Part II
Object-Oriented and Event-Driven Programming Concepts

Part II
Object-Oriented and Event-Driven Programming Concepts

Chapter 4
Classes, Objects, Methods, and Parameters

"The Queen of Hearts, she made some tarts,
All on a summer day:
The Knave of Hearts, he stole those tarts,
And took them quite away!"

Instructions for Making a Strawberry Tart

1 crust, baked
3 cups strawberries, hulled
1 pkg. strawberry gelatin
1 1/2 cups of water
2 Tbs. corn starch

Place strawberries in the crust.
Mix gelatin, water, and corn starch in a small pan.
Stir, while heating to a boil.
Let cool and then pour over strawberries.
Chill.

When you created your own animations in earlier chapters, you may have started to think about more complicated scenarios with more twists and turns in the storyline, perhaps games or simulations. Naturally, as the storyline becomes more intricate, so does the program code for creating the animation. The program code can quickly increase to many, many lines—sort of an "explosion" in program size and complexity. Animation programs are not alone in this complexity. Real-world software applications can have thousands, even millions, of lines of code. How does a programmer deal with huge amounts of program code? One technique is to divide a very large program into manageable "pieces," making it easier to design and think about. Smaller pieces are also easier to read and debug. Object-oriented programming uses classes, objects, and methods as basic program components, which will help you organize large programs into small manageable pieces. In this chapter and the next, you will learn how to write more intricate and impressive programs by using objects (instances of classes) and writing your own methods.

Classes

A class defines a particular kind of object. In Alice, classes are predefined as 3D models provided in the gallery, categorized into groups such as Animals, People, Buildings, Sets and Scenes, Space, and so on. Figure 4-0-1 shows some of the classes in the Animals folder. Notice that the name of a class begins with a capital letter.

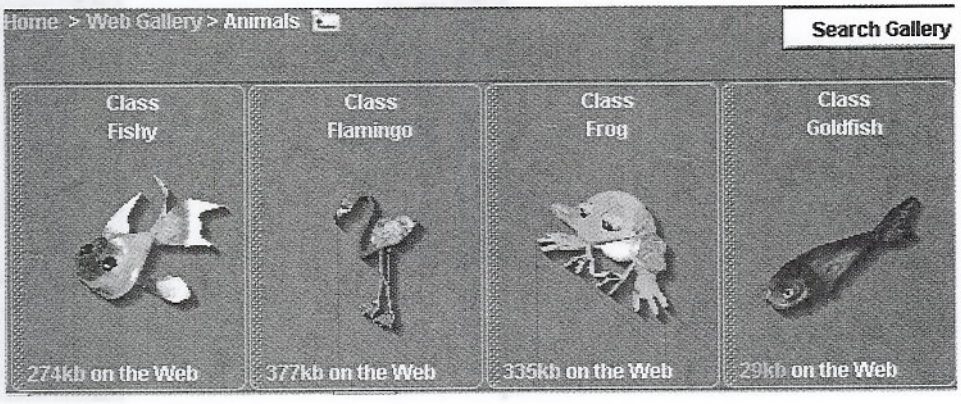


Figure 4-0-1. Classes of 3D Models in the Animals folder

Each class is a blueprint that tells Alice exactly how to create and display an object from that class. When an object is created and displayed, we call this instantiating the class because an object is an instance of that class.

Objects

In Figure 4-0-2, Person and Dog are classes. Joe, stan, and cindy are instances of the Person class while spike, scamp, and fido are instances of the Dog class. Notice that the name of an object begins with a lowercase letter. This naming style helps us to easily distinguish the name of a class from the name of an object. All objects of the same class share some commonality. All Person objects have properties such as two legs, two arms, height, and eye color. Person objects can perform walking and speaking actions. All Dog objects have properties including four legs, height, and fur-color, and have the ability to run and bark. Although each object belongs to a class, it is still unique in its own way. Joe is tall and has green eyes. Cindy is short and has blue eyes. Spike has brown fur and his bark is a low growl, and scamp has golden-color fur and his bark is a high-pitched yip.

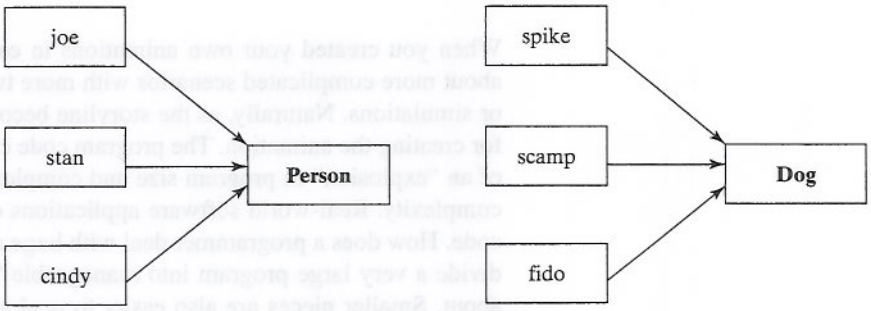


Figure 4-0-2. Organizing objects into classes

In Figure 4-0-3, larry, lila, and louis are all instances of the Lemur class (Animals) in Alice. We named the lemurs in this world, made them different heights, and changed the color of lila. Larry, lila and louis are all objects of the same Lemur character class and have many common characteristics. They also differ in that larry is the tallest, lila has rich, dark fur, and louis is the shortest.

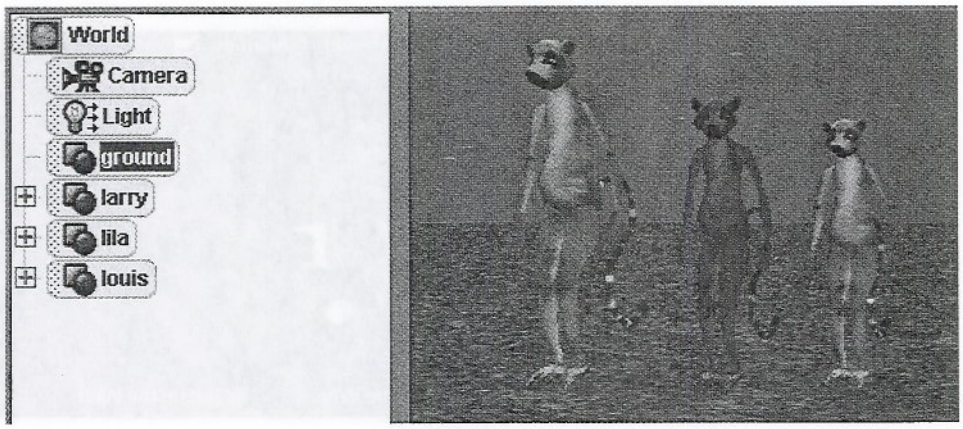


Figure 4-0-3. Objects of the Lemur class in Alice

Methods

A method is a coordinated sequence of instructions that will be carried out when requested. You have already used methods in your animations. Every object in an Alice world has a repertoire of instructions it knows how to do—*move, turn, turn to face*, etc. These instructions are actually primitive methods, built-in to the Alice software. The primitive methods can be organized into a method of your own—to carry out a small piece of the overall program. Each method performs its own job, but all the methods in a program work together to create the overall animation.

As your animation programs grow larger, it will become increasingly important to use many, many methods as a way of organizing the program. Methods divide a program into small manageable pieces that work together to create a meaningful whole. Just as paragraphs, sections, and chapters make a book easier to read, methods make a program easier to read. Methods also provide a number of advantages. For example, once a method is written it allows us to think about an overall task instead of all the small actions that were needed to complete the task. This is called abstraction.

Some methods need to be sent certain pieces of information to carry out an action. For example a *move* method needs a *direction* (*forward, backward, left, right, up,* or *down)* and a *distance* (in meters). A parameter acts like a basket to receive information that we send to a method. In a way, you can think of a method as somewhat like a recipe—a set of instructions that describe how to perform some action. (As an example, see the recipe at the beginning of this chapter for making a strawberry tart.) Parameters hold onto the specific items of information. In a recipe, a parameter could specify the amount of water. In a method, a parameter could specify the distance a spaceship is to move.

In Alice, you can define methods for an object acting alone or for two or more objects interacting with one another. This is similar to the way a director works with the cast of actors in a play. The director gives instructions sometimes to a single cast member and at other times to several cast members to coordinate their actions. Methods that specifically reference more than one object are world-level methods. Methods that define behaviors for a single object may be considered class-level methods.

In Section 4-1, our focus is on learning how to create and run your own world-level methods. This section will demonstrate how to call your own method. Calling a method causes Alice to animate the instructions in the method. We revise our storyboard design process to use a technique of breaking the overall task down into abstract tasks and then break each task down into smaller pieces and then define the steps in each piece. This design technique is known as stepwise refinement.

Section 4-2 launches a discussion of parameters. A parameter helps us send information to a method—a form of communication with the method. The information that gets sent to a method can be of many different types (e.g., a numeric value, an object, or some property value such as a color).

Section 4-3 presents an introduction to class-level methods. An advantage of class-level methods is that once new methods are defined, we can create a new class with all the new methods (and also the old methods) as available actions. This is a form of inheritance—the new class inherits methods from the old class.

4-1 World-level methods

In this section, an example will be used to demonstrate how to organize several primitive instructions into a method. Each method in a program performs its own job, but all the methods work together to create the overall animation. A method allows the programmer to think about a collection of instructions as if it was just one instruction—this is called abstraction. Furthermore, each individual method can be tested to be sure it works properly. Finding a bug in a few lines of code is much easier than trying to find a bug in hundreds of lines of code where everything is interrelated.

