (as I did in Listing 9-1), Java can create a constructor declaration automatically. If you want, you can type the declaration's code manually. Listing 9-5 shows you what the declaration's code would look like:

Listing 9-5: The Parameterless Constructor

```
package com.allmycode.andy;

public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  BagOfCheese() {
  }
}
```

In Listing 9-5, the boldface code

```
BagOfCheese() {
}
```

is a very simple constructor declaration. This declaration (unlike most constructor declarations) has no statements inside its body. This declaration is simply a *header* (`BagOfCheese()`) and an empty body (`{}`).

You can type Listing 9-5 exactly as it is. Alternatively, you can omit the code in boldface type, and Java creates that constructor for you automatically. (You don't see the constructor declaration in the Eclipse editor, but Java behaves as if the constructor declaration exists.) To find out when Java creates a constructor declaration automatically and when it doesn't, see the "Constructors with parameters" section, later in this chapter.

A constructor's declaration looks much like a method declaration. But a constructor's declaration differs from a method declaration in two ways:

- **A constructor's name is the same as the name of the class whose objects the constructor constructs.**

  In Listing 9-5, the class name is `BagOfCheese`, and the constructor's header starts with the name `BagOfCheese`.
- **Before the constructor's name, the constructor's header has no type.**

  Unlike a method header, the constructor's header doesn't say `int BagOfCheese()` or even `void BagOfCheese()`. The header simply says `BagOfCheese()`.

The constructor declaration in Listing 9-5 contains no statements. That isn't typical of a constructor, but it's what you get in the constructor that Java creates automatically. With or without statements, calling the constructor in Listing 9-5 creates a brand-new `BagOfCheese` object.

More About Classes and Objects (Adding Methods to the Mix)

In Chapters 5 and 7, I introduce parameter passing. In those chapters, I unobtrusively avoid details about passing objects to methods. (At least, I hope it's unobtrusive.) In this chapter, I shed my coy demeanor and face the topic (passing objects to methods) head-on.

I start with an improvement on an earlier example. The code in Listing 9-2 contains two nasty-looking `showMessageDialog` calls. You can streamline the code there by moving the calls to a method. Here's how:

1. **View the code from Listing 9-2 in the Eclipse editor.**

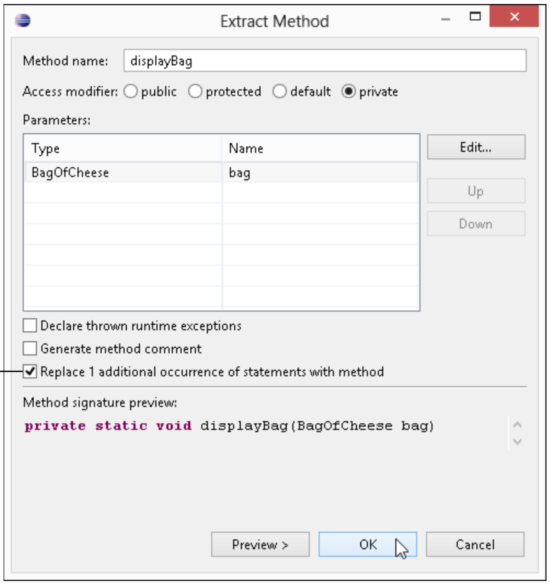
The `CreateBags.java` file is in the 09-01 project that you import in Chapter 2.

2. **Use the mouse to select the entire statement containing the first call to `JOptionPane.showMessagedialog`.**

   Be sure to highlight all words in the statement, starting with the word `JOptionPane` and ending with the semicolon four lines later.

3. **On the Eclipse main menu, choose Refactor➪Extract Method.**

   The Extract Method dialog box in Eclipse appears, as shown in Figure 9-9.



Extract Method

Method name: displayBag

Access modifier: ○ public ○ protected ○ default ● private

Parameters:

Type	Name
BagOfCheese	bag

☐ Declare thrown runtime exceptions
☐ Generate method comment
☑ Replace 1 additional occurrence of statements with method

Method signature preview:

```
private static void displayBag(BagOfCheese bag)
```

Figure 9-9: The Extract Method dialog box.

4. **In the Method Name field in the Extract Method dialog box, type** displayBag**.**

5. **(Optional) In the Name column in the Extract Method dialog box, change** `bag1` **to** `bag`**.**

6. **Make sure that a check mark appears in the box labeled Replace 1 Additional Occurrence of Statements with Method.**

   This check mark indicates that Eclipse will replace both `show MessageDialog` calls with a call to the new `displayBag` method.

7. **Click OK.**

   Eclipse dismisses the Extract Method dialog box and replaces your Java code with the new code in Listing 9-6.

Listing 9-6: A Method Displays a Bag of Cheese

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag1 = new BagOfCheese();
    bag1.kind = "Cheddar";
    bag1.weight = 2.43;
    bag1.daysAged = 30;
    bag1.isDomestic = true;

    BagOfCheese bag2 = new BagOfCheese();
    bag2.kind = "Blue";
    bag2.weight = 5.987;
    bag2.daysAged = 90;
    bag2.isDomestic = false;

    displayBag(bag1);

    displayBag(bag2);
  }

  private static void displayBag(BagOfCheese bag) {
    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);
  }
}
```

According to the `displayBag` declaration (Listing 9-6), the `displayBag` method takes one parameter. That parameter must be a `BagOfCheese` instance. Inside the body of the method declaration, you refer to that instance with the parameter name `bag`. (You refer to `bag.kind`, `bag.weight`, `bag.daysAged`, and `bag.isDomestic`.)

In the `main` method, you create two `BagOfCheese` instances: `bag1` and `bag2`. You call `displayBag` once with the first instance (`displayBag(bag1)`), and call it a second time with the second instance (`displayBag(bag2)`).

Constructors with parameters

Listing 9-7 contains a variation on the theme from Listing 9-2.

Listing 9-7: Another Way to Create Two Objects

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag1 =
        new BagOfCheese("Cheddar", 2.43, 30, true);
    BagOfCheese bag2 =
        new BagOfCheese("Blue", 5.987, 90, false);

    displayBag(bag1);

    displayBag(bag2);
  }

  private static void displayBag(BagOfCheese bag) {
    JOptionPane.showMessageDialog(null,
        bag.kind + ", " +
        bag.weight + ", " +
        bag.daysAged + ", " +
        bag.isDomestic);
  }
}
```

Listing 9-7 calls a `BagOfCheese` constructor with four parameters, so the code has to have a four-parameter constructor. In Listing 9-8, I show you how to declare that constructor.

Listing 9-8: A Constructor with Parameters

```
package com.allmycode.andy;

public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  BagOfCheese() {
  }

  BagOfCheese(String pKind, double pWeight,
               int pDaysAged, boolean pIsDomestic) {
    kind = pKind;
    weight = pWeight;
    daysAged = pDaysAged;
    isDomestic = pIsDomestic;
  }
}
```

Listing 9-8 borrows some tricks from Chapters 5 and 7. In those chapters, I introduce the concept of *overloading* — reusing a name by providing different parameter lists. Listing 9-8 has two different `BagOfCheese` constructors — one with no parameters and another with four parameters. When you call a `BagOfCheese` constructor (as in Listing 9-7), Java knows which declaration to execute by matching the parameters in the constructor call. The call in Listing 9-7 has parameters of type `String`, `double`, `int`, and `boolean`, and the second constructor in Listing 9-8 has the same types of parameters in the same order, so Java calls the second constructor in Listing 9-8.

You might also notice another trick from Chapter 7. In Listing 9-8, in the second constructor declaration, I use different names for the parameters and the class's fields. For example, I use the parameter name `pKind` and the field name `kind`. So what happens if you use the same names for the parameters and the fields, as in this example:

```
// DON'T DO THIS
BagOfCheese(String kind, double weight,
            int daysAged, boolean isDomestic) {
  kind = kind;
  weight = weight;
  daysAged = daysAged;
  isDomestic = isDomestic;
}
```

Figure 9-10 shows you exactly what happens. (***Hint:*** Nothing good happens!)

