Chapter 3
Variables and Functions

The first step towards wisdom is calling things by their right names.

OLD CHINESE PROVERB

Figuratively speaking, killing two birds with one stone may be good, but killing three, four, or even more birds with one stone is even better.

V. OREHCK III

Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.

THE BRIDGEKEEPER (TERRY GILLIAM), IN *MONTY PYTHON AND THE HOLY GRAIL*

Objectives

Upon completion of this chapter, you should be able to:

- ❑ Use variables to store values for use later in a method
- ❑ Use a variable to store the value of an arithmetic expression
- ❑ Use a variable to store the value produced by a function
- ❑ Use parameters to write methods that are more broadly useful
- ❑ Define and access property variables
- ❑ Use the **`vehicle`** property to synchronize the movements of two objects
- ❑ Create functions — messages that return a value to their sender

In Chapter 2, we saw how to define world and object methods. In this chapter, we turn our attention to **variables**, the use of which can make it easier to define methods. In computer programming, *a variable is a name that refers to a piece of the program's memory, in which a value can be stored, retrieved, and changed.*

Alice provides several different kinds of variables that we will examine in this chapter. The first kind is the **method variable**, which lets us store a value within a method for later use. The second kind is the **parameter**, which lets us write methods that are more broadly useful. These first two kinds of variables are created using the two buttons that appear on the right edge of every Alice method, as shown in Figure 3-1.

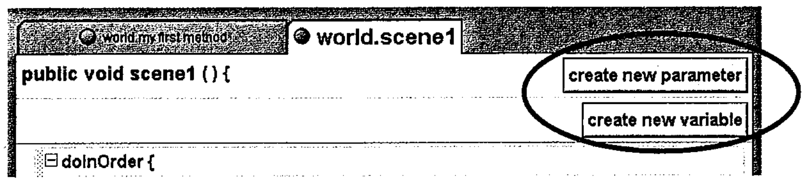


FIGURE 3-1 The buttons to create variables and parameters

The third and final kind of variable is the **object variable** or **property variable**, which lets us store a property of an object. Object variables are created using the `create new variable` button under the *properties* pane of the *details area*.

In this chapter, we'll see how to create and use all three kinds of variables.

3.1 Method Variables

Method variables are names defined within a method that refer to program memory in which values can be stored. When we click the `create new variable` button within a method, Alice asks us what we want to *name* the variable, the *type* of information we want to store in it, and its initial value. When we have told it these things, Alice reserves as much program memory as is needed for that type of information, and associates the name with that memory, which is called **defining** the variable. Method variables are often called **local variables**, because they can only be accessed from within the method in which they are defined — they are *local* to it.

One common use of method variables is to compute and store values that will be used later, especially values that will be used more than once. Another common use is to store values that the user enters. In the rest of this section, we present these two uses.

3.1.1 Example 1: Storing a Computed Value

Suppose that in Scene 2 of a story, a girl and a horse are positioned as seen in Figure 3-2.

FIGURE 3-2 Girl and horse: initial positions

Suppose our scene calls for the girl to move toward the horse and stop when she is directly in front of it. We can send the girl the **move()** message to move her toward the horse, but how far should we ask her to move? One way would be to use trial-and-error to find a suitable value. But trial-and-error is tedious, especially when there is a better way. The better way is to:

1. Define a variable to store the distance from the girl to the horse.
2. Ask the girl how far she is from the horse, and store her reply in the variable.
3. Use that variable in the **move()** message to get her to move the right distance.

To accomplish the first step, we just click the **create new variable** button we saw in Figure 3-1. To get the information it needs to define the variable, Alice pops up a **Create New Local Variable** dialog box in which we can enter the variable's **name**, **type**, and **initial value**.

A variable's name should be a noun that describes the value it stores.

For example, this variable is storing the distance from the girl to the horse, so we will name it **distanceToHorse**. Like method names, variable names always use lowercase letters, capitalizing the first letter of words after the first word.

A variable's **type** describes the kind of value we intend to store in it. Alice provides four basic types:

- **Number**, for storing numeric values (for example, **-3**, **-1.5**, **0**, **1**, **3.14159**, and so on)
- **Boolean**, for storing logical (**true** or **false**) values
- **Object**, for storing references to Alice objects (for example, **troll**, **wizard**, **castle**, and so on)
- **Other**, for storing things like **Strings**, **Colors**, **Sounds**, and other kinds of values

Since the distance from the girl to the horse is a numerical value, **Number** is the appropriate type for this variable.

As its name suggests, the **initial value** is the value the variable will contain when the method begins. We will usually use a value like 0 or 1, as shown in Figure 3-3.

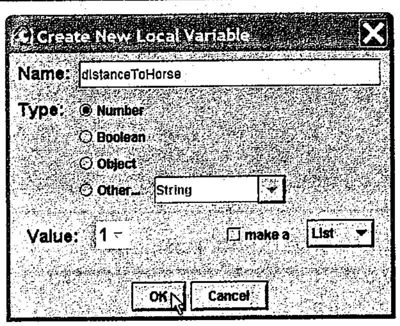


FIGURE 3-3 The **Create New Local Variable** dialog box

When we click the **OK** button, Alice defines a new variable in the method, in the space above the *editing area*, as shown in Figure 3-4.

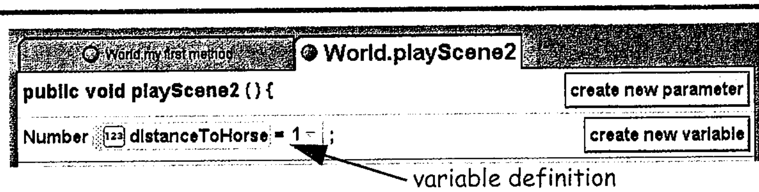


FIGURE 3-4 The **distanceToHorse** variable

Next, we want to ask the girl how far she is from the horse, and set the value of this variable to her response. In Alice, it is easiest to do these steps in reverse order.

Setting the value of a variable is done in a way similar to how we set the value of a property back in Chapter 1: we drag its definition into the *editing area*, and Alice generates a menu of potential values, as shown in Figure 3-5.

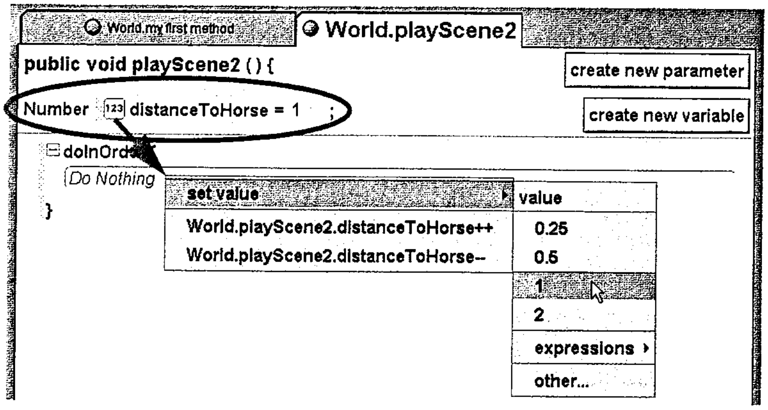


FIGURE 3-5 Setting a variable's value (part I)

If we wished to add 1 to **distanceToHorse**, we would choose **World.playScene2.distanceToHorse++** from the menu (**++** is called the **increment operator**). If we wanted to subtract 1 from its value, we would choose **World.playScene2.distanceToHorse--** (**--** is called the **decrement operator**). Since we want to *set* the variable's value, we choose the **set value** choice.

The value to which we want to set **distanceToHorse** is the result of asking the girl how far she is from the horse. Unfortunately, this value is not present in the menu. In this situation, we can choose any value from the menu to act as a **placeholder** for the function. (In Figure 3-5, we are choosing **1** as the placeholder.) The result is the **set()** statement shown in Figure 3-6.

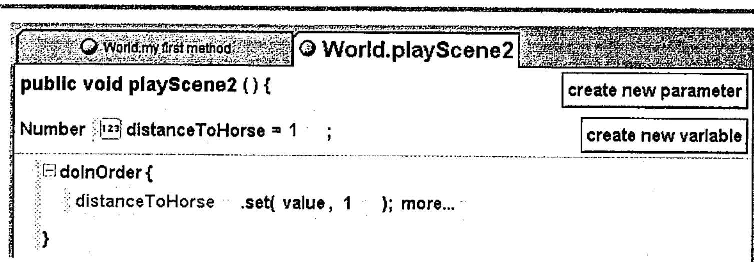


FIGURE 3-6 Setting a variable's value using a placeholder

With a **set()** statement in place, we are ready to ask the girl how far she is from the horse. To do so, we make sure we have the girl selected in the *object tree*, and then click the *functions* tab in the *details area*. "How far are you from the horse?" is a proximity question, so we look in the proximity section of the functions. Since the girl is in front of the horse and we see a **distanceInFrontOf()** proximity function, we drag it into the *editing area* to replace the **1** in the **set()** message, as shown in Figure 3-7.

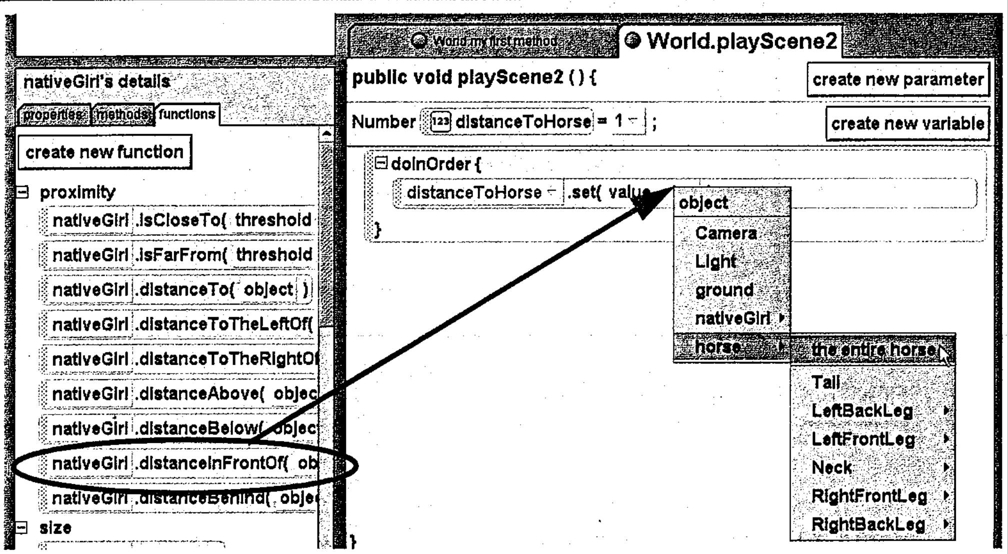


FIGURE 3-7 Setting a variable's value to a function's answer

When we drag the function onto the placeholder (**1**), the box around the **1** turns green, indicating we can drop it. Alice then asks us for the object whose distance we want to compute, and displays a menu of the available options. When we select **horse -> the entire horse** (see Figure 3-7), Alice replaces the placeholder **1** with the function, as can be seen in Figure 3-8.

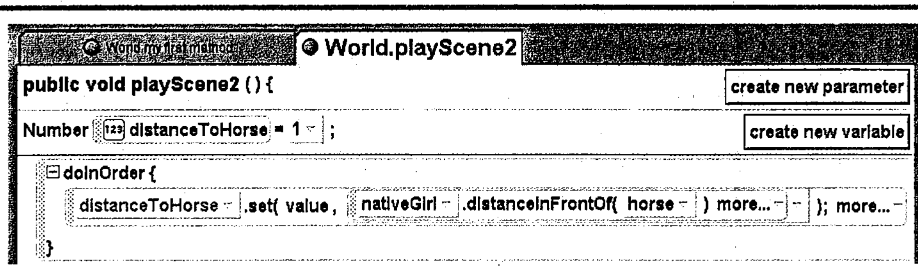


FIGURE 3-8 Setting a variable's value (part III)

You may be wondering why we used the **distanceInFrontOf()** function instead of the **distanceTo()** function. The reason is that the **distanceTo()** function returns the distance from the *center* of one object to the *center* of the other object. If we moved the girl that far, she and the horse would occupy the same space, which looks really weird! (Try it and see.) By contrast, the other proximity methods all measure from the *outer edge* of one object's bounding box to the *outer edge* of the other object's bounding box.

Once we have a variable containing the distance from the girl to the front of the horse, we can use it in the **`move()`** message. When we drag the **`move()`** message into the *editing area*, we can specify that we want the girl to move forward the value of the variable by selecting **`expressions -> distanceToHorse`**. Alice's **`expressions`** menu usually contains a list of all the variables (and parameters, and functions we define) that are available for use within the current method. Figure 3-9 illustrates this.

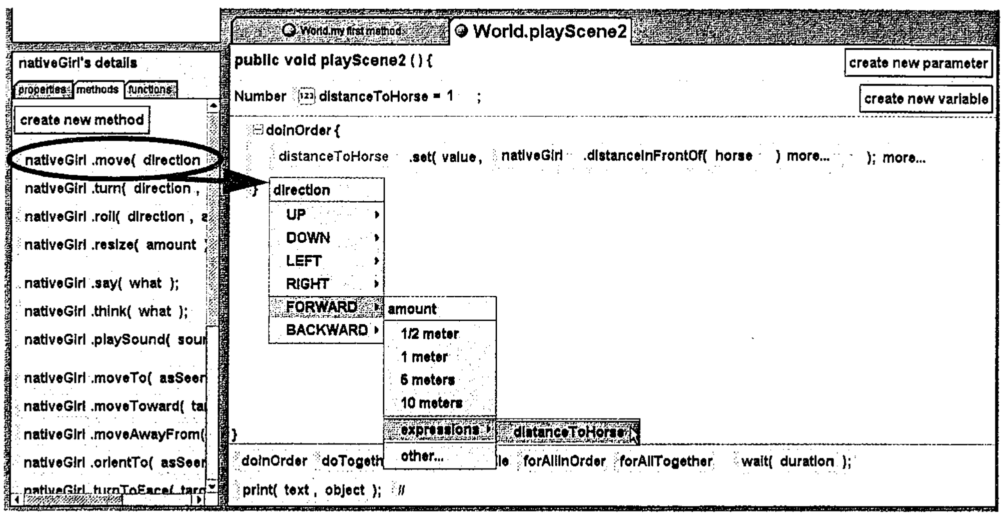


FIGURE 3-9 Using a variable's value in a message (part I)

Figure 3-10 shows the statement Alice generates when we select **`distanceToHorse`**.

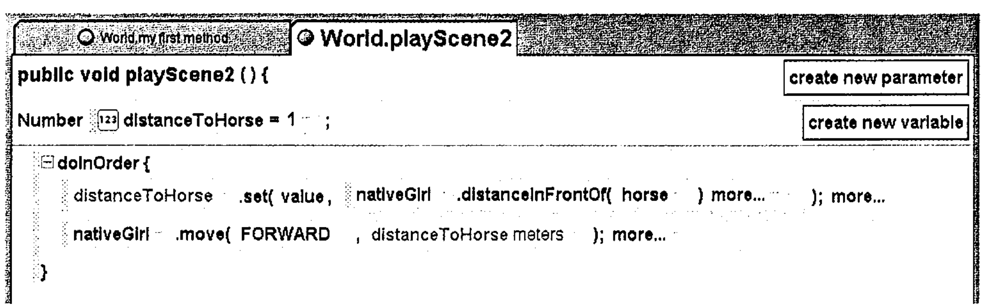


FIGURE 3-10 Using a variable's value in a message (part II)

When we play this method, we get the result shown in Figure 3-11.

FIGURE 3-11 The girl too close to the horse

This looks a bit too close for comfort — the girl is invading the personal space of the horse! We can easily remedy that by moving her slightly less than **distanceToHorse**. In the **move()** statement, clicking the list arrow next to **distanceToHorse** reveals a drop-down menu we can use to modify the distance the girl moves, as shown in Figure 3-12.

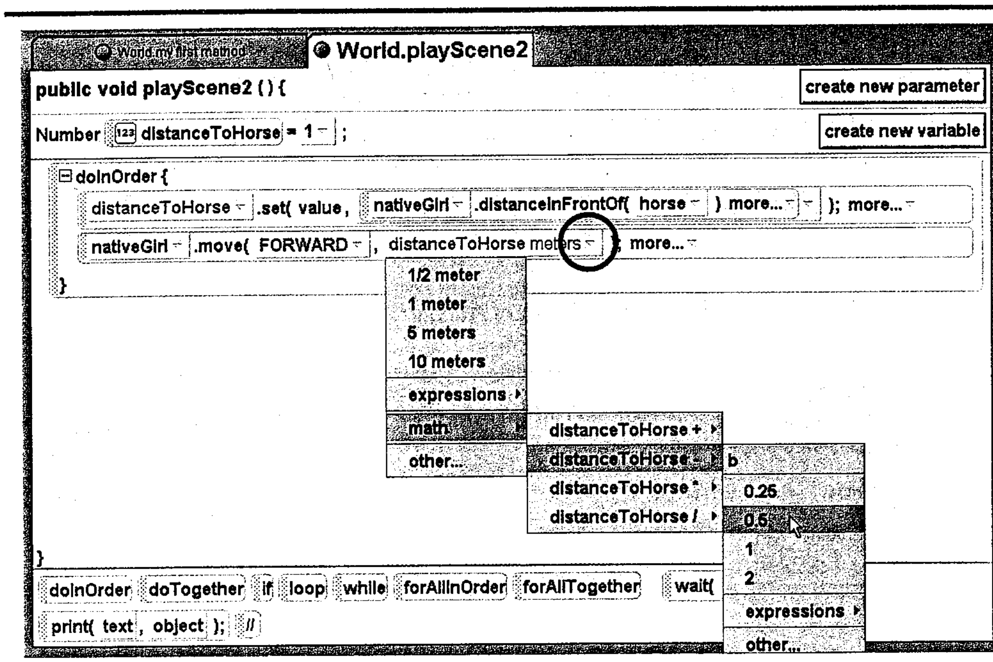


FIGURE 3-12 Adjusting a value in a message

As can be seen in Figure 3-12, Alice's **math** menu choice provides the basic arithmetic calculations of addition, subtraction, multiplication, and division. Selecting **distanceToHorse - 0.5** produces the statement shown in Figure 3-13.

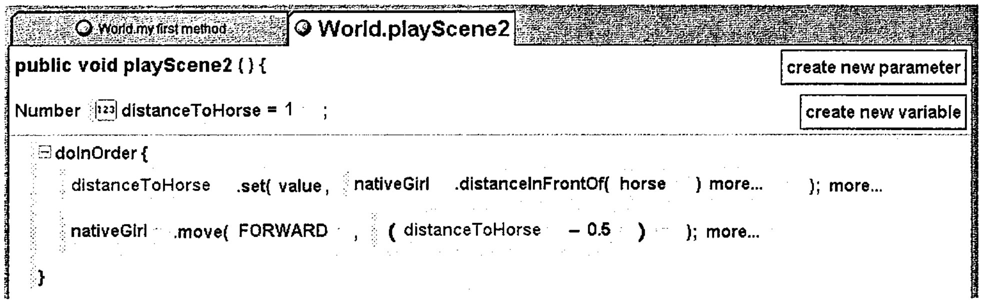


FIGURE 3-13 Decreasing how far she moves

Now, when we play the method, the girl stops a comfortable distance from the horse, as shown in Figure 3-14.

FIGURE 3-14 Stopping a comfortable distance from the horse

Using functions and variables has a major advantage over trial-and-error: it yields the right behavior even if we reposition the girl or the horse! If we had used trial-and-error to find the exact distance to move the girl, and then later repositioned the girl or horse, the value we had found using trial-and-error would no longer be correct, so we would have to fix it (either with another round of trial-and-error, or by getting smart and using a variable and a function).

Once you get used to using variables and functions, they often provide a much better way to make a character move a distance relative to another object.

3.1.2 Example 2: Storing a User-Entered Value

Another common use of variables is to store values that the user enters, for later use. To illustrate, suppose your geometry teacher gives you a list of right triangles' leg-lengths, and tells you to calculate each triangle's hypotenuse length using the Pythagorean Theorem:

$$c = \sqrt{a^2 + b^2}$$

We could either get out our calculators and grind through the list, or we could write an Alice program to help us. Which sounds like more fun? (Writing an Alice program, of course!)

As always, we start with a user story. We might write something like this:

Scene: There is a girl on the screen. She says, "I can calculate hypotenuse-lengths in my head!" Then she says, "Give me the lengths of the two edges of a right triangle..." A dialog box appears, prompting us for the first edge length. When we enter it, a second dialog box appears, prompting us for the second edge length. When we enter it, the girl says, "The hypotenuse-length is X." (Where X is the correct answer.)

The nouns in our story include girl, hypotenuse-length, first edge length, second edge length, and two dialog boxes. For the girl, we will use the **skaterGirl** from the Alice Gallery. For the hypotenuse-length, first edge length, and second edge length, we will create **Number** variables named **hypotenuse**, **edge1**, and **edge2**, respectively. For the dialog boxes, Alice provides a function that will build and display dialog objects for us (see below).

Since the scene has just one object (girl), we will create a **skaterGirl** object method named **computeHypotenuse()** to animate her with the desired behavior. Within this method, we declare the three **Number** variables, and then begin programming the desired behavior. Using what we have seen so far, we can get to the point shown in Figure 3-15:

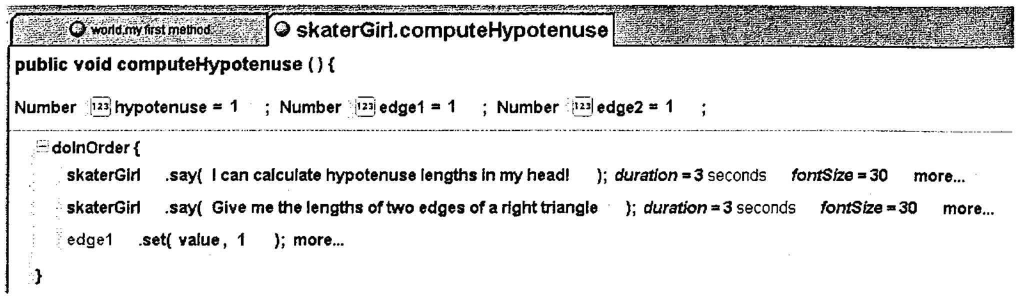


FIGURE 3-15 Getting started

But how do we generate a dialog box to set the value of **edge1**? The trick is to look in the **World**'s functions! The **World**'s *functions* pane provides an entirely different set of function-messages from those we can send to an object. If we scroll down a bit, we find the **NumberDialog** function that we can drag over to replace the **1** placeholder, as we saw in Figure 3-7. When we drop it on the **1**, Alice displays a menu of questions we can have the dialog box ask, as shown in Figure 3-16.