A problem

In the *FirstEncounter* world, a spiderRobot has traveled through space to land on the surface of a distant moon. The spiderRobot surprisingly encounters an alienOnWheels, investigates the alien, then sends a message back to earth. To construct this first animation program, we used a rather straightforward technique of designing a storyboard and then writing the program instructions all in *World.my first method*. The code for this program (as written in Chapter 2) is shown in Figure 4-1-1.

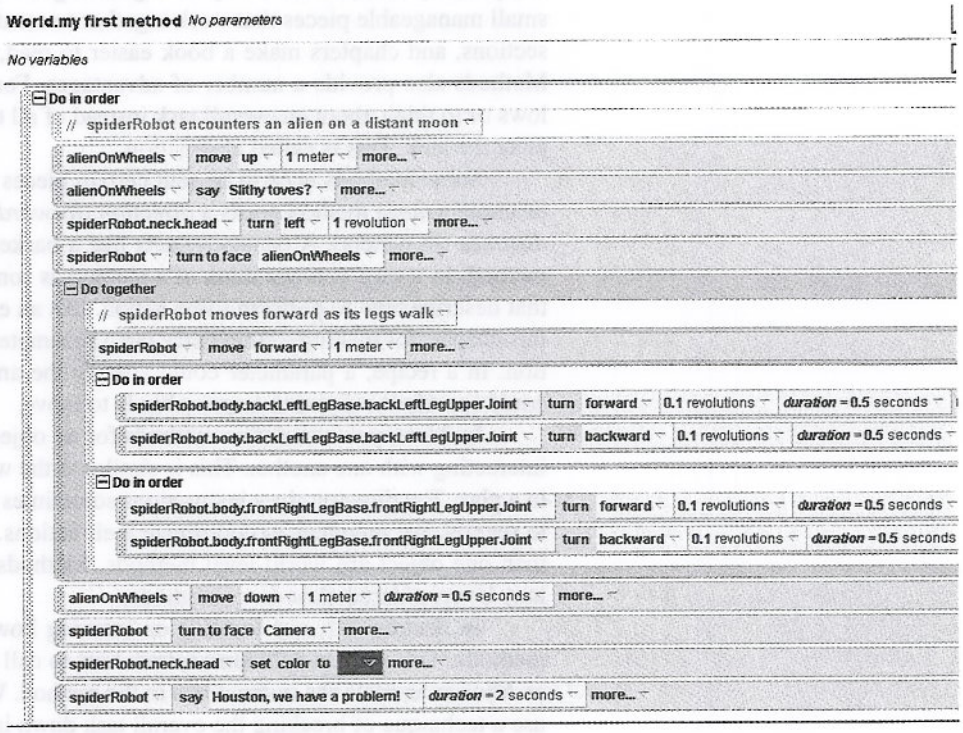


Figure 4-1-1. Program code for First Encounter world (as created in Chapter 2)

As we constructed this program, the code just seemed to grow and grow until we ended up with a large number of instructions all together. If we continue to write our programs this way, our program code is likely to grow to hundreds of lines of code all in one big block. The problem with many lines of code all in one big block is that it becomes difficult to read and even more difficult to find and remove bugs.

We need a way to better organize the instructions to make it easier to read and debug a program. One way to do this is to organize the instructions into smaller methods. Once the method is defined, we can tell Alice to run it from one main method. Another advantage of using small methods is that the methods can be called from several different places in the program, without having to copy all the instructions again and again into the editor. The following illustrates how to organize your program by using methods.

Creating your own method

If we had a chance to start again to write the First Encounter animation program, this time using methods, how would we begin? Well, the first step is to think about the animation in terms of large tasks, without all the details. For the First Encounter animation, we could write the storyboard like this:

```
Do in order
  surprise–spiderRobot and alienOnWheels surprise each other
  investigate–spiderRobot gets a closer look at alienOnWheels
  react–alienOnWheels hides and spiderRobot sends message
```

The next step is to break down each major task of our storyboard design into simpler steps. As an example, let's do this for the *surprise* task.

```
Do in order
  surprise–spiderRobot and alienOn Wheels surprise each other
  investigate–spiderRobot moves closer to alienOnWheels
  react–alienOn Wheels hides and spiderRobot sends message
```

```
surprise

Do in order
  alien moves up
  alien says "Slithy toves?"
  robot's head turns around
```

The steps in the *surprise* storyboard are the same as the first four instructions in the program we wrote in Chapter 2. But now we are thinking of these instructions as one abstract idea—the spiderRobot and alienOnWheels surprise each other. In a similar way, we can construct storyboards for the *investigate* and *react* tasks.

```
investigate

Do in order
  robot turns to look at alien
  Do together
    robot moves forward (toward the alien)
    robot's legs walk
```

```
react

  Do in order
    alien moves down
    robot turns to look at the camera
    robot's head turns red (to signal danger)
    robot says "Houston, we have a problem!"
```

The process of breaking a problem down into large tasks and then breaking each task down into simpler steps is called stepwise refinement—a design technique used in many of the examples presented in the next few chapters. Now that we have organized our storyboard into three abstract tasks, the program can be constructed by writing a method for each task. Let's write a method for the task named *surprise*. (Notice that method names conventionally begin with a

lowercase letter.) We know that when objects carry out instructions in an Alice world, they may be acting alone—that is, not affecting or being affected by other objects. On the other hand, objects are often interacting in some way with other objects. For a method where objects are interacting with other objects, we write a world-level method. The *surprise* method will involve both the spiderRobot and the alienOnWheels, so it will be a world-level method.

Although the instructions we will use are the same as those presented in Chapter 2, we will go through some of the steps of creating the animation once again (starting with the initial scene, containing no program code) so as to illustrate the process of writing our own method. In the object tree, the World object is selected and then the *methods* tab in the Details panel (located in the lower left of the screen). Then the **create new method** button (in the methods detail panel) is clicked. Figure 4-1-2 illustrates the **create new method** button selection. When the **create new method** button is clicked, a popup box allows you to enter the name of the new method. In this example, we entered *surprise*.

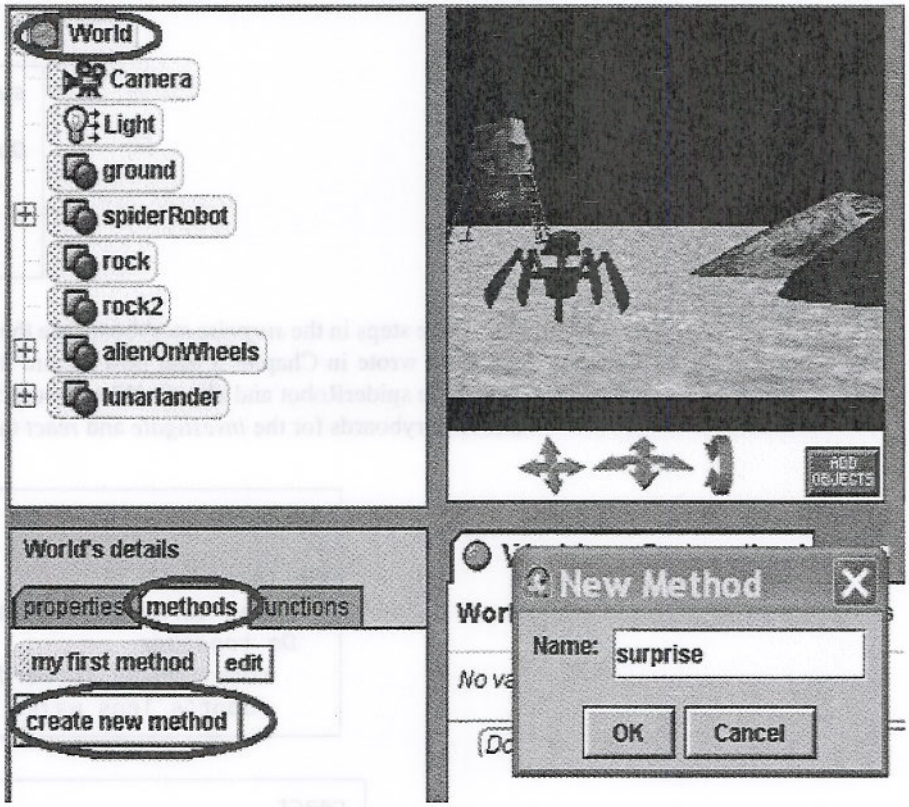


Figure 4-1-2. Selecting World's **create new method** button and entering the name of the new method

Alice automatically opens a new editor tab, where the code can be written for the new method. (See Figure 4-1-3.)

Note that the particular method being edited has its tab colored yellow, and that all other method tabs are greyed out. To switch back and forth between the editor tabs, click the edit button to the right of the method you want to work on (in the details panel on the left).

Now we can add instructions for the *World.surprise* method, using instructions similar to those in the program in Chapter 2, as in Figure 4-1-4.