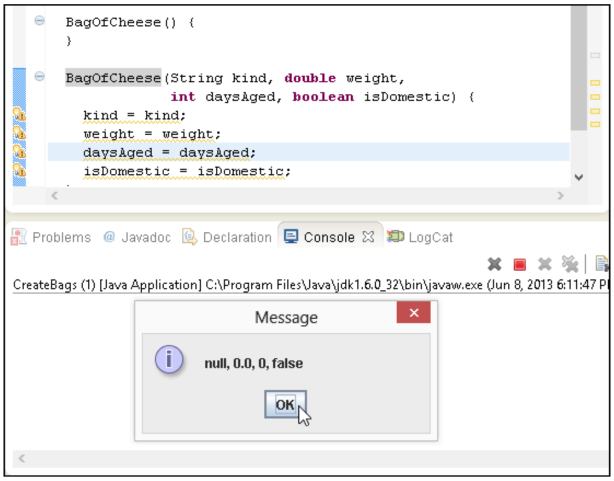
Aside from all the yellow warning markers in the Eclipse editor, the code with duplicate parameter and field names gives you the useless results from Figure 9-10. The code in Listing 9-8 makes the mistake of containing two `kind` variables — one inside the constructor and another outside of the constructor, as shown in Figure 9-11.

When you have a field and a parameter with the same name, `kind`, the parameter name *shadows* the field name inside the method or the constructor. So, outside the constructor declaration, the word `kind` refers to the field name. Inside the constructor declaration, however, the word `kind` refers only to the parameter name. So, in the horrible code with duplicate names, the statement

```
kind = kind;
```

does nothing to the `kind` field. Instead, this statement tells the computer to make the `kind` parameter refer to the same string that the `kind` parameter already refers to.

If this explanation sounds like nonsense to you, it is.

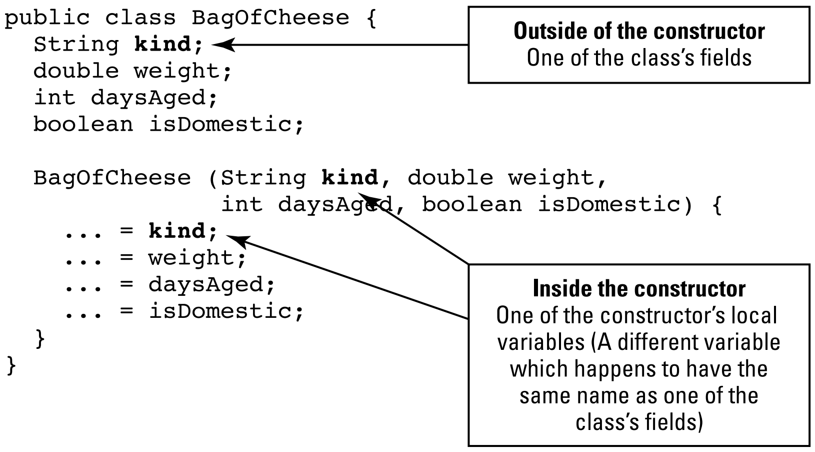


```
BagOfCheese() {
}

BagOfCheese(String kind, double weight,
        int daysAged, boolean isDomestic) {
    kind = kind;
    weight = weight;
    daysAged = daysAged;
    isDomestic = isDomestic;
```

```
CreateBags (1) [Java Application] C:\Program Files\Java\jdk1.6.0_32\bin\javaw.exe (Jun 8, 2013 6:11:47 P
```

```
null, 0.0, 0, false
```

Figure 9-10: Some unpleasant results.



```
public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  BagOfCheese (String kind, double weight,
                int daysAged, boolean isDomestic) {
     ... = kind;
     ... = weight;
     ... = daysAged;
     ... = isDomestic;
  }
}
```

Outside of the constructor
One of the class's fields
Inside the constructor
One of the constructor's local variables (A different variable which happens to have the same name as one of the class's fields)

Figure 9-11: Two `kind` variables.

The `kind` variable in the constructor declaration's parameter list is *local* to the constructor. Any use of the word `kind` outside the constructor cannot refer to the constructor's local `kind` variable.

Fields are different. You can refer to a field anywhere in the class's code. For example, in Listing 9-8, the second constructor declaration has no local `kind` variable of its own. Inside that constructor's body, the word `kind` refers to the class's field.

One way or another, the second constructor in Listing 9-8 is cumbersome. Do you always have to make up peculiar names like `pKind` for a constructor's parameters? No, you don't. To find out why, see the "This is it!" section.

The default constructor

In Listing 9-1, I don't explicitly type a parameterless constructor into my program's code, and Java creates a parameterless constructor for me. (I don't see a parameterless constructor in Listing 9-1, but I can still call `new BagOfCheese()` in Listing 9-2.) But in Listing 9-8, if I didn't explicitly type the parameterless constructor in my code, Java wouldn't have created a parameterless constructor for me. A call to `new BagOfCheese()` would have been illegal. (The Eclipse editor would tell me that *The BagOfCheese() constructor is undefined.*)

Here's how it works: When you define a class, Java creates a parameterless constructor (known formally as a *default constructor*) if, and only if, you haven't explicitly defined any constructors in your class's code. When Java encounters Listing 9-1, Java automatically adds a parameterless constructor to your `BagOfCheese` class. But when Java encounters Listing 9-8, you have to type the lines

```
BagOfCheese() {
}
```

into your code. If you don't, calls to `new BagOfCheese()` (with no parameters) will be illegal.

This is it!

The naming problem that crops up earlier in this chapter, in the "Constructors with parameters" section, has an elegant solution. Listing 9-9 illustrates the idea.

Listing 9-9: Using Java's `this` Keyword

```
package com.allmycode.andy;

public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  BagOfCheese() {
  }

  public BagOfCheese(String kind, double weight,
                     int daysAged, boolean isDomestic) {
    super();
    this.kind = kind;
    this.weight = weight;
    this.daysAged = daysAged;
    this.isDomestic = isDomestic;
  }
}
```

To use the class in Listing 9-9, you can run the `CreateBags` code in Listing 9-7. When you do, you see the run shown earlier, in Figure 9-8.

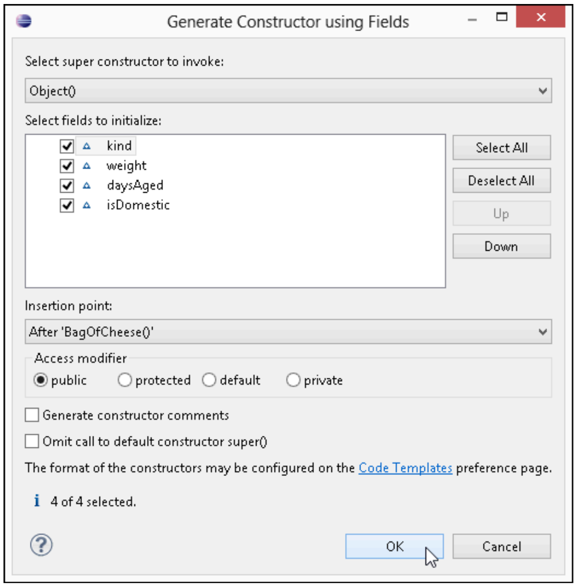
You can persuade Eclipse to create the oversized constructor that you see in Listing 9-9. Here's how:

1. **Start with the code from Listing 9-1 (or Listing 9-3) in the Eclipse editor.**
2. **Click the mouse cursor anywhere inside the editor.**
3. **On the Eclipse main menu, select Source➪ Generate Constructor Using Fields.**

   The Generate Constructor Using Fields dialog box in Eclipse appears, as shown in Figure 9-12.
4. **In the Select Fields to Initialize pane in the dialog box, make sure that all four of the `BagOfCheese` fields are selected.**

   Doing so ensures that the new constructor will have a parameter for each of the class's fields.
5. **Click OK.**

   That does it! Eclipse dismisses the dialog box and adds a freshly brewed constructor to the editor's code.


Generate Constructor using Fields

Select super constructor to invoke:

Object()

Select fields to initialize:

- [x] kind
- [x] weight
- [x] daysAged
- [x] isDomestic

Insertion point:

After 'BagOfCheese()'

Access modifier

(● public) ○ protected ○ default ○ private

- [ ] Generate constructor comments
- [ ] Omit call to default constructor super()

The format of the constructors may be configured on the Code Templates preference page.