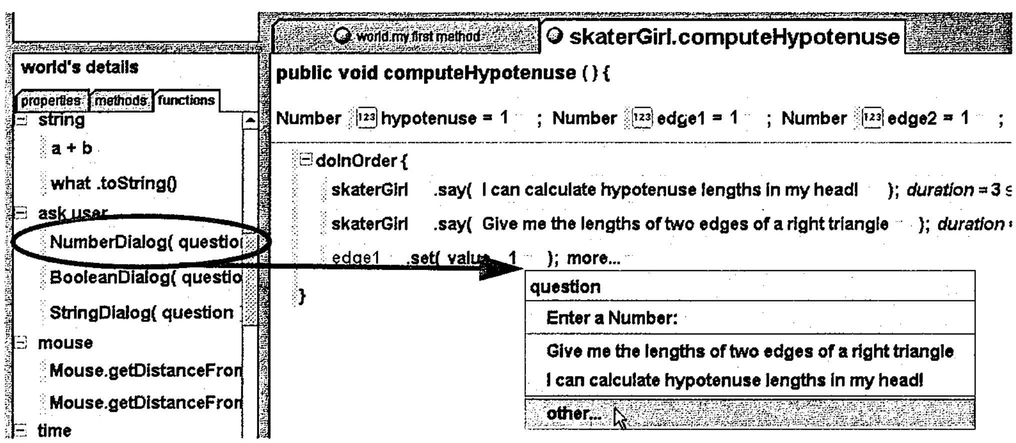


FIGURE 3-16 Dragging a dialog function

In this case, we want the dialog box to ask for the length of one edge of a right triangle, so we choose **other...** Alice then lets us enter the prompt to be displayed, as shown in Figure 3-17.

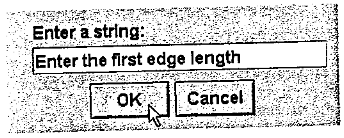


FIGURE 3-17 Customizing a dialog box's prompt message

This yields the **set()** message shown at the bottom of Figure 3-18.

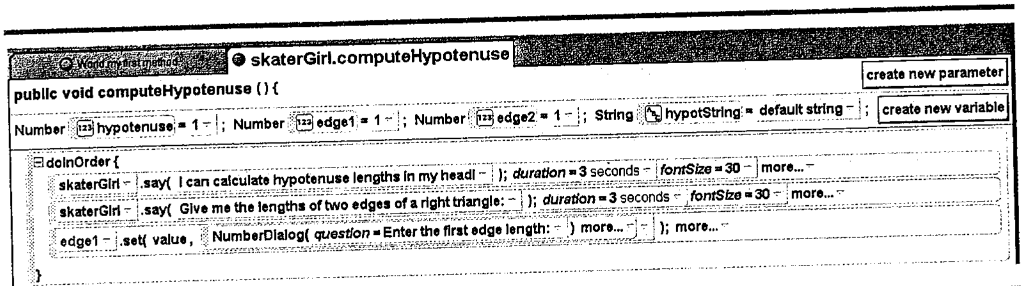


FIGURE 3-18 Setting a variable to a dialog box's result

Now, when the program flows through the **set()** message, it will send **World** the **NumberDialog()** message, which will display a dialog box asking the user to enter the first edge length. When the user enters a number in that dialog box, the **NumberDialog()** function will return that number, which the **set()** method will then use to set the value of **edge1**.

We can use a similar approach to get the value for **edge2**, and once we have the two edge lengths, we are ready to compute the **hypotenuse** value. We get as far as shown in Figure 3-19 before we hit a snag.

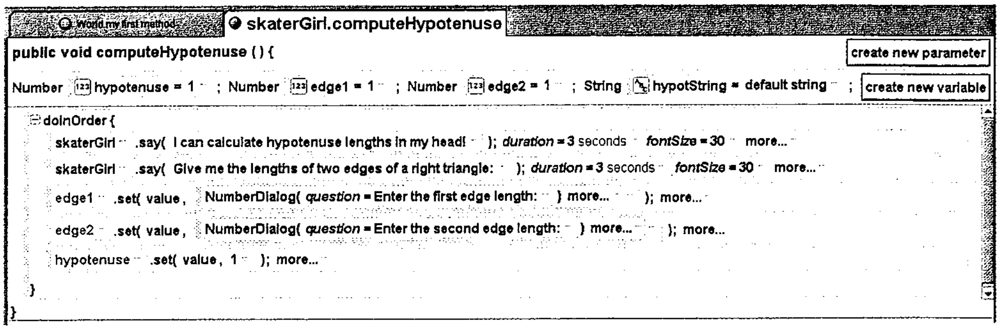


FIGURE 3-19 How to compute the hypotenuse

Looking back at the Pythagorean Theorem, we see that we need the square root function. Like the dialog box function, square root is available in the **world**'s *functions* pane, under the *advanced math* category. We thus drag and drop **Math.sqrt()** to replace the placeholder **1** in the **set()** message. From there, we can use the list arrow, the **expressions** menu choice, and the **math** menu choice several times to build the **set()** statement shown in Figure 3-20.

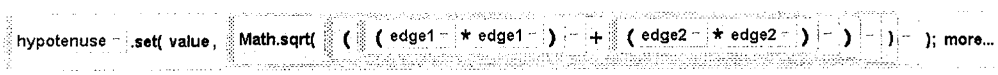


FIGURE 3-20 Computing the hypotenuse

Now that we have the **hypotenuse** calculated, how do we get the **skaterGirl** to say it? We can easily get her to say **"The hypotenuse length is "**, but how do we get her to say the value of **hypotenuse** at the same time? The answer has to do with *types*. As you know, the type of hypotenuse is **Number**. The type of the value we send with the **say()** message must be a **String**. Resolving this dilemma takes several steps.

The first step is to declare a new variable that will contain the value of **hypotenuse**, converted to a **String**. We'll call it **hypotString**, make its type **String**, and leave its initial value as **<None>**. We can then set its value to a placeholder value, like any other variable.

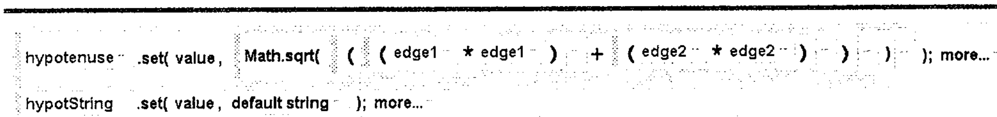


FIGURE 3-21 Converting the hypotenuse to a string (part I)

The next step is to use this variable to store a **String** representation of the (**Number**) value of **hypotenuse**. To do this, we go back to the **World**'s *functions* pane again, and under *string operations* we find a function named **toString()**. We drag this function into the **set()** statement to replace its **default string** value. When we drop it, Alice displays a menu from which we can choose **expressions -> hypotenuse** as the thing that we convert to a **String**. The result is the statement in Figure 3-22.

```
hypotString .set( value, hypotenuse .toString() ); more...
```

FIGURE 3-22 Converting the hypotenuse to a string (part II)

We now have a **String** version of the **hypotenuse**. The next step in the algorithm is for the **skaterGirl** to say "The hypotenuse length is X" where X is **hypotString**. To make this happen, we need a way to combine **"The hypotenuse is "** with **hypotString**. In programming, combining two strings **a** and **b** into a single string **ab** is called **concatenating** the strings, and for **String** values, the **+** sign is called the **concatenation operator**. In a concatenation **a + b**, the order of **a** and **b** matters: **"en" + "list"** makes **"enlist"**, but **"list" + "en"** makes **"listen"**.

We can start by having **skaterGirl** say the first part of what we want her to say: **"The hypotenuse length is "**. It doesn't show up well in Figure 3-23, but we must take care to leave a space after the word **is**, to separate it from the next part.

```
skaterGirl .say( The hypotenuse length is ); duration = 5 seconds fontSize = 30 more...
```

FIGURE 3-23 Converting the hypotenuse to a string (part III)

To make her also say the second part, we make a final trip to the **World**'s *functions* pane, from which we drag the other **String** function (**a+b**) onto **"The hypotenuse length is "** in the **set()** statement. When we drop the **a+b** function onto **"The hypotenuse length is "** in Figure 3-23, Alice takes the **String** that's there (**"The hypotenuse length is "**) as its **a** value. Alice then displays the menu we have seen before, from which we can select **expressions -> hypotString** as the (**a+b**) function's **b** value, as shown in Figure 3-24.

```
skaterGirl .say( The hypotenuse length is + hypotString ); duration = 5 seconds fontSize = 30 more...
```

FIGURE 3-24 Concatenating two strings

Since that is the last step, the method is done! The complete method is shown in Figure 3-25.

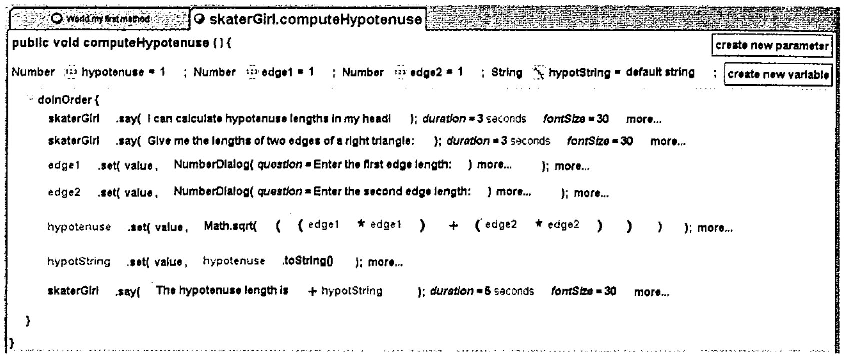


FIGURE 3-25 The `computeHypotenuse()` method (final version)

We then send **`skaterGirl`** the **`computeHypotenuse()`** message in **`my_first_method()`** to finish the program.

To test our work, we enter commonly known values. Figure 3-26 shows the result after we have entered edge lengths of 3 and 4 (the corresponding hypotenuse length is 5).

FIGURE 3-26 Testing `computeHypotenuse()`

Variables thus provide a convenient way to store values for later use in a program.

3.2 Parameters

A value that we pass to an object via a message is called an **argument**. While the word may be new to you, you have actually been using arguments ever since Chapter 1. For example, our very first program began with the code shown in Figure 3-27.

```
aliceLiddell.neck.head .pointAt( camera ); more...
aliceLiddell .say( Oh, hello there! ); duration = 2 seconds fontSize = 30 more...
```

FIGURE 3-27 Two statements from our first program

In the first statement, **camera** is an argument being passed to **aliceLiddell.neck.head** — the *value* at which **aliceLiddell.neck.head** should point. Each of the statements in Figure 3-27 has a single argument: **camera** (an **Object**) in the first statement, and **Oh, hello there!** (a **String**) in the second statement. Other methods we have seen require us to pass multiple arguments, as shown in Figure 3-28.

```
flappingDragon.left wing .roll( LEFT , .2 revolutions ); more...
flappingDragon.right wing .roll( RIGHT , 0.2 revolutions ); more...
```

FIGURE 3-28 The **roll()** message requires two arguments

Here, we see that the **roll()** message requires two arguments: the *direction* the object is to roll, and the *amount* it is to roll.

When you send an object a message accompanied by an argument, that argument must be stored somewhere so that the receiving object can access it.

A parameter is a variable that stores an argument, so that the receiver of the message can access it!

Thus, the **pointAt()** and **say()** methods each have a single parameter, while the **roll()** method has two parameters. There is no limit to the number of parameters a method can have.

To make all of this a bit more concrete, let's see some examples.

3.2.1 Example 1: Old MacDonald Had A Farm

Suppose we have a user story containing a scene in which a scarecrow is supposed to sing the song "Old MacDonald," one line at a time. Some of the lyrics to this song are below:

Old MacDonald had a farm, E-I-E-I-O. And on this farm he had a cow, E-I-E-I-O. With a moo-moo here, and a moo-moo there, here a moo, there a moo, everywhere a moo-moo. Old MacDonald had a farm, E-I-E-I-O.	*Old MacDonald had a farm, E-I-E-I-O. And on this farm he had a duck, E-I-E-I-O. With a quack-quack here, and a quack-quack there, here a quack, there a quack, everywhere a quack-quack. Old MacDonald had a farm, E-I-E-I-O.*
Old MacDonald had a farm, E-I-E-I-O. And on this farm he had a horse, E-I-E-I-O. With a neigh-neigh here, and a neigh-neigh there, here a neigh, there a neigh, everywhere a neigh-neigh. Old MacDonald had a farm, E-I-E-I-O.	*Old MacDonald had a farm, E-I-E-I-O. And on this farm he had a dog, E-I-E-I-O. With a ruff-ruff here, and a ruff-ruff there, here a ruff, there a ruff, everywhere a ruff-ruff. Old MacDonald had a farm, E-I-E-I-O.*

Subsequent verses introduce other farm animals (for example, chicken, cat, pig, etc.). For now, we will just have the character sing these four verses.

Clearly, we *could* use divide-and-conquer to have the scarecrow sing four verses; in each verse we send the scarecrow five **`say()`** messages. For example, **`singVerse1()`** would contain statements like these:

```
scarecrow.say("Old MacDonald had a farm, E-I-E-I-O.");
scarecrow.say("And on this farm he had a cow, E-I-E-I-O.");
scarecrow.say("With a moo-moo here and a moo-moo there,");
scarecrow.say("here a moo, there a moo, everywhere a moo-moo.");
scarecrow.say("Old MacDonald had a farm, E-I-E-I-O.");
```

However, this approach has several disadvantages. One is that if later we want to add a fifth verse, then we must write a new method, containing five more **`say()`** messages, and add it to the program. With this approach, every new verse we want the scarecrow to sing will require a new method containing five more statements. This seems like a lot of repetitious work.

A related disadvantage of this approach is that each verse-method we write is identical, except for (1) the animal, and (2) the noise it makes.

Whenever you find yourself programming the same thing more than once, there is usually a better way to write the program.

In this case, the better way is to write a single "generic" **`singVerse()`** method, to which we can pass a given animal and its noise as arguments. That is, we want a message like this:

```
scarecrow.singVerse("cow", "moo");
```

to make the scarecrow sing the first verse; a message like this:

```
scarecrow.singVerse("horse", "neigh");
```

to make him sing the second verse, and so on.

The trick to making this happen is to build a method with a generic *animal* parameter to store whatever animal we want to pass, and a generic *noise* parameter to store the noise it makes. The statements of this method then contain the lyrics that are common to each verse, but using the *animal* parameter in place of the specific cow, duck, horse, or dog; and using the *noise* parameter in place of the specific moo, quack, neigh, or ruff.

Assuming we have created a world containing a **`scarecrow`** (from Alice's Web Gallery) and whatever other farm-related objects we desire, we can start by creating a new **`scarecrow`** method named **`singVerse()`**. With this method open, we click the **`create new parameter`** button we saw back in Figure 3-1. When clicked, this button generates a **`Create New Parameter`** dialog box similar to the **`Create New Local Variable`** dialog box we saw in Figure 3-3. As in that dialog box, we can specify the *name* of the parameter and its *type*. When we click this dialog box's **`OK`** button, it defines a new parameter with the given *name* and *type* between the method's parentheses. In Figure 3-29, we have used this button to create the **`animal`** and **`noise`** parameters.

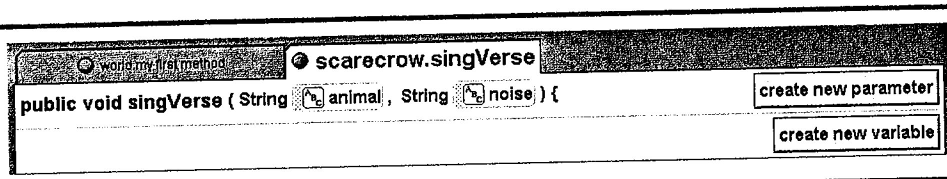


FIGURE 3-29 Parameters for animal and noise

With the parameters defined, we can proceed to add statements to the method to make the scarecrow sing a verse. Like a variable, a parameter's name appears in the **`expressions`** menu choice that appears when we drag and drop a statement into the method. Figure 3-30 shows one way we might define the **`singVerse()`** method.

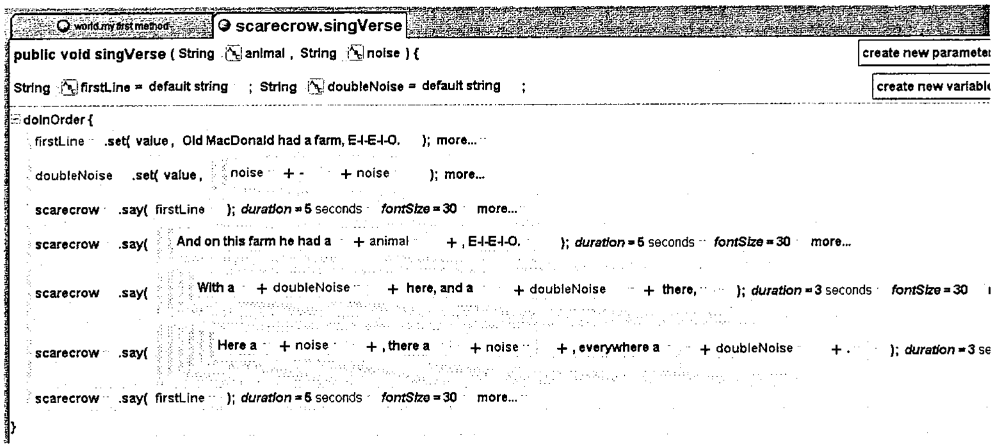


FIGURE 3-30 The `singVerse()` method

Recognizing that the first and last lines are the same, we defined a variable named **`firstLine`** to store those lines, so that we need not write them twice. Also, seeing that a verse uses the string *noise-noise* three times, we defined a variable named **`doubleNoise`**, and defined its value as **`noise + "-" + noise`**, using the string concatenation operator (+) we saw in the last section. In fact, we used the concatenation operator *14 times* in building this method, most often in the statements in which the scarecrow sings the 3rd and 4th lines of the verse.

Given this method, we can now define a **`singOldMacDonald()`** method quite simply (Figure 3-31).

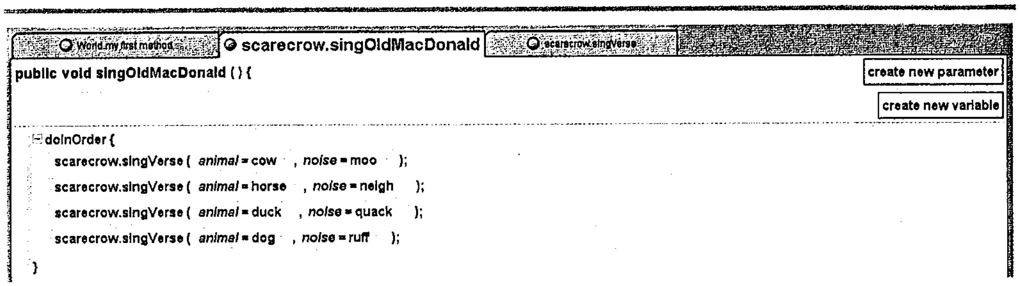


FIGURE 3-31 The `singOldMacDonald()` method

Figure 3-32 presents the program running, partway through its third verse.

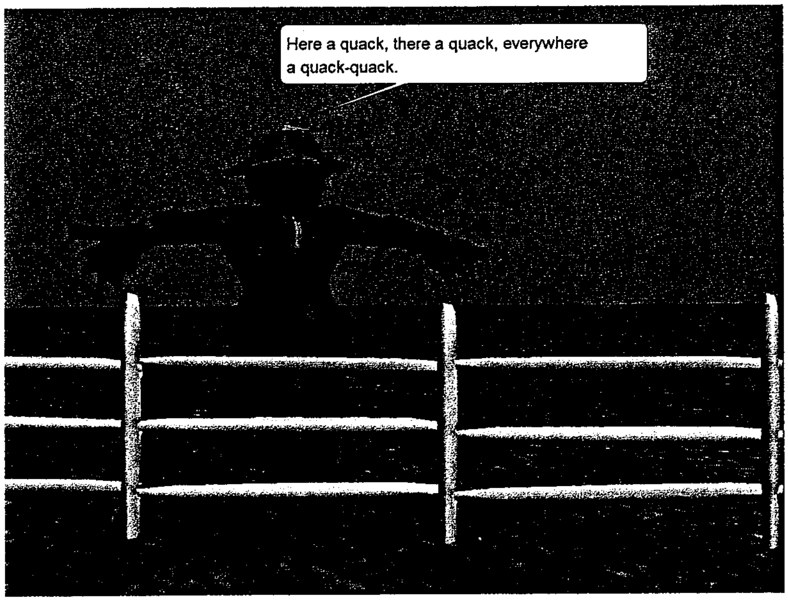


FIGURE 3-32 Testing `singOldMacDonald()`

If we should subsequently decide to add a new verse, doing so is as easy as sending the **`scarecrow`** another **`singVerse()`** message, with the desired *animal* and *noise* arguments.

3.2.2 Example 2: Jumping Fish!

Suppose we have a user story in which a fish jumps out of the water, tracing a graceful arc through the air before re-entering the water. If we examine the various fish classes in the Alice Gallery, none of them offers a **`jump()`** method that solves the problem. Choosing one that will contrast with the water, we will define a **`jump()`** method for the **`Pinkminnow`** class.

If we think about what kinds of arguments we might want to pass to a **`jump()`** message, one possibility is the *distance* we want the fish to jump. Another possibility would be the *height* we want it to jump. (These are very different behaviors, as indicated by there being separate *high jump* and *long jump* events in track and field.) In this section, we will have the fish do the equivalent of the long jump, and pass the *distance* we want it to jump.