Note: If the **Play** button is clicked at this time, the animation will NOT run. Although the *surprise* method has been defined, Alice has not been told to execute it. That is, the method has

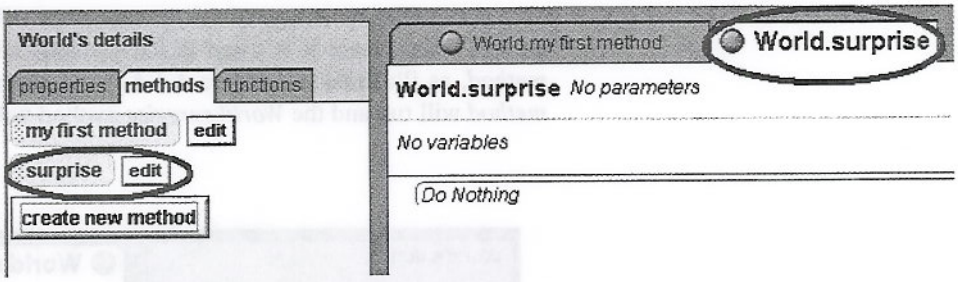


Figure 4-1-3. The new *World.surprise* code editor

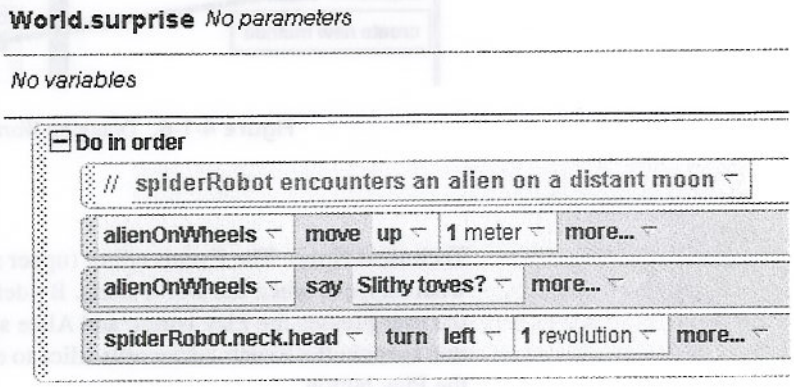


Figure 4-1-4. Defining the *World.surprise* method

not been called into action. (Another phrase commonly used for "calling a method" is "invoking a method"—but in this book we will use the phrase "calling a method.")

Calling a method

How is your new method called into action? You have no doubt discovered that when you (as the human "user") click on the Play button, Alice automatically executes *World.my first method*. You can see why this happens by looking carefully at the Events editor, located in the top right of the Alice interface as seen in Figure 4-1-5. The instruction in this editor tells Alice *When the world starts, do World.my first method*. We didn't put this instruction here—the Alice interface is automatically programmed this way. So, when the user clicks on the Play button, the world starts and *World.my first method* is called.

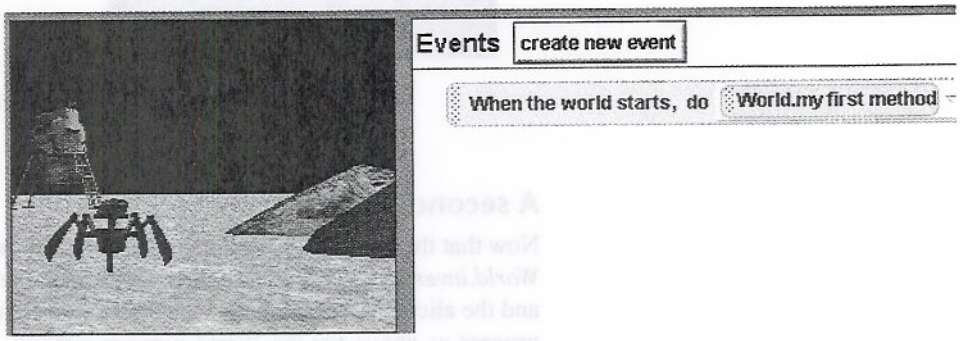


Figure 4-1-5. *When the world starts* is linked to *World.my first method*

Let's take advantage of this arrangement. All we have to do is drag the *World.surprise* method from where it is listed in the methods tab of the details panel into *World.my first method*, as illustrated in Figure 4-1-6. Now, whenever the Play button is clicked, *my first method* will run and the *World.surprise* method will be called.

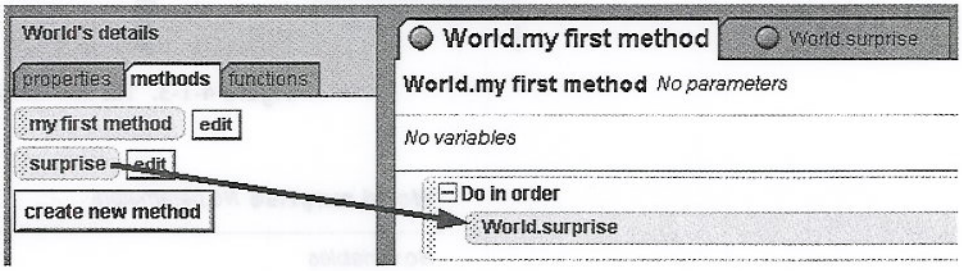


Figure 4-1-6. Dragging *World.surprise* into *World.my first method*

Technical Note: The Events editor (upper right of the Alice interface) can be used to modify what happens when the world starts. By default (meaning "unless you tell Alice otherwise"), the user presses the Play button and Alice starts executing *World.my first method*. Modifying this event in the Events editor tells Alice to execute a different method when the user clicks on the Play button.

To modify the *When the world starts* event, click on the right of the *World.my first method* tile in the Events editor. Then, select *surprise* from the drop down list, as seen in Figure 4-1-7. Now, when the world starts, the *World.surprise* method will run instead of *World.my first method*.

Figure 4-1-7. Modifying the *When the world starts* event

A second method

Now that the *World.surprise* method is complete, we can construct the second method, named *World.investigate*. *World.investigate* is a world-level method because both the spiderRobot and the alienOnWheels objects are directly referenced in the instructions. Following the same process as above for the *World.surprise* method, we created a *World.investigate* method as shown in Figure 4-1-8.

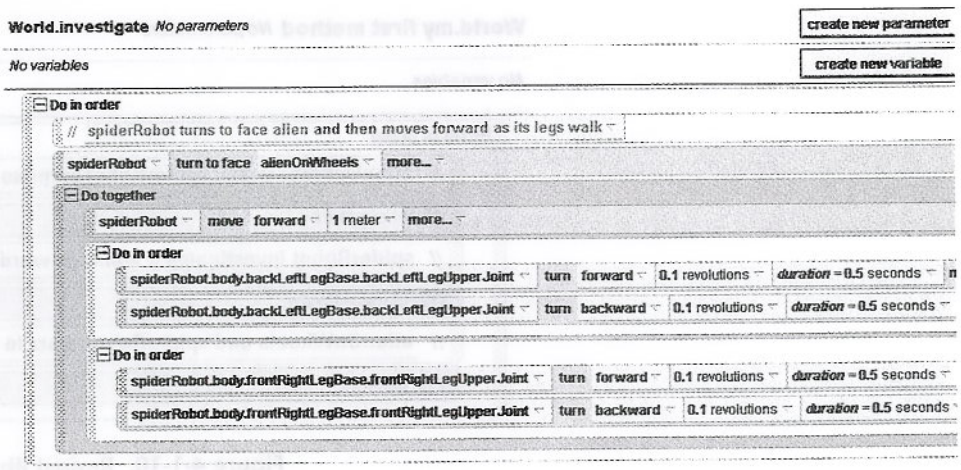


Figure 4-1-8. Defining the *World.investigate* method

The third method

Finally, a world-level method is created for the *react* task. As with the *World.surprise* and *World.investigate* methods, *World.react* is world-level because the spiderRobot and the alienOnWheels are both directly referenced in the instructions. The *World.react* method is shown in Figure 4-1-9.

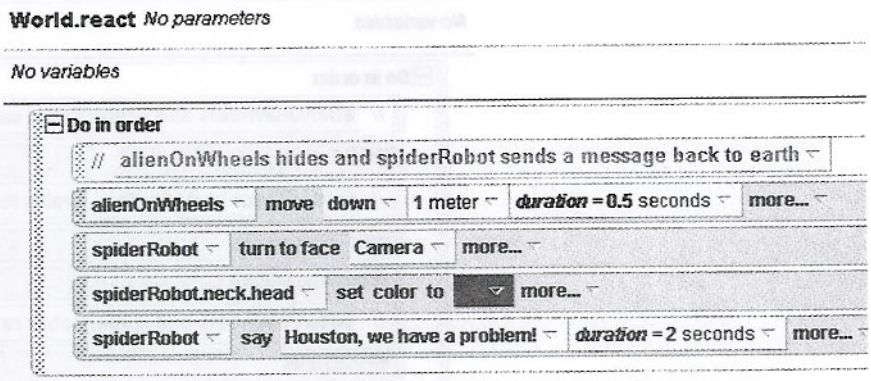


Figure 4-1-9. A world-level method, *react*

Now, we are ready to call each of the methods in *World.my first method*. The code is illustrated in Figure 4-1-10.

Compare the revised program code in Figure 4-1-10 to the code in Figure 4-1-1. One thing that you should notice immediately is that the revised code in Figure 4-1-10 has fewer lines of code. The overall program has been broken down into methods—small collections of instructions that carry out specific abstract tasks. *World.my first method* acts as a driver that calls the methods. This organization makes the program easier to read and understand. Also, writing and testing short methods makes it easer to debug your programs.