4 of 4 selected.

Figure 9-12: The Generate Constructor Using Fields dialog box.

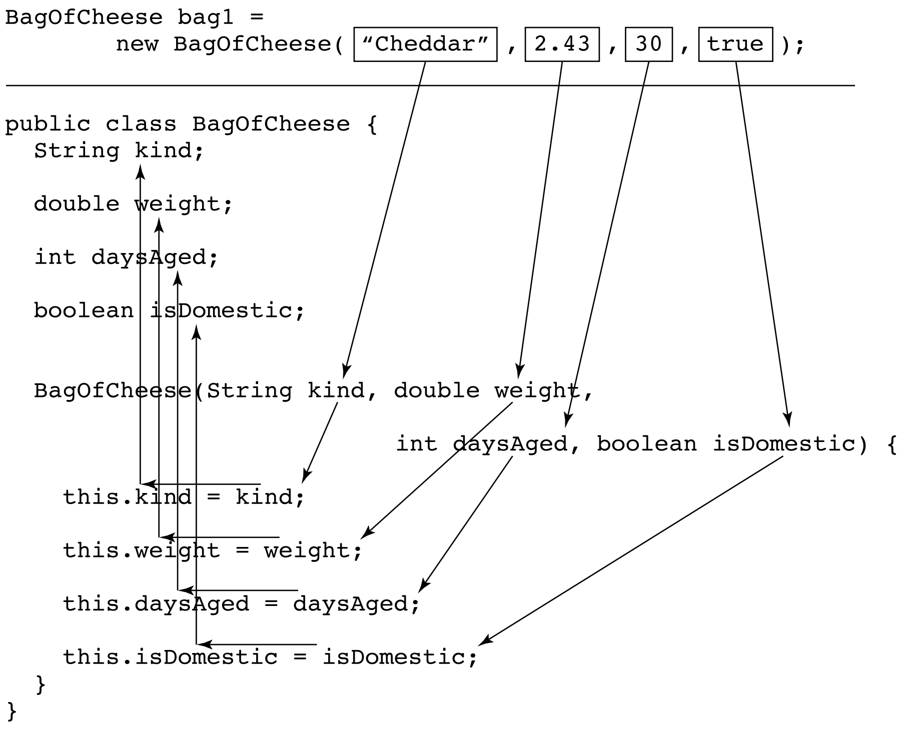
Java's `this` keyword refers to "the object that contains the current line of code." So in Listing 9-9, the word `this` refers to an instance of `BagOfCheese` (that is, to the object that's being constructed). That object has a `kind` field, so `this.kind` refers to the first of the object's four fields (and not to the constructor's `kind` parameter). That object also has `weight`, `daysAged`, and `isDomestic` fields, so `this.weight`, `this.daysAged`, and `this.isDomestic` refer to that object's fields, as shown in Figure 9-13. And the assignment statements inside the constructor give values to the new object's fields.

Listing 9-9 contains the call `super()`. To find out what `super()` means, see Chapter 10.

Giving an object more responsibility

You have a printer and you try to install it on your computer. It's a capable printer, but it didn't come with your computer, so your computer needs a program to *drive* the printer: a printer *driver*. Without a driver, your new printer is nothing but a giant paperweight.

But, sometimes, finding a device driver can be a pain in the neck. Maybe you can't find the disk that came with the printer. (That's always my problem.)



```
BagOfCheese bag1 =
       new BagOfCheese( "Cheddar" , 2.43 , 30 , true );

public class BagOfCheese {
  String kind;

  double weight;

  int daysAged;

  boolean isDomestic;


  BagOfCheese(String kind, double weight,

                         int daysAged, boolean isDomestic) {

    this.kind = kind;

    this.weight = weight;

    this.daysAged = daysAged;

    this.isDomestic = isDomestic;
  }
}
```

Figure 9-13: Assigning values to an object's fields.

I have one off-brand printer whose driver is built into its permanent memory. When I plug the printer into a USB port, the computer displays a new storage location. (The location looks, to ordinary users, like another of the computer's disks.) The drivers for the printer are stored directly on the printer's internal memory. It's as though the printer knows how to drive itself!

Now consider the code in Listings 9-7 and 9-8. You're the `CreateBags` class (refer to Listing 9-7), and you have a new gadget to play with — the `BagOfCheese` class in Listing 9-8. You want to display the properties of a particular bag, and you don't enjoy reinventing the wheel. That is, you don't like declaring your own `displayBag` method (the way you do in Listing 9-7). You'd rather have the `BagOfCheese` class come with its own `displayBag` method.

Here's the plan: Move the `displayBag` method from the `CreateBags` class to the `BagOfCheese` class. That is, make each `BagOfCheese` object be responsible for displaying itself. With the Andy's Cheese Emporium metaphor that starts this chapter, each bag's form has its own Display button, as shown in Figure 9-14.

The interesting characteristic of a Display button is that when you press it, the message you see depends on the bag of cheese you're examining. More precisely, the message you see depends on the values in that particular form's fields.

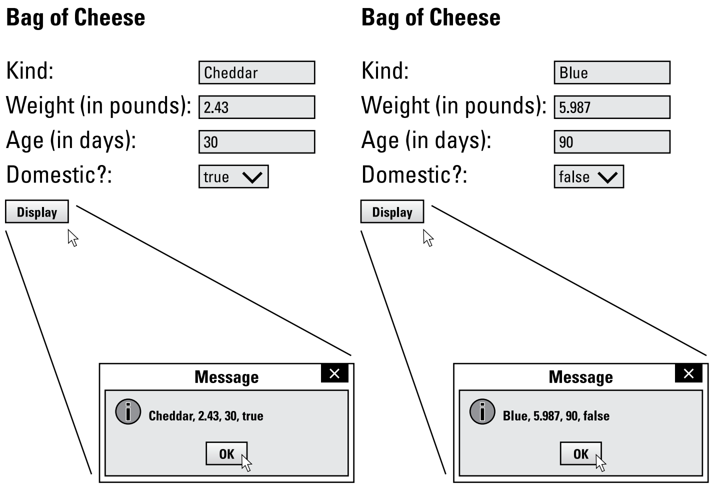


Figure 9-14: Two bag objects and two displays.

The same thing happens in Listing 9-11 when you call `bag1.displayBag()`. Java runs the `displayBag` method shown in Listing 9-10. The values used in that method call — `kind`, `weight`, `daysAged`, and `isDomestic` — are the values in the `bag1` object's fields. Similarly, the values used when you call `bag2.displayBag()` are the values in the `bag2` object's fields.

Listing 9-10: A Self-Displaying Class

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  BagOfCheese() {
  }

  public BagOfCheese(String kind, double weight,
```

(continued)

Listing 9-10 *(continued)*

```
                      int daysAged, boolean isDomestic) {

    super();
    this.kind = kind;
    this.weight = weight;
    this.daysAged = daysAged;
    this.isDomestic = isDomestic;
  }

  public void displayBag() {
    JOptionPane.showMessageDialog(null,
        kind + ", " +
        weight + ", " +
        daysAged + ", " +
        isDomestic);
  }
}
```

Listing 9-11: Having a Bag Display Itself

```
package com.allmycode.andy;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag1 =
        new BagOfCheese("Cheddar", 2.43, 30, true);
    BagOfCheese bag2 =
        new BagOfCheese("Blue", 5.987, 90, false);

    bag1.displayBag();

    bag2.displayBag();
  }
}
```

In Listing 9-10, the `BagOfCheese` object has its own, parameterless `displayBag` method. And in Listing 9-11, the following two lines make two calls to the `displayBag` method — one call for `bag1` and another call for `bag2`:

