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 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 *shapes*.[1] You should see a window similar to that shown in Figure 1.1.

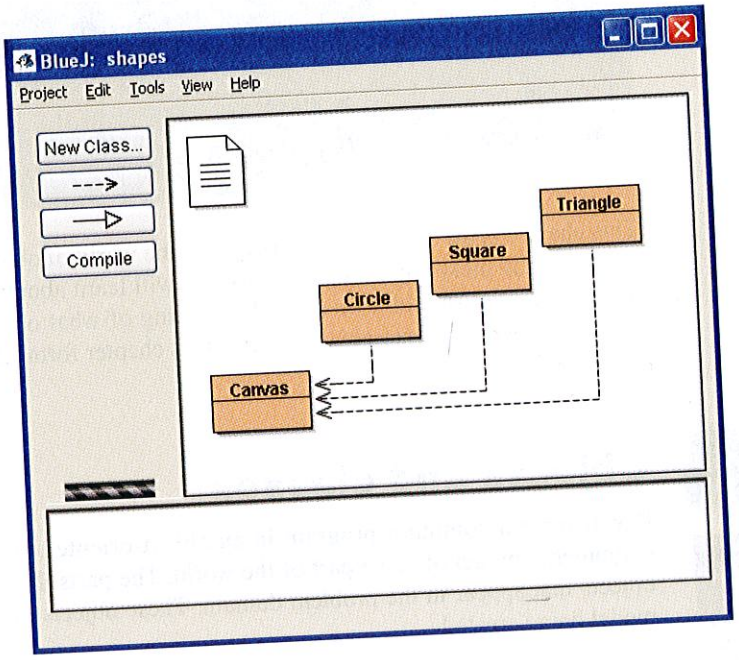

Figure 1.1
The *shapes* project in BlueJ

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`, and `Canvas`.

Right-click on the `Circle` class and choose

```
new Circle()
```

from the popup 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).

[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.

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 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 *shapes*.[1] You should see a window similar to that shown in Figure 1.1.

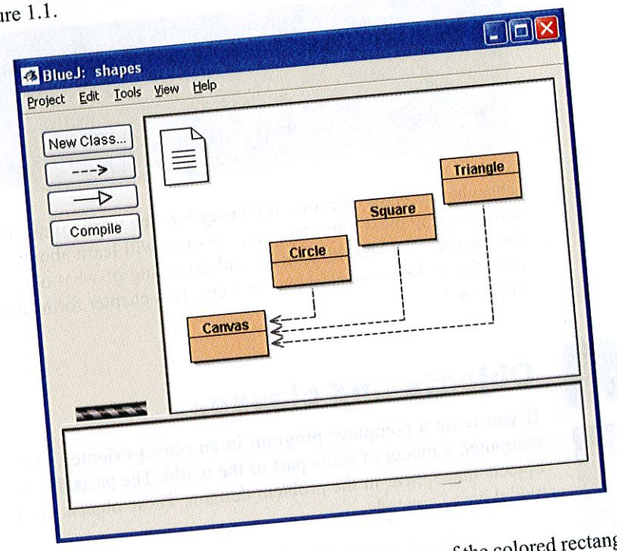

Figure 1.1
The *shapes* project in BlueJ

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`, and `Canvas`.

Right-click on the `Circle` class and choose

```
new Circle()
```

from the popup 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).

[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.

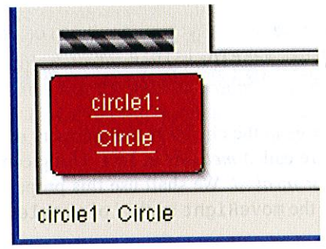

Figure 1.2
An object on the object bench

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.

Exercise 1.1 Create another circle. Then create a square.

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*.

1.3 Calling methods

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

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 center of the screen. You may also like to try `makeInvisible` and `makeVisible` to hide and show the circle.

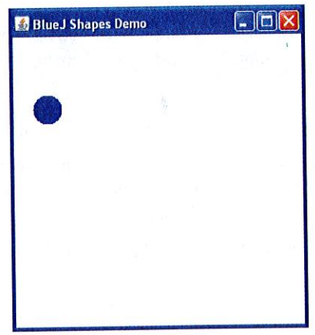

Figure 1.3
A drawing of a circle

Concept:

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

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 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 1.4). 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.