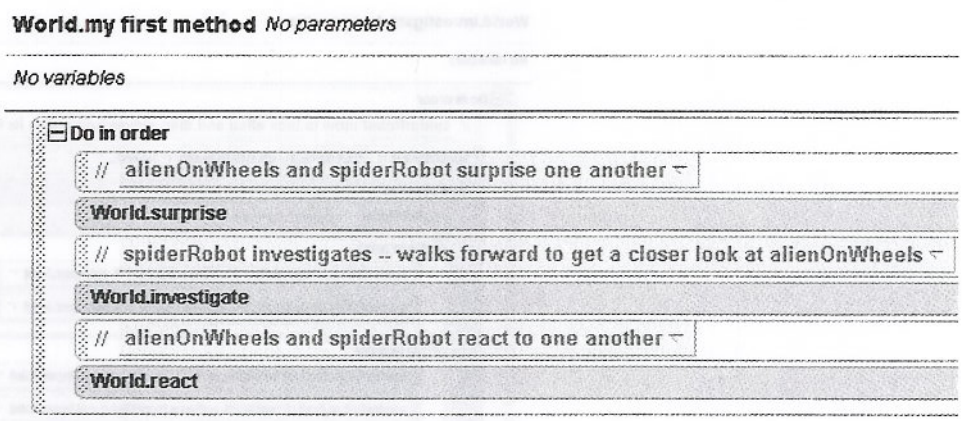


Figure 4-1-10. Revised *World.my first method*

One benefit of writing methods this way is that some methods may be called more than once. For example, the *World.investigate* method can be called multiple times to make the spiderRobot walk toward the alienOnWheels. This reduces the amount of program code and saves us time as we create our programs. Figure 4-1-11 illustrates two calls to the *World.investigate* method.

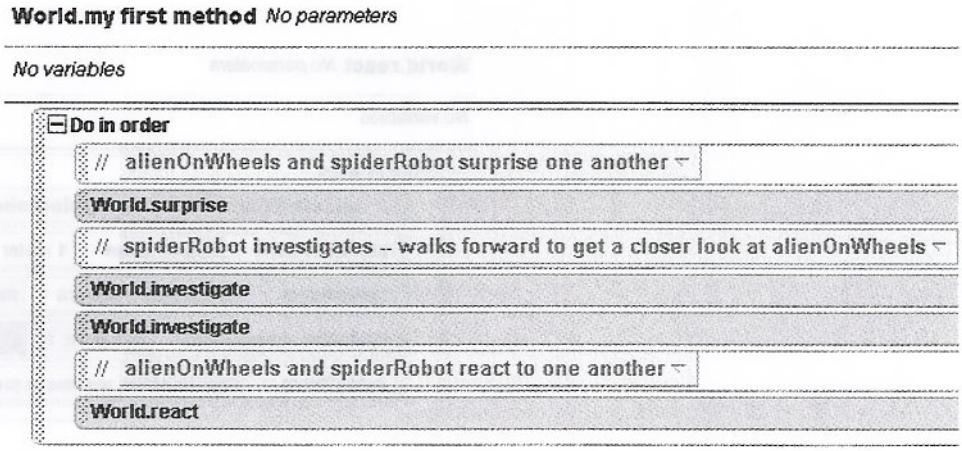


Figure 4-1-11. The *World.investigate* method is called twice

Deleting a method

Built-in methods and methods that are predefined for a model should not be deleted. In other words, delete a method only if you have written the method and want to discard it. Before deleting a method, delete all calls to the method and then close its edit window (right click on the method's edit panel and select "close"), if open. To delete a method, drag its name tile (from the methods list) to the trash bin, as illustrated in Figure 4-1-12.

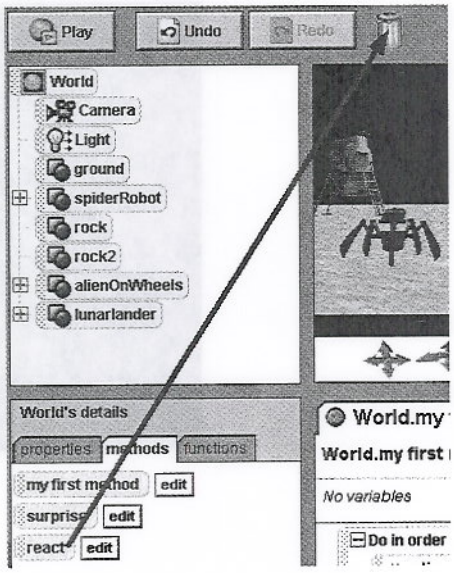


Figure 4-1-12. Deleting a method you have written

4-2 Parameters

It is clear from the examples in the preceding section that one program may be made up of several methods. Each method is its own small block of instructions, designed to perform a specific task when requested. We can appreciate that some communication might need to occur when a method is called. In this section, we look at parameters. Parameters are used to communicate with a method. We arrange to communicate values (for example, a number or a color) or names of objects from one method to another by using parameters in our methods.

Example

An example world will illustrate the creation and use of parameters. For a spring concert, our entertainment committee has hired a popular music group—the Beetle Band. Our job is to create an animation to advertise the concert. In the animation, each band member wants to show off his musical skills in a short solo performance.

Setting the stage: Figure 4-2-1 shows an initial scene. The world is simple to set up. To a new world, add a table as a stage (Furniture on CD or Web gallery), georgeBeetle, lennonBeetle, paulBeetle, and ringoBeetle (Animals). Give each band member a musical instrument: bass, saxophone, timbalesCowBell, and guitar (Musical Instruments).

Make the vehicle of each musical instrument be the band member who plays the instrument. For example, the vehicle of the bass guitar is georgeBeetle. Using the vehicle property is a convenient way to make the musical instrument move with the band member as the band member jumps up and down in his solo.

In the scene in Figure 4-2-1, we used a table to simulate a stage for the band and conserve memory. (Worlds that take up less memory load faster.) If you want a fancier scene, you could use the concert stage (Environments on CD or Web gallery) as shown in Figure 4-2-2.

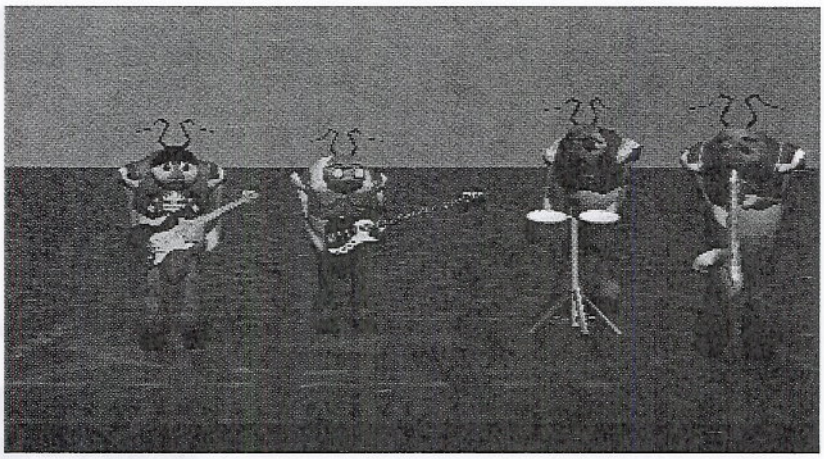


Figure 4-2-1. Beetle Band on stage

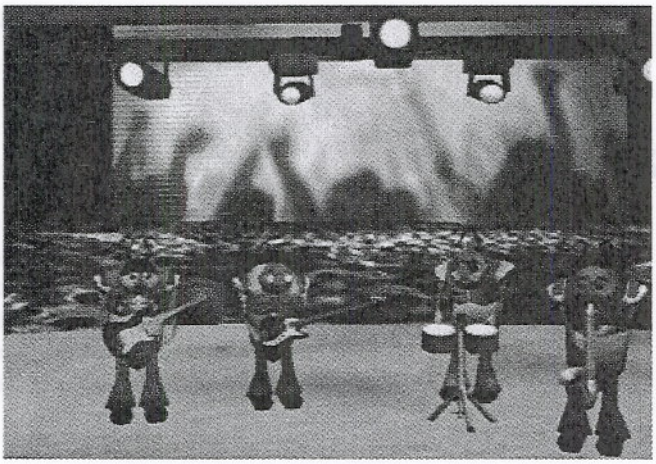


Figure 4-2-2. Beetle Band on a concert stage

Storyboard

The storyline for this animation is that each band member will perform a short solo—the band member will jump up and down at the same time as a sound plays. (If your computer does not have sound, you can have the band member *say* lyrics, rather than play a sound.) Let's create a storyboard for each band member's solo. Because we have four band members (georgeBeetle, ringoBeetle, paulBeetle, and lennonBeetle), four textual storyboards can be composed:

```
Do together
  Do in order
    georgeBeetle move up
    georgeBeetle move down
  play sound
```

```
Do together
  Do in order
    ringoBeetle move up
    ringoBeetle move down
  play sound
```

```
Do together
  Do in order
    paulBeetle move up
    paulBeetle move down
  play sound
```

```
Do together
  Do in order
    lennonBeetle move up
    lennonBeetle move down
  play sound
```

Now a method can be written for each storyboard. We begin with a method for the solo performed by georgeBeetle. Of course, the bass guitar will move with georgeBeetle when he moves up and down because the bass guitar's vehicle property is set to georgeBeetle. In a *Do together* block, a *play sound* instruction will be used to play a sound at the same time as georgeBeetle and the bass guitar instrument move up and down. Before a sound can be played, a sound file must be imported into Alice. (Alice will play either MP3 or WAV sound files.) Alice provides a few sounds for your use. Many non-copyrighted sound files are available on the internet. You can also use sound editing software to record your own sound files or you can purchase sound recordings on the internet (for educational projects, only).