```
    bag1.displayBag();

    bag2.displayBag();
```

A call to `displayBag` behaves differently depending on the particular bag that's being displayed. When you call `bag1.displayBag()`, you see the field values for `bag1`, and when you call `bag2.displayBag()`, you see the field values for `bag2`.

To call one of an object's methods, follow a reference to the object with a dot and then the method's name.

Members of a class

Notice the similarity between fields and methods:

- As I say earlier in this chapter, in the "Creating objects" section:

  To refer to one of an object's fields, follow a reference to the object with a dot and then the field's name.

- As I say earlier in this chapter, in the "Giving an object more responsibility" section:

  To call one of an object's methods, follow a reference to the object with a dot and then the method's name.

The similarity between fields and methods stretches far and wide in object-oriented programming. The similarity is so strong that special terminology is necessary to describe it. In addition to each `BagOfCheese` object having its own values for the four fields, you can think of each object as having its own copy of the `displayBag` method. So the `BagOfCheese` class in Listing 9-10 has five *members*. Four of the members are the fields `kind`, `weight`, `daysAged`, and `isDomestic`, and the remaining member is the `displayBag` method.

Reference types

Here's a near-quotation from the earlier section "Creating objects:"

> *In Listing 9-2, the initialization of* `bag1` *makes the* `bag1` *variable refer to the newly created bag.*

In the quotation, I choose my words carefully. "The initialization makes the `bag1` variable *refer to* the newly created bag." Notice how I italicize the words *refer to*. A variable of type `int` *stores* an `int` value, but the `bag1` variable in Listing 9-2 *refers to* an object.

What's the difference? The difference is similar to holding an object in your hand versus pointing to it in the room. Figure 9-15 shows you what I mean.

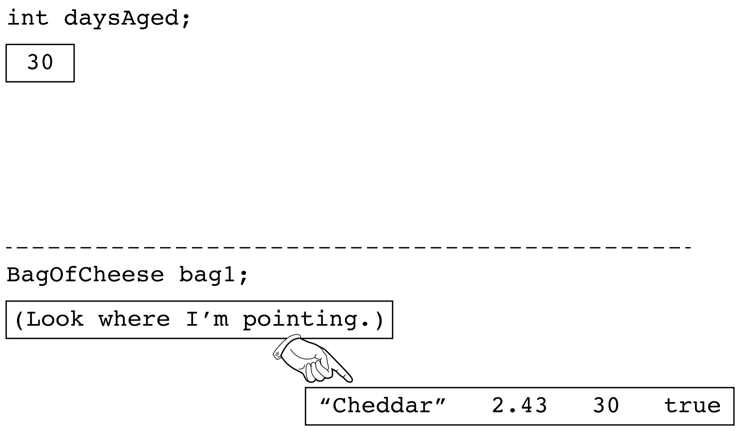


Figure 9-15: Primitive types versus reference types.

Java has two kinds of types: primitive types and reference types.

- I cover primitive types in Chapter 6. Java's eight primitive types are `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, and `float`.
- A *reference type* is the name of a class or (as you see in Chapter 10) an interface.

In Figure 9-15, the variable `daysAged` contains the value `30` (indicating that the cheese in a particular bag has been aged for 30 days). I imagine the value `30` being right inside the `daysAged` box because the `daysAged` variable has type `int` — a primitive type.

But the variable `bag1` has type `BagOfCheese`, and `BagOfCheese` isn't a primitive type. (I know of no computer programming language in which a bag of cheese is a built-in, primitive type!) So the `bag1` variable doesn't contain `"Cheddar" 2.43 30 true`. Instead, the variable `bag1` contains the information required to locate the `"Cheddar" 2.43 30 true` object. The variable `bag1` stores information that *refers to* the `"Cheddar" 2.43 30 true` object.

The types `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, and `float` are primitive types. A primitive type variable (`int daysAged`, `double weight`, `boolean`, and `isDomestic`, for example) stores a value. In contrast, a class is a reference type, such as `String`, which is defined in Java's API, and `BagOfCheese`, which you or I declare ourselves. A reference type variable (`BagOfCheese bag` and `String kind`, for example) *refers* to an object.

Figure 9-15 would be slightly more accurate (but a bit more complicated) if the bottommost box contained a picture of a hand followed by the values 2.43 30 true. The hand would point outside of the box to the string "Cheddar".

In this section, I say that the `bag1` variable *refers to* the `"Cheddar" 2.43 30 true` object. It's also common to say that the `bag1` variable *points to* the `"Cheddar" 2.43 30 true` object. Alternatively, you can say that the `bag1` variable stores the number of the memory address where the `"Cheddar" 2.43 30 true` object's values begin. Neither the pointing language nor the memory language expresses the truth of the matter, but if the rough terminology helps you understand what's going on, there's no harm in using it.

Pass by reference

In the previous section, I emphasize that classes are reference types. A variable whose type is a class contains something that refers to blah, blah, blah. You might ask, "Why should I care?"

Look at Listing 7-4, over in Chapter 7, and notice the result of passing a primitive type to a method:

When the method's body changes the parameter's value, the change has no effect on the value of the variable in the method call.

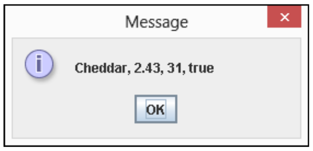
This principle holds true for reference types as well. But in the case of a reference type, the value that's passed is the information about where to find an object, not the object itself. When you pass a reference type in a method's parameter list, you can change values in the object's fields.

See, for example, the code in Listing 9-12.

Listing 9-12: Another Day Goes By

```
package com.allmycode.andy;

public class CreateBags {
  public static void main(String[] args) {
    BagOfCheese bag1 =
        new BagOfCheese("Cheddar", 2.43, 30, true);

    addOneDay(bag1);

    bag1.displayBag();
  }

  static void addOneDay(BagOfCheese bag) {
    bag.daysAged++;
  }
}
```

A run of the code in Listing 9-12 is shown in Figure 9-16. In that run, the constructor creates a bag that is aged 30 days, but the `addOneDay` method successfully adds a day. In the end, the display in Figure 9-16 shows 31 days aged.