If we think through the behavior this method should provide, we might sketch it as the sequence of steps shown in Figure 3-33.

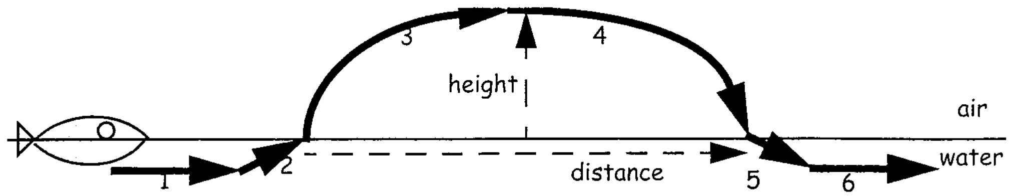


FIGURE 3-33 Sketching a fish's jumping behavior

We can write out these steps as an algorithm as follows:

1 fish swims forward a starting distance (to get its speed up)
2 fish angles upward
3 fish moves upward the height and half the distance, angling upward
4 fish moves downward the height and forward half the distance, angling downward
5 fish angles upward (levels off)
6 fish swims forward a stopping distance (coasting to a stop)

If we consider how an animal jumps, when an animal jumps a short distance, it doesn't spring very high; but if it jumps a longer distance, it springs higher. The height and distance of an animal's jump are thus related. For the sake of simplicity, we will approximate the height as 1/3 of the distance. (If this proves too simplistic, we can always change it.) Similarly, if a fish is to jump farther, it may need a longer starting distance to get its speed up, and the distance it glides before it stops will be greater. For simplicity's sake, we will assume that the starting and stopping distances are 1/4 of the distance to be jumped.

Using our algorithm and our sketch, we might identify these objects: fish, height, distance, half the distance, angle, starting distance, and stopping distance. We have already selected the **Pinkminnow** class for the fish. Since we intend to pass the distance to be jumped as an argument, and such a value is numeric, we will create a **Number** parameter to store this value using the **create new parameter** button. The remaining objects are all numeric values, so we will define a **Number** variable for each of them, using the **create new variable** button we saw in Figure 3-1. We will use the names **height**, **halfDist**, and **angle** for three of these objects. If we assume that the starting and stopping distances are the same, we can use one variable for both, which we will name **`startStopDist`**, as shown in Figure 3-34.

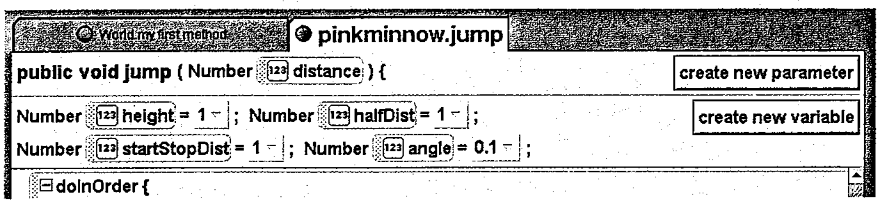


FIGURE 3-34 The `jump()` parameter and variables

Given our algorithm and these variables, building the method consists of setting their values appropriately, and then dragging the right statements into the method to elicit the behavior required by our algorithm. Figure 3-35 shows the completed definition.

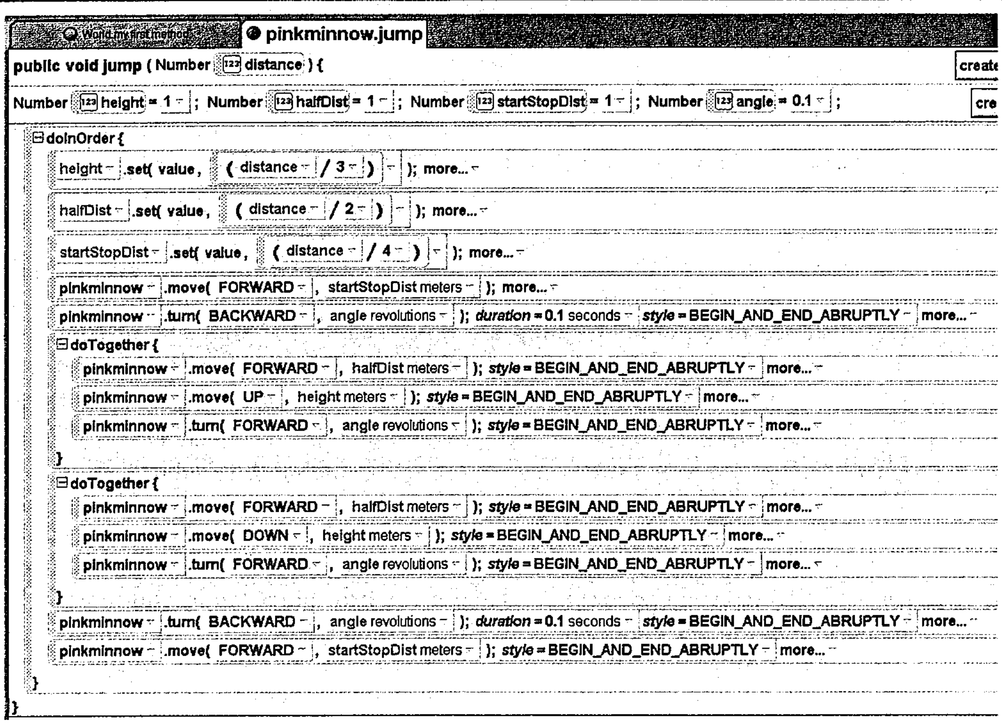


FIGURE 3-35 The `jump()` method (complete)

We can see in Figure 3-35 that each variable's value is accessed multiple times. One of the benefits of using variables this way is that if we later decide to change a value (for example the height of the jump, or its angle), we only have to change it in one place, instead of in several places. This can be a big time-saver when you are using trial-and-error to find just the right value.

To test our program, we send **`pinkminnow`** the **`jump()`** message. To test it thoroughly, we use a variety of argument values (for example, 0.25, 0.5, 1, 2, ...), to check that its behavior is appropriate in each case. Figure 3-36 shows a test using one of these values.

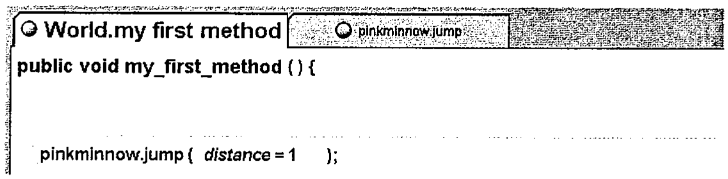


FIGURE 3-36 Testing the `jump()` method

Figure 3-37 is a montage of snapshots, showing the behavior produced by the **`jump()`** method.

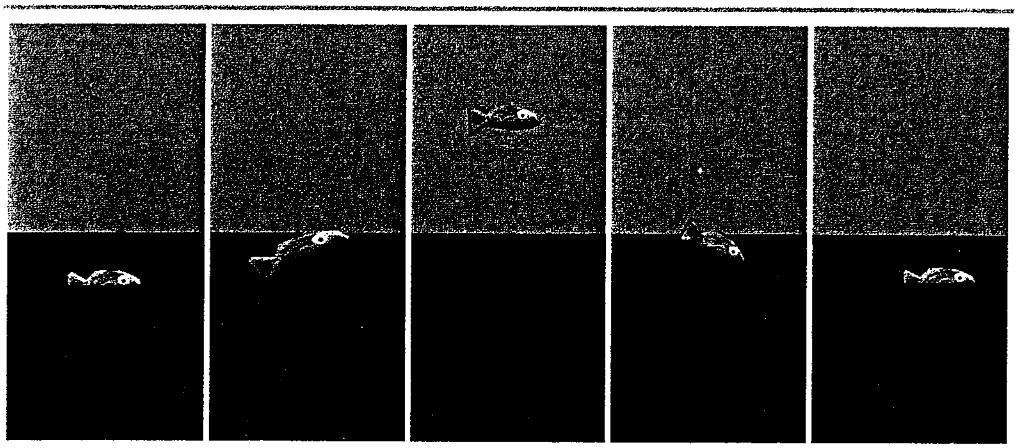


FIGURE 3-37 A jumping fish

Parameters are thus variables through which we can pass arguments to a method. By passing different arguments to the same method, that method can produce different (related) behaviors. For example, the **`singVerse()`** method allows the **`scarecrow`** to sing different verses of the same song, depending on what *animal* and *noise* values we pass it. Similarly, the **`jump()`** method makes the **`pinkminnow`** jump different distances, depending on what *distance* we pass it.

The key to using parameters well is to anticipate that you will want to pass different values to the method as arguments, and then create a parameter to store such values. A well-written method with parameters is like a stone that (figuratively speaking) lets you kill multiple birds.

3.3 Property Variables

Now that we have seen method variables and parameters, it is time to take a brief look at Alice's third kind of variable: **object variables**, which are also known as **instance variables** or **properties**. Whereas method variables and parameters are defined within a method, an object variable is defined within an object. More precisely, an object variable is defined within the *properties* pane of an object's *details area*.

An object variable allows an object to *remember* one of its properties. Each object has its own variable for the property, in which it can store a value distinct from any other object.

To clarify this, let's look at a concrete example. Suppose a user story calls for twin wizards named *Jim* and *Tim*, and each wizard needs to know his own name. One way to make this happen is to add a **wizard** to our world and define within it an object variable whose name is **myName**, whose type is **String**, and whose value is **"Jim"**. If we then make a copy of the **wizard**, the new wizard will have its own **myName** variable, whose value we can change to **"Tim"**.

To define an object variable in the wizard, we click on **wizard** in the *object tree*, click the *properties* tab in the *details area*, and then click the **create new variable** button we see there[1], as shown in Figure 3-38.

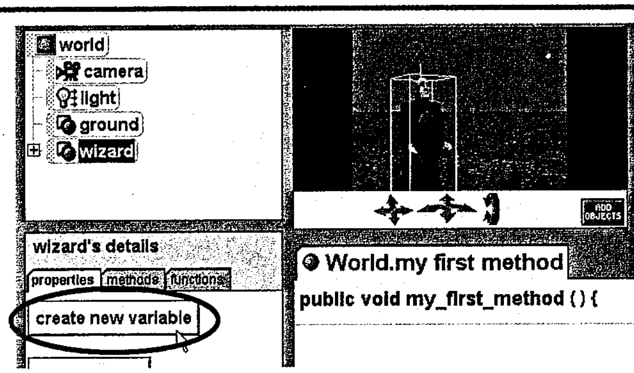


FIGURE 3-38 The properties pane's `create new variable` button

1. Just below the **create new variable** button is a **capture pose** button. When pressed, this button saves the object's current *pose* (the positions+orientations of its subparts) in a new property variable of type **Pose**. If you want to pose your character manually before running your program, this button lets you save such poses. You can use the **setPose()** method within your program to change an object's pose to a saved pose. (The **getCurrentPose()** function can be used to retrieve an object's current pose while your program is running.)

Clicking this button causes the **create new variable** dialog box to appear, which is almost identical to the **Create New Local Variable** dialog box we saw back in Figure 3-3. In it, we enter **myName** for the name, select **Other -> String** as its type, and enter **Jim** for its value. When we click the dialog box's **OK** button, Alice creates a new **String** variable named **myName** whose value is **Jim** in the wizard's *properties* pane, as shown in Figure 3-39.

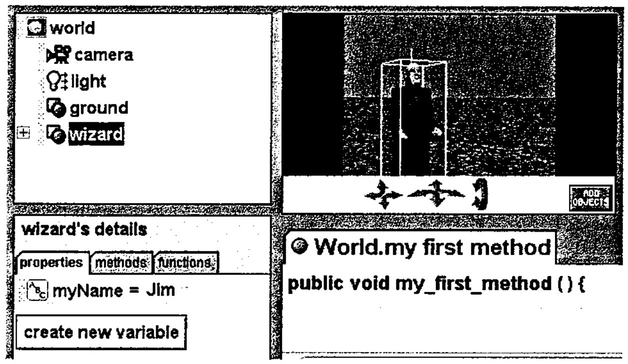


FIGURE 3-39 A new property variable

To make the wizard's twin, we can use the **copy** button (the rightmost control in the **Add Objects** window), as was covered in the Alice Tutorial. Copying the wizard this way gives us two wizards named **Jim**, so we close the **Add Objects** window, click on the second wizard in the *object tree*, click the *properties* tab in the *details area*, and there change the value of the new wizard's **myName** property from **Jim** to **Tim**. See Figure 3-40.

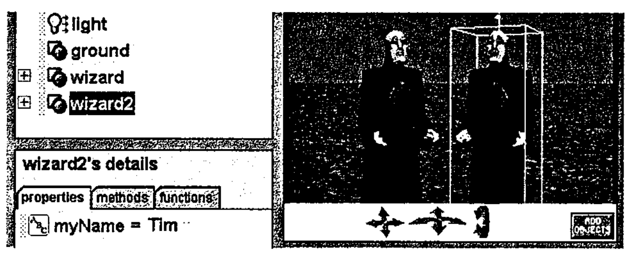


FIGURE 3-40 Twin wizards

A program can now access each wizard's name, as shown in Figure 3-41.

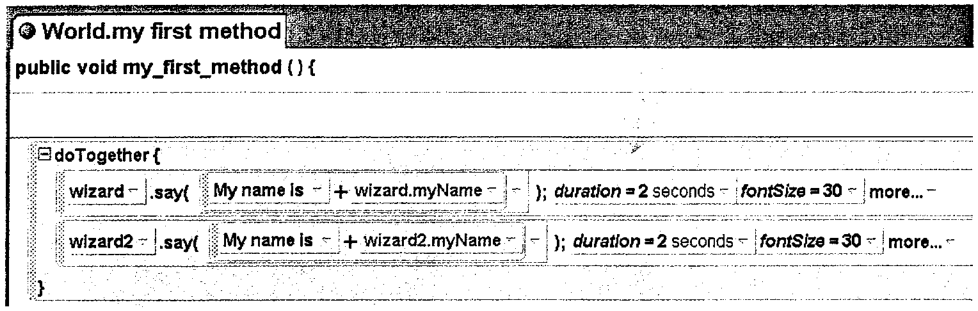


FIGURE 3-41 Accessing property variables

When we click Alice's **`Play`** button, we see that each wizard "knows" his own name, as shown in Figure 3-42.

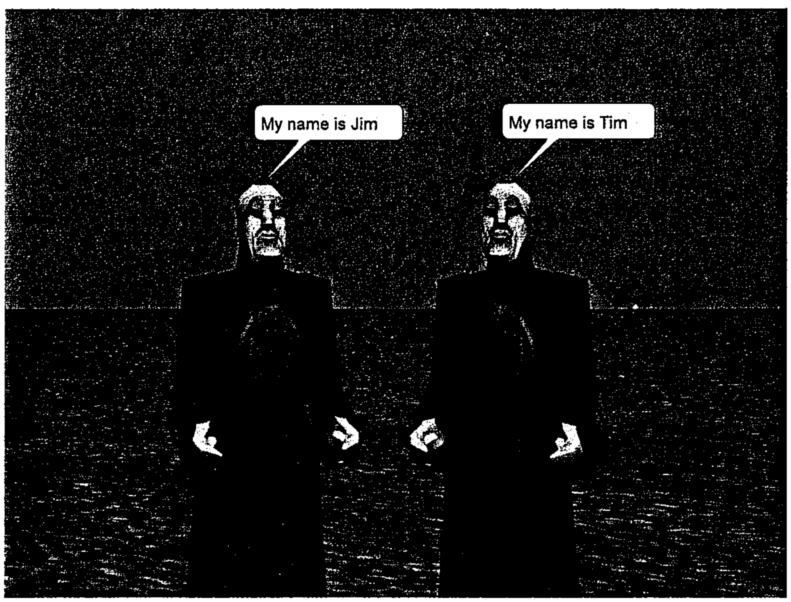


FIGURE 3-42 The twin wizards introduce themselves

A property variable thus provides a place for us to store an *attribute* of an object, such as its name, its size, its weight, and anything else we want an object to know about itself.

As we have seen, each Alice object has a number of predefined property variables. These variables store the object's **`color`** (essentially a filter through which we see the object), its **`opacity`** (what percentage of light the object reflects), its **`vehicle`** (what can move this object?), its **skinTexture** (the graphical appearance of the object), its **fillingStyle** (how much of the object gets drawn), its **pointOfView** (the object's position and orientation), and its **isShowing** property (whether or not the object is visible). If you have not done so already, take the time to experiment with each of these properties, to get a feel for what role each plays.

In the next section of this chapter, we will take a closer look at the **vehicle** property.

3.4 Alice Tip: Using the Vehicle Property

In some user stories, it may be desirable to **synchronize** the movements of two objects, so that when one of the objects moves, the other moves with it. To illustrate, let us return to the example from Section 3.1.1, in which Scene 2 had a girl approaching a horse. Suppose that Scene 4 calls for her to ride the horse across the screen. We might set the scene as shown in Figure 3-43.

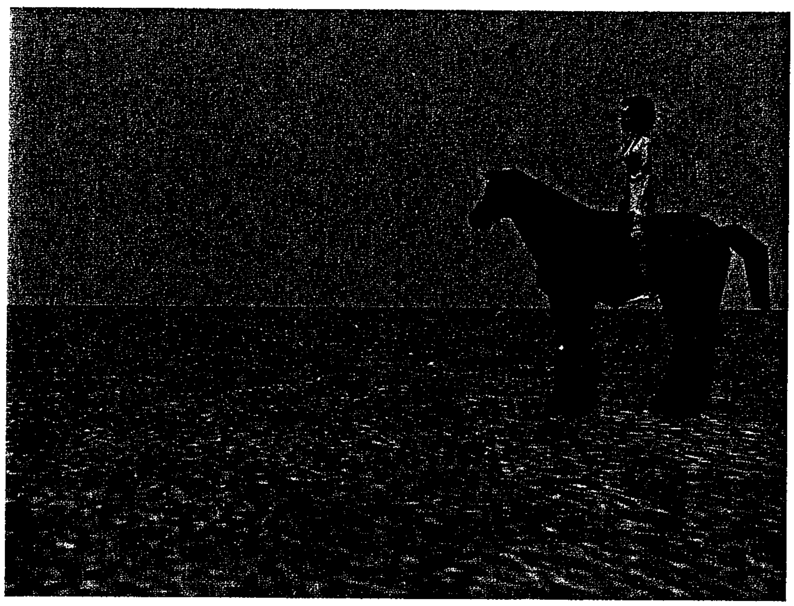


FIGURE 3-43 The girl on the horse

With the girl on the horse, we can use a **move()** message to move the horse across the screen (Figure 3-44).

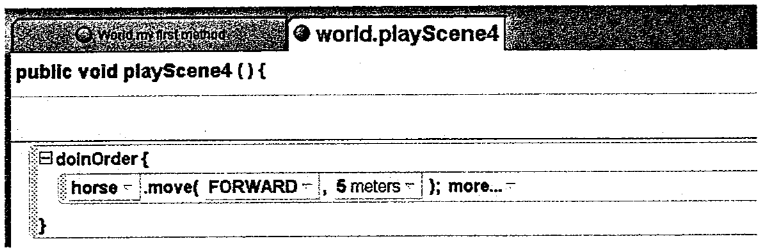


FIGURE 3-44 Moving the horse across the screen

However, as shown in Figure 3-45, when we do so, the horse moves, leaving the girl hanging suspended in mid-air!

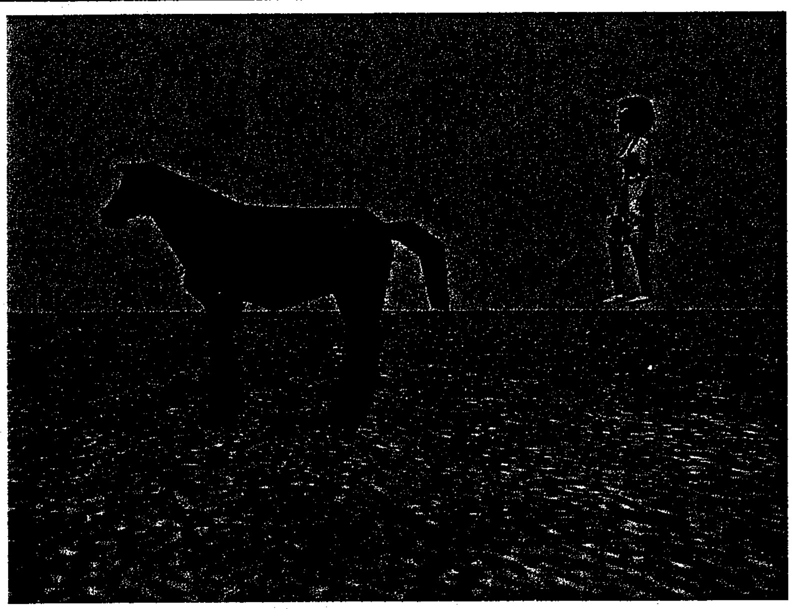


FIGURE 3-45 Moving the horse leaves the girl hanging

We could solve this problem using a **doTogether** block, in which we make the girl and the horse move together. But doing so would force us to write twice as many statements anytime we wanted her to ride the horse, and the additional statements to move the girl will be virtually identical to those we are using to move the horse. It would be much better if we could somehow make the girl "ride" the horse, so that if the horse moves, the girl moves with it.

The way to achieve this better solution is by using the **vehicle** property. As its name implies, an object's **vehicle** is the thing on which it "rides," which is by default, the **world**. If we want the girl to ride the horse, we need to change her **vehicle** property. This can be done by setting her **vehicle** property (using the approach we saw back in Section 1.5.1) at the beginning of the scene, as shown in Figure 3-46.

```
World.my first method | world.playScene4

public void playScene4 ( ) {

  doInOrder {
    nativeGirl   .set( vehicle , horse   ); more...
    horse   .move( FORWARD   , 5 meters   ); more...
  }
```

FIGURE 3-46 Changing the girl's `vehicle` property

As soon as we have made this change, playing the scene causes the girl to "ride" the horse across the screen, as shown in Figure 3-47.