In this world, a sound is associated with a musical instrument. For this reason, the sound will be imported for the instrument object. To illustrate how to import a sound file, let's import a bass guitar sound for the bass instrument. Click on the bass instrument in the Object tree and then on the **import sound** button in the bass object's property list. A file selection box appears. Navigate to a directory where you have stored your sound files and then select the sound file to be used, as shown in Figure 4-2-3. (Note that our sound files have been stored in a directory we created and named Sounds, but you may have them stored in some other directory. It may be necessary to navigate through several folders on your computer to find a sound file you wish to import.) Once the file has been selected, click on the **Import** button.

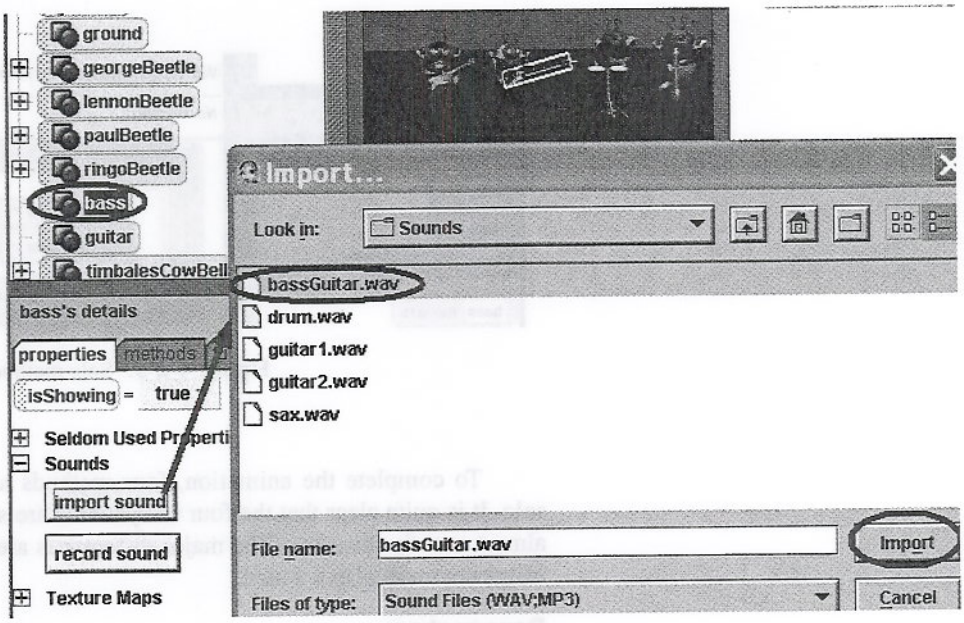


Figure 4-2-3. Importing a sound file for an object

The name of the sound file automatically appears in the list of properties for the bass object, as shown in Figure 4-2-4. The green arrow is a preview button and can be clicked to test the sound file.

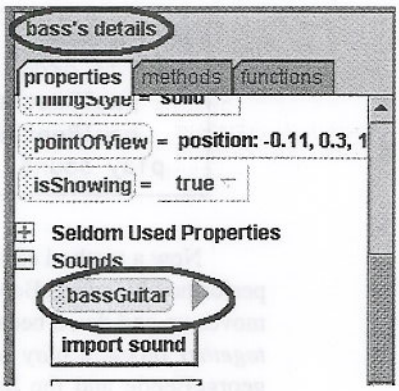


Figure 4-2-4. Imported sound file in the properties panel

For this world, we used this technique to import four sound files—one for each musical instrument (bass, sax, timbalesCowbell, and guitar) in the world. Now that the sound files have been imported, a method can be created for a solo performed by one of the members of the Beetle Band. The code for a solo by georgeBeetle is shown in Figure 4-2-5. The *play sound* instruction is created by dragging the play sound method tile into the editor and selecting the appropriate sound from a popup menu. The *play sound* instruction automatically includes the *duration* of the sound. The bassGuitar sound used in our example will play for a *duration* of 1.845 seconds. If you wish to change the duration of the sound, we recommend that you use a sound editor to modify the sound (rather than modify the duration in the Alice instruction).

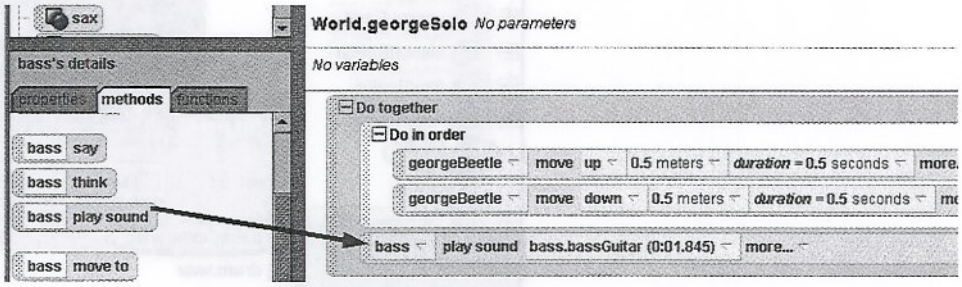


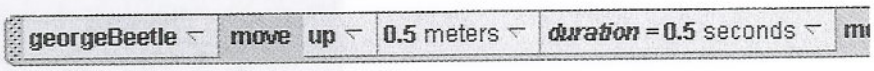
Figure 4-2-5. Solo method for georgeBeetle

To complete the animation, four methods are needed—one for each band member's solo. It is quite clear that the four storyboards are strikingly similar and four methods will be almost exactly the same. The major differences are which band member will solo and which instrument will play a sound.

Parameters

This is where parameters come in. A parameter allows you to send information to a method when the method is called. You have been using parameters all along. Most primitive methods have parameters that allow you to send information to the method. For example, a *move* instruction has parameters that allow you to send in the direction, distance, and duration. In the

move instruction shown below, the direction is up, the distance is .25 meters and the duration is 0.5 seconds. We say that the values are sent in as arguments to the parameters.



We can use parameters in our own methods. In the Beetle Band example, the four storyboards are so similar that we can collapse them into one storyboard by using a parameter to communicate to the method which band member will perform the solo. The storyboard with a parameter is:

```
solo

Parameter: bandMember
Do together
  Do in order
    bandMember move up
    bandMember move down
  play sound
```

The *bandMember* parameter name (an arbitrary name) is taking the place of the name of the specific object that will perform the solo. You can think of a parameter as acting like someone who stands in a cafeteria line for you until you arrive—sort of a placeholder. For example, when georgeBeetle is sent in as an argument, then *bandMember* represents georgeBeetle. But, when lennonBeetle is sent in as an argument, *bandMember* represents lennonBeetle. By creating a parameter, we can write just one method (instead of four methods) and use the parameter to communicate which band member will perform the solo.

An object parameter

A new world-level method, named *solo*, is created. The editor creates a new editor tab for the World.solo method, as seen in Figure 4-2-6. A **create new parameter** button automatically appears in the upper right corner of the editor. When the **create new parameter** button is clicked, a dialog box pops up as shown in Figure 4-2-7. The name of the parameter is entered and its *type* is selected. The *type* of a parameter can be a *Number*, *Boolean* ("true" or "false"), *Object*, or *Other* (for example, a color or sound). In this example, the name of the parameter is *bandMember* and its type is *Object*.

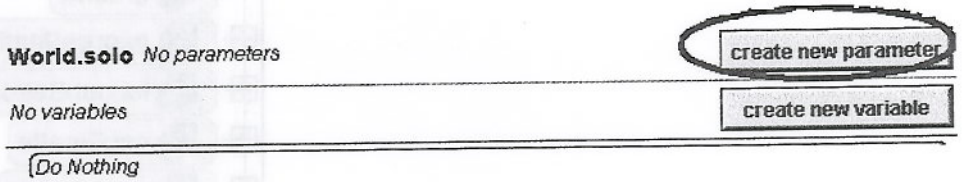


Figure 4-2-6. *World.solo* method pane

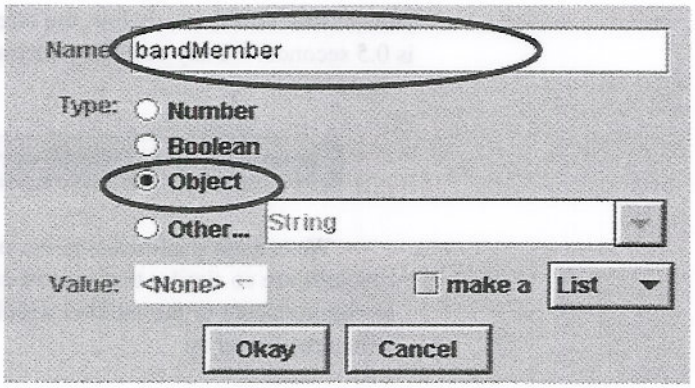


Figure 4-2-7. Enter the name and select a type for a parameter declaration

When completed, the parameter name is in the upper left of the method panel, as shown in Figure 4-2-8. Now, whenever the *World.solo* method is called, an object must be sent as an argument to the *bandMember* parameter.