```
Cheddar, 2.43, 31, true
```

Figure 9-16: Thirty-one days old.

Unlike the story with `int` values, you can change a bag of cheese's `daysAged` value by passing the bag as a method parameter. Why does it work this way?

When you call a method, you make a copy of each parameter's value in the call. You initialize the declaration's parameters with the copied values. Immediately after making the `addOneDay` call in Listing 9-12, you have two variables: the original `bag1` variable in the `main` method and the new `bag` variable in the `addOneDay` method. The new `bag` variable has a copy of the value from the `main` method, as shown in Figure 9-17. That "value" from the `main` method is a reference to a `BagOfCheese` object. In other words, the `bag1` and `bag` variables refer to the same object.

The statement in the body of the `addOneDay` method adds `1` to the value stored in the object's `daysAged` field. After one day is added, the program's variables look like the information in Figure 9-18.

Notice how both the `bag1` and `bag` variables refer to an object whose `daysAged` value is 31. After returning from the call to `addOneDay`, the `bag` variable goes away. All that remains is the original `main` method and its `bag1` variable, as shown in Figure 9-19. But `bag1` still refers to an object whose `daysAged` value has been changed to 31.

In Chapter 7, I show you how to pass primitive values to method parameters. Passing a primitive value to a method parameter is called *pass-by value*. In this section, I show you how to pass both primitive values and objects to method parameters. Passing an object (such as `bag1`) to a method parameter is called *pass-by reference*.

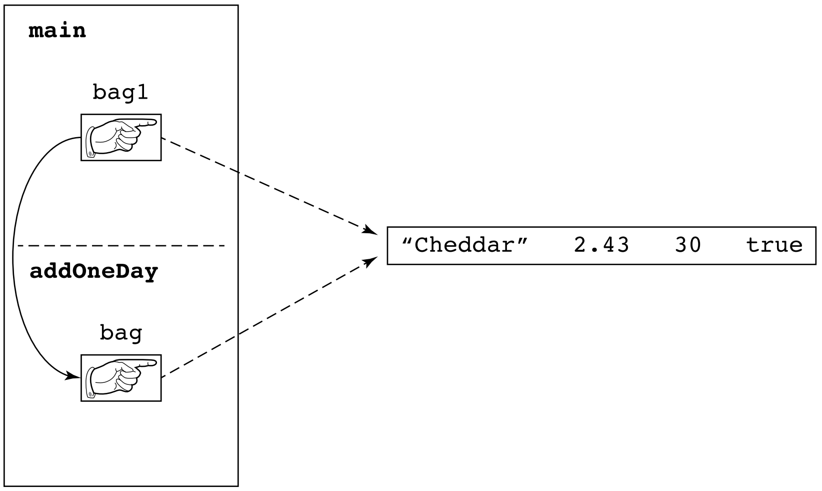


Figure 9-17: Java copies a pointer.

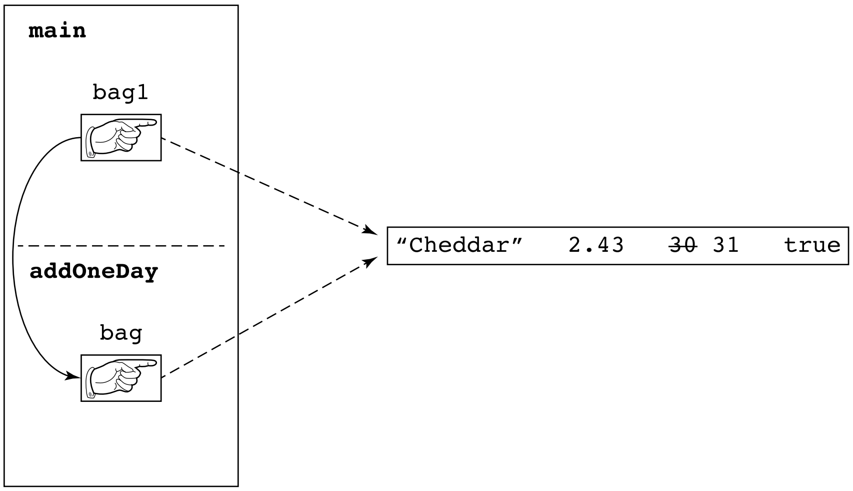


Figure 9-18: Java adds 1 to `daysAged`.

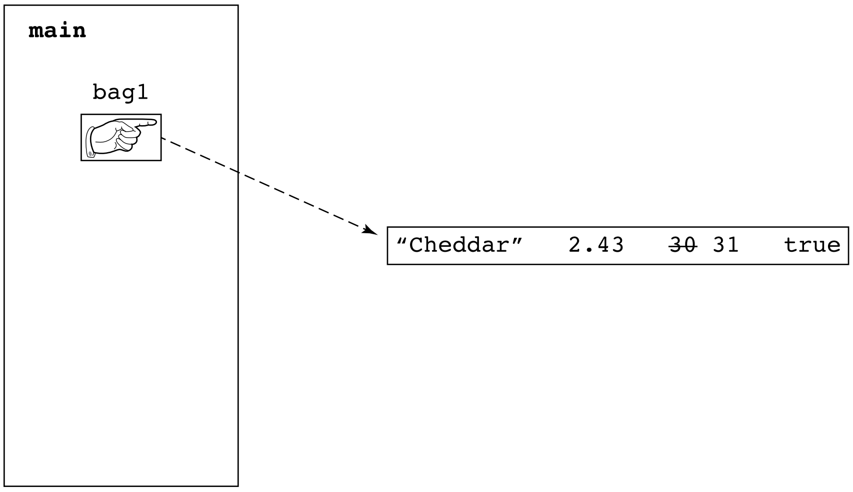


Figure 9-19: The original bag is aged 31 days.

Java's Modifiers

Throughout this book, you see words like `static` and `public` peppered throughout the code listings. You might wonder what these words mean. (Actually, if you're reading from front to back, you might have grown accustomed to seeing them and started thinking of them as background noise.) In the next few sections, I tackle some of these *modifier* keywords.

Public classes and default-access classes

Most of the classes in this chapter's listings begin with the word `public`. When a class is public, any program in any package can use the code (or at least some of the code) inside that class. If a class isn't public, then for a program to use the code inside that class, the program must be inside the same package as the class. Listings 9-13, 9-14, and 9-15 illustrate these ideas.

Listing 9-13: What Is a Paragraph?

```
package org.allyourcode.wordprocessor;

class Paragraph {
  int alignment;
  int borders;
  double leftIndent;
  double lineSpacing;
  int style;
}
```

Listing 9-14: Making a Paragraph with Code in the Same Package

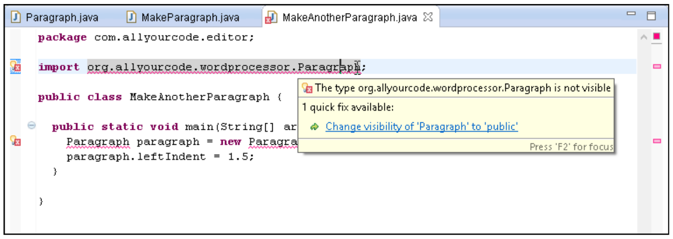
```
package org.allyourcode.wordprocessor;

class MakeParagraph {

  public static void main(String[] args) {
    Paragraph paragraph = new Paragraph();
    paragraph.leftIndent = 1.5;
  }

}
```

Listing 9-15: Making a Paragraph with Code in Another Package

```
package com.allyourcode.editor;

import org.allyourcode.wordprocessor.Paragraph;

public class MakeAnotherParagraph {

  public static void main(String[] args) {
    Paragraph paragraph = new Paragraph();
    paragraph.leftIndent = 1.5;
  }

}
```

The `Paragraph` class in Listing 9-13 has *default access* — that is, the `Paragraph` class isn't public. The code in Listing 9-14 is in the same package as the `Paragraph` class (the `org.allyourcode.wordprocessor` package). So In Listing 9-14, you can declare an object to be of type `Paragraph`, and you can refer to that object's `leftIndent` field.

The code in Listing 9-15 isn't in the same org.allyourcode.wordprocessor package. For that reason, the use of names like `Paragraph` and `leftIndent` (from Listing 9-13) aren't legal in Listing 9-15, even if Listings 9-13 and 9-15 are in the same Eclipse project. When you type Listings 9-13, 9-14, and 9-15 into the Eclipse editor, you see a red, blotchy mess for Listing 9-15, as shown in Figure 9-20.



```
package com.allyourcode.editor;

import org.allyourcode.wordprocessor.Paragraph;

public class MakeAnotherParagraph {

  public static void main(String[] ar
    Paragraph paragraph = new Paragra
    paragraph.leftIndent = 1.5;
  }

}
```

The type org.allyourcode.wordprocessor.Paragraph is not visible

1 quick fix available:

Change visibility of 'Paragraph' to 'public'

Press 'F2' for focus

Figure 9-20: Errors in Listing 9-15.

An Android activity can invoke the code from another package (that is, another Android app). To do this, you don't use names from the other package in your activity's code. For details, see the discussion of `start Activity` in Chapter 12.

The `.java` file containing a public class must have the same name as the public class, so the file containing the code in Listing 9-1 must be named `BagOfCheese.java`.

Even the capitalization of the filename must be the same as the public class's name. You see an error message if you put the code in Listing 9-1 inside a file named `bagofcheese.java`. In the file's name, you have to capitalize the letters `B`, `O`, and `C`.

Because of the file-naming rule, you can't declare more than one public class in a `.java` file. If you put the public classes from Listings 9-1 and 9-2 into the same file, would you name the file `BagOfCheese.java` or `CreateBags.java`? Neither name would satisfy the file-naming rule. For that matter, *no* name would satisfy it.

It's customary to declare a class containing a `main` method to be public. I sometimes ignore this convention, but when I do, the code looks strange to me later. Once, I faced a situation in which a Java class had to be public simply because that class contained a `main` method. I promised myself that I'd use this example in my writing later, but since then I haven't been able to remember the situation. Oh, well!

Access for fields and methods

A class can have either public access or nonpublic (default) access. But a member of a class has four possibilities: public, private, default, and protected.

A class's fields and methods are the class's *members*. For example, the class in Listing 9-10 has five members: the fields `kind`, `weight`, `daysAged`, and `isDomestic` and the method `displayBag`.

Here's how member access works:

- 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.
- A private member of a class cannot be used in any code outside the class.

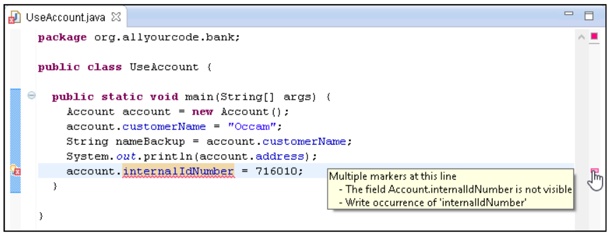
- A public member of a class can be used wherever the class itself can be used; that is:
    - Any program in any package can refer to a public member of a public class.
    - For a program to reference a public member of a default access class, the program must be inside the same package as the class.

To see these rules in action, check out the public class in Listing 9-16.

Listing 9-16: A Class with Public Access

