CHAPTER

1 Objects and Classes

Main concepts discussed in this chapter:

- objects
- classes
- methods
- parameters

It's time to jump in and get started with our discussion of object-oriented programming. Learning to program requires a mix of some theory and a lot of practice. In this book, we will present both, so that the two reinforce each other.

At the heart of object orientation are two concepts that we have to understand first: *objects* and *classes*. These concepts form the basis of all programming in object-oriented languages. So let us start with a brief discussion of these two foundations.

1.1 Objects and classes

Concept

Java **objects** model objects from a problem domain.

If you write a computer program in an object-oriented language, you are creating, in your computer, a model of some part of the world. The parts that the model is built up from are the *objects* that appear in the problem domain. These objects must be represented in the computer model being created. The objects from the problem domain vary with the program you are writing. They may be words and paragraphs if you are programming a word processor, users and messages if you are working on a social-network system, or monsters if you are writing a computer game.

Objects may be categorized. A class describes,—in an abstract way,—all objects of a particular kind.

We can make these abstract notions clearer by looking at an example. Assume you want to model a traffic simulation. One kind of entity you then have to deal with is cars. What is a car in our context? Is it a class or an object? A few questions may help us to make a decision.

What color is a car? How fast can it go? Where is it right now?

Concept

Objects are created from **classes**. The class describes the kind of object; the objects represent individual instances of the class.

You will notice that we cannot answer these questions until we talk about one specific car. The reason is that the word "car" in this context refers to the *class* car; we are talking about cars in general, not about one particular car.

If I speak of "my old car that is parked at home in my garage," we can answer the questions above. That car is red, it doesn't go very fast, and it is in my garage. Now I am talking about an object—about one particular example of a car.

We usually refer to a particular object as an *instance*. We shall use the term "instance" quite regularly from now on. "Instance" is roughly synonymous with "object"; we refer to objects as instances when we want to emphasize that they are of a particular class (such as "this object is an instance of the class car").

Before we continue this rather theoretical discussion, let us look at an example.

1.2 Creating objects

Start BlueJ and open the example named *figures*.[1] You should see a window similar to that shown in Figure 1.1.

In this window, a diagram should become visible. Every one of the colored rectangles in the diagram represents a class in our project. In this project, we have classes named **`Circle`**, **`Square`**, **`Triangle`**, **`Person`**, and **`Canvas`**.

Right-click on the **`Circle`** class and choose

```
new Circle()
```

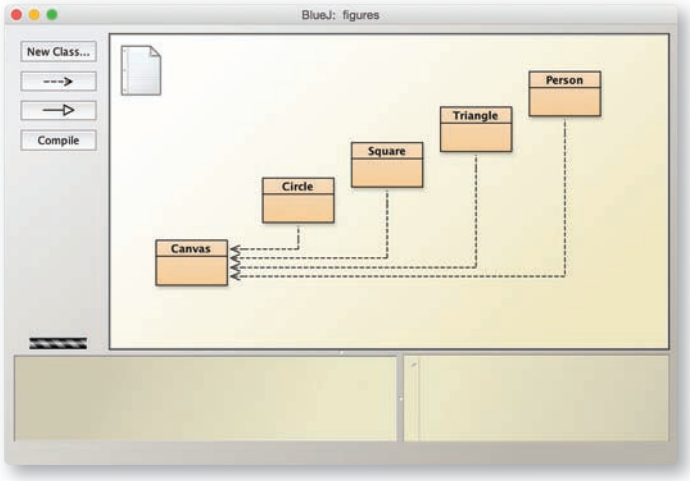

Figure 1.1
The *figures* project in BlueJ

[1] We regularly expect you to undertake some activities and exercises while reading this book. At this point, we assume that you already know how to start BlueJ and open the example projects. If not, read Appendix A first.

Figure 1.2
An object on the object bench

from the pop-up menu. The system asks you for a "name of the instance"; click OK—the default name supplied is good enough for now. You will see a red rectangle toward the bottom of the screen labeled "circle1" (Figure 1.2).

