How to code instance variables

Figure 7-6 shows how to code the instance variables that define the types of data that are used by the objects that are created from a class. When you declare an instance variable, you should use an access modifier to control its accessibility. If you use the private keyword, the instance variable can be used only within the class that defines it. In contrast, if you use the public keyword, the instance variable can be accessed by other classes. You can also use other access modifiers that give you finer control over the accessibility of your instance variables. You'll learn about those modifiers in the chapters that follow.

This figure shows four examples of declaring an instance variable. The first example declares a variable of the double type. The second one declares a variable of the int type. The third one declares a variable that's an object of the String class. And the last one declares an object from the Product class...the class that you're learning how to code right now.

Although instance variables work like regular variables, they must be declared within the class body, but not inside methods or constructors. That way, they'll be available throughout the entire class. In this book, all of the instance variables for a class are declared at the beginning of the class. However, when you read through code from other sources, you may find that the instance variables are declared at the end of the class or at other locations within the class.

Figure 7-6

The syntax for declaring instance variables

```
public|private primitiveType|ClassName variableName;
```

Examples

```
private double price;
private int quantity;
private String code;
private Product product;
```

Where you can declare instance variables

```
public class Product
{
    //common to code instance variables here
    private String code;
    private String description;
    private double price;

    //the constructors and methods of the class
    public Product(){}
    public void setCode(String code){}
    public String getCode(){ return code; }
    public void setDescription(String description){}
    public String getDescription(){ return description; }
    public void setPrice(double price){}
    public double getPrice(){ return price; }
    public String getFormattedPrice(){ return formattedPrice; }

    //also possible to code instance variables here
    private int test;
}
```

Description

- An instance variable may be a primitive data type, an object created from a Java class such as the String class, or an object created from a user-defined class such as the Product class.
- To prevent other classes from accessing instance variables, use the private keyword to declare them as private.
- You can declare the instance variables for a class anywhere outside the constructors and methods of the class.

Figure 7-6 How to code instance variables

How to code constructors

Figure 7-7 shows how to code a constructor for a class. When you code one, it's a good coding practice to assign a value to all of the instance variables of the class as shown in the four examples. You can also include any additional statements that you want to execute within the constructor. For instance, the fourth example ends by calling two different get methods from the current class.

When you code a constructor, you must use the public access modifier and the same name, including capitalization, as the class name. Then, if you don't want to accept arguments, you must code an empty set of parentheses as shown in the first example. On the other hand, if you want to accept arguments, you code the parameters for the constructor as shown in the next three examples. When you code the parameters for a constructor, you must code a data type and a name for each parameter. For the data type, you can code a primitive data type or the class name for any class that defines an object.

The second example shows a constructor with three parameters. Here, the first parameter is a String object named code; the second parameter is a String object named description; and the third parameter is a double type named price. Then, the three statements within the constructor use these parameters to initialize the three instance variables of the class.

In this example, the names of the parameters are the same as the names of the instance variables. As a result, the constructor must distinguish between the two. To do that, it uses the this keyword to refer to the instance variables of the current object. You'll learn about other ways you can use this keyword later in this chapter.

The third example works the same as the second example, but it doesn't need to use the this keyword because the parameter names aren't the same as the names of the instance variables. In this case, though, the parameter names aren't very descriptive. As a result, the code in the second example is easier for other programmers to read than the code in the third example.

The fourth example shows a constructor with one parameter. Here, the first statement assigns the first parameter to the first instance variable of the class. Then, the second statement calls a method of a class named ProductDB to get a Product object for the specified code. Finally, the last two statements call methods of the Product object, and the values returned by these methods are assigned to the second and third instance variables.

When you code a constructor, the class name plus the number of parameters and the data type for each parameter form the *signature* of the constructor. You can code more than one constructor per class as long as each constructor has a unique signature. For example, the first two constructors shown in this figure have different signatures so they could both be coded within the Product class. This is known as *overloading* a constructor.