```
package org.allyourcode.bank;

public class Account {
  public String customerName;
  private int internalIdNumber;
  String address;
  String phone;
  public int socialSecurityNumber;
  int accountType;
  double balance;

  public static int findById(int internalIdNumber) {
    Account foundAccount = new Account();
    // Code to find the account goes here.
    return foundAccount.internalIdNumber;
  }
}
```

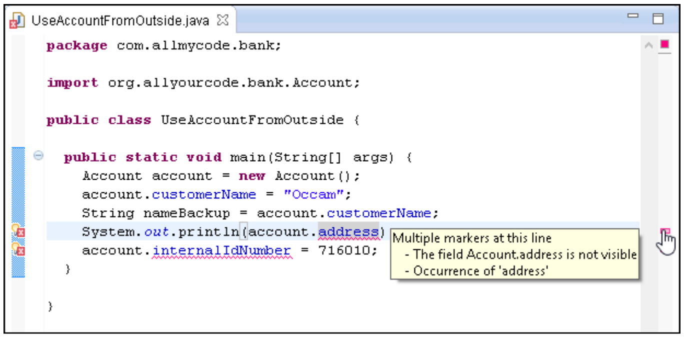
The code in Figures 9-21 and 9-22 uses the `Account` class and its fields.



```
package org.allyourcode.bank;

public class UseAccount {

  public static void main(String[] args) {
    Account account = new Account();
    account.customerName = "Occam";
    String nameBackup = account.customerName;
    System.out.println(account.address);
    account.internalIdNumber = 716010;
  }

}
```

```
Multiple markers at this line
  - The field Account.internalIdNumber is not visible
  - Write occurrence of 'internalIdNumber'
```

Figure 9-21: Referring to a public class in the same package.



UseAccountFromOutside.java

```
package com.allmycode.bank;

import org.allyourcode.bank.Account;

public class UseAccountFromOutside {

  public static void main(String[] args) {
    Account account = new Account();
    account.customerName = "Occam";
    String nameBackup = account.customerName;
    System.out.println(account.address)
    account.internalIdNumber = 716010;
  }

}
```

```
Multiple markers at this line
  - The field Account.address is not visible
  - Occurrence of 'address'
```

Figure 9-22: Referring to a public class in a different package.

In Figures 9-21 and 9-22, notice that

- The `UseAccount` class is in the same package as the `Account` class.
- The `UseAccount` class can create a variable of type `Account`.
- The `UseAccount` class's code can refer to the public `customerName` field of the `Account` class and to the default `address` field of the `Account` class.
- The `UseAccount` class cannot refer to the private `internalIdNumber` field of the `Account` class, even though `UseAccount` and `Account` are in the same package.
- The `UseAccountFromOutside` class is not in the same package as the `Account` class.
- The `UseAccountFromOutside` class can create a variable of type `Account`. (An `import` declaration keeps me from having to repeat the fully qualified `org.allyourcode.bank.Account` name everywhere in the code.)
- The `UseAccountFromOutside` class's code can refer to the public `customerName` field of the `Account` class.
- The `UseAccountFromOutside` class's code cannot refer to the default `address` field of the `Account` class or to the private `internalIdNum-ber` field of the `Account` class.

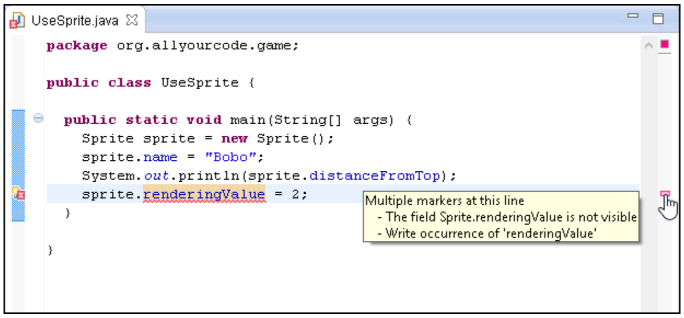
Now examine the nonpublic class in Listing 9-17.

Listing 9-17: A Class with Default Access

```
package org.allyourcode.game;

class Sprite {
  public String name;
```

```
    String image;
    double distanceFromLeftEdge, distanceFromTop;
    double motionAcross, motionDown;
    private int renderingMethod;

    void render() {
      if (renderingMethod == 2) {
        // Do stuff here
      }
    }
}
```

The code in Figures 9-23 and 9-24 uses the `Sprite` class and its fields.

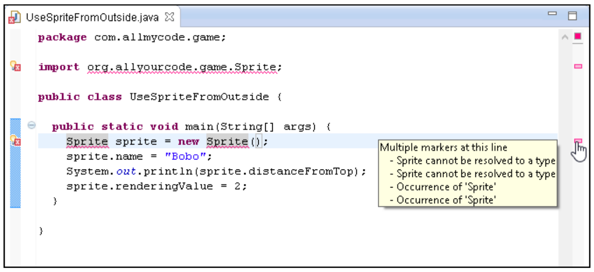


UseSprite.java

```
package org.allyourcode.game;

public class UseSprite {

  public static void main(String[] args) {
    Sprite sprite = new Sprite();
    sprite.name = "Bobo";
    System.out.println(sprite.distanceFromTop);
    sprite.renderingValue = 2;
  }

}
```

```
Multiple markers at this line
  - The field Sprite.renderingValue is not visible
  - Write occurrence of 'renderingValue'
```

Figure 9-23: Referring to a default access class in the same package.



UseSpriteFromOutside.java

```
package com.allmycode.game;

import org.allyourcode.game.Sprite;

public class UseSpriteFromOutside {

  public static void main(String[] args) {
    Sprite sprite = new Sprite();
    sprite.name = "Bobo";
    System.out.println(sprite.distanceFromTop);
    sprite.renderingValue = 2;
  }

}
```

```
Multiple markers at this line
  - Sprite cannot be resolved to a type
  - Sprite cannot be resolved to a type
  - Occurrence of 'Sprite'
  - Occurrence of 'Sprite'
```

Figure 9-24: Referring to a default access class in a different package.

In Figures 9-23 and 9-24, notice that

- The `UseSprite` class is in the same package as the `Sprite` class.
- The `UseSprite` class can create a variable of type `Sprite`.
- The `UseSprite` class's code can refer to the public `name` field of the `Sprite` class and to the default `distanceFromTop` field of the `Sprite` class.

- The UseSprite class cannot refer to the private renderingValue field of the Sprite class, even though UseSprite and Sprite are in the same package.
- The UseSpriteFromOutside class isn't in the same package as the Sprite class.
- The UseSpriteFromOutside class cannot create a variable of type Sprite. (Not even an import declaration can save you from an error message here.)
- Inside the UseAccountFromOutside class, references to sprite.name, sprite.distanceFromTop, and sprite.renderingValue are all meaningless because the sprite variable doesn't have a type.

Using getters and setters

In Figures 9-21 and 9-22, the UseAccount and UseAccountFromOutside classes can set an account's customerName and get the account's existing customerName:

```
account.customerName = "Occam";
String nameBackup = account.customerName;
```

But neither the UseAccount class nor the UseAccountFromOutside class can tinker with an account's internalIdNumber field.

What if you want a class like UseAccount to be able to get an existing account's internalIdNumber but not to change an account's internalIdNumber? (In many situations, getting information is necessary, but changing existing information is dangerous.) You can do all this with a *getter* method, as shown in Listing 9-18.

Listing 9-18: Creating a Read-Only Field

```
package org.allyourcode.bank;

public class Account {
  public String customerName;
  private int internalIdNumber;
  String address;
  String phone;
  public int socialSecurityNumber;
  int accountType;
  double balance;

  public static int findById(int internalIdNumber) {
    Account foundAccount = new Account();
    // Code to find the account goes here.
    return foundAccount.internalIdNumber;
  }

  public int getInternalIdNumber() {
    return internalIdNumber;
  }
}
```

With the `Account` class in Listing 9-18, another class's code can call