Convention We start names of classes with capital letters (such as `Circle`) and names of objects with lowercase letters (such as `circle1`). This helps to distinguish what we are talking about.

You have just created your first object! "Circle," the rectangular icon in Figure 1.1, represents the class **`Circle`**; **`circle1`** is an object created from this class. The area at the bottom of the screen where the object is shown is called the *object bench.*

Exercise 1.1 Create another circle. Then create a square.

1.3 Calling methods

Right-click on one of the circle objects (not the class!), and you will see a pop-up menu with several operations (Figure 1.3). Choose **`makeVisible`** from the menu—this will draw a representation of this circle in a separate window (Figure 1.4).

Concept

We can communicate with objects by invoking **methods** on them. Objects usually do something if we invoke a method.

You will notice several other operations in the circle's menu. Try invoking **`moveRight`** and **`moveDown`** a few times to move the circle closer to the corner of the screen. You may also like to try **`makeInvisible`** and **`makeVisible`** to hide and show the circle.

Exercise 1.2 What happens if you call **`moveDown`** twice? Or three times? What happens if you call **`makeInvisible`** twice?

The entries in the circle's menu represent operations that you can use to manipulate the circle. These are called *methods* in Java. Using common terminology, we say that these

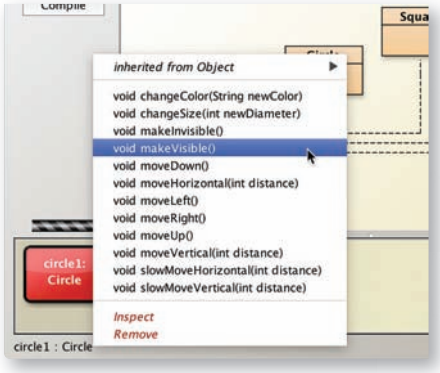

Figure 1.3
An object's pop-up menu, listing its operations

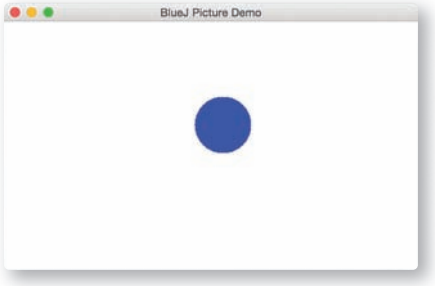

Figure 1.4
A drawing of a circle

methods are *called* or *invoked*. We shall use this proper terminology from now on. We might ask you to "invoke the `moveRight` method of `circle1`."

1.4 Parameters

Concept

Methods can have **parameters** to provide additional information for a task.

Now invoke the `moveHorizontal` method. You will see a dialog appear that prompts you for some input (Figure 1.5). Type in 50 and click OK. You will see the circle move 50 pixels to the right.[2]

The `moveHorizontal` method that was just called is written in such a way that it requires some more information to execute. In this case, the information required is the distance—how far the circle should be moved. Thus, the `moveHorizontal` method is more flexible than the `moveRight` and `moveLeft` methods. The latter always move the circle a fixed distance, whereas `moveHorizontal` lets you specify how far you want to move the circle.

[2] A pixel is a single dot on your screen. Your screen is made up of a grid of single pixels.

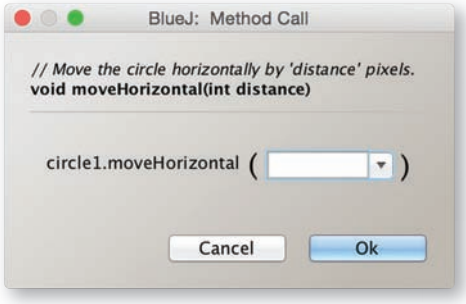

Figure 1.5
A method-call dialog

Exercise 1.3 Try invoking the `moveVertical`, `slowMoveVertical`, and `changeSize` methods before you read on. Find out how you can use `moveHorizontal` to move the circle 70 pixels to the left.