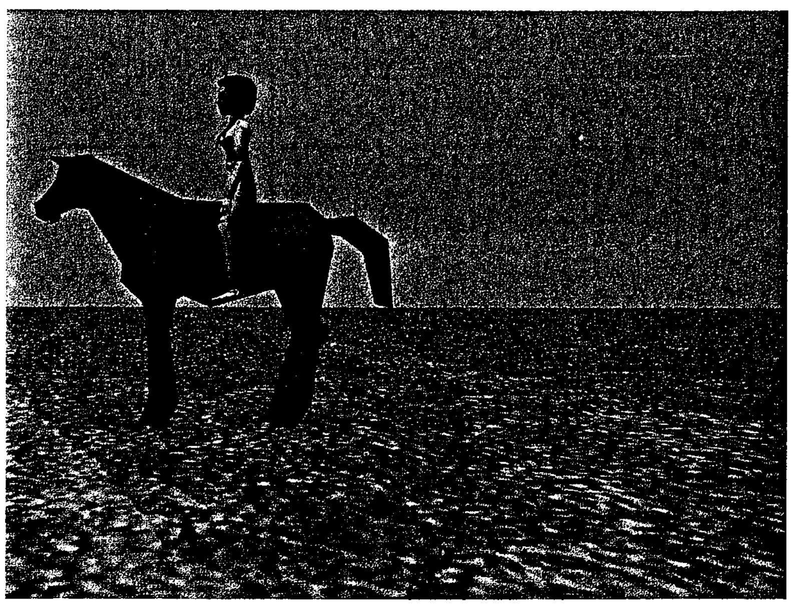


FIGURE 3-47 The girl rides the horse across the screen

By setting the **vehicle** of the girl to the horse, any **move()** messages we send to the horse will cause her to move as well, effectively synchronizing her movements with those of the horse.

Note that if a subsequent scene calls for the girl and the horse to move independently, we will need to reset her **vehicle** to be the **world**. If we neglect to do this, then **move()** messages we send to the horse will make her move too, since their movements will still be synchronized.

3.5 Functions

We have seen how to use a function to send an object a message in order to get information from it. Suppose we wanted to be able to get information from an object, but there was no predefined function providing that information? In such circumstances, we can define our own function.

3.5.1 Example: Retrieving an Attribute From an Object

Let us return to the twin wizards we met in Section 3.3. Suppose that in addition to their names, the wizards have titles that, together with their names, they use on formal occasions. For example, suppose that the wizard *Jim* goes by the title *The Enchanter*, while the wizard *Tim* goes by the title *The Magus*. (Yes, these sound pretentious to me, too.) It should be evident that we can use the same approach we used in Section 3.3 to define a second property variable for each of the wizards to store his title. We will name this property variable **myTitle**, and define it to be of type **String**. Once we have defined this property, we can set its value to the appropriate value in each of the wizards, as shown in Figure 3-48.

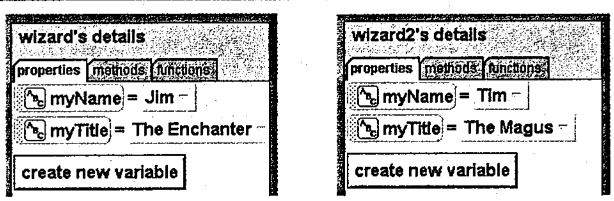


FIGURE 3-48 The wizards' **myTitle** properties

Now suppose that, at times, we need to access a wizard's name, at other times we need to access a wizard's title, and at other times we need to access a wizard's full name (that is, title plus name). In the first case, we can retrieve the wizard's name using the **myName** property. In the second case, we can retrieve the wizard's title using the **myTitle** property. But how can we access the wizard's full name?

One approach would be to concatenate **myTitle** and **myName** with a space in between:

```
wizard.say("I am " + myTitle + " " + myName);
```

This approach is okay, so long as we don't have to access the full name very often. If we have to access it frequently, it can get tiresome to have to repeatedly rebuild the wizard's full name. In such a situation, we can define a function that, when sent to a wizard, produces his full name as its value. To do so, we select the wizard in the *object tree*, click the *functions* tab in the *details area*, and then click the **create new function** button, as shown in Figure 3-49.

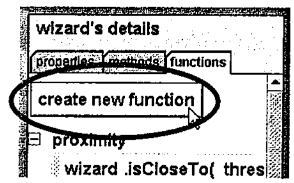


FIGURE 3-49 The `create new function` button

Alice then displays a **New Function** dialog box in which we can enter the name of the function and select the type of value it should produce. We will call the function *getFullName*, and the value it produces is a **String**, as shown in Figure 3-50.

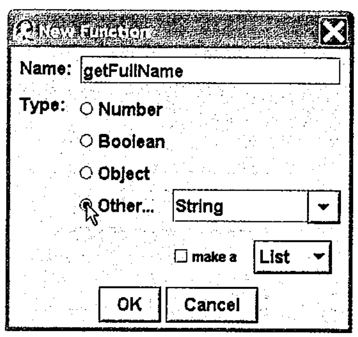


FIGURE 3-50 The `New Function` dialog box

When we click the **OK** button, Alice adds the new function to the wizard's *functions* in the *details area* and opens this function in the *editing area*, as shown in Figure 3-51.

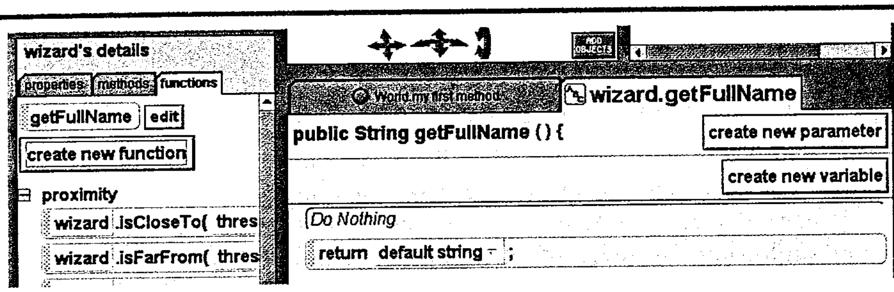


FIGURE 3-51 An "empty" string-returning function

Unlike an Alice method (which produces no value), *a function produces a value*. The value the function produces is whatever value appears in the function's **`return`** statement, whose form is:

```
return Value ;
```

When Alice performs this statement, the function produces ***`Value`***, sending it back to the place from which the function-message was sent. Note that when Alice defines an "empty" function, it supplies the **`return`** statement with a default ***`Value`*** appropriate for the function's type.

Figure 3-52 shows one way we could define the function.

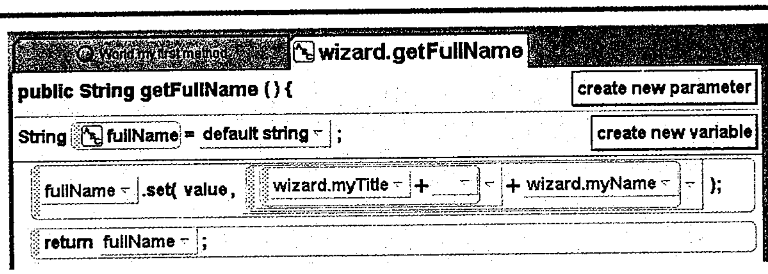


FIGURE 3-52 The **`getFullName()`** function (version 1)

This approach uses a local variable named **`fullName`** to store the computation of concatenating the wizard's title, a space, and the wizard's name, and then returns the value of **`fullName`**. Alternatively, we can eliminate the local variable and just return the value produced by the concatenation operators. Figure 3-53 uses this approach to define **`getFullName()`** for **`wizard2`**.

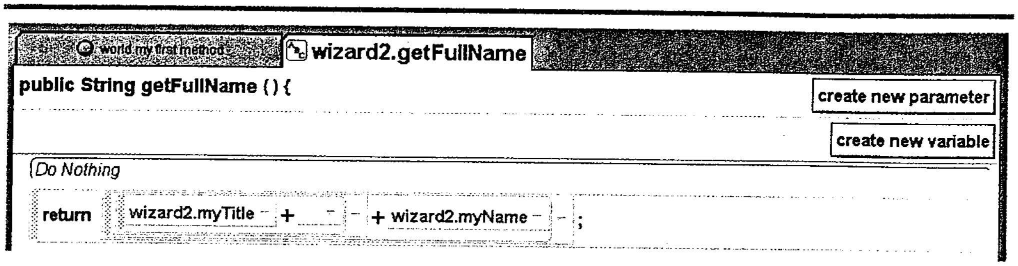


FIGURE 3-53 The `getFullName()` function (version 2)

This version is equivalent to that in Figure 3-52, but it requires no local variable. Now, we can use these functions in a program like that shown in Figure 3-54.

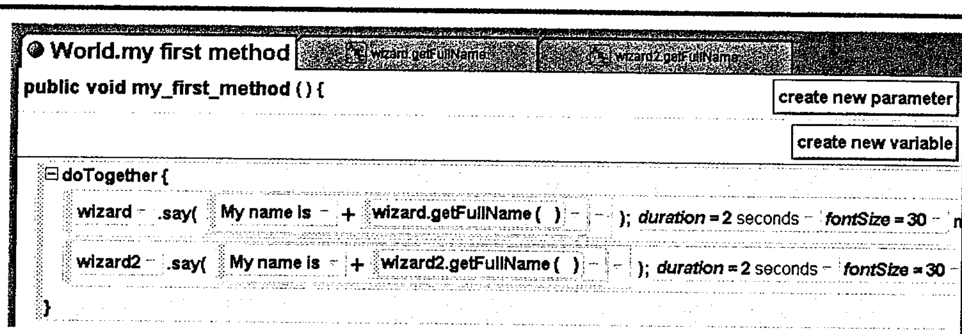


FIGURE 3-54 The wizards introduce themselves

The behavior these functions produce can be seen in Figure 3-55.

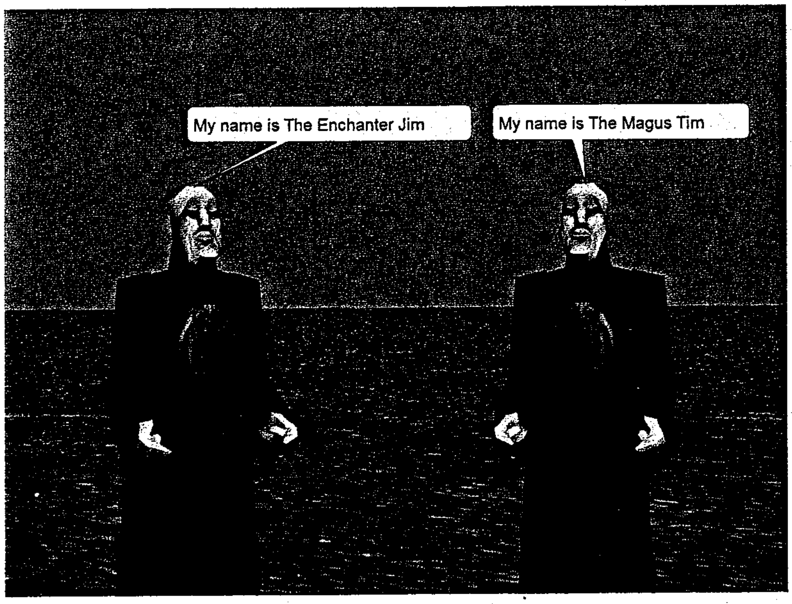


FIGURE 3-55 The wizards introducing themselves

3.5.2 Functions With Parameters

Like methods, functions can have parameters to store arguments passed by the sender of the message. The arguments can then be accessed through the parameters. To illustrate, recall that in Section 3.1.2, we built a world in which **skaterGirl** could compute hypotenuse-lengths in her head. The method we wrote there inputs values for the two leg lengths, computes the hypotenuse, and then outputs the result. There might be situations where we just want to calculate the numerical hypotenuse-length, without the input or output:

```
hypotenuse.set(value, skaterGirl.calculateHypotenuse(3, 4) );
```

To define such a function, we make sure **skaterGirl** is selected in the *object tree*, click the *functions* tab in the *details area*, and then click the **create new function** button as before. When the **New Function** dialog box appears, we enter its name (*calculate-Hypotenuse*), but this time we select **Number** as the type of value it produces, as shown in Figure 3-56.

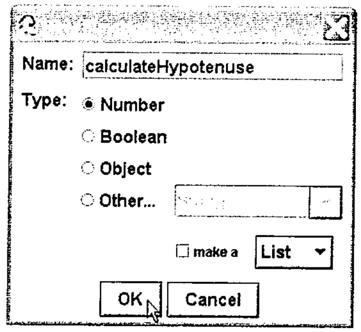


FIGURE 3-56 Creating a number-producing function

When we click the **OK** button, Alice adds **calculateHypotenuse** to **skaterGirl**'s *functions* in the *details area*, and opens the new function in the *editing area*, as shown in Figure 3-57.

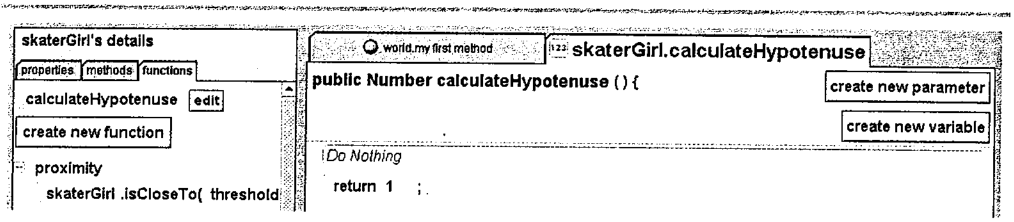


FIGURE 3-57 An empty number-returning function

To store whatever arguments the sender of this message passes for the two leg lengths, we need two parameters, which we can make using the function's **create new parameter** button. This displays a dialog box like the one shown in Figure 3-3, in which we can enter a parameter's name and its type. Doing this for each of the two parameters gives us the function shown in Figure 3-58.

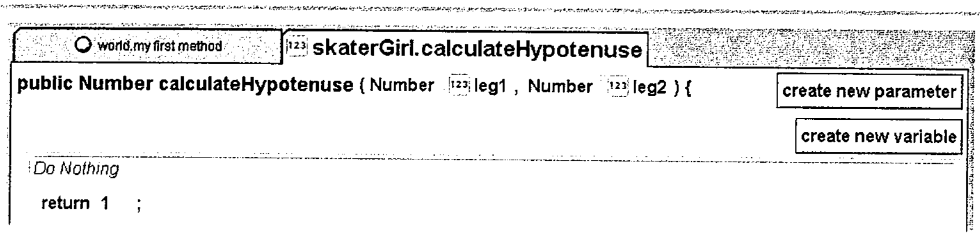


FIGURE 3-58 A function with parameters

To finish the function, we add the necessary operations to make it compute the hypotenuse length using its parameters. Figure 3-59 shows one way to do so.

```
world.my first method | skaterGirl.calculateHypotenuse
public Number calculateHypotenuse ( Number leg1 , Number leg2 ) {
Number hypot = 1 ;
  hypot .set( value, Math.sqrt( ( ( leg1 * leg1 ) + ( leg2 * leg2 ) ) ) );
  return hypot ;
```

FIGURE 3-59 Calculating the hypotenuse

Given this function, we can now send **`skaterGirl`** the **`calculateHypotenuse()`** message, and pass it arguments for the leg lengths. Figure 3-60 shows a revised version of Figure 3-25.

```
world.my first method | skaterGirl.computeHypotenuse
public void computeHypotenuse ( ) {
Number hypotenuse = 1 ; Number edge1 = 1 ; Number edge2 = 1 ; String hypotString = default string ;
  doInOrder {
    skaterGirl .say( I can calculate hypotenuse lengths in my head! ); duration = 3 seconds fontSize = 30 more...
    skaterGirl .say( Give me the lengths of two edges of a right triangle: ); duration = 3 seconds fontSize = 30 more...
    edge1 .set( value, NumberDialog( question = Enter the first edge length ) more... ); more...
    edge2 .set( value, NumberDialog( question = Enter the second edge length ) more... ); more...
    hypotenuse .set( value, skaterGirl.calculateHypotenuse( leg1 = edge1 , leg2 = edge2 ) ); more...
    hypotString .set( value, hypotenuse .toString() ); more...
    skaterGirl .say( The hypotenuse length is + hypotString ); duration = 5 seconds fontSize = 30 more...
  }
}
```

FIGURE 3-60 Sending a function-message

When this program is performed, it prompts the user to enter the lengths of the two triangle legs, and then **`skaterGirl`** "says" the corresponding hypotenuse length. For example, if the user enters **`3`** and **`4`** for the leg lengths, the program behaves as shown in Figure 3-61.

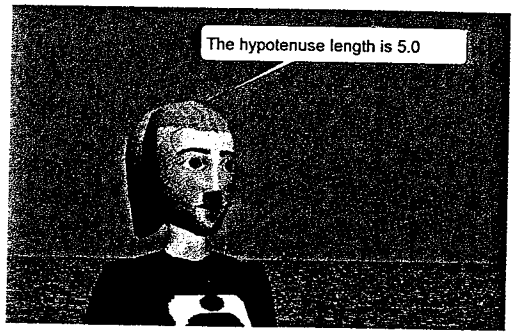


FIGURE 3-61 Testing the function

Functions are thus much like methods. We can create parameters and local variables within each of them, and perform just about any computation we can envision. The difference between the two is that a function-message returns a value to its sender, while a method-message does not. Because of this difference, a function-message must be sent from a place where a *value* can appear, such within a **`set()`** statement. By contrast, a method-message can only be sent from a place where a *statement* can appear.

Being able to define messages — both method and function — is central to object-based programming. In the chapters to come, we will see many more examples of each.

3.6 Chapter Summary

- Method variables let us store computed and user-entered values for later use.
- Parameters let us store and access arguments passed by the sender of a message.
- Properties (object variables) let us store and retrieve an object's attributes.
- Alice's **`vehicle`** property lets us synchronize the movements of two objects.
- A function lets us send a message to an object, and get a value in response.

3.6.1 Key Terms

argument
concatenation
define a variable
function
initial value
local variable
method variable
object variable
parameter
placeholder
property variable
`return` statement
synchronized movements
variable
variable name
variable type
vehicle
world functions

Programming Projects

3.1 Following the approach used in Section 3.1.1, build a scene containing two people who walk toward each other from opposite sides of the screen. When they meet, they should turn and walk off together toward a building, and enter the building when they get there.

3.2 Using the horse we used in Section 3.4, build a **`gallop()`** method for the horse that makes its legs move realistically through the motions for one stride of a gallop. Then modify the **`playScene4()`** method so that the horse gallops across the screen. (For now, you may send the **`gallop()`** message multiple times.)

3.3 Using the **`heBuilder`** or **`sheBuilder`**, build a person. For your person, define an object method named **`walkInSquare()`** that has a parameter named **`edgeLength`**. When **`walkInSquare(dist)`** is sent to your person, he or she should walk in a square with edges that are each ***`dist`*** meters long. Make certain your person begins and ends at the same spot. When the person is done, have the person say the area and perimeter of the square.

3.4 Using the ideas in this chapter, build a world containing a person who can calculate Einstein's formula $e = m^*c^2$ in his or her head, where the user enters the m value (mass, in kilograms), and c is the speed of light (299,792,458 meters per second). Define descriptive variables for each quantity, and use the **`World`** function **`pow()`** to compute c^2.

3.5 Choose a hopping animal from the Alice Gallery (for example, a frog, a bunny, etc.). Write a **`hop()`** method that makes it hop in a realistic fashion, with a parameter that lets the sender of the message specify how far the animal should hop. Using your **`hop()`** method, have your animal hop around a building in four hops.

3.6 *The Farmer in the Dell* is an old folk song with the lyrics below. Create an Alice program containing a character who sings this song. Use a **`singVerse()`** method, parameters, and variables to write your program efficiently.

The farmer in the dell. The farmer in the dell. Heigh-ho, the derry-o. The farmer in the dell.	*The farmer takes a wife. The farmer takes a wife. Heigh-ho, the derry-oh. The farmer takes a wife.*
The wife takes a child. The wife takes a child. Heigh-ho, the derry-oh. The wife takes a child.	*The child takes a nurse. The child takes a nurse. Heigh-ho, the derry-oh. The child takes a nurse.*
The nurse takes a cow. The nurse takes a cow. Heigh-ho, the derry-oh. The nurse takes a cow.	*The cow takes a dog. The cow takes a dog. Heigh-ho, the derry-oh. The cow takes a dog.*

The dog takes a cat. *The dog takes a cat.* *Heigh-ho, the derry-oh.* *The dog takes a cat.*	*The cat takes a rat.* *The cat takes a rat.* *Heigh-ho, the derry-oh.* *The cat takes a rat.*
The rat takes the cheese. *The rat takes the cheese.* *Heigh-ho, the derry-oh.* *The rat takes the cheese.*	*The cheese stands alone.* *The cheese stands alone.* *Heigh-ho, the derry-oh.* *The cheese stands alone.*

3.7 Using the **heBuilder** or **sheBuilder** (or any of the other persons in the Alice Gallery with enough detail), build male and female persons and add them to your world. Using your persons, build a program in which your people dance the waltz (or a similar dance in which the partners' movements are synchronized). Have your world play music while your people dance.

3.8 Build a world in which two knights on horseback joust, using the techniques from this chapter.

3.9 In Section 2.4, we developed Scene 2 of a program, in which a wizard confronts three trolls. Write a wizard method **castChangeSizeSpell(*obj*, *newSize*)**, that takes an object ***obj*** and a number ***newSize*** as arguments. The method should cause the wizard to turn towards ***obj***, raise his arms, say a magic word or phrase, and then lower his arms. The method should resize ***obj*** the amount specified by ***newSize***, and then make certain ***obj*** is standing on the ground. Create a scene 3 in which the wizard uses the **castChangeSizeSpell()** message to defeat the trolls by shrinking most of them to 1/10 their original size.