```
System.out.println(account.getInternalIdNumber());
```

or

```
int backupIdNumber = account.getInternalIdNumber();
```

The `Account` class's `internalIdNumber` field is still private, so another class's code has no way to assign a value to an account's `internalId Number` field. To enable other classes to change an account's private `internalIdNumber` value, you can add a setter method to the code in Listing 9-18, like this:

```
public void setInternalIdNumber(int internalIdNumber) {
  this.internalIdNumber = internalIdNumber;
}
```

Getter and setter methods aren't built-in features in Java — they're simply ordinary Java methods. But this pattern (having a method whose purpose is to access an otherwise inaccessible field's value) is used so often that programmers use the terms *getter* and *setter* to describe it.

Getter and setter methods are *accessor* methods. Java programmers almost always follow the convention of starting an accessor method name with `get` or `set` and then capitalizing the name of the field being accessed. For example, the field `internalIdNumber` has accessors named `getInternal IdNumber` and `setInternalIdNumber`. The field `renderingValue` has accessors named `getRenderingValue` and `setRenderingValue`.

You can have Eclipse create getters and setters for you. Here's how:

1. **Start with the code from Listing 9-16 in the Eclipse editor.**
2. **Click the mouse cursor anywhere inside the editor.**
3. **On the Eclipse main menu, select Source➪Generate Getters and Setters.**

The Generate Getters and Setters dialog box in Eclipse appears, as shown in Figure 9-25.

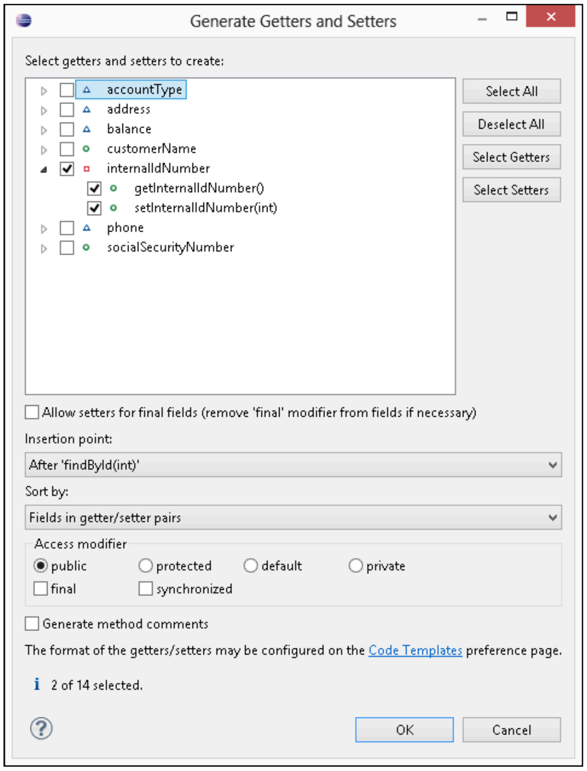


Figure 9-25: The Generate Getters and Setters dialog box.

4. **In the Select Getters and Setters to Create pane in the dialog box, expand the** `internalIdNumber` **branch.**
5. **Within the** `internalIdNumber` **branch, select either or both of the getInternalIdNumber() or setInternalIdNumber(int) check boxes.**

   Eclipse creates only the getters and setters whose check boxes you select.
6. **Click OK.**

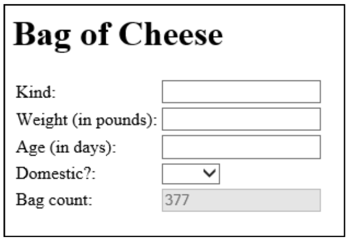
   Eclipse dismisses the dialog box and adds freshly brewed getter and setter methods to the editor's code.

I cover protected access in Chapter 10.

What does static mean?

This chapter begins with a discussion of cheese and its effects on Andy's business practices. Andy has a blank form that represents a class. He also has a bunch of filled-in forms, each of which represents an individual bag-of-cheese object.

One day, Andy decides to take inventory of his cheese by counting all the bags of cheese (see Figure 9-26).



Bag of Cheese

Kind:	
Weight (in pounds):	
Age (in days):	
Domestic?:	
Bag count:	377

Figure 9-26: Counting bags of cheese.

Compare the various fields shown in Figure 9-27. From the object-oriented point of view, how is the `daysAged` field so different from the `count` field?

The answer is that a single bag can keep track of how many days it has been aged, but it shouldn't count *all* the bags. As far back as Listing 9-1, a `BagOfCheese` object has its own `daysAged` field. That makes sense. (Well, it makes sense to an object-oriented programmer.)

But giving a particular object the responsibility of counting all objects in its class doesn't seem fair. To have each `BagOfCheese` object speak on behalf of all the others violates a prime directive of computer programming: The structure of the program should imitate the structure of the real-life data. For example, I can post a picture of myself on Facebook, but I can't promise to count everyone else's pictures on Facebook. ("All you other Facebook users, count your own @#!% pictures!")

A field to count all bags of cheese belongs in one central place. That's why, in Figure 9-27, I have one, and only one, `count` field. Each object has its own `daysAged` value, but only the class itself has a `count` value.

A field or method that belongs to an entire class rather than to each individual object is a *static* member of the class. To declare a static member of a class, you use Java's `static` keyword (what a surprise!), as shown in Listing 9-19.

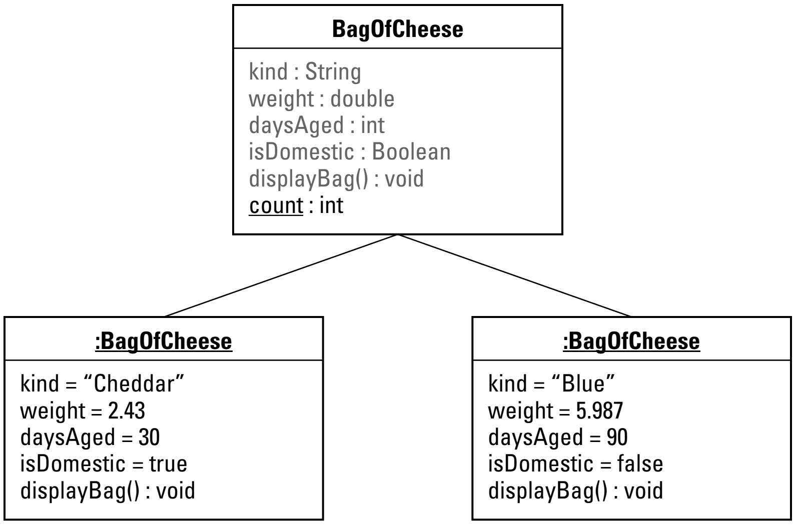


Figure 9-27: The UML diagram has only one `count` variable.

Listing 9-19: Creating a Static Field

```
package com.allmycode.andy;

class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;

  static int count = 0;

  public BagOfCheese() {
    count++;
  }
}
```

To refer to a class's static member, you preface the member's name with the name of the class, as shown in Listing 9-20.

Listing 9-20: Referring to a Static Field

```
package com.allmycode.andy;

import javax.swing.JOptionPane;

public class CreateBags {

  public static void main(String[] args) {
    new BagOfCheese();
    new BagOfCheese();
    new BagOfCheese();
    JOptionPane.showMessageDialog
            (null, BagOfCheese.count);
  }

}
```

Knowing when to create a static member

In many situations, you declare an element to be static in order to mirror the structure of real-life data — but sometimes you declare it to be static for technical reasons. For example, a program's `main` method has to be static in order to provide the Java virtual machine with easy access to the method.

Listing 9-21 is a copy of an example from Chapter 7. In the listing, the `main` method has to be static. I've learned to live with that fact.

Listing 9-21: Declaring and Calling a Static Method