Concept

The method name and the parameter types of a method are called its **signature**. They provide the information needed to invoke that method.

The additional values that some methods require are called *parameters*. A method indicates what kinds of parameters it requires. When calling, for example, the `moveHorizontal` method as shown in Figure 1.5, the dialog displays the line near the top.

```
void moveHorizontal(int distance)
```

This is called the *header* of the method. The header provides some information about the method in question. The part enclosed by parentheses (`int distance`) is the information about the required parameter. For each parameter, it defines a *type* and a *name*. The header above states that the method requires one parameter of type `int` named `distance`. The name gives a hint about the meaning of the data expected. Together, the name of a method and the parameter types found in its header are called the method's *signature*.

1.5 Data types

Concept

Parameters have **types**. The type defines what kinds of values a parameter can take.

A type specifies what kind of data can be passed to a parameter. The type `int` signifies whole numbers (also called "integer" numbers, hence the abbreviation "int").

In the example above, the signature of the `moveHorizontal` method states that, before the method can execute, we need to supply a whole number specifying the distance to move. The data entry field shown in Figure 1.5 then lets you enter that number.

In the examples so far, the only data type we have seen has been `int`. The parameters of the move methods and the `changeSize` method are all of that type.

Closer inspection of the object's pop-up menu shows that the method entries in the menu include the parameter types. If a method has no parameter, the method name is followed by an empty set of parentheses. If it has a parameter, the type and name of that parameter

is displayed. In the list of methods for a circle, you will see one method with a different parameter type: the **changeColor** method has a parameter of type **String**.

The **String** type indicates that a section of text (for example, a word or a sentence) is expected. Strings are always enclosed within double quotes. For example, to enter the word *red* as a string, type

```
"red"
```

The method-call dialog also includes a section of text called a *comment* above the method header. Comments are included to provide information to the (human) reader and are described in Chapter 2. The comment of the **changeColor** method describes what color names the system knows about.

Exercise 1.4 Invoke the **changeColor** method on one of your circle objects and enter the string **"red"**. This should change the color of the circle. Try other colors.

Exercise 1.5 This is a very simple example, and not many colors are supported. See what happens when you specify a color that is not known.

Exercise 1.6 Invoke the **changeColor** method, and write the color into the parameter field *without* the quotes. What happens?

Pitfall A common error for beginners is to forget the double quotes when typing in a data value of type **String**. If you type **green** instead of **"green"**, you will get an error message saying something like "Error: cannot find symbol - variable green."

Java supports several other data types, including decimal numbers and characters. We shall not discuss all of them right now, but rather come back to this issue later. If you want to find out about them now, see at Appendix B.

1.6 Multiple instances

Once you have a class, you can create as many objects (or instances) of that class as you like. From the class **Circle**, you can create many circles. From **Square**, you can create many squares.

Exercise 1.7 Create several circle objects on the object bench. You can do so by selecting **new Circle()** from the pop-up menu of the **Circle** class. Make them visible, then move them around on the screen using the "move" methods. Make one big and yellow; make another one small and green. Try the other shapes too: create a few triangles, squares, and persons. Change their positions, sizes, and colors.

Concept

Multiple instances. Many similar objects can be created from a single class.

Every one of those objects has its own position, color, and size. You change an attribute of an object (such as its size) by calling a method on that object. This will affect this particular object, but not others.

You may also notice an additional detail about parameters. Have a look at the **changeSize** method of the triangle. Its header is

```
void changeSize(int newHeight, int newWidth)
```

Here is an example of a method with more than one parameter. This method has two, and a comma separates them in the header. Methods can, in fact, have any number of parameters.

1.7 State

Concept

Objects have state. The **state** is represented by storing values in fields.