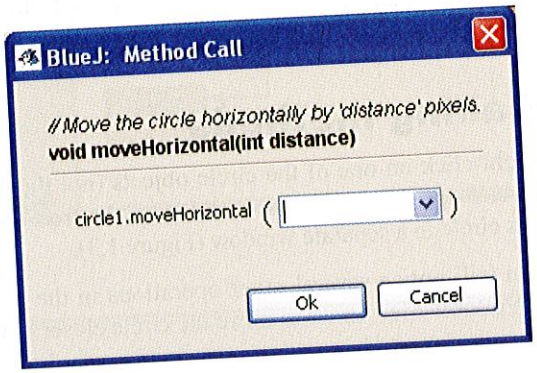

Figure 1.4
A method call dialog

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.

The additional values that some methods require are called *parameters*. A method indicates wh kinds of parameters it requires. When calling, for example, the `moveHorizontal` method shown in Figure 1.4, the dialog displays the line

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

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

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

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 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 1.4). 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.

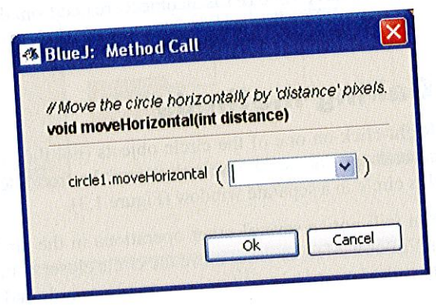

Figure 1.4
A method call dialog

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.

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 1.4, the dialog displays the line

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

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

Concept: The header of a method is called its **signature.** It provides information needed to invoke that method.

near the top. This is called the *signature* of the method. The signature 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 signature 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.

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 1.4 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 popup 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 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 signature. 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 resolve symbol.'

Java supports several other data types, including decimal numbers and characters. We shall discuss all of them right now, but rather come back to this issue later. If you want to find about them now, look at Appendix B.

1.6 Multiple instances

Exercise 1.7 Create several circle objects on the object bench. You can do so b selecting `new Circle( )` from the popup menu of the `Circle` class. Make thei visible, then move them around on the screen using the 'move' methods. Make one bi and yellow; make another one small and green. Try the other shapes too: create a fe triangles and squares. Change their positions, sizes, and colors.

Concept:

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

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

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

You may also notice an additional detail about parameters. Have a look at the `changeS` method of the triangle. Its signature is

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

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

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, di eter, and visibility status for a circle) is also referred to as the object's *state.* This is anc 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 object's popup menu. When an object is inspected, a window similar to that shown in Figure is displayed. This window is called the *object inspector.*

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

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, look at Appendix B.

1.6 Multiple instances

Exercise 1.7 Create several circle objects on the object bench. You can do so by selecting `new Circle( )` from the popup 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 and squares. Change their positions, sizes, and colors.

Concept:

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

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.

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 signature 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 signature. 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 popup menu. When an object is inspected, a window similar to that shown in Figure 1.5 is displayed. This window is called the *object inspector*.

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.

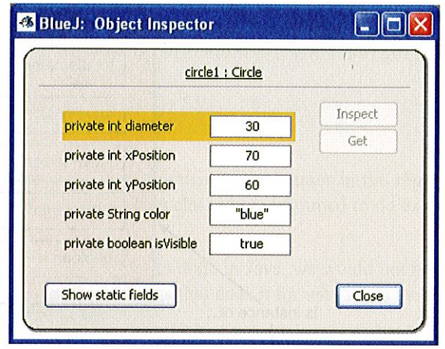

Figure 1.5
An object inspection dialog

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*.

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 field 'diameter,' 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 these fields are stored in the object. That ensures that each circle has a color, for instance, and each can have a different color (Figure 1.6).

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 Use the shapes from the *shapes* project to create an image of a house and a sun, similar to that shown in Figure 1.7. While you are doing this, write down what you have to do to achieve this. Could it be done in different ways?