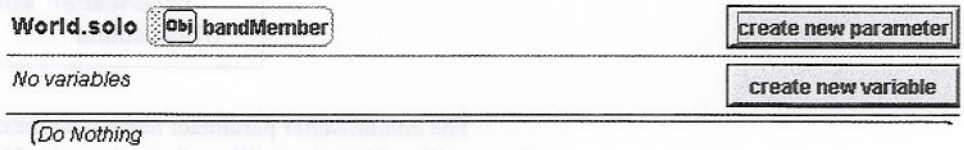


Figure 4-2-8. Resulting parameter

Now, we can translate the storyboard into program code. The first part of the storyboard is a *Do in order* block to make the *bandMember* jump (move up and then down). Intuitively, we look at the Object tree to find *bandMember* so that its *move* instruction can be dragged into the editor, but *bandMember* is not in the Object tree. (See Figure 4-2-9.) This makes sense because, as mentioned earlier, *bandMember* is not an actual object—it is acting as a placeholder for an object.

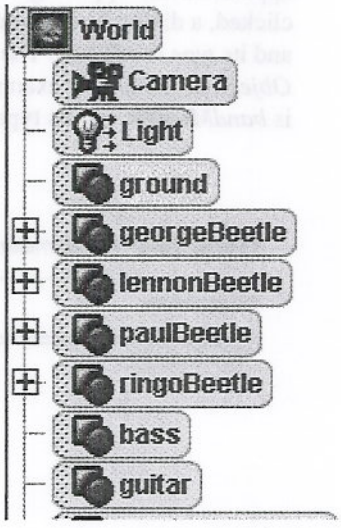


Figure 4-2-9. The *bandMember* parameter is not in the Object tree

Instead of dragging a *move* method into the editor, drag the parameter tile into the editor. For example, in Figure 4-2-10, the *bandMember* parameter tile is dragged into the editor and *move, up,* and 1/2 *meter* are selected from the popup menus.

Figure 4-2-10. Instruction for an arbitrary object

The resulting instruction looks like this:

Using the same procedure, another instruction is written to *move bandMember* down 0.5 meters. The duration for the *move up* and *move down* instructions is selected as 0.5 seconds. The resulting method is shown in Figure 4-2-11.

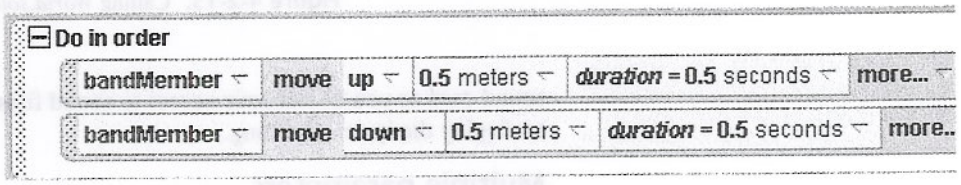


Figure 4-2-11. The *bandMember* jumps (moves up and down)

Test with arguments

This is a good time to save and test the code. To test the *World.solo* method, the *solo* method is called from *my first method*. When *solo* is dragged into *my first method*, a popup menu (Figure 4-2-12) allows the selection of an object that *bandMember* will represent for that call of the method. To be certain the *solo* method works for each Beetle Band musician, four statements are written, as seen in Figure 4-2-13. In this example, georgeBeetle, lennonBeetle, ringoBeetle, and paulBeetle are each used as an argument in a call to the *solo* method. For example, in the first call the *solo* method will be performed with *bandMember* representing georgeBeetle; in the second call *bandMember* will represent lennonBeetle.

Completing the animation

You may have noticed that the above code does not yet complete the animation. In each solo, the band member should not only move but also a musical instrument should play a sound. An instruction is needed in the *solo* method to play a sound. If your computer does not have a

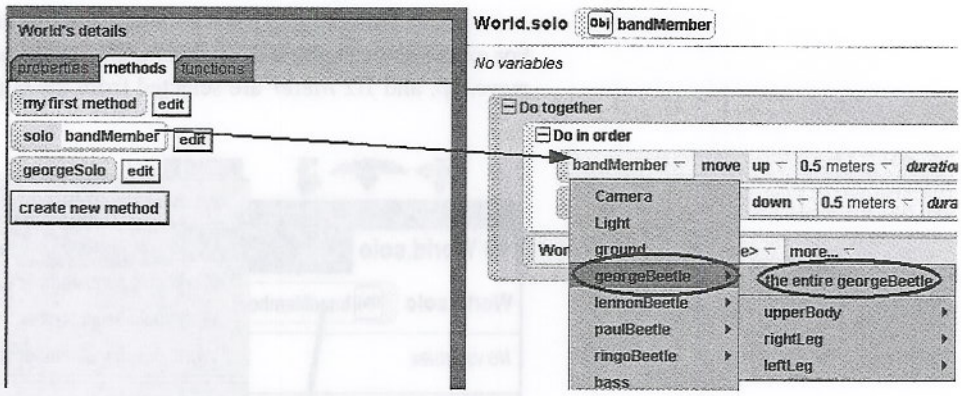


Figure 4-2-12. Selecting an argument for the *bandMember* parameter

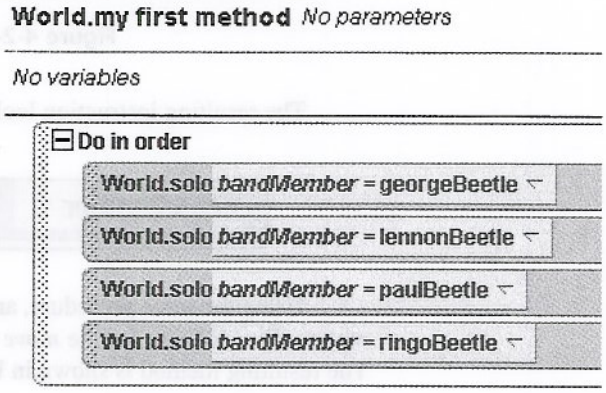


Figure 4-2-13. Calling *World.solo* with different arguments

sound card or you do not have access to sound files, a *say* instruction (see below) can be used to display the lyrics of a song.

Multiple parameters

Each *bandMember* plays a different musical instrument, and each instrument should have a different sound. (In this example, we want a bass sound for georgeBeetle, saxophone for paulBeetle, drum for ringoBeetle, and guitar for lennonBeetle's performance.) Let's create two additional parameters: *instrument*, for the object playing the sound, and *music*, for the sound to be played. The type of the *instrument* parameter is Object (created in the same way as *bandMember*, above) and the type of the *music* parameter is Sound. Figure 4-2-14 illustrates creating a Sound parameter named *music*.

As with the *bandMember* parameter above, the *instrument* and *music* parameters are placeholders, and do not appear in the Object tree. When a parameter does not represent an object, the parameter tile often must be dragged into the editor to replace the tile in an existing instruction. In this example, the instrument tile is dragged in to the editor to create a play sound instruction, as illustrated in Figure 4-2-15. A popup menu allows the selection of *play sound* as the method and *music* as the sound.

The code shown in Figure 4-2-15 is complete. A *Do in order* block is used to have the *bandMember* move up and then down. And, a *Do together* block causes the sound to play at the same time as the *bandMember* moves up and down.

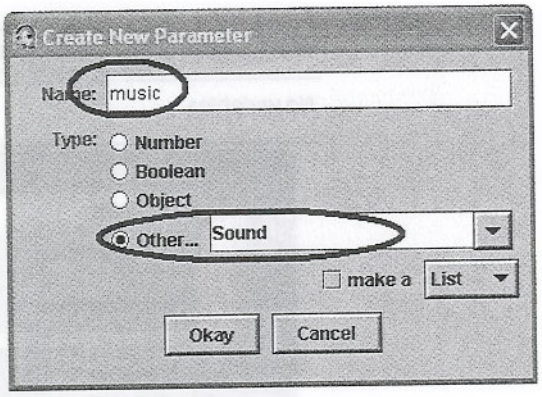


Figure 4-2-14. The *music* parameter *Type* is *Sound*

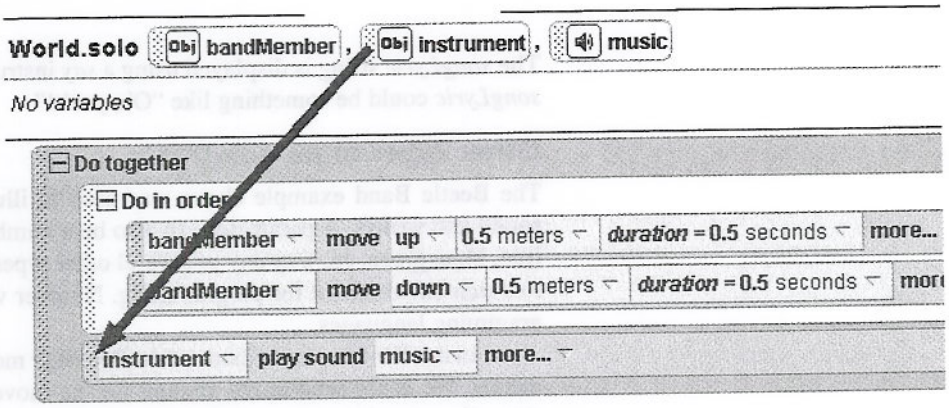


Figure 4-2-15. Dragging the *instrument* and *music* parameters in to create a *play sound* instruction

Calling the revised method

Calls to *World.solo* are revised in *my first method* to pass in three arguments (two objects and a sound), as in Figure 4-2-16.