If you don't code a constructor, Java will create a *default constructor* that doesn't accept any parameters and initializes all instance variables to null, zero, or false. If you code a constructor that accepts parameters, though, Java won't create this default constructor. So if you need a constructor like that, you'll need to code it explicitly. To avoid this confusion, it's a good practice to code all of

The syntax for coding constructors

```
public ClassName([parameterList])
{
    // the statements of the constructor
}
```

Example 1: A constructor that assigns default values

```
public Product()
{
    code = "";
    description = "";
    price = 0.0;
}
```

Example 2: A custom constructor with three parameters

```
public Product(String code, String description, double price)
{
    this.code = code;
    this.description = description;
    this.price = price;
}
```

Example 3: Another way to code the constructor shown above

```
public Product(String c, String d, double p)
{
    code = c;
    description = d;
    price = p;
}
```

Example 4: A constructor with one parameter

```
public Product(String code)
{
    this.code = code;
    Product p = ProductDB.getProduct(code);
    description = p.getDescription();
    price = p.getPrice();
}
```

Description

- The constructor must use the same name and capitalization as the name of the class.
- If you don't code a constructor, Java will create a *default constructor* that initializes all numeric types to zero, all boolean types to false, and all objects to null.
- To code a constructor that has parameters, code a data type and name for each parameter within the parentheses that follow the class name.
- The name of the class combined with the parameter list forms the *signature* of the constructor. Although you can code more than one constructor per class, each constructor must have a unique signature.
- In the second and fourth examples above, the this keyword is used to refer to an instance variable of the current object.

Figure 7-7 How to code constructors

your own constructors. That way, it's easy to see which constructors are available to a class, and it's easy to check the values that each constructor uses to initialize the instance variables.

How to code methods

Figure 7-8 shows how to code the methods of a class. To start, you code an access modifier. Most of the time, you can use the public keyword to declare the method so it can be used by other classes. However, you can also use the private keyword to hide the method from other classes.

After the access modifier, you code the return type for the method, which refers to the data type that the method returns. After the return type, you code the name of the method followed by a set of parentheses. Within the parentheses, you code the parameter list for the method. Last, you code the opening and closing braces that contain the statements of the method.

Since a method name should describe the action that the method performs, it's a common coding practice to start each method name with a verb. For example, methods that set the value of an instance variable usually begin with *set*. Conversely, methods that return the value of an instance variable usually begin with *get*. These types of methods are typically referred to as *accessors* because they let you access the values of the instance variables. Methods that perform other types of tasks also begin with verbs such as print, save, read, and write.

The first example shows how to code a method that doesn't accept any parameters or return any values. To do that, it uses the void keyword for the return type and it ends with a set of empty parentheses. When this method is called, it prints the instance variables of the Product object to the console, separating each instance variable with a pipe character (|).

The next three examples show how to code methods that return data. To do that, these methods specify a return type, and they include a return statement to return the appropriate variable. When coding a method like this, you must make sure that the return type that you specify matches the data type of the variable that you return. Otherwise, your code won't compile.

In the fourth example, the getFormattedPrice method uses a NumberFormat object to apply standard currency formatting to the double variable named price. This also converts the double variable to a String object. Then, the return statement returns the String object to the calling method.

The fifth and sixth examples show two possible ways to code a set method. In the fifth example, the method accepts a parameter that has the same name as the instance variable. As a result, the assignment statement within this method uses the this keyword to identify the instance variable. In the sixth example, the parameter has a different name than the instance variable. As a result, the assignment statement doesn't need to use the this keyword. Since the parameter name for both examples are descriptive, both of these examples work equally well.

The syntax for coding a method

```
public|private returnType methodName([parameterList])
{
    // the statements of the method
}
```

Example 1: A method that doesn't accept parameters or return data

```
public void printToConsole()
{
    System.out.println(code + "|" + description +  "|" + price);
}
```

Example 2: A get method that returns a string

```
public String getCode()
{
    return code;
}
```

Example 3: A get method that returns a double value

```
public double getPrice()
{
    return price;
}
```

Example 4: A custom get method

```
public String getFormattedPrice()
{
    NumberFormat currency = NumberFormat.getCurrencyInstance();
    return currency.format(price);
}
```

Example 5: A set method

```
public void setCode(String code)
{
    this.code = code;
}
```

Example 6: Another way to code a set method

```
public void setCode(String productCode)
{
    code = productCode;
}
```

Description

- To allow other classes to access a method, use the public keyword. To prevent other classes from accessing a method, use the private keyword.
- To code a method that doesn't return data, use the void keyword for the return type. To code a method that returns data, code a return type in the method declaration and code a return statement in the body of the method.
- When you name a method, you should start each name with a verb. It's a common coding practice to use the verb *set* for methods that set the values of instance variables and to use the verb *get* for methods that return the values of instance variables. These methods are typically referred to as set and get *accessors*.

Figure 7-8 How to code methods

The code for the Product class

Figure 7-5 presents the code for the Product class. This code implements the fields and methods of the class diagram in figure 7-2. In the next six pages, you'll learn the details of writing code like the code shown here. For now, I'll just present a preview of this code so you have a general idea of how it works.