The set of values of all attributes defining an object (such as *x*-position, *y*-position, color, diameter, and visibility status for a circle) is also referred to as the object's *state*. This is another example of common terminology that we shall use from now on.

In BlueJ, the state of an object can be inspected by selecting the *Inspect* function from the object's pop-up menu. When an object is inspected, an *object inspector* is displayed. The object inspector is an enlarged view of the object that shows the attributes stored inside it (Figure 1.6).

Exercise 1.8 Make sure you have several objects on the object bench, and then inspect each of them in turn. Try changing the state of an object (for example, by calling the **moveLeft** method) while the object inspector is open. You should see the values in the object inspector change.

Some methods, when called, change the state of an object. For example, **moveLeft** changes the **xPosition** attribute. Java refers to these object attributes as *fields*.

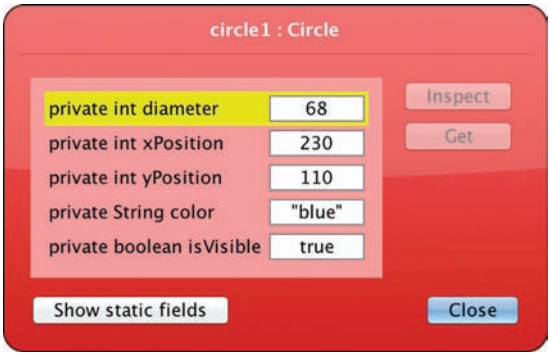

Figure 1.6
An object inspector, showing details of an object

1.8 What is in an object?

On inspecting different objects, you will notice that objects of the *same* class all have the same fields. That is, the number, type, and names of the fields are the same, while the actual value of a particular field in each object may be different. In contrast, objects of a *different* class may have different fields. A circle, for example, has a "diameter" field, while a triangle has fields for "width" and "height."

The reason is that the number, types, and names of fields are defined in a class, not in an object. So the class **`Circle`** defines that each circle object will have five fields, named **`diameter`**, **`xPosition`**, **`yPosition`**, **`color`**, and **`isVisible`**. It also defines the types for these fields. That is, it specifies that the first three are of type **`int`**, while the color is of type **`String`** and the **`isVisible`** flag is of type **`boolean`**. (**`boolean`** is a type that can represent two values: **`true`** and **`false`**. We shall discuss it in more detail later.)

When an object of class **`Circle`** is created, the object will automatically have these fields. The values of the fields are stored in the object. That ensures that each circle has a color, for instance, and each can have a different color (Figure 1.7).