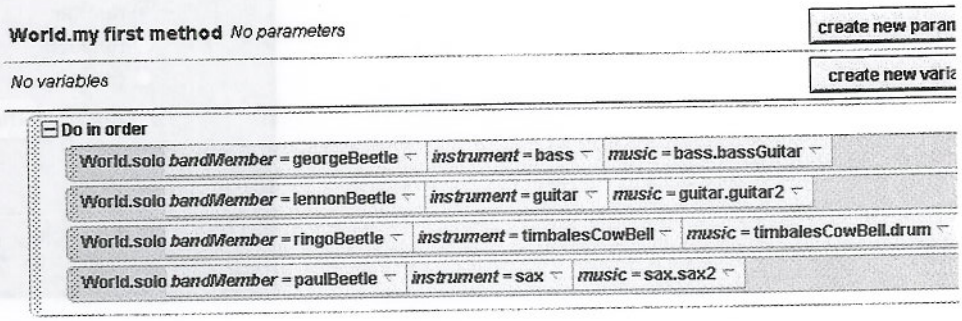


Figure 4-2-16. Completed *World.my first method*

As mentioned above, this animation can be completed without the use of sound. An alternate version of the *solo* method is shown in Figure 4-2-17. A string parameter, *songLyric*, is used instead of a sound parameter. (A string is just several text characters or words.)

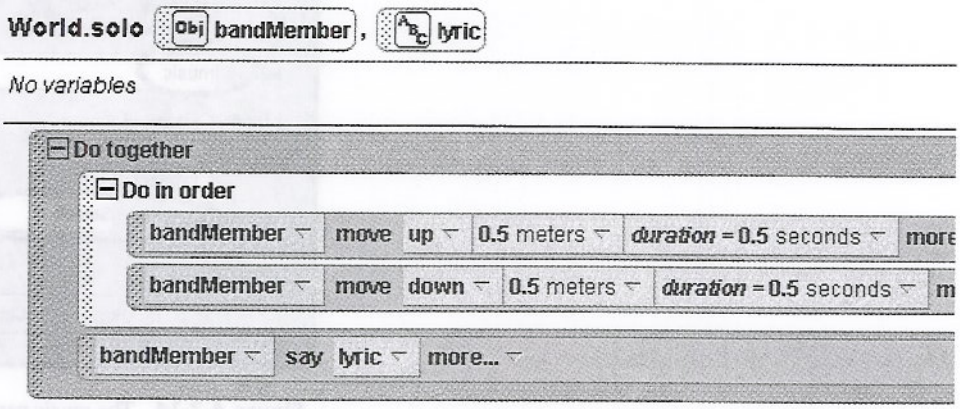


Figure 4-2-17. Lyric version of *solo*

The *songLyric* string is displayed using a *say* instruction. To call this method, an argument for *songLyric* could be something like "Oh, yeah!"

Other types of parameters

The Beetle Band example above was used to illustrate three types of parameters: objects, sound and strings. A parameter can also be a number, a Boolean value (*true* or *false*), a color (red, blue, green, etc.), or any of several other types. Each of these types of values contributes to a rich environment for programming. Number values play an important role in many programming languages.

In the *World.solo* method, a *bandMember* moved up and down an arbitrary amount, 0.5 meters. We could send in the amount for the move by adding a number parameter. The first step, of course, is to create a number parameter, as shown in Figure 4-2-18. We used the name *height* and selected Number as the type.

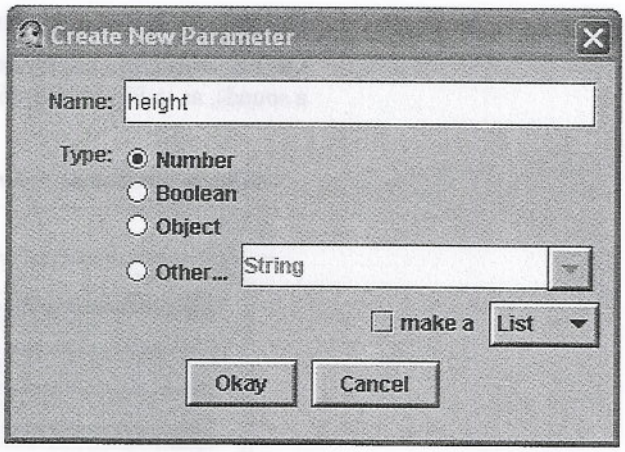


Figure 4-2-18. Creating a number parameter

The *height* parameter can then be dragged into the *move up* and *move down* instructions. Figure 4-2-19 shows a revised *solo* method where the *bandMember* moves up and down an amount specified by the *height* parameter.

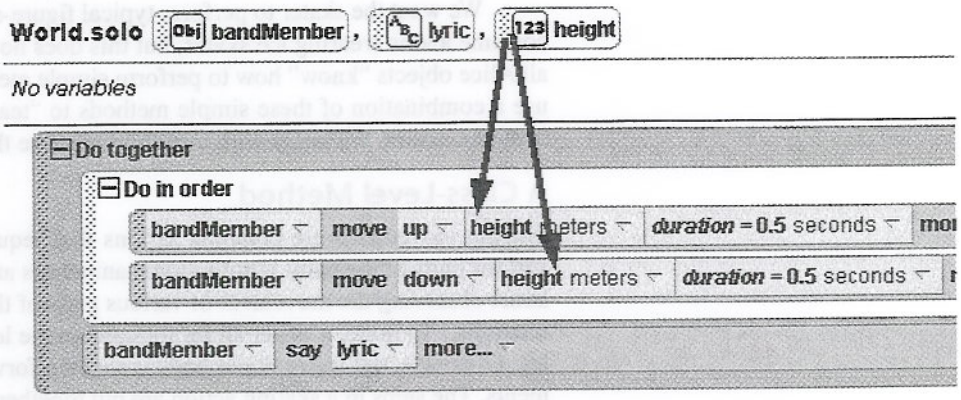


Figure 4-2-19. Using a number parameter

4-3 Class-level methods and inheritance

The galleries of 3D models in Alice give us a choice of diverse and well-designed classes of objects for populating and creating a scenic backdrop in a virtual world. When you add an instance of a 3D model to an Alice world, it already "knows" how to perform a set of methods—*move, turn, roll*, and *resize* (to name a few). The 3D model class already defines these methods. After writing several programs, it is natural to think about extending the actions an object "knows" how to perform.

In this section, you will learn how to write new methods that define new actions to be carried out by an object acting alone (rather than several objects acting together). We call these class-level methods. Class-level methods are rather special, because we can save an object along with its newly defined method(s) as a new kind of object. In Alice, the new kind of object is saved as a new 3D class model. Later instances of the new class still know how to perform all the actions defined in the original class but will also be able to perform all the actions in the newly defined methods. We say that the new class inherits all the properties and methods of the original class.

Example

Consider the iceSkater shown in the winter scene of Figure 4-3-1. (The IceSkater class is from the People collection, and the Lake class is from the Environments collection in the gallery.)

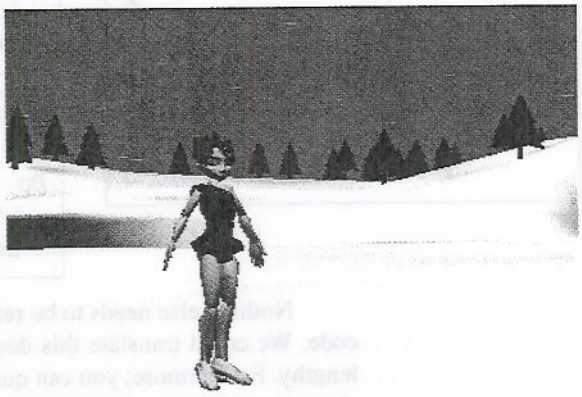


Figure 4-3-1. The iceSkater

We want the skater to perform typical figure-skating actions. She is dressed in a skating costume and is wearing ice skates, but this does not mean she knows how to skate. However, all Alice objects "know" how to perform simple methods such as *move, turn*, and *roll*. We can use a combination of these simple methods to "teach" the ice skater how to perform a more complex action. We begin with a method to make the skater perform a skating motion.

A Class-Level Method

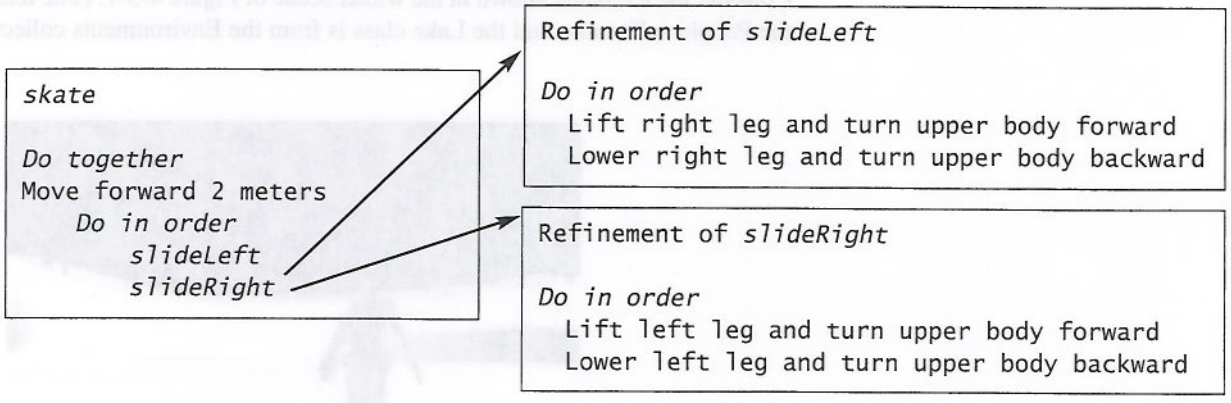
Skating movements are complex actions that require several motion instructions involving various parts of the body. (Professional animators at Disney and Pixar may spend many, many hours observing the movement of various parts of the human body so as to create realistic animations.) To skate, a skater slides forward on the left leg and then slides forward on the right leg. Of course, the entire skater body is moving forward as the legs perform the sliding movements. The steps in a skating action are put together as a sequence of motions in a storyboard, as shown next.

```
skate

Do together
Move skater forward 2 meters
    Do in order
        slide on left leg
        slide on right leg
```

Notice that the storyboard breaks down the skating action into two pieces—slide on the left leg and slide on the right leg. The sliding motions can each be broken down into simpler methods. Breaking a complex action down into simpler actions is called refinement. Here we are using a design technique known as stepwise refinement. We first describe general actions, and then break each action down into smaller and smaller steps (successively refined) until the whole task is defined in simple actions. Each piece contributes a small part to the overall animation; the pieces together accomplish the entire task.