The first three statements in this class are declarations for the fields of the class. The *fields* are the variables or constants that are available to the class and its objects. In this example, all three fields define *instance variables*, which store the data for the code, description, and price variables that apply to each Product object.

After the field declarations, this class declares the *constructor* of the Product class. This constructor creates an instance of the Product class and initializes its instance variables to their default values. As you'll see later in this chapter, you can also code constructors that accept parameters. Then, the constructor can use the parameter values to initialize the instance variables.

Next are the declarations for the methods of the Product class. In this class, the methods provide access to the values stored in the three fields. For each field, a *get method* returns the value stored in the field, while a *set method* assigns a new value to the field. Of these methods, the getFormattedPrice method is the only method that does any work beyond getting or setting the value provided by the instance variable. This method applies the standard currency format to the price variable and returns the resulting string.

Although the Product class includes both a get and a set method for each field, you don't always have to code both of these methods for a field. In particular, it's common to code just a get method for a field so that its value can be retrieved but not changed. This can be referred to as a *read-only field*. Although you can also code just a set method for a field, that's uncommon.

The private and public keywords determine which *members* of a class are available to other classes. Since all of the instance variables of the Product class use the private keyword, they are only available within that class. The constructor and the methods, however, use the public keyword. As a result, they are available to all classes. Keep in mind, though, that you can include both public and private instance variables and methods in any class.

By the way, this class follows the three coding rules that are required for a *JavaBean*. First, it includes a constructor that requires no arguments. Second, all of the instance variables are private. Third, it includes get and set methods for all instance variables that you want to be able to access. As you progress with Java, you'll find many advantages to creating classes that are also JavaBeans. For example, if you develop JavaServer Pages (JSPs) for a web application, you can use special JSP tags to create a JavaBean and to access its get and set methods.

Now that you've seen the code for the Product class, you might want to consider how it uses encapsulation. First, the three fields are hidden from other classes because they're declared with the private keyword. In addition, all of the code contained within the constructor and methods is hidden. Because of that, you can change any of this code without having to change the other classes that use this class.

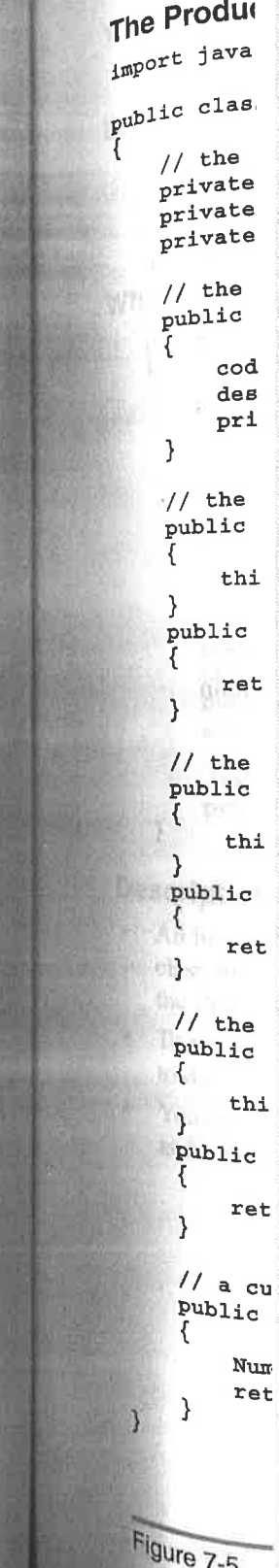
The Produ
import java
public clas
{
// the
private
private
private
// the
public
{
cod
des
pri
}
// the
public
{
thi
}
public
{
ret
}
// the
public
{
thi
}
public
{
ret
}
// the
public
{
thi
}
public
{
ret
}
// a cu
public
{
Num
ret
}
}

Figure 7-5

The Product class

```
import java.text.NumberFormat;

public class Product
{
    // the instance variables
    private String code;
    private String description;
    private double price;

    // the constructor
    public Product()
    {
        code = "";
        description = "";
        price = 0;
    }

    // the set and get methods for the code variable
    public void setCode(String code)
    {
        this.code = code;
    }
    public String getCode()
    {
        return code;
    }

    // the set and get methods for the description variable
    public void setDescription(String description)
    {
        this.description = description;
    }
    public String getDescription()
    {
        return description;
    }

    // the set and get methods for the price variable
    public void setPrice(double price)
    {
        this.price = price;
    }
    public double getPrice()
    {
        return price;
    }

    // a custom get method for the price variable
    public String getFormattedPrice()
    {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        return currency.format(price);
    }
}
```

Figure 7-5 The code for the Product class