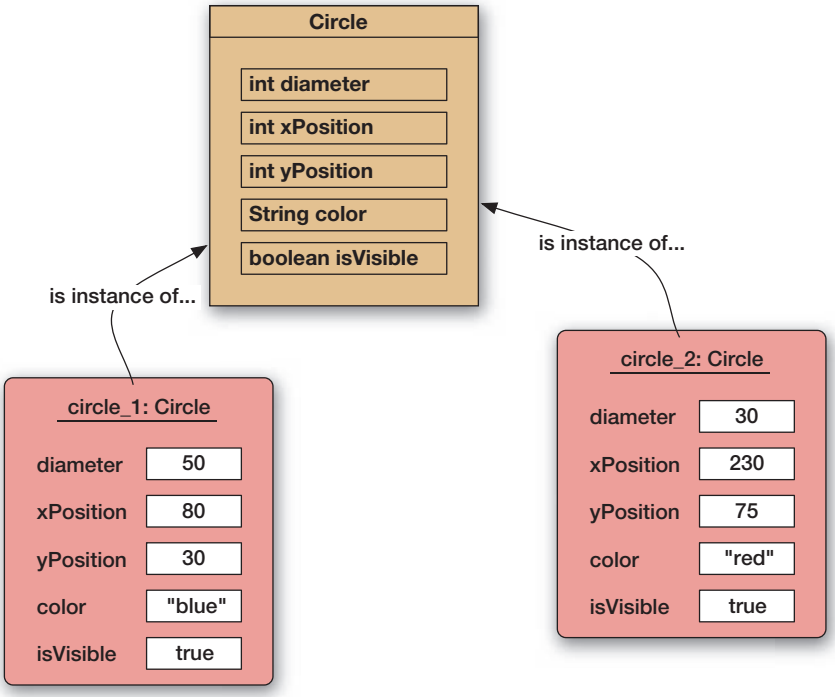

Figure 1.7
A class and its objects with fields and values

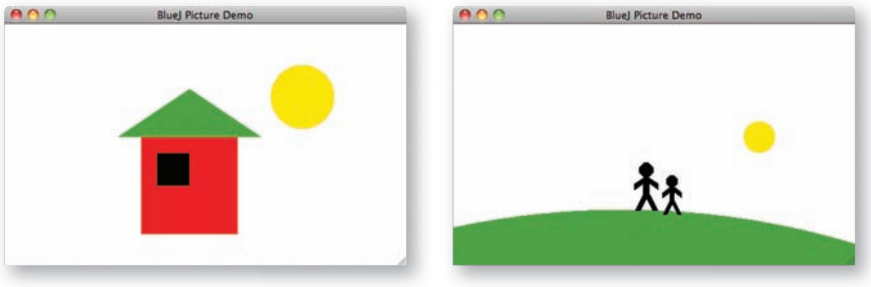

Figure 1.8
Two images created from a set of shape objects

The story is similar for methods. Methods are defined in the class of the object. As a result, all objects of a given class have the same methods. However, the methods are invoked on objects. This makes it clear which object to change when, for example, a `moveRight` method is invoked.

Exercise 1.9 Figure 1.8 shows two different images. Choose one of these images and recreate it using the shapes from the *figures* project. While you are doing this, write down what you have to do to achieve this. Could it be done in different ways?

1.9 Java code

When we program in Java, we essentially write down instructions to invoke methods on objects, just as we have done with our figure objects above. However, we do not do this interactively by choosing methods from a menu with the mouse, but instead we type the commands down in textual form. We can see what those commands look like in text form by using the BlueJ Terminal.

Exercise 1.10 Select *Show Terminal* from the *View* menu. This shows another window that BlueJ uses for text output. Then select *Record method calls* from the terminal's *Options* menu. This function will cause all our method calls (in their textual form) to be written to the terminal. Now create a few objects, call some of their methods, and observe the output in the terminal window.

Using the terminal's *Record method calls* function, we can see that the sequence of creating a person object and calling its `makeVisible` and `moveRight` methods looks like this in Java text form:

```
Person person1 = new Person();
person1.makeVisible();
person1.moveRight();
```

Here, we can observe several things:

- We can see what creating an object and giving it a name looks like. Technically, what we are doing here is *storing the Person object into a variable*; we will discuss this in detail in the next chapter.
- We see that, to call a method on an object, we write the name of the object, followed by a dot, followed by the name of the method. The command ends with a parameter list—an empty pair of parentheses if there are no parameters.
- All Java statements end with a semicolon.

Instead of just looking at Java statements, we can also type them. To do this, we use the *Code Pad*. (You can switch off the *Record method calls* function now and close the terminal.)

Exercise 1.11 Select *Show Code Pad* from the *View* menu. This should display a new pane next to the object bench in your main BlueJ window. This pane is the *Code Pad*. You can type Java code here.

In the Code Pad, we can type Java code that does the same things we did interactively before. The Java code we need to write is exactly like that shown above.

Exercise 1.12 In the Code Pad, type the code shown above to create a Person object and call its `makeVisible` and `moveRight` methods. Then go on to create some other objects and call their methods.

Tip

You can recall previously used commands in the Code Pad by using the up arrow.

Typing these commands should have the same effect as invoking the same command from the object's menu. If instead you see an error message, then you have mistyped the command. Check your spelling. You will note that getting even a single character wrong will cause the command to fail.