3.10 Alice provides the **Pose** type, which can be used to store the position of each of an object's subparts. Under the *properties* pane, the **capture pose** button allows you to save an object's current **Pose** in a property variable before the program is run. The function named **getCurrentPose()** can be used (as the value of a **set()** message) to save an object's pose in a **Pose** variable as the program is running. The **setPose()** method can be used to set an object's pose to a pose stored in a **Pose** variable. Rewrite the **march()** method we wrote in Section 2.2.2. Discard the **moveLeftLegForward()** and **moveRightLegForward()** methods we used, using three **Pose** variables instead.

Chapter 4
Flow Control

Controlling complexity is the essence of computer programming.

BRIAN KERNIGHAN

When you get to the fork in the road, take it.

YOGI BERRA

If you build it, he will come.

THE VOICE (JAMES EARL JONES), IN *FIELD OF DREAMS*

While you're at it, why don't you give me a nice paper cut and pour some lemon juice on it?

MIRACLE MAX (BILLY CRYSTAL), IN *THE PRINCESS BRIDE*

Objectives

Upon completion of this chapter, you will be able to:

- Use the **`Boolean`** type and its basic operations
- Use the **`if`** statement to perform some statements while skipping others
- Use the **`for`** and **`while`** statements to perform (other) statements more than once
- Use **`Boolean`** variables and functions to control **`if`** and **`while`** statements
- Use the **`wait()`** message to temporarily suspend program execution

In Chapter 1, we saw that the flow of a program is the sequence of steps the program follows in performing a story. From the perspective of an Alice program, we can think of a flow as the sequence of statements that are performed when we click the **Play** button.

In the preceding chapters, the programs we have written have mostly used the **doInOrder** statement, which produces a sequential execution. However, we sometimes used a **doTogether** statement, which produces a parallel execution. If we consider a group of *N* statements within a **doInOrder** statement compared to a **doTogether** statement, we can visualize the difference in behavior of these two statements in a **flow diagram** like the one shown in Figure 4-1.

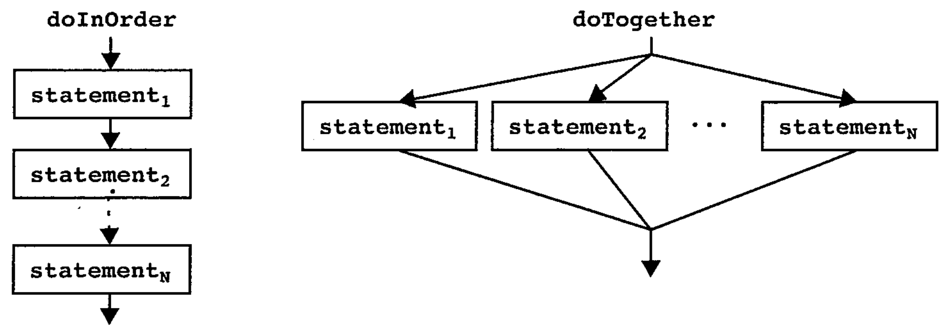


FIGURE 4-1 The flows produced by the **doInOrder** and **doTogether** statements

The **doInOrder** and **doTogether** are thus **flow control** statements, because their effect is to *control the flow* of the program through the statements within them. Computer scientists often describe flow control statements as **control structures.**

In this chapter, we will examine several of Alice's flow control statements, including the following:

- the **if** statement, which directs the flow through one group of statements and away from another group of statements
- the **for** statement, which directs the flow through a group of statements a fixed number of times
- the **while** statement, which directs the flow through a group of statements an arbitrary number of times

Before we examine these statements, let's briefly look at a related topic: the **Boolean** type.

4.1 The Boolean Type

You may recall from Chapter 3 that **Boolean** is one of Alice's basic types (for defining variables). The **Boolean** type is named after George Boole, a 19th century English mathematician who studied *true/false* values and the kinds of operations that can be used with them.

Whereas a **Number** variable can have any of millions of (numeric) values, and an **Object** variable can refer to any Alice object, a **Boolean** variable can have either of just two values: **true** or **false**. At first, this may seem rather limiting: what good is a type that only provides two values? As we shall see, the **Boolean** type is extremely useful when we want the program to make decisions. Decision-making depends on current circumstances or *conditions*, so a piece of a program that produces a **true** or **false** value is called a **boolean expression** or **condition.**

4.1.1 Boolean Functions

The *functions* pane of Alice's *details area* contains questions we can ask an object. When the answer to a question is **true** or **false**, the function is a condition. Many of the questions we can ask an object produce a **Boolean** value for their answer, including those shown in Figure 4-2.

Function	Value Produced
***obj*.isCloseTo(*dist*, *obj2*)**	**true**, if ***obj2*** is within ***dist*** meters of ***obj***; **false**, otherwise.
***obj*.isFarFrom(*dist*, *obj2*)**	**true**, if ***obj2*** is at least ***dist*** meters away from ***obj***; **false**, otherwise.
***obj*.isSmallerThan(*obj2*)**	**true**, if ***obj2***'s volume exceeds that of ***obj***; **false**, otherwise.
***obj*.isLargerThan(*obj2*)**	**true**, if ***obj***'s volume exceeds that of ***obj2***; **false**, otherwise.
***obj*.isNarrowerThan(*obj2*)**	**true**, if ***obj2***'s width exceeds that of ***obj***; **false**, otherwise.
***obj*.isWiderThan(*obj2*)**	**true**, if ***obj***'s width exceeds that of ***obj2***; **false**, otherwise.
***obj*.isShorterThan(*obj2*)**	**true**, if ***obj2***'s height exceeds that of ***obj***; **false**, otherwise.
***obj*.isTallerThan(*obj2*)**	**true**, if ***obj***'s height exceeds that of ***obj2***; **false**, otherwise.

FIGURE 4-2 Boolean functions

continued

Function	Value Produced
obj.isToTheLeftOf(*obj2*)	**true**, if *obj*'s position is beyond *obj2*'s left edge; **false**, otherwise.
obj.isToTheRightOf(*obj2*)	**true**, if *obj*'s position is beyond *obj2*'s right edge; **false**, otherwise.
obj.isAbove(*obj2*)	**true**, if *obj*'s position is above *obj2*'s top edge; **false**, otherwise.
obj.isBelow(*obj2*)	**true**, if **obj**'s position is below *obj2*'s bottom edge; **false**, otherwise.
obj.isInFrontOf(*obj2*)	**true**, if *obj*'s position is before *obj2*'s front edge; **false**, otherwise.
obj.isBehind(*obj2*)	**true**, if *obj*'s position is beyond *obj2*'s rear edge; **false**, otherwise.
obj.isToTheLeftOf(*obj2*)	**true**, if *obj*'s position is beyond *obj2*'s left edge; **false**, otherwise.

FIGURE 4-2 Boolean functions *(continued)*

Note that most of these functions refer to an object's bounding box. For example, the function ***obj*.isBehind(*obj2*)** uses the rear edge of ***obj2***'s bounding box.

These functions can be used with an **if** or **while** statement (see below) to make a decision or otherwise control an object's behavior.

4.1.2 **Boolean** Variables

Another kind of condition is the **Boolean** variable or parameter. **Boolean** variables, parameters, or properties can be created by clicking the appropriate **create new variable** (or **parameter**) button, and then specifying **Boolean** as the type of the new variable (or parameter). Such variables can be used to store **true** or **false** values until they are needed, and can serve as a condition in an **if** or **while** statement, which we describe below.

4.1.3 Relational Operators

Another kind of condition is produced by an *operator* that computes a **true** or **false** value. The six most common operators that produce **Boolean** values are called the **relational operators**, and they are shown in Figure 4-3.

Relational Operator	Name	Value Produced
`val1 == val2`	equality	**true**, if **val1** and **val2** have the same value; **false**, otherwise.
`val1 != val2`	inequality	**true**, if **val1** and **val2** have different values; **false**, otherwise.
`val1 < val2`	less-than	**true**, if **val1** is less than **val2**; **false**, otherwise.
`val1 <= val2`	less-than-or-equal	**true**, if **val1** is less than or equal to **val2**; **false**, otherwise.
`val1 > val2`	greater-than	**true**, if **val1** is greater than **val2**; **false**, otherwise.
`val1 >= val2`	greater-than-or-equal	**true**, if **val1** is greater than or equal to **val2**; **false**, otherwise.

FIGURE 4-3 The relational operators

In Alice, the six relational operators are located in the *functions* pane of the **world**'s *details area*. These are most often used to compare **Number** values. For example, suppose a person is to receive overtime pay if he or she works 40 hours or more in a week. If **hoursWorked** is a **Number** variable in which a person's weekly working hours are stored, then the condition

```
hoursWorked > 40
```

will produce **true** if the person should receive overtime pay, and **false** if he or she should not. Relational operators compare two values and produce an appropriate **true** or **false** value.

Beyond numeric values, the equality (==) and inequality (!=) operators can be used to compare **String**, **Object**, and **Other** values. We will see an example of this in Section 4.2.

4.1.4 Boolean Operators

The final three conditional operators are used to combine or modify relational operations. These are called the **boolean operators**, and they are shown in Figure 4-4.

Boolean Operation	Name	Value Produced
`val1 && val2`	AND	**true**, if **val1** and **val2** are both **true**; **false**, otherwise.
`val1 || val2`	OR	**true**, if either **val1** or **val2** is **true**; **false**, otherwise.
`!val`	NOT	**true**, if **val** is **false**; **false**, if **val** is **true**.

FIGURE 4-4 The boolean operators

Like the relational operators, Alice provides the boolean operators in the *functions* pane of the **World**'s *details area*. To illustrate their use, suppose we want to know if a person is a teenager, and their age is stored in a **Number** variable named **age**. Then the condition

```
age > 12 && age < 20
```

will produce the value **true** if the person is a teenager; otherwise it will produce the value **false**. Similarly, suppose that a valid test score is in the range 0 to 100, and we want to guard against data-entry mistakes. If the score is in a **Number** variable named **testScore**, then we can decide if it is invalid with the condition

```
testScore < 0 || testScore > 100
```

since the condition will produce **true** if either **testScore < 0** or **testScore > 100** is **true**, but will produce **false** if neither of them is **true**.

Now that we have seen the various ways to build a condition, let's see how we can make use of them to control the flow of a program.

4.2 The `if` Statement

4.2.1 Introducing Selective Flow Control

Suppose we have a user story in which the following scene occurs:

Scene 3: A princess meets a mute dragon, and says "Hello." The dragon just looks at her. She asks it, "Can you understand me?" The dragon shakes its head up and down to indicate *yes*. She says, "Can you speak?" The dragon shakes its head sideways to indicate *no*. She says, "Can you only answer yes or no questions?" The dragon shakes its head *yes*. She says, "Are you a tame dragon?" The dragon shakes its head *no*.

The co-star of the scene is a mute dragon, who answers yes-or-no questions by shaking his head up and down for *yes*, and shaking it sideways for *no*. We could write two separate **dragon** methods, one named **shakeHeadYes()**, and another named **shakeHeadNo()**. Instead, let's "kill two birds with one stone" and write one **shakeHead()** method providing both behaviors.

As we saw in Chapter 3, the key to making one method do the work of two (or more) is to use a parameter to produce the different behaviors. In this case, we will pass the argument **yes** when we want the dragon to shake its head up and down, and pass the argument **no** when we want it to shake its head sideways. To store this argument, we will need a parameter whose type is **String**. For lack of a better name, we will name the parameter **yesOrNo**.

If we write out the behavior this method should produce, we might write the following:

Parameter: yesOrNo, a String.

If *yesOrNo* is equal to "yes", the dragon shakes his head up and down;

Otherwise, the dragon shakes his head sideways.

The key idea here is that if the parameter has one value, we want one thing to happen; otherwise, we want something else to happen. That *if* is the magic word. Any time we use the word *if* to describe a desired behavior, we can use Alice's **if statement** to produce that behavior.

To build this method in Alice, we might start by opening a world, adding a **playScene3()** method to the world, adding a dragon to the world; positioning the camera so that we can see the dragon's head clearly; selecting **dragon** in the *object tree*; creating a new method named **shakeHead()**; and then within this method, creating a new parameter named **yesOrNo**, whose type is **String**. The result is shown in Figure 4-5.

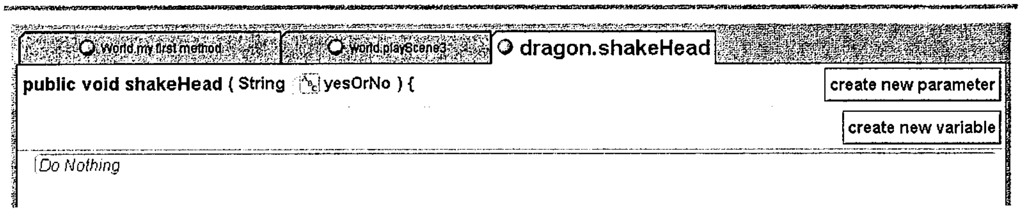


FIGURE 4-5 The empty `shakeHead()` method

Looking at the algorithm for this method, we see the magic word *if*. There is a control named **if** at the bottom of Alice's *editing area*, so we drag it into the method. When we drop it, Alice produces a **condition** menu, with the choices **true** or **false**, as shown in Figure 4-6.

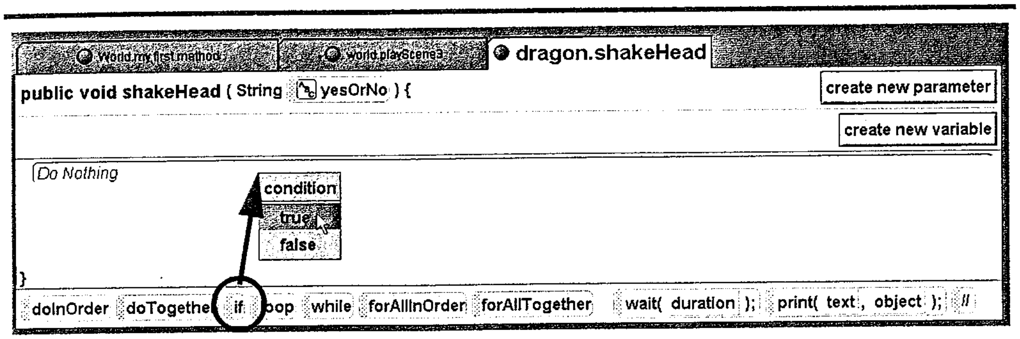


FIGURE 4-6 Dragging the `if` control

For the moment, we will just choose **`true`** as a *placeholder* value. Alice then generates an **`if`** statement in the method, as shown in Figure 4-7.

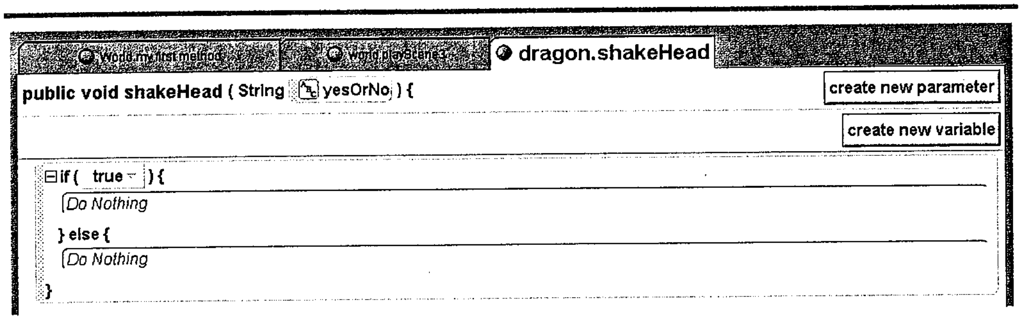


FIGURE 4-7 The Alice `if` statement

4.2.2 `if` Statement Mechanics

An **`if`** statement is a flow control statement that directs the flow according to the value of a condition. Alice's **`if`** statement has the following structure:

```
if ( Condition ) {
   Statements1
} else {
   Statements2
}
```

and we might visualize the **`if`** statement's flow-behavior as shown in Figure 4-8.

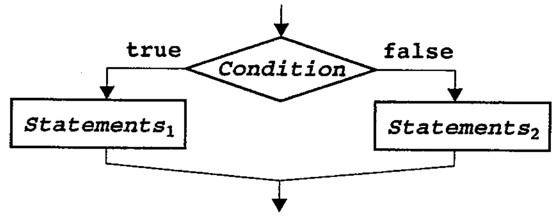


FIGURE 4-8 Flow through an `if` statement

Figure 4-8 shows that when the flow reaches an **`if`** statement, it reaches a "fork" in its path. Depending on its ***`Condition`***, the flow proceeds one way or the other, but not both. That is, when the flow first reaches an **`if`** statement, its ***`Condition`*** is evaluated. If the value of the ***`Condition`*** is **`true`**, then the flow is directed through the first group of statements (and the second group is ignored); if the ***`Condition`***'s value is **`false`**, then the flow is directed through the second group of statements (ignoring the first group). Put differently, when the **`if`** statement's ***`Condition`*** is **`true`**, then the first group of statements is *selected* and the second group is skipped; otherwise, the second group of statements is *selected* and the first group is skipped. The **`if`** statement's behavior is sometimes called **selective flow**, or **selective execution**.

4.2.3 Building `if` Statement Conditions

Back in the user story, we want the dragon to shake its head up and down if **`yesOrNo`** is equal to **`yes`**; otherwise, it should shake its head sideways. We saw in Figure 4-3 that the **equality operator** is ==, so that is what we need. To use it, we can click on the **`yesOrNo`** parameter, drag it into the *editing area*, and drop it on the placeholder in the **`if`** statement's condition. Alice will display a menu from which we can choose **`yesOrNo ==`** , followed by a second menu from which we can choose the **`b`**-value, as shown in Figure 4-9.

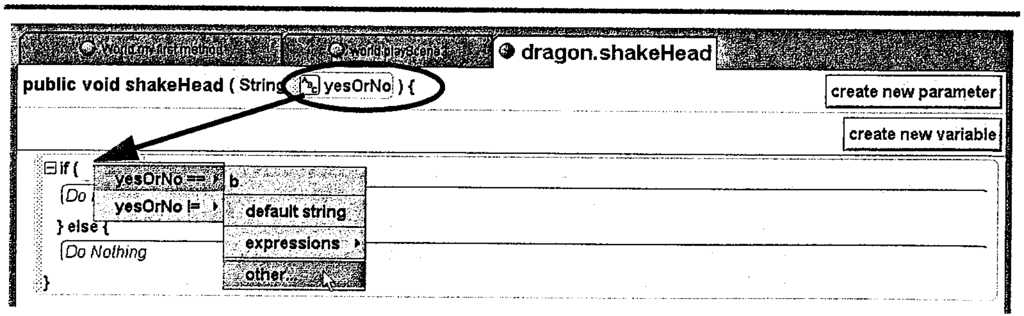


FIGURE 4-9 Dragging a parameter to an `if` statement's condition

Choosing **other** for the **b**-value produces a dialog box into which we can type **"yes"**. When we click its **OK** button, Alice generates the condition shown in Figure 4-10.

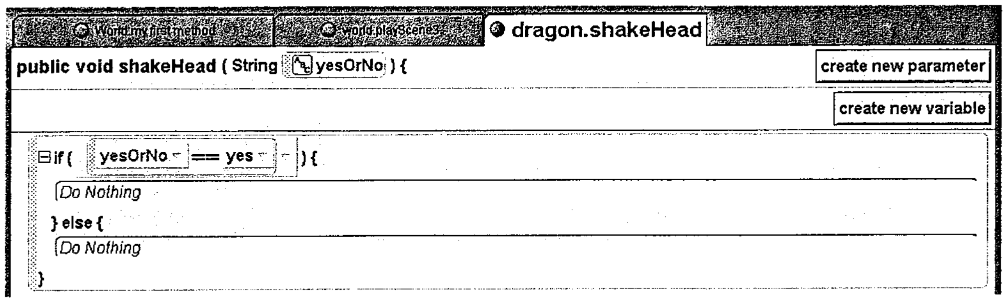


FIGURE 4-10 An `if` statement's condition using a parameter

With the condition in place, finishing the method consists only of placing messages in the top *Do Nothing* area to shake the dragon's head up and down, and placing messages in the bottom *Do Nothing* area to shake its head sideways. Figure 4-11 shows the finished method.

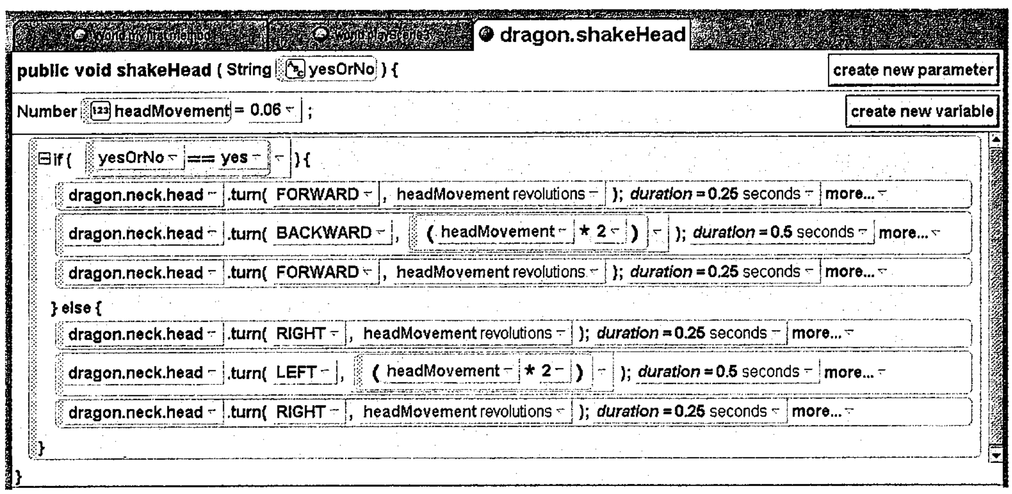


FIGURE 4-11 The `dragon.shakeHead()` method (final version)

In Figure 4-11, we used a local **Number** variable named **headMovement** to store how far the dragon turns his head. By using it in each of the **turn()** messages instead of actual numbers, we simplify the task of finding the right amount by which the dragon should shake his head, since trying a given value only requires one change (to the variable) instead of six changes.

To test the **shakeHead()** method, we build the scene method, as shown in Figure 4-12.