The following diagram illustrates the refinement of the *slideLeft* and *slideRight* actions. The actions needed to slide on the left leg are to lift the right leg and turn the upper body slightly forward. Then, lower the right leg and turn the upper body backward (to an upright position). Similar actions are carried out to slide on the right leg.



Nothing else needs to be refined. We are now ready to translate the design into program code. We could translate this design to instructions in just the one method, but it would be lengthy. Furthermore, you can quickly see that we have used stepwise refinement to break the skate task down into distinct pieces. So, we will demonstrate how to write several small methods and make them work together to accomplish a larger task.

Skate is a complex action that is designed specifically for the iceSkater and involves no other objects. Likewise, the *slideLeft* and *slideRight* actions are designed specifically for the iceSkater. The methods should be written as class-level methods because they involve only the ice skater. We begin with the *slideLeft* method. The iceSkater is selected in the Object tree and the **create new method** button is clicked in the details panel. We enter *slideLeft* as the name of the new method. (The result is shown in Figure 4-3-2.) Notice that the editor tab is labeled *iceSkater.slideLeft* (not *World.slideLeft*)—indicating that the method is a class-level method.

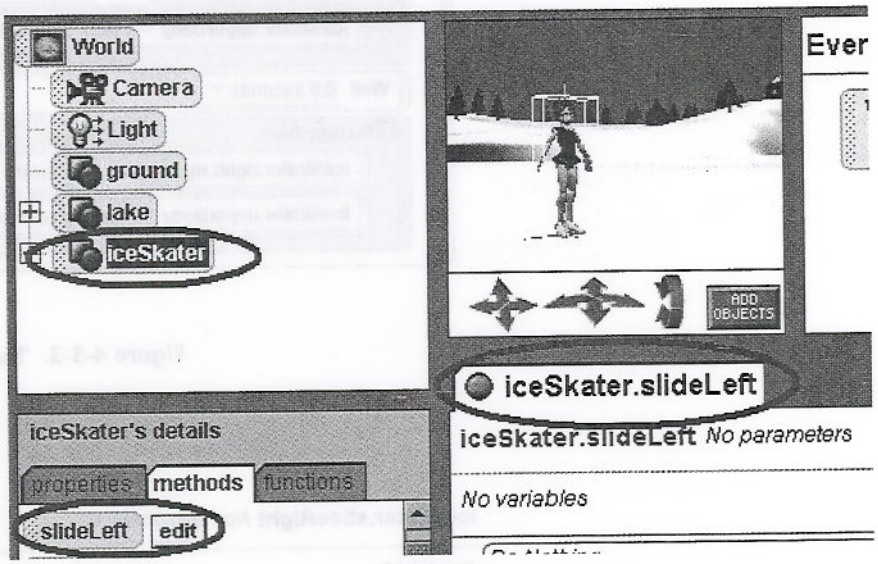


Figure 4-3-2. A *slideLeft* class-level method

To implement the *slideLeft* method, we enter instructions in the editor. The idea is to translate the design into actual program instructions. For example, to translate the design steps for sliding on the left leg, we use the following:

Design step	Instruction
Lift the right leg	*turn* the rightLeg forward
Turn upper body forward	*turn* the upperBody forward
(We inserted a short *wait* to allow time for forward movement.)	
Lower the right leg	*turn* the rightLeg backward
Turn the upper body backward	*turn* the upperBody backward

Figure 4-3-3 illustrates translating the textual storyboard into instructions for sliding on the left leg. The instructions for sliding forward on the right leg are similar. So, writing the *slideRight* method is rather easy. Figure 4-3-4 illustrates translating the textual storyboard into instructions to slide forward on the right leg.

With the *slideLeft* and *slideRight* methods written, we are now ready to write the *skate* method. The *skate* method is really quite simple: *slideLeft* and then *slideRight* at the same time as the entire skater is moving forward. The skate method moves the skater forward and calls the *slideLeft* method and then calls the *slideRight* method. Note that the calls to the *slideLeft* and *slideRight* methods are enclosed in a *Do in order* block, nested within a *Do together*. The *Do together* block is needed to ensure that the instruction that moves the skater forward is performed simultaneously with the left and right sliding motions.

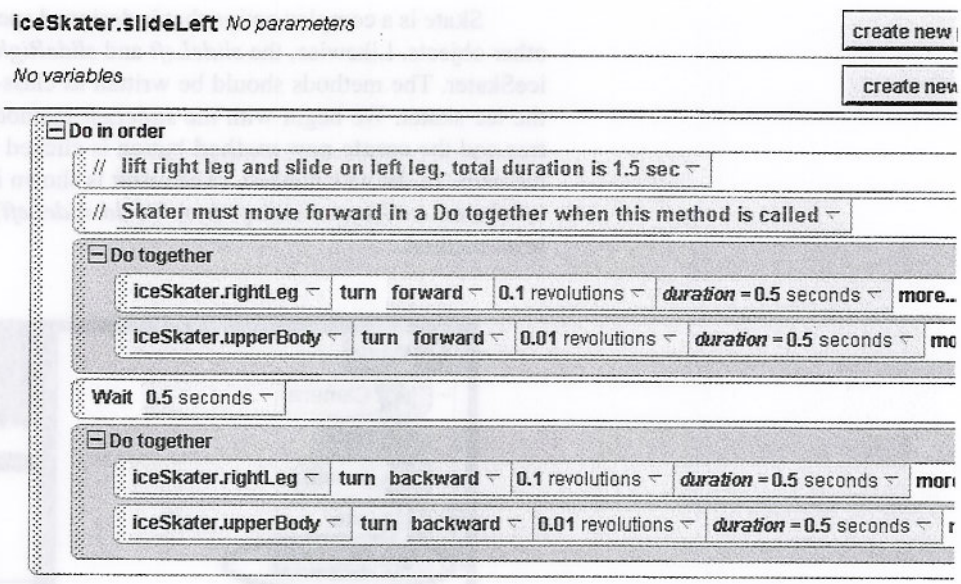


Figure 4-3-3. The *slideLeft* method

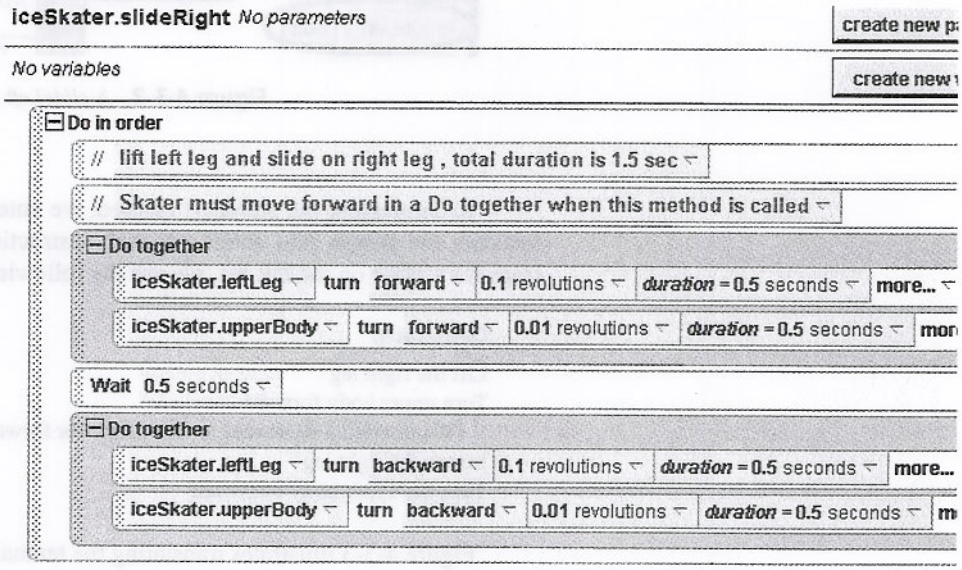


Figure 4-3-4. The *slideRight* method

The duration of the forward movement of the skater is the sum of the durations of the left and right slides. Paying attention to the durations of the instructions in a *Do together* block will help coordinate the motions to begin and end at the same time. In this case, we wanted to coordinate the *slideLeft* and *slideRight* motions with the forward motion of the entire body. When the *skate* method is called, the skater glides forward in a realistic motion. Figure 4-3-5 illustrates the *skate* method.