1.10 Object interaction

For the next section, we shall work with a different example project. Close the *figures* project if you still have it open, and open the project called *house.*

Exercise 1.13 Open the *house* project. Create an instance of class `Picture` and invoke its `draw` method. Also, try out the `setBlackAndWhite` and `setColor` methods.

Exercise 1.14 How do you think the `Picture` class draws the picture?

Five of the classes in the *house* project are identical to the classes in the *figures* project. But we now have an additional class: **`Picture`**. This class is programmed to do exactly what we have done by hand in Exercise 1.9.

In reality, if we want a sequence of tasks done in Java, we would not normally do it by hand. Instead, we would create a class that does it for us. This is the **`Picture`** class.

The **`Picture`** class is written so that, when you create an instance, the instance creates two square objects (one for the wall, one for the window), a triangle, and a circle. Then, when you call the `draw` method, it moves them around and changes their color and size, until the canvas looks like the picture we see in Figure 1.8.

Concept

Method calling. Objects can communicate by **calling** each other's **methods**.

The important points here are that: objects can create other objects; and they can call each other's methods. In a normal Java program, you may well have hundreds or thousands of objects. The user of a program just starts the program (which typically creates a first object), and all other objects are created—directly or indirectly—by that object.

The big question now is this: How do we write the class for such an object?

1.11 Source code

Each class has some *source code* associated with it. The source code is text that defines the details of the class. In BlueJ, the source code of a class can be viewed by selecting the *Open Editor* function from the class's pop-up menu, or by double-clicking the class icon.

Concept

The **source code** of a class determines the structure and behavior (the fields and methods) of each of the objects of that class.

Exercise 1.15 Look at the pop-up menu of class **`Picture`** again. You will see an option labeled *Open Editor.* Select it. This will open a text editor displaying the source code of the class.

The source code is text written in the Java programming language. It defines what fields and methods a class has, and precisely what happens when a method is invoked. In the next chapter, we shall discuss exactly what the source code of a class contains, and how it is structured.

A large part of learning the art of programming is learning how to write these class definitions. To do this, we shall learn to use the Java language (although there are many other programming languages that could be used to write code).

When you make a change to the source code and close the editor, the icon for that class appears striped in the diagram.[3] The stripes indicate that the source has been changed. The class now needs to be compiled by clicking the *Compile* button. (You may like to read the "About compilation" note for more information on what is happening when you compile a class.) Once a class has been compiled, objects can be created again and you can try out your change.

[3] In BlueJ, there is no need to explicitly save the text in the editor before closing. If you close the editor, the source code will automatically be saved.

About compilation When people write computer programs, they typically use a "higher-level" programming language such as Java. A problem with that is that a computer cannot execute Java source code directly. Java was designed to be reasonably easy to read for humans, not for computers. Computers, internally, work with a binary representation of a machine code, which looks quite different from Java. The problem for us is that it looks so complex that we do not want to write it directly. We prefer to write Java. What can we do about this?

The solution is a program called the *compiler*. The compiler translates the Java code into machine code. We can write Java and run the compiler—which generates the machine code—and the computer can then read the machine code. As a result, every time we change the source code, we must first run the compiler before we can use the class again to create an object. Otherwise, the machine code version that the computer needs will not exist.

Exercise 1.16 In the source code of class **`Picture`**, find the part that actually draws the picture. Change it so that the sun will be blue rather than yellow.

Exercise 1.17 Add a second sun to the picture. To do this, pay attention to the field definitions close to the top of the class. You will find this code:

```
private Square wall;
private Square window;
private Triangle roof;
private Circle sun;
```

You need to add a line here for the second sun. For example:

```
private Circle sun2;
```

Then write the appropriate code in two different places for creating the second sun and making it visible when the picture is drawn.