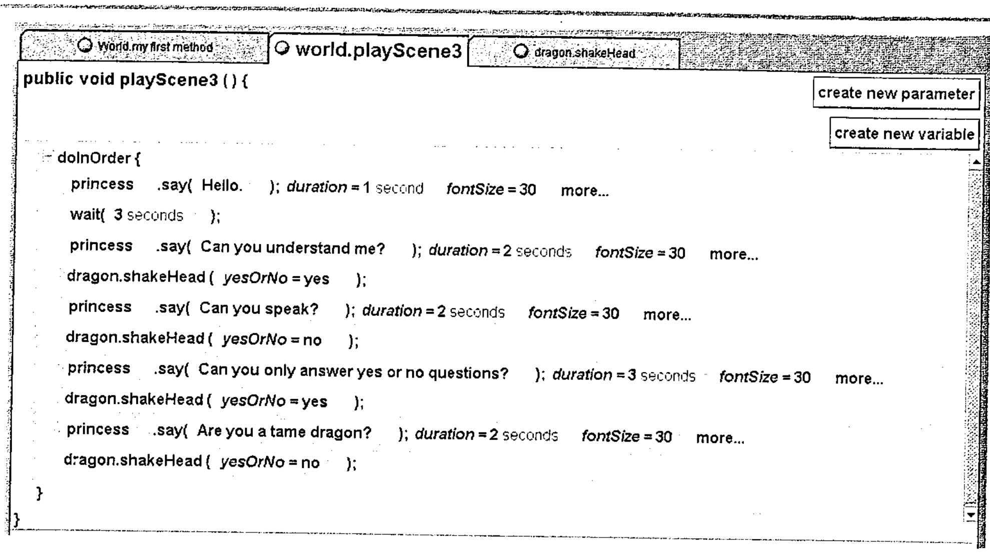


FIGURE 4-12 Testing `shakeHead()` in `playScene3()`

When we click Alice's **Play** button, we see that the **shakeHead()** method works as intended, as shown in Figure 4-13.

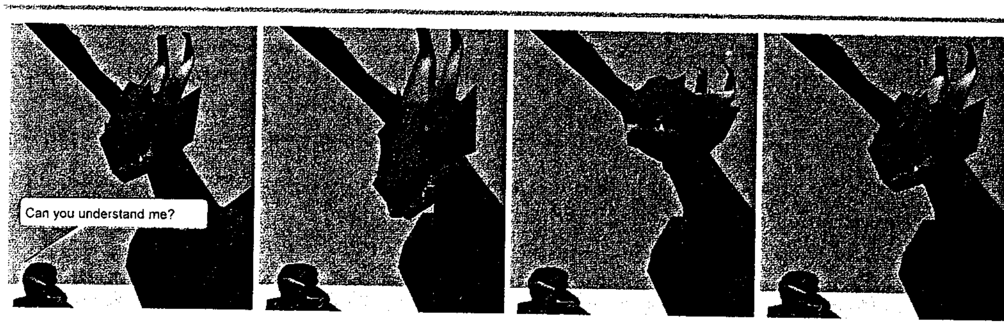


FIGURE 4-13 Testing the `shakeHead()` method

4.2.4 The `wait()` Statement

To introduce a time delay between the princess's first and second statements in Figure 4-12, we used another flow control statement named **wait()**, whose form is as follows:

```
wait(numSecs);
```

When the flow reaches this statement, Alice *pauses* the program's flow, sets an internal timer to **numSecs** seconds, and starts this timer counting down towards zero. When the timer reaches zero, Alice *resumes* the program's flow at whatever statement follows the **wait()**.

4.2.5 Validating Parameter Values

In the previous example, we saw how the **if** statement can be used to direct the flow of a program through one group of statements while bypassing another group, where each group of statements was equally valid. A different use of the **if** statement is to *guard* a group of statements, and only allow the flow to enter them if "everything is ok."

To illustrate, let us return to the jumping fish example from Section 3.2. There, we built a method for the **Pinkminnow** class named **jump()**, with a parameter named **distance** to which we could pass an argument indicating how far we wanted the fish to jump. Something we did not discuss in Section 3.2 was whether or not there are any *restrictions* or *preconditions* on the value of this argument (that is, limitations to how *far* the fish can jump). This situation — where a parameter's value needs to be checked for validity before we allow the flow to proceed — is called **validating the parameter.**

If we assume that the fish can only jump forward, then one easy restriction is that the argument passed to **distance** must be positive. We can check this with the condition **distance > 0**. Passing an argument that is 0 or less can be treated as an error.

There may also be an upper bound on how far a **PinkMinnow** can jump, but identifying such a bound is more difficult. Minnows are rather small fish, so 2 meters might be a reasonable upper bound. However if a minnow were bigger than normal, or were super-strong, maybe it could jump farther, so we want to make this upper bound easy to change. We can do so by defining a variable named **MAX_DISTANCE**, and then using the condition **distance <= MAX_DISTANCE** to check that the argument passed to parameter **distance** is within this bound.

If a variable's value will not change, and its purpose is to improve a program's readability, name it with all uppercase letters, to distinguish it from normal variables.

We now have two conditions that need to be met in order for the argument passed to the parameter to be deemed valid: **distance > 0** and **distance <= MAX_DISTANCE**. Since *both* of these must be true in order for our argument to be acceptable, we use the boolean AND operator (**&&**) to combine them: **distance > 0 && distance <= MAX_DISTANCE**.

We will use these ideas to revise the **jump()** method, as follows:

```
if (distance > 0 && distance <= MAX_DISTANCE) {
   // ... statements performed when distance is valid
   // (make the fish jump)
} else {  // ... distance is invalid
   if (distance <= 0) {
      // ... statements performed when distance is too low
   } else {
      // ... statements performed when distance is too high
   }
}
```

Here, we are using an **if** statement with a second **if** statement nested within its **else** statements. The first **if** is often called the **outer if**, and the second **if** is often called the **inner if**, or the **nested if**.

Figure 4-14 presents a revised version of the **jump()** method, using this approach to validate the parameter.

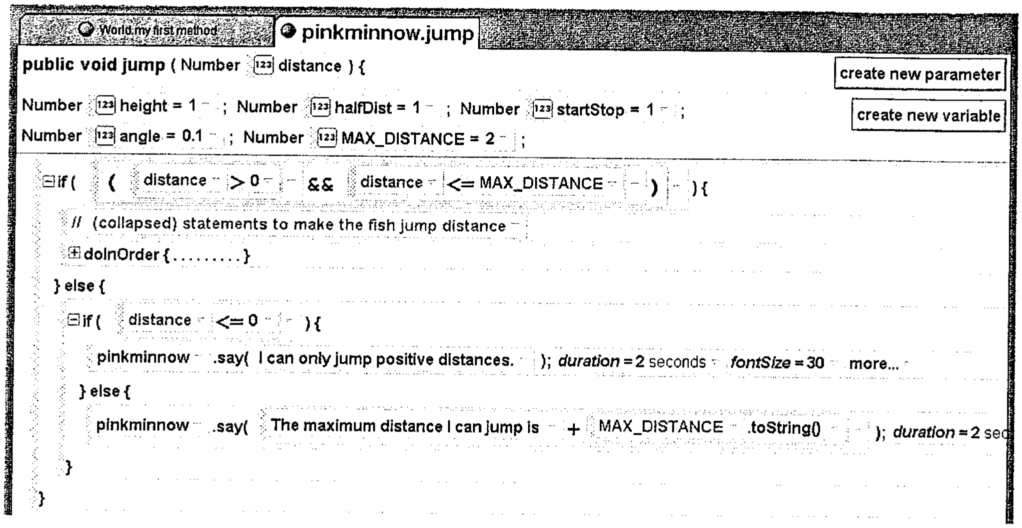


FIGURE 4-14 Validating a parameter's value with nested **if** statements

To save space, we have collapsed the **doInOrder** statement that contains the statements that make the fish jump, using the plus (+) sign at the beginning of the statement.

Let us take a moment to trace the program flow through the revised method:

- When **distance** is valid, the outer **if**'s condition will be **true**, so flow will proceed into the statements that make the fish jump, as we saw in Figure 3-36 in Chapter 3.
- When **distance** is invalid, the first condition will be **false**, so flow will proceed into the **else** statements of the outer **if**. The only statement there is the inner **if** statement, which determines *why* **distance** is invalid (too small or too large?):
  - If **distance** is zero or less, the flow proceeds to the statement in which we send the fish the first **say()** message.
  - Otherwise, **distance** must be greater than **MAX_DISTANCE**, so the flow proceeds to the statement in which we send the fish the second **say()** message.

To illustrate, Figure 4-15 shows the fish's behavior when we send it the message `jump(-2)`.

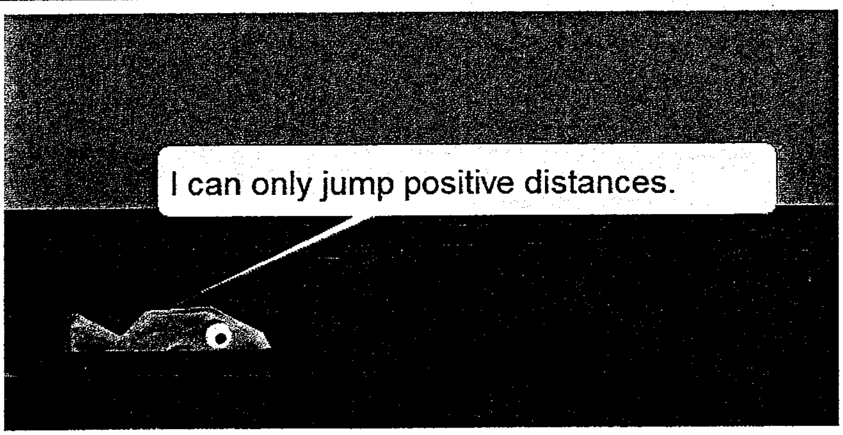


FIGURE 4-15 Asking the fish to jump a negative distance

Similarly, Figure 4-16 shows the fish's behavior when we send it the message `jump(3)`.

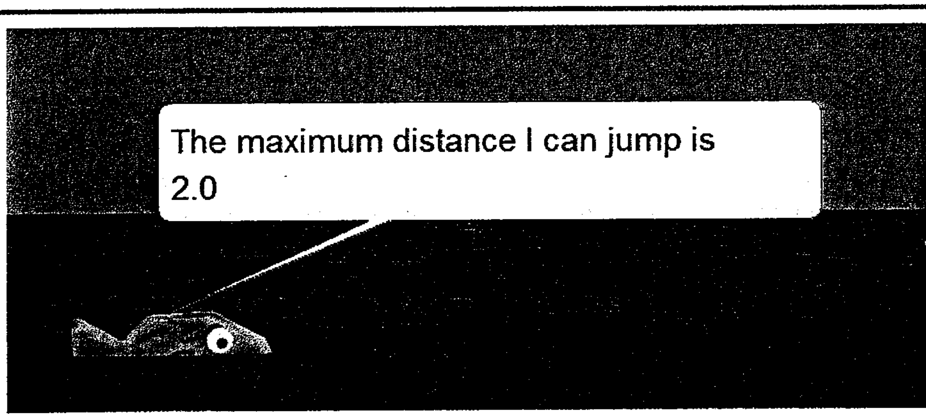


FIGURE 4-16 Asking the fish to jump too far

When building a method with a parameter, think about whether there are any "bad" arguments that could be passed to the parameter. If so, use an `if` statement to guard against such values.

The `if` statement thus provides a way to build **if-then-else logic** into a method. When such logic uses a method's parameter for its condition, then the method can produce different behaviors, based on what argument is passed to that parameter when the message is sent.

4.3 The `for` Statement

4.3.1 Introducing Repetition

In Section 2.2.1, we built a **`flapWings()`** method for the dragon, and in Section 2.3, we saw how to rename, save, and import the dragon as a **`flappingDragon`**. One drawback to the **`flapWings()`** method is that the **`flappingDragon`** will only flap its wings once. Now that we have learned about parameters, we might improve this method by passing it an argument specifying how many times the dragon should flap its wings. To store this argument, we will need a **`Number`** parameter, which we will name **`numTimes`**. We might describe the behavior we want this way:

Parameter: *numTimes*, a Number.

For each value count = 1, 2, ..., *numTimes*:

    The dragon flaps its wings once.

Since we already know how to make the dragon flap its wings once, the idea is to have the method redirect the flow so as to *repeat* the wing-flapping behavior **`numTimes`** times.

We can start by opening the **`flapWings()`** method from Figure 2-16. To make the dragon's wing-flapping seem more realistic, we might adjust the **`duration`** values of the wing movements, so that downstrokes (that is, beating against the air) take longer than upstrokes (that is, resetting for a downstroke). In the version below, we've made the complete cycle (down-stroke and up-stroke) require 1 second.

To make the **`flapWings()`** method flap the dragon's wings more than once, we define a **`Number`** parameter named **`numTimes`**, as shown in Figure 4-17. Next, we drag the **`loop`** control from the bottom of the *editing area* into the method. Since we want to repeat the method's wing-flap behavior, we drop the **`loop`** control at the very beginning of the method. When we drop it, Alice displays an **`end`** menu from which we can choose the number of repetitions we want, as shown in Figure 4-17.

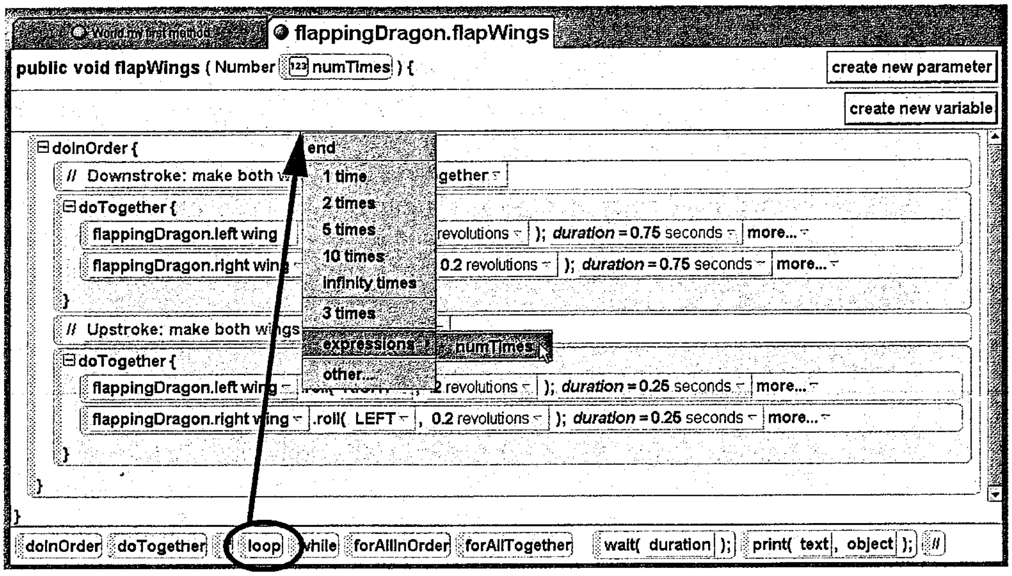


FIGURE 4-17 Dragging the `loop` control

When we select **`numTimes`**, Alice inserts an empty **`for` statement** in the method, as shown in Figure 4-18.

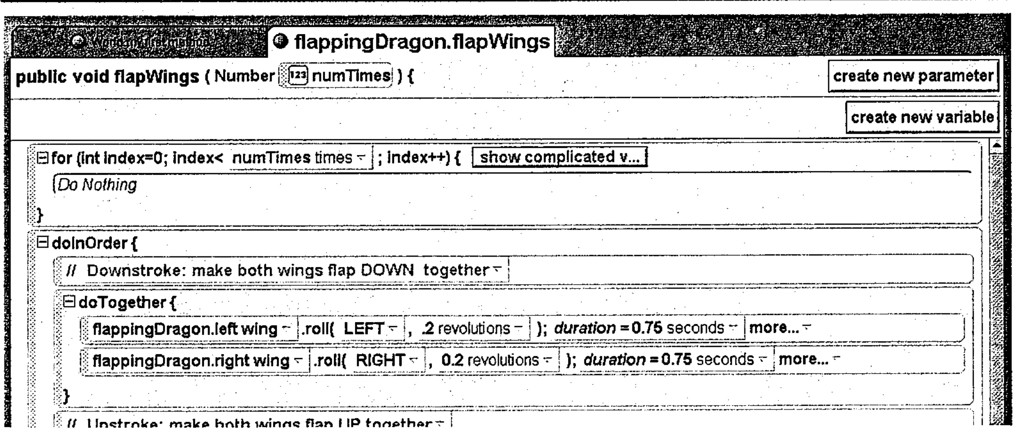


FIGURE 4-18 An empty `for` loop

To finish the method, we drag the **`doInOrder`** statement below the **`for`** statement into the **`for`** statement, resulting in the method definition shown in Figure 4-19.

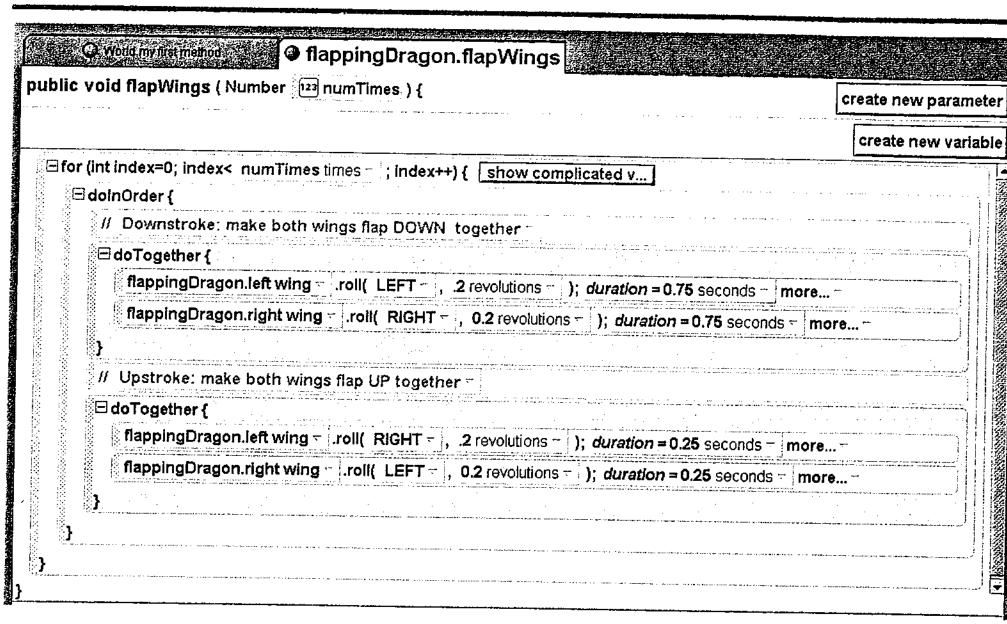


FIGURE 4-19 The revised `flapWings()` method

With this definition, if we send the **`dragon`** the message **`flapWings(3)`**, then it will flap its wings three times. If we send it the message **`flapWings(8)`**, it will flap its wings eight times.

4.3.2 Mechanics of the `for` Statement

The **`for`** statement is a flow control statement whose purpose is to direct the program's flow through the statements within it, while *counting* through a range of numbers. For this reason, it is sometimes called a **counting loop**. If we were to send the **`dragon`** the message **`flapWings(3);`** then the **`for`** statement would count **`0, 1, 2`** (performing the statements within it once for each number), and then quit. If we were to send **`dragon.flapWings(8);`** then the **`for`** statement would count **`0, 1, 2, 3, 4, 5, 6, 7`** (again, performing the statements within it once for each number), and then quit. More generally, the **`for`** statement in **`flapWings()`** will always count from **`0`** to **`numTimes-1`**.

How does it work? Alice's "simple" **`for`** statement has the structure shown below:

```
for (int index = 0; index < limit; index++ ) {
   Statements
}
```

When the program's flow reaches this statement, the flow behaves as shown in Figure 4-20.

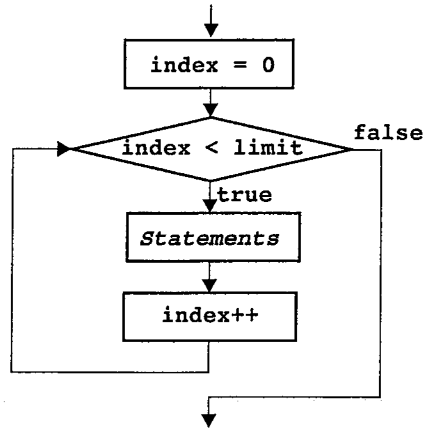


FIGURE 4-20 Flow through a **for** statement

As indicated in Figure 4-20, the **index = 0** in a **for** statement is performed just once, when the flow first reaches the statement. The **for** statement's condition **index < limit** is then checked. If the condition is **false**, then the flow is directed *around* the ***Statements*** within it to whatever statement follows the **for** statement. If the condition is **true**, then the ***Statements*** within the **for** statement are performed, followed by the **index++** (recall that ++ is the **increment operator**). The flow is then redirected *back* to the condition, restarting the cycle.

In Figure 4-21, we trace the behavior of the **for** statement in Figure 4-19 when we send **dragon** the message **flapWings(3)**.

Step	Flow is in...	Effect	Comment
1	**index = 0;**	Initialize **index**	**index**'s value is **0**
2	**index < numTimes (0 < 3)**	The condition is **true**	Flow is directed into the loop
3	**doInOrder**	Flap wings	The first repetition
4	**index++**	Increment **index**	**index**'s value changes from **0** to **1**
5	**index < numTimes (1 < 3)**	The condition is **true**	Flow is directed into the loop
6	**doInOrder**	Flap wings	The second repetition
7	**index++**	Increment **index**	**index**'s value changes from **1** to **2**

FIGURE 4-21 Tracing the flow of **flapWings(3)**

continued

Step	Flow is in...	Effect	Comment
8	**`index < numTimes (2 < 3)`**	The condition is **`true`**	Flow is directed into the loop
9	**`doInOrder`**	Flap wings	The third repetition
10	**`index++`**	Increment **`index`**	**`index`**'s value changes from **2** to **3**
11	**`index < numTimes (3 < 3)`**	The condition is **`false`**	Flow is directed *out of* the loop
12	Flow leaves the **`for`** statement, moving to the end of the method		

FIGURE 4-21 Tracing the flow of `flapWings(3)` *(continued)*

The simple version of the Alice **`for`** statement always begins counting with **`0`**, uses **`index < limit`** as the condition (for whatever **`limit`** value we specify), and uses **`index++`** as the way to increase the index. If we want different values for any of these, we can click the **`show complicated version`** button on the first line of the **`for`** statement. (The button appears as **`show complicated v...`** in Figure 4-19). Clicking this button "expands" the first line of the **`for`** statement into the form shown in Figure 4-22.