```
import javax.swing.JOptionPane;

public class Scorekeeper {

  public static void main(String[] args) {
    int score = 50000;
    int points = 1000;
    score = addPoints(score, points);
    JOptionPane.showMessageDialog(null, score,
        "New Score", JOptionPane.INFORMATION_MESSAGE);
  }

  static int addPoints(int score, int points) {
    return score + points;
  }

}
```

But what about the `addPoints` method in Listing 9-21? Why is the `addPoints` method static? If you remove the word `static` from the `addPoints` method's declaration, you get this ferocious-looking error: `Cannot make a static reference to non-static method`. What gives?

To understand what's going on, consider the three ways to refer to a member (a field or a method):

- **You can preface the member name with a name that refers to an object.**

  For example, in Listing 9-11, I preface calls to `displayBag` with the names `bag1` and `bag2`, each of which refers to an object:

  ```
  bag1.displayBag();
  bag2.displayBag();
  ```

  When you do this, you're referring to something that belongs to each individual object. (You're referring to the object's nonstatic field, or calling the object's nonstatic method.)

- **You can preface the member name with a name that refers to a class.**

  For example, in Listing 9-20, I prefaced the field name `count` with the class name `BagOfCheese`.

  When you do this, you're referring to something that belongs to the entire class. (You're referring to the class's static field, or calling the class's static method.)

- **You can preface the member name with nothing.**

  For example, in Listing 9-10, inside the `displayBag` method, I use the names `kind`, `weight`, `daysAged`, and `isDomestic` with no dots in front of them:

  ```
  public void displayBag() {
    JOptionPane.showMessageDialog(null,
        kind + ", " +
        weight + ", " +
        daysAged + ", " +
        isDomestic);
  }
  ```

  In Listing 9-21, I preface the static method name `addPoints` with no dots in front of the name:

  ```
  score = addPoints(score, points);
  ```

  When you do this, you're referring to either a nonstatic member belonging to a particular object or to a static member belonging to a particular class. It all depends on the location of the code containing the member name, as described in this list:

  - If the code is inside a nonstatic method, the name refers to an element belonging to an object. That is, the name refers to an object's nonstatic field or method.

    For example, in Listing 9-10, the following code snippet is in the non-static `displayBag` method:

  ```
  kind + ", " +
  weight + ", " +
  daysAged + ", " +
  isDomestic);
  ```

In this context, the names `kind`, `weight`, `daysAged`, and `isDomestic` refer to a particular object's properties.

- If the code is inside a static method, the name refers to something belonging to an entire class. That is, the name refers to a class's static field or method.

  In Listing 9-21, the line

```
score = addPoints(score, points);
```

  is inside the static `main` method, so the name `addPoints` refers to the `Scorekeeper` class's static `addPoints` method.

Java provides a loophole in which you break one of the three rules I just described. You can preface a member name with a name that refers to an object. If the member is static, it's the same as prefacing the member name with the name of a class (whatever class you used when you declared that name).

Consider the code in Listing 9-21. If the `addPoints` method isn't static, each instance of the `Scorekeeper` class has its own `addPoints` method, and each `addPoints` method belongs to an instance of the `Scorekeeper` class. The trouble is that the code in Listing 9-21 doesn't construct any instances of the `Scorekeeper` class. (The listing declares the `Scorekeeper` class itself, but doesn't create any instances.) The listing has no copies of `addPoints` to call. (See Figure 9-28.)Without `addPoints` being static, the statement `score = addPoints(score, points)` is illegal.

Sure, you can call the `Scorekeeper` constructor to create a `Scorekeeper` instance:

```
Scorekeeper keeper = new Scorekeeper();
```

But that doesn't solve the problem. The call to `addPoints` is inside the `main` method, and the `main` method is static. So the `addPoints` call doesn't come from the new `keeper` object, and the call doesn't refer to the `keeper` object's `addPoints` method.

You can fix the problem (of `addPoints` not being static) by using a two-step approach: Create a `Scorekeeper` instance, *and* call the new instance's `addPoints` method, as shown here and in Figure 9-29:

```
Scorekeeper keeper = new Scorekeeper();
keeper.addPoints(score, points);
```

But this approach complicates the example from Chapter 7.

In Listing 9-21, the one and only static `addPoints` method belongs to the entire `Scorekeeper` class, as shown in Figure 9-30. Also, the static `main` method and the call to `addPoints` belong to the entire `Scorekeeper` class, so the `addPoints` call in Listing 9-21 has a natural target, as shown in Figure 9-29.

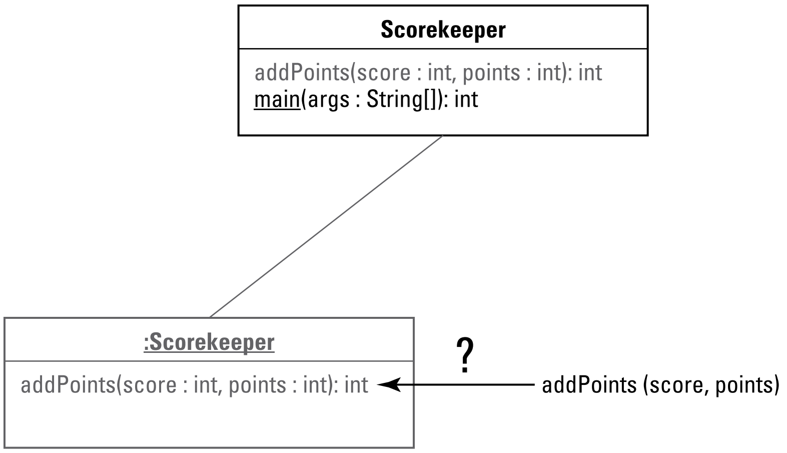

Scorekeeper
addPoints(score : int, points : int): int
main(args : String[]): int
:Scorekeeper
addPoints(score : int, points : int): int
?
addPoints (score, points)

Figure 9-28: Failing when trying to call a nonstatic `addPoints` method.

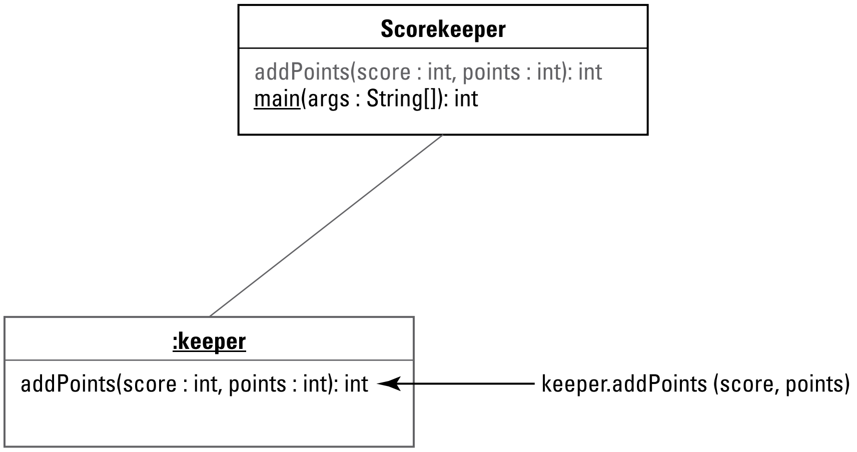

Scorekeeper
addPoints(score : int, points : int): int
main(args : String[]): int
:keeper
addPoints(score : int, points : int): int
keeper.addPoints (score, points)

Figure 9-29: Succeeding in calling a nonstatic `addPoints` method.

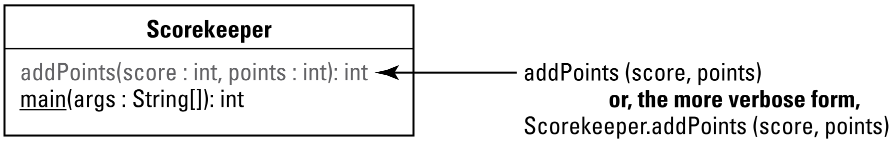

Scorekeeper
addPoints(score : int, points : int): int
main(args : String[]): int
addPoints (score, points)
or, the more verbose form,
Scorekeeper.addPoints (score, points)

Figure 9-30: Calling a static method.

What's Next?

This chapter talks about individual classes. Most classes don't exist in isolation from other classes. Most classes belong to hierarchies of classes, subclasses, and sub-subclasses, so the next chapter covers the relationships among classes.