Exercise 1.18 *Challenge exercise* (This means that this exercise might not be solved quickly. We do not expect everyone to be able to solve this at the moment. If you do, great. If you don't, then don't worry. Things will become clearer as you read on. Come back to this exercise later.) Add a sunset to the single-sun version of **`Picture`**. That is, make the sun go down slowly. Remember: The circle has a method **`slowMoveVertical`** that you can use to do this.

Exercise 1.19 *Challenge exercise* If you added your sunset to the end of the **`draw`** method (so that the sun goes down automatically when the picture is drawn), change this now. We now want the sunset in a separate method, so that we can call **`draw`** and see the picture with the sun up, and then call **`sunset`** (a separate method!) to make the sun go down.

Exercise 1.20 *Challenge exercise* Make a person walk up to the house after the sunset.

1.12 Another example

In this chapter, we have already discussed a large number of new concepts. To help in understanding these concepts, we shall now revisit them in a different context. For this, we use another example. Close the *house* project if you still have it open, and open the *lab-classes* project.

This project is a simplified part of a student database designed to keep track of students in laboratory classes, and to print class lists.

Exercise 1.21 Create an object of class **`Student`**. You will notice that this time you are prompted not only for a name of the instance, but also for some other parameters. Fill them in before clicking OK. (Remember that parameters of type **`String`** must be written within double quotes.)

1.13 Return values

As before, you can create multiple objects. And again, as before, the objects have methods that you can call from their pop-up menu(s).

Exercise 1.22 Create some `Student` objects. Call the **`getName`** method on each object. Explain what is happening.

Concept

Result. Methods may return information about an object via a **return value**.

When calling the **`getName`** method of the **`Student`** class, we notice something new: methods may return a result value. In fact, the header of each method tells us whether or not it returns a result, and what the type of the result is. The header of **`getName`** (as shown in the object's pop-up menu) is defined as

```
String getName()
```

The word **`String`** before the method name specifies the return type. In this case, it states that calling this method will return a result of type **`String`**. The header of **`changeName`** states:

```
void changeName(String replacementName)
```

The word **`void`** indicates that this method does not return any result.

Methods with return values enable us to get information from an object via a method call. This means that we can use methods either to change an object's state, or to find out about its state. The return type of a method is not part of its signature.

1.14 Objects as parameters

Exercise 1.23 Create an object of class **`LabClass`**. As the signature indicates, you need to specify the maximum number of students in that class (an integer).

Exercise 1.24 Call the **`numberOfStudents`** method of that class. What does it do?

Exercise 1.25 Look at the signature of the **`enrollStudent`** method. You will notice that the type of the expected parameter is **`Student`**. Make sure you have two or three students and a **`LabClass`** object on the object bench, then call the **`enrollStudent`** method of the **`LabClass`** object. With the input cursor in the dialog entry field, click on one of the student objects; this enters the name of the student object into the parameter field of the **`enrollStudent`** method (Figure 1.9). Click OK and you have added the student to the **`LabClass`**. Add one or more other students as well.

Exercise 1.26 Call the **`printList`** method of the **`LabClass`** object. You will see a list of all the students in that class printed to the BlueJ terminal window (Figure 1.10).

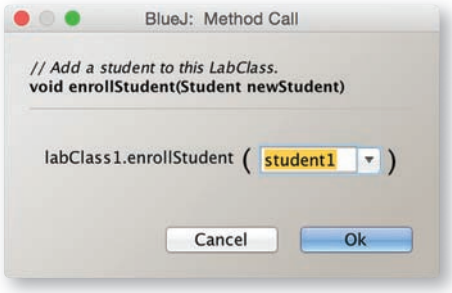

Figure 1.9
Adding a student to a **`LabClass`**

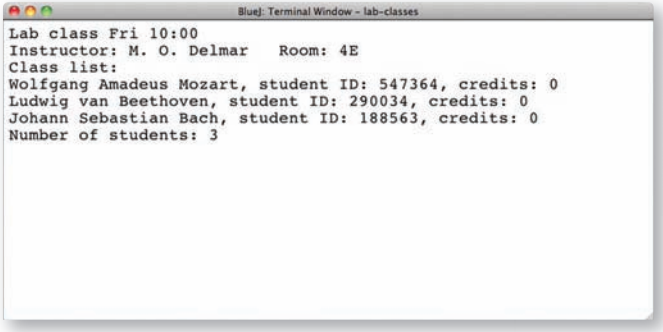

Figure 1.10
Output of the **`LabClass`** class listing

As the exercises show, objects can be passed as parameters to methods of other objects. In the case where a method expects an object as a parameter, the expected object's class name is specified as the parameter type in the method signature.

Explore this project a bit more. Try to identify the concepts discussed in the *figures* example in this context.

Exercise 1.27 Create three students with the following details:

Snow White, student ID: *A00234,* credits: *24*

Lisa Simpson, student ID: *C22044,* credits: *56*

Charlie Brown, student ID: *A12003*, credits: *6*

Then enter all three into a lab and print a list to the screen.

Exercise 1.28 Use the inspector on a **`LabClass`** object to discover what fields it has.

Exercise 1.29 Set the instructor, room, and time for a lab, and print the list to the terminal window to check that these new details appear.

1.15 Summary

In this chapter, we have explored the basics of classes and objects. We have discussed the fact that objects are specified by classes. Classes represent the general concept of things, while objects represent concrete instances of a class. We can have many objects of any class.

Objects have methods that we use to communicate with them. We can use a method to make a change to the object or to get information from the object. Methods can have parameters, and parameters have types. Methods have return types, which specify what type of data they return. If the return type is **`void`**, they do not return anything.

Objects store data in fields (which also have types). All the data values of an object together are referred to as the object's state.

Objects are created from class definitions that have been written in a particular programming language. Much of programming in Java is about learning to write class definitions. A large Java program will have many classes, each with many methods that call each other in many different ways.

To learn to develop Java programs, we need to learn how to write class definitions, including fields and methods, and how to put these classes together well. The rest of this book deals with these issues.

Terms introduced in this chapter:

object, class, instance, method, signature, parameter, type, state, source code, return value, compiler

Concept summary

- **object** Java objects model objects from a problem domain.
- **class** Objects are created from classes. The class describes the kind of object; the objects represent individual instances of the class.
- **method** We can communicate with objects by invoking methods on them. Objects usually do something if we invoke a method.
- **parameter** Methods can have parameters to provide additional information for a task.
- **signature** The method name and the parameter types of a method are called its signature. They provide the information needed to invoke that method.
- **type** Parameters have types. The type defines what kinds of values a parameter can take.
- **multiple instances** Many similar objects can be created from a single class.
- **state** Objects have state. The state is represented by storing values in fields.
- **method calling** Objects can communicate by calling each other's methods.
- **source code** The source code of a class determines the structure and behavior (the fields and methods) of each of the objects of that class.
- **result** Methods may return information about an object via a return value.

Exercise 1.30 In this chapter we have mentioned the data types `int` and `String`. Java has more predefined data types. Find out what they are and what they are used for. To do this, you can check Appendix B, or look it up in another Java book or in an online Java language manual. One such manual is at

```
http://download.oracle.com/javase/tutorial/java/nutsandbolts/
datatypes.html
```

Exercise 1.31 What are the types of the following values?

```
0
"hello"
101
-1
true
"33"
3.1415
```

Exercise 1.32 What would you have to do to add a new field, for example one called `name`, to a circle object?

Exercise 1.33 Write the header for a method named **`send`** that has one parameter of type `String`, and does not return a value.

Exercise 1.34 Write the header for a method named **`average`** that has two parameters, both of type **`int`**, and returns an **`int`** value.

Exercise 1.35 Look at the book you are reading right now. Is it an object or a class? If it is a class, name some objects. If it is an object, name its class.

Exercise 1.36 Can an object have several different classes? Discuss.