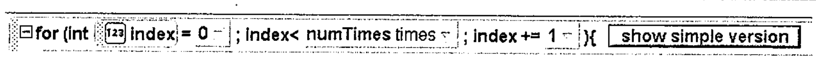


FIGURE 4-22 The complicated `for` loop

Where the simple version just lets you modify the **`limit`** value, the complicated version also lets you set the initial value of **`index`** to a value other than zero, and increase **`index`** by a value other than **`1`** each repetition.

In our experience, the simple version of the **`for`** loop is sufficient most of the time, but Alice provides the complicated version for situations where the simple version is inadequate. Both versions will only count up; if you need to count down, you will need to use a **`while`** statement (see Section 4.4) with a **`Number`** variable that you explicitly set, test, and decrement.

4.3.3 Nested Loops

Suppose the first scene of a user-story is as follows:

> A castle sits in a peaceful countryside. A dragon appears, flying toward the castle. When it gets close, it circles the castle's tower three times, and then descends, landing on the castle's drawbridge.

Using divide-and-conquer, we might divide this scene into three shots:

1. A castle sits in a peaceful countryside. A dragon appears, flying toward the castle.
2. When it gets close, it circles the castle's tower three times.
3. It then descends, landing on the castle's drawbridge.

The first shot can be built several ways. One way is to position the dragon off-screen, store the distance from the dragon to the castle's drawbridge in a variable, and then use a **`move()`** statement to move the dragon that distance, as we have seen before. Another way is to go into the **`Add Objects`** window, position the dragon above the castle's drawbridge, move it upwards until it is even with the castle's tower, and then (using **`more controls`**) click the **`drop dummy at selected object`** button. If we then drag the dragon off-screen, the program can move it to the dummy's position above the drawbridge using the **`setPointOfView()`** message.

The third shot can also be built in several ways. Section 4.4 presents one approach.

To build the second shot, we will use a **`for`** statement controlling other statements that make the dragon fly around the castle tower, as shown in Figure 4-23.

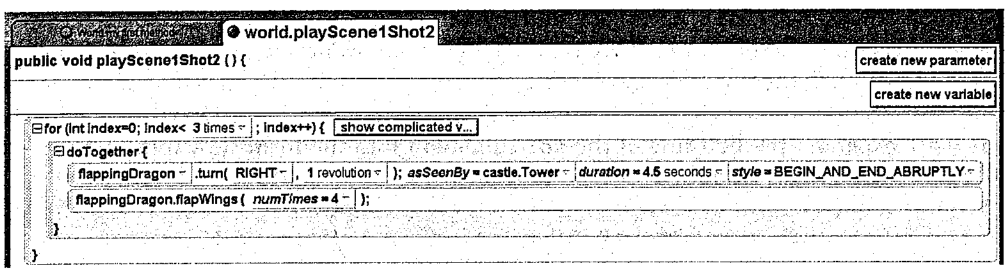


FIGURE 4-23 Making the dragon circle the castle

As defined in Figure 4-23, the **`for`** statement contains a **`doTogether`** statement that causes the dragon to simultaneously fly around the castle (taking 4.5 seconds per circuit), and flap its wings four times. As shown above, this behavior will repeat three times. If, after testing the method, we were to decide that two circuits around the castle tower would be preferable, all we need to do is change the **`for`** statement's **`limit`** value from **`3`** to **`2`**.

Figure 4-23 is deceptively simple. It contains several subtleties that we discuss next.

Nested `for` Statements

One subtlety is that this method is actually using *two* **`for`** statements: the one visible in Figure 4-23, plus the one that is hidden within the **`flapWings()`** method. This situation — where one **`for`** statement is controlling another **`for`** statement — is called **nested `for` statements**, because one **`for`** statement is nested within another.

In Figure 4-23, the **inner `for` statement** (the one hidden within **`flapWings()`**) repeats 4 times for every 1 repetition of the **outer `for` statement** (the one that is visible). With the outer statement repeating 3 times, the dragon flaps its wings a total of $3 \times 4 = 12$ times. Nested

4.4

loops thus have a **multiplying effect**: if the outer loop repeates **i** times and the inner loop repeats **j** times, then the statements in the inner loop will be repeated a total of **i × j** times.

The asSeenBy Attribute

The second subtlety is how the **turn()** message in Figure 4-23 causes the dragon to circle the tower. Alice's **turn()** message has a special **asSeenBy** attribute. Normally, this attribute is set to **None**, in which case **turn()** just causes its receiver to revolve about its **LR** axis or its **FB** axis. However, if we specify another object (like **castle.tower**) as the value of the **asSeenBy** attribute, then the **turn()** message causes its receiver to *revolve around that object*. Figure 4-23 uses this trick to make the dragon revolve around the castle tower once for each repetition of the outer **for** statement.

The duration Attribute

In testing the method, we initially set the **duration** of the **turn()** message to **4** seconds, to match the dragon's 4 wing-flaps (1 per second) per circuit of the tower. This produced a "hitch" in the animation as the dragon finished each circuit. The problem is that while each wing-flap takes 1 second to complete, the **flapWings(4)** message consumes slightly longer than 4 seconds.[1] As a result, the 4-second **turn()** message was finishing before the 4 wing-flaps. We were able to smooth the animation by increasing the **duration** of the **turn()** message slightly, and setting the message's **style** attribute to **BEGIN_AND_END_ABRUPTLY**, as shown in Figure 4-23.

4.4 The while Statement

The **for** statement is a means of causing flow to repeatedly move through the same group of statements a fixed number of times. For this reason, the **for** statement is often called a counting statement, or a **counting loop**. The program must "know" (that is, be able to compute) how many repetitions are needed when flow reaches the **for** statement, to set its **limit** value.

This raises a problem: What do we do when we encounter a situation for which we need repetitive flow-behavior, but we do not know in advance how many repetitions are required? For such statements, Alice (and other programming languages) provides the **while** statement.

4.4.1 Introducing the while Statement

In Section 4.3.3, we began work on a scene consisting of three shots:

1. A castle sits in a peaceful countryside. A dragon appears, flying toward the castle.
2. When it gets close, it circles the castle's tower three times.
3. It then descends, landing on the castle's drawbridge.

---

1. For each repetition of a **for** statement, its **index++** statement and the **index < limit** condition must be processed, which consumes time. A **flapWings(*n*)** message thus consumes more than *n* seconds.

We have seen how to build the first two shots, and it is possible to build the third shot using a variable, a function, and a **doTogether** statement containing a **move()** message and the **flapWings()** method. The drawback to this approach is that we must coordinate the **move()** and **flapWings()** messages, so that the duration of the **move()** (that is, how long the descent will take) coincides with the wing-flaps of the dragon. If we later change the elevation of the dragon above the drawbridge, we will have to recoordinate the **move()** and **flapWings()** messages.

In this section, we will see an alternative way to build this shot, using a **while** statement, a function, and a **doTogether** statement containing a **move()** message and the **flapWings()** message. The idea is to repeatedly (1) have the dragon flap its wings, and (2) move it downwards whatever distance it drops in one wing-flap, so long as it is above the drawbridge.

We begin by moving the camera closer (via a dummy we'll rename shot1-3, using the techniques described in Section 2.4.), to better see the dragon's descent, as shown in Figure 4-24.

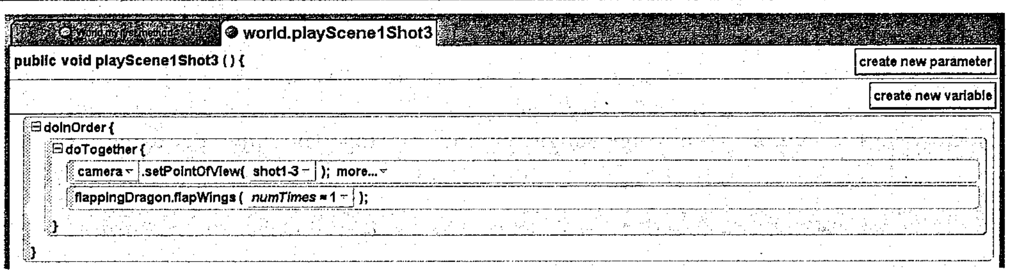


FIGURE 4-24 Moving the camera closer

With the camera in position, we are ready to make the dragon descend. To do so, we click the **while** control at the bottom of the *editing area*, drag it into the method, and drop it at the last position within the **doInOrder** statement. See Figure 4-25.

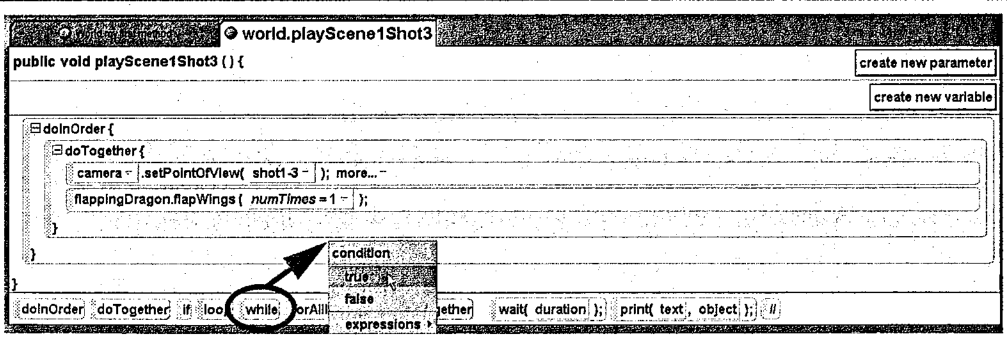


FIGURE 4-25 Dragging the **while** control

When we drop it there, Alice generates a **`condition`** menu from which we can choose a condition to control the **`while`** statement. For the moment, we just choose **`true`** as a *placeholder*. Alice then generates the empty **`while`** statement shown in Figure 4-26.

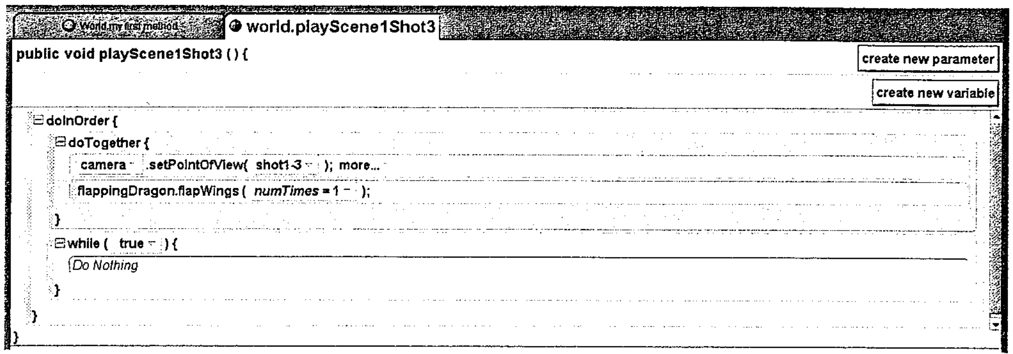


FIGURE 4-26 An empty `while` statement

For each repetition of the **`while`** statement, we want the dragon to flap its wings once and move downward a short distance (still to be determined). We want this behavior to repeat as many times as necessary, so long as the dragon is above the drawbridge. For the **`while`** statement's condition, we can thus drag the dragon's **`isAbove()`** function into the **`while`** statement's placeholder condition, and when we drop it, choose the castle's drawbridge as its argument, as shown in Figure 4-27.

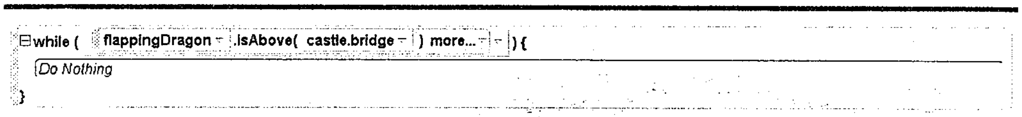


FIGURE 4-27 Repeating so long as the dragon is above the drawbridge

Any statements we place within the **`while`** statement will be repeated so long as the condition **`flappingDragon.isAbove(castle.Bridge)`** produces the value **`true`**. Those statements must ensure that the condition eventually becomes **`false`**, or else an **infinite loop** will result. That is, if the flow reaches the **`while`** statement shown in Figure 4-27, the flow will remain there sending **`flappingDragon`** the **`isAbove()`** message over and over forever, or until we terminate the program, whichever comes first. Any time the flow reaches a **`while`** loop whose statements do not cause its condition to eventually become **`false`**, this infinite looping behavior is the result.

To avoid an infinite loop, the loop's statements should flap the dragon's wings and move it down a small distance, so that its bounding box eventually touches that of the bridge. When that happens, the **`isAbove()`** condition will become **`false`** and the loop will terminate. We can use these ideas to complete the method as shown in Figure 4-28.

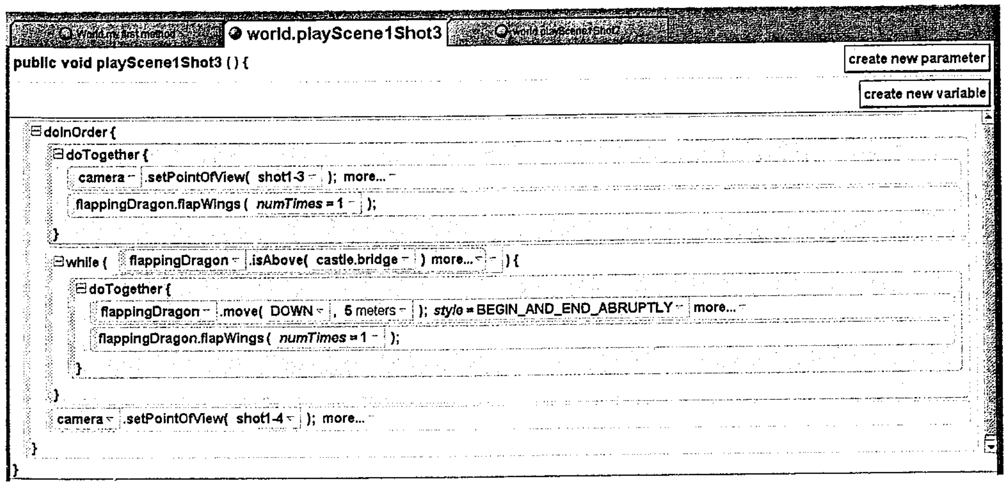


FIGURE 4-28 The `playScene1Shot3()` method (final version)

Each repetition of the **`while`** statement in Figure 4-28 takes 1 second, during which the dragon simultaneously flaps its wings and moves down 5 meters. If we decide this descent is too slow, we can double its descent rate by changing the **`5`** to a **`10`**; or if it seems too fast, we can slow the descent by changing the **`5`** to a **`4`**, a **`2`**, or a **`1`**. The key decision in this approach is how far a dragon should descend in 1 second (which is simpler than the use-a-variable approach).

The final statement in the method zooms the camera in (using another dummy) for a closer shot of the dragon on the bridge after its descent, yielding the shot in Figure 4-29.

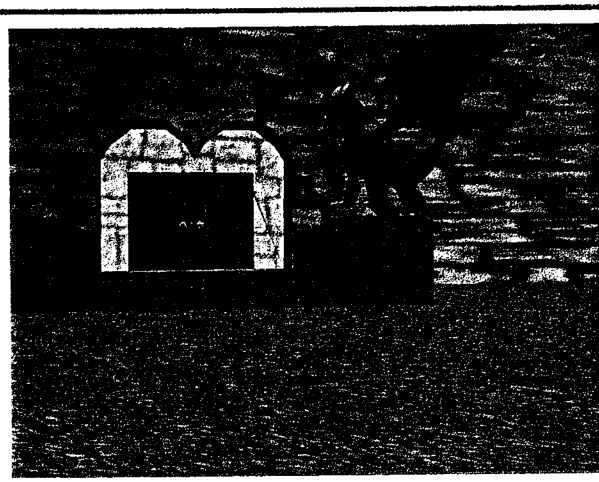


FIGURE 4-29 The dragon on the drawbridge

4.4.2 `while` Statement Mechanics

Where the **`for`** statement is a counting loop, the **`while`** statement is a **general**, or **indefinite loop**, meaning the number of repetitions to be performed need not be known in advance. The structure of the Alice **`while`** statement is as follows:

```
while ( Condition ) {
   Statements
}
```

When flow reaches a **`while`** statement, it proceeds as shown in Figure 4-30.

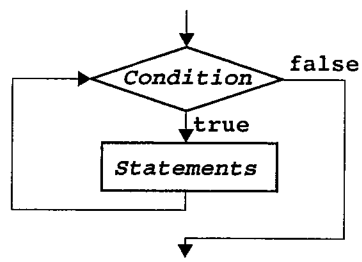


FIGURE 4-30 Flow through a `while` statement

In Figure 4-30, when flow first reaches a **`while`** statement, its ***`Condition`*** is evaluated. If it is **`false`**, then the flow leaves the **`while`** statement, bypassing its ***`Statements`***. However, if it is **`true`**, then the ***`Statements`*** within the **`while`** statement are performed, after which the flow is redirected back to recheck its ***`Condition`***, where the process begins again.

4.4.3 Comparing the `for` and `while` Statements

If you compare Figure 4-30 to Figure 4-20, you will see that the **`while`** statement's behavior is actually much simpler than that of the **`for`** statement. This is because the **`while`** is the more general flow-control statement; whereas the **`for`** statement is useful mainly in counting situations, the **`while`** statement can be used in any situation where repetition is required.

So when should you use each statement? Whenever you are working to produce a behavior that needs to be repeated, ask yourself this question: "Am I counting something?" If the answer is "yes," then use a **`for`** statement; otherwise, use a **`while`** statement. For example, in Figure 4-19 and Figure 4-23, we counted wing-flaps and tower-circuits, respectively. By contrast, in Figure 4-28, we were not counting anything, just controlling the dragon's descent.

Both the **`while`** and the **`for`** statements test their condition *before* the statements within the loop are performed. In both cases, if the condition is initially **`false`**, then statements within the loop will be bypassed (that is, not performed). If you write a program containing a loop statement that seems to be having no effect, it is likely that the loop's condition is **false** when flow reaches it. To remedy this, either choose a different condition, or ensure that its condition is **true** before flow reaches the loop.

4.4.4 A Second Example

As a second example of the **while** statement, suppose that Scene 1 of a story has a girl named Jane dropping a soccer ball (that is, a football everywhere outside of the U.S.). Jane lets it bounce until it stops on its own. Our problem is to get it to bounce realistically.

When dropped, a ball falls until it strikes a surface beneath it. It then rebounds upwards some distance (depending on some bounce factor that combines its elasticity, the hardness of the surface it hits, etc.), drops again, rebounds again, drops again, rebounds again, and so on. We can sketch the behavior as being something like that shown in Figure 4-31.

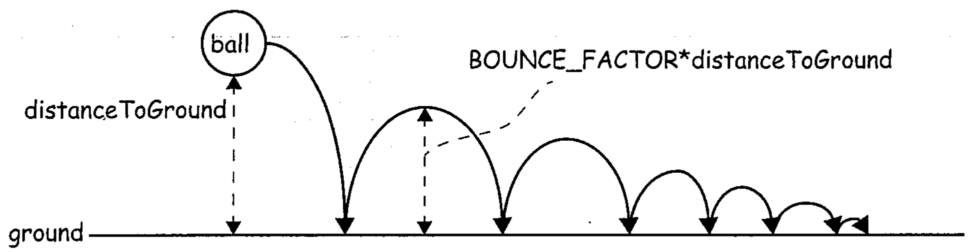


FIGURE 4-31 Sketch of the up-down motion of a bouncing ball

For simplicity, we will just have the soccer ball bounce straight up and down.

Using the **sheBuilder** (located in the **People** folder in the Alice Gallery), the **SoccerBall** class from Alice's **Web Gallery**, and the **quad-view** window, we might start by building a scene like the one shown in Figure 4-32.

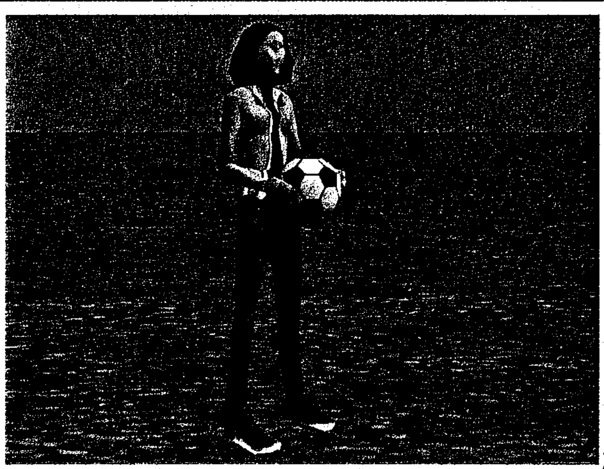


FIGURE 4-32 Jane with the soccer ball

To produce the desired bouncing behavior, we can write a **dropAndBounce()** method for the **soccerBall**, which is shown in Figure 4-33.

```
world.my first method | world.janeDropsBall | soccerBall.dropAndBounce
public void dropAndBounce ( ) {                                   create new parameter
Number  distanceToGround = 0  ; Number  BOUNCE_FACTOR = 0.67  ;   create new variable
  doInOrder {
    distanceToGround .set( value , soccerBall .distanceAbove( ground ) more... ); more...
    while ( distanceToGround > 0 ) {
      soccerBall .move( DOWN , distanceToGround meters ); duration = distanceToGround seconds style = BEGIN_GENTLY_AND_END_ABRUPTLY more
      distanceToGround .set( value , ( distanceToGround * BOUNCE_FACTOR ) ); more...
      soccerBall .move( UP , distanceToGround meters ); duration = distanceToGround seconds style = BEGIN_ABRUPTLY_AND_END_GENTLY more
    }
  }
```

FIGURE 4-33 Method soccerBall.dropAndBounce()

When Jane drops the ball, we do not know in advance how many times it is going to rebound, so we have used a **while** statement instead of a **for** statement. The condition controlling the loop is this: the ball should continue to bounce so long as its distance above the ground exceeds zero.

We have assumed that on each bounce, the ball will rebound to 2/3 of the distance it fell previously. (If this proves to be a poor assumption, we have made it easy to change by storing the 2/3 in a variable called **BOUNCE_FACTOR**.) By storing the (initial) distance from the ball to the ground in a variable named **distanceToGround**, then for each repetition of the loop, we

- move the ball *down* **distanceToGround** meters
- change the value of **distanceToGround** to **distanceToGround*BOUNCE_FACTOR**
- move the ball *up* **distanceToGround** meters (which is now 2/3 of its previous value)

To make the ball's behavior seem more realistic, we set the **duration** of each bounce-movement to the current value of the **distanceToGround** variable. Thanks to this, each successive bounce-movement will occur faster as **distanceToGround** gets smaller.

Another refinement to increase the realism was to set the style of the **move()** causing the ball's drop to **BEGIN_GENTLY_AND_END_ABRUPTLY**, and set the style of the **move()** causing the ball's rebound to **BEGIN_ABRUPTLY_AND_END_GENTLY**. The net effect is to make a fast down-to-up transition when the ball bounces, and to make a slow up-to-down transition as the ball reaches the peak of its bounce.

Given the method in Figure 4-33, we can easily build a world method (since it animates two different objects) in which Jane drops the ball, as shown in Figure 4-34.

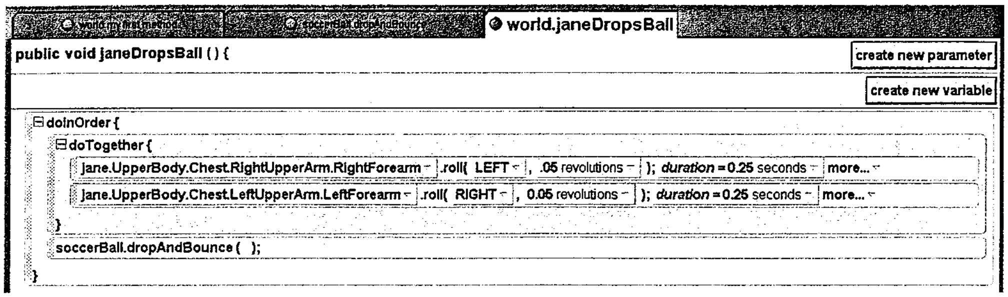


FIGURE 4-34 Method `world.janeDropsBall()`

Try this yourself, and experiment with the statements and settings shown in Figure 4-33, to see how each one affects the ball's behavior. (There's always the **Undo** button!)

4.5 Flow-Control in Functions

At the end of Chapter 3, we saw that if we want to ask an object a question for which there is not already a function, we can define our own function to provide the answer. The functions we wrote there used sequential flow, and were fairly simple. The flow-control statements we have seen in this chapter allow us to build functions that answer more complex questions.

4.5.1 Spirals and the Fibonacci Function

Suppose that we have a story in which a girl finds an old book. The book tells her that there is a treasure hidden near a certain palm tree in the middle of the desert. The book contains a map showing how to find the tree, plus instructions for locating the treasure from the tree. Suppose that Scene 1 of the story has the girl finding the old book and reading its contents. In Scene 2, the girl uses the map to locate the palm tree. In Scene 3 she follows the instructions:

Scene 3: The girl is at the tree, her back to the camera. She says, "Now that I am at the tree, I turn to face North." She turns to face the camera. "Then I walk in a spiral of six quarter turns to the left, and then say the key phrase." She walks in a spiral of six quarter turns to her left, says a key phrase, and an opening appears in the ground at her feet.

The main challenge in building this user story is getting the girl to move in a spiral pattern. Mathematicians have discovered that many of the spirals that occur in nature — for example, the spiraling chambers inside a nautilus shell, the spiral of petals in a rose, and the spiraling seeds in sunflowers and pinecones — all use a pattern given in the following numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Can you see a pattern in these numbers? The first known mention of them is by the Indian scholar Gospala sometime before 1135 AD. The first European to discover them was Leonardo Pisano, a 13th century mathematician who found that they predict the growth of rabbit populations. Leonardo was the son of Guglielmo Bonaccio, and often called himself Fibonacci (short for "son of Bonaccio"). Today, these numbers are called the **Fibonacci series**.

To draw a spiral from the series, we draw a series of squares whose lengths and widths are the Fibonacci numbers. Starting with the smallest square, we draw a series of quarter turn arcs, crossing from one corner of the square to the opposite corner, as shown in Figure 4-35.

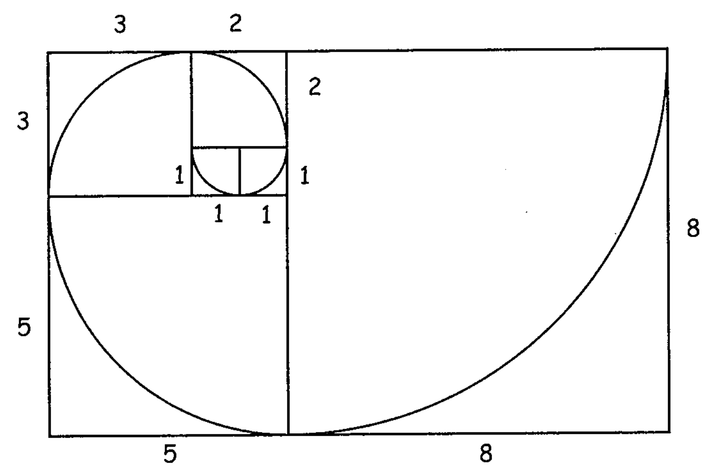


FIGURE 4-35 A Fibonacci spiral pattern

To move the girl in the story in a spiral pattern, we can use a similar approach. More precisely, we can move her in a close approximation of the Fibonacci spiral as follows:

1. move her forward 1 meter while turning left 1/4 revolution
2. move her forward 1 meter while turning left 1/4 revolution
3. move her forward 2 meters while turning left 1/4 revolution
4. move her forward 3 meters while turning left 1/4 revolution
5. move her forward 5 meters while turning left 1/4 revolution
6. move her forward 8 meters while turning left 1/4 revolution

More concisely, we can have her move 6 times, each time moving a distance equal to the next Fibonacci number while turning left 1/4 revolution. That is, if we had a function that, given a positive number i, computes the i^{th} Fibonacci number, we could write the **playScene3()** method as shown in Figure 4-36.

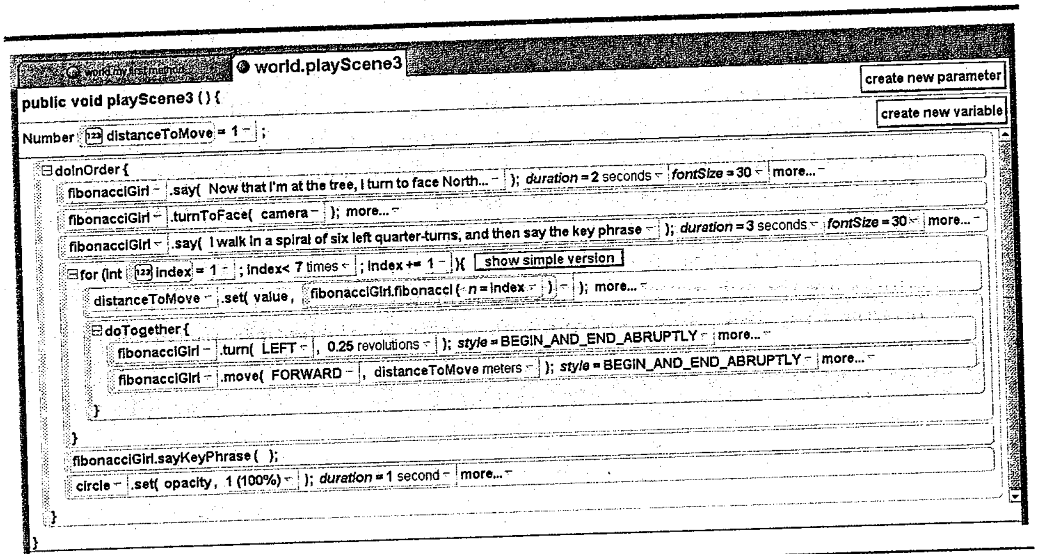


FIGURE 4-36 The **playScene3()** Method

In just a moment, we will build such a **fibonacci()** function. Since it seems possible we may want to reuse it someday, we will define it within the girl, whom we have renamed **fibonacciGirl** in Figure 4-36. (In the Alice Gallery, her name was **RandomGirl3**).

4.5.2 The Fibonacci Function

To create the **fibonacci()** function that is invoked in Figure 4-36, we select the girl in the *object tree*, click the *functions* tab in her *details area*, and then click the **create new function** button. Alice prompts us for the name of the function, so we enter **fibonacci**.

To invoke this function, we must pass it a positive **Number** argument indicating which Fibonacci number we want it to return. To store this argument, the function must have a **Number** parameter. We will name this parameter **n**.

Design

The question the function must answer is this: Given n, what is the n^{th} Fibonacci number? If we look at the series carefully

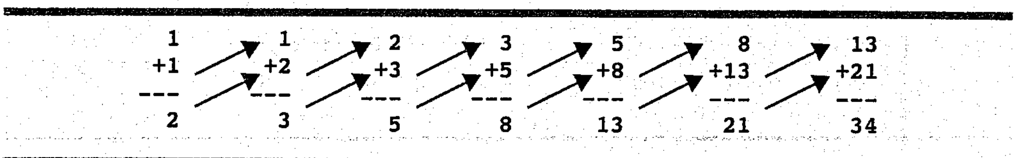
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

we can see this pattern: after the initial two 1s, every subsequent number is the *sum of the preceding two numbers*. That is, there are two cases we must deal with:

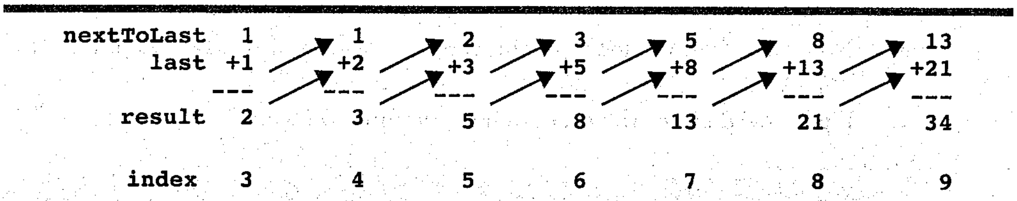
if (n is 1 or n is 2) the function's result is 1;

otherwise, the function's result is the sum of the preceding two values in the series.

The tricky part is figuring out "the preceding two values in the series." As we have seen before, let's first try doing this by hand. For example, to compute **`fibonacci(9)`**:



Since we are doing the same thing over and over, we can do this using a loop. To do so, we store each value used per iteration in a variable: one for the next-to-last term, one for the last term, and one for the result; we can then use a **`for`** loop to count from 3 to **`n`**. When **`n`** is 9:



Putting all of this together yields the following algorithm for the function:

```
Parameter: n, a Number.
Number result = 0; Number nextToLast = 1; Number last = 1;
if (n == 1 or n == 2) {
   result = 1;
} else {
   for (int index = 3; index < n+1; index++) {
      result = last + nextToLast;
      nextToLast = last;
      last = result;
   }
}
return result;
```

Coding in Alice

We can encode the algorithm in Alice as shown in Figure 4-37.

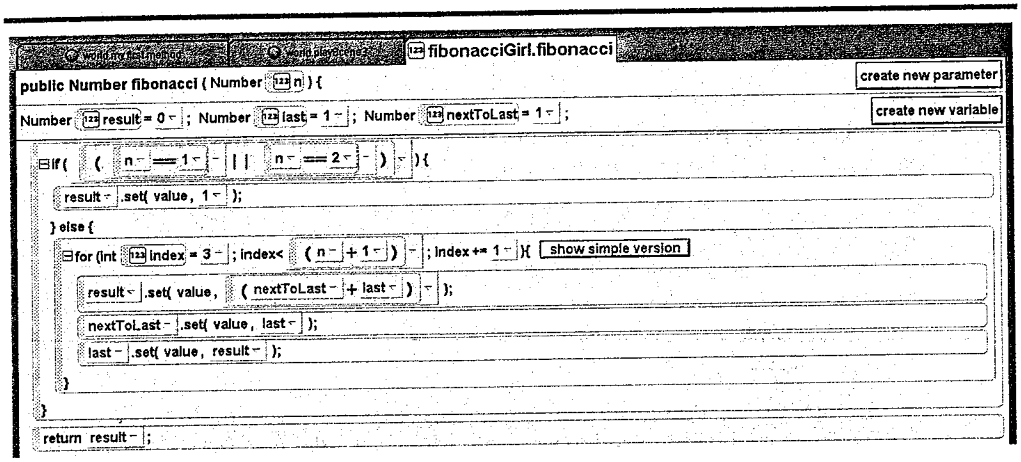


FIGURE 4-37 The **`fibonacci()`** function

Note that the function uses the complex version of the **`for`** loop, because it begins counting at 3.

Figure 4-38 traces the execution of the function when **`4`** is passed to **`n`**.

Step	Flow is in...	Effect	Comment
1	**`if`** ***`Condition`***	***`Condition`*** is **`false`**	Control flows to the **`if`**'s **`else`** branch
2	**`index = 3`**	For loop is initialized	**`index`** is **`3`**
3	**`index < n+1`**	The condition is **`true`**	Flow is directed into the loop
4	**`result = ...`**	Compute fibonacci(3)	**`result`** is **`2`**
5	**`nextToLast = ...`**	Update **`nextToLast`**	**`nextToLast`** is **`1`**
6	**`last = ...`**	Update **`last`**	**`last`** is **`2`**
7	**`index++`**	Increase **`index`**	**`index`** is **`4`**
8	**`index < n+1`**	The condition is **`true`**	Flow is directed into the loop
9	**`result = ...`**	Compute fibonacci(4)	**`result`** is **`3`**

FIGURE 4-38 Tracing the **`fibonacci()`** function

continued

Step	Flow is in...	Effect	Comment
10	`nextToLast = ...`	Update **nextToLast**	**nextToLast** is **2**
11	`last = ...`	Update **last**	**last** is **3**
12	`index++`	Increase **index**	**index** is **5**
13	`index < n+1`	The condition is **false**	Flow is directed out of the loop
14	`return result;`	The function terminates	**result** is **3**, the 4th Fibonacci number
15	Flow leaves the function, returning **result** to the point where the function was invoked.		

FIGURE 4-38 Tracing the **fibonacci()** function *(continued)*

Note that we initialize **result** to zero. If the user passes an invalid argument (for example, zero or a negative number), then the function returns this zero. First, control flows into the **if** statement's **else** branch. However when its **for** loop tests the condition **(3 < (n+1))**, that condition will be **false** if **n** is negative or zero, so the body of the **for** loop will be skipped. Control then flows to the **return** statement, and since **result** has not been modified, the function returns zero.

Using this function, the **for** loop in Figure 4-35 will cause **fibonacciGirl** to move in a spiral pattern, after which she says the key phrase and a dark opening appears in the ground at her feet. What happens next? It's up to you!

4.6 Chapter Summary

- **Boolean** operators allow us to build *conditions*.
- The **if** statement uses a condition to direct program flow *selectively* through one group of statements while bypassing others.
- The **for** statement uses a condition to direct program flow through a group of statements *repeatedly*, a fixed number of times.
- The **while** statement uses a condition to direct program flow through a group of statements *repeatedly*, where the number of repetitions is not known in advance.
- The **wait()** message lets us suspend a program's flow for a fixed length of time.
- The **asSeenBy** attribute alters the behavior of the **turn()** message.

4.6.1 Key Terms

boolean expression
boolean operators
(`&&`, `||`, `!`)
`Boolean` type
boolean variables
condition
control structure
counting loop
flow control
flow diagram
general loop
if-then-else logic
`if` statement
indefinite loop
infinite loop
nested statement
(inner statement, outer statement)
relational operators
(`==`, `!=`, `<`, `>`, `<=`, `>=`)
repetitive control
selective control
selective execution
selective flow
validating parameter values
`wait()` statement
`while` statement

Programming Projects

4.1 Choose a hopping animal from the Alice Gallery (for example, a frog or a bunny). Write a `hop()` method that makes it hop in a realistic fashion, with a (validated) parameter that lets the sender of the message specify how far the animal should hop. Then build a method containing just one `hop()` message that causes your animal to hop around a building.

4.2 *Johnny Hammers* is a traditional song with the lyrics below. Create an Alice program containing a character who sings this song. Write your program using as few statements as possible.

Johnny hammers with 1 hammer, 1 hammer, 1 hammer. Johnny hammers with 1 hammer, all day long.	*Johnny hammers with 2 hammers, 2 hammers, 2 hammers. Johnny hammers with 2 hammers, all day long.*
Johnny hammers with 3 hammers, 3 hammers, 3 hammers. Johnny hammers with 3 hammers, all day long.	*Johnny hammers with 4 hammers, 4 hammers, 4 hammers. Johnny hammers with 4 hammers, all day long.*
Johnny hammers with 5 hammers, 5 hammers, 5 hammers. Johnny hammers with 5 hammers, all day long.	*Johnny's very tired now, tired now, tired now. Johnny's very tired now, so he goes to sleep.*

4.3 Using the horse we used in Section 3.4, build a `gallop()` method for the horse that makes its legs move realistically through the motions of a gallop, with a (validated) parameter that specifies the number of strides (or alternatively, the distance to gallop). Then create a story containing a scene that uses your method to make the horse gallop across the screen.

4.4 *The Song That Never Ends* is a silly song with the lyrics below. Create an Alice program containing a character who sings this song, using as few statements as possible. (If your computer has a microphone, get your character to "sing" a recording of the song as well as "say" the lyrics. If you do not know the tune, find and listen to the song on the World Wide Web.)

This is the song that never ends, and it goes on and on my friends. Some people started singing it not knowing what it was, and now they'll keep on singing it forever just because.	*This is the song that never ends, and it goes on and on my friends. Some people started singing it not knowing what it was, and now they'll keep on singing it forever just because.*
This is the song that never ends, and it goes on and on my friends. Some people started singing it not knowing what it was, and now they'll keep on singing it forever just because.	*... (ad infinitum, ad annoyeum, ad nauseum)*

4.5 Build a world containing a person who can calculate the average of a sequence of numbers in his or her head. Have the person ask the user how many numbers are in the sequence, and then display a **`NumberDialog`** that many times to get the numbers from the user. When all the numbers have been entered, have your person "say" the average of those numbers.

4.6 Proceed as in Problem 4.5, but instead of having your person ask the user in advance how many numbers are in the sequence, have your person and each **`NumberDialog`** tell the user to enter a special value (for example, -999) after the last value in the sequence.

4.7 *99 Bottles of Pop* is a silly song with the lyrics below. Create an Alice program in which a character sings this song. Use as few statements as possible. (Hint: Even though this is a counting problem, you will need to use a **`while`** statement instead of a **`for`** statement. Why?)

99 bottles of pop on the wall, 99 bottles of pop, take one down, pass it around, 98 bottles of pop on the wall.	*98 bottles of pop on the wall, 98 bottles of pop, take one down, pass it around, 97 bottles of pop on the wall.*
(96 verses omitted) ...	*1 bottle of pop on the wall, 1 bottle of pop, take one down, pass it around 0 bottles of pop on the wall.*

4.8 Using the **`heBuilder`** or **`sheBuilder`** (or any of the other persons in the Alice Gallery with enough detail), build a person and add him or her to your world. Using your person, build an aerobic exercise video in which the person leads the user through an exercise routine. Using repetition statements, your person should do each exercise a fixed number of times. (Hint: Use **`Pose`** variables and the **`capture pose`** button.)

4.9 Proceed as in Problem 4.8, but at the beginning of the program, ask the user to specify the difficulty level of the workout (1, 2, 3, 4, or 5). If the user specifies 1, have your person do each exercise 10 times. If they specify 2, 20 times. If they specify 3, 40 times, If they specify 4, 80 times. If they specify 5, 100 times.

4.10 From the Alice Gallery, choose a clock class that has subparts for the minute and hour hands.

a. Build a clock method named **`run()`** that moves the minute and hour hands realistically (that is, each time the minute hand completes a rotation, the hour hand should advance 1 hour). Define a parameter named **`speedUp`** that controls the **`duration`**s of the hand movements, such that **`run(0)`** will make the clock run at normal speed, **`run(60)`** will make the clock run at 60 times its normal speed, **`run(3600)`** will make the clock run at 3600 times its normal speed, and so on.

b. Build a clock method **`setTime(h, m)`** that sets the clock's time to **`h:m`** (**`m`** minutes after hour **`h`**).

c. Build three functions for your clock: one that returns its current time (as a **`String`**), one that returns its current hours value (as a **`Number`**), and one that returns its current minutes value (as a **`Number`**).

d. Build a clock method **`setAlarm(h, m)`** that lets you set the clock's alarm to **`h:m`**. Then modify your **`run()`** method so that when the clock's current time is equal to **`m`** minutes after hour **`h`**, the clock plays a sound (for example, Alice's **`gong`** sound).

4.11 Using appropriately colored **`Shapes`** from the Alice Gallery, build a chessboard. Then choose objects from the Gallery to serve as chess pieces. Build a class-level method named **`chessMove()`** for each piece that makes it move appropriately (for example, a bishop should move diagonally). For pieces that can move varying distances, the definition of **`chessMove()`** should have a (validated) parameter indicating the distance (in squares) of the move, plus any other parameters necessary. When your "pieces" are finished, build a program that simulates the opening moves of a game of chess, using your board and pieces.

4.12 Design an original 3–5 minute story that uses each of the statements presented in this chapter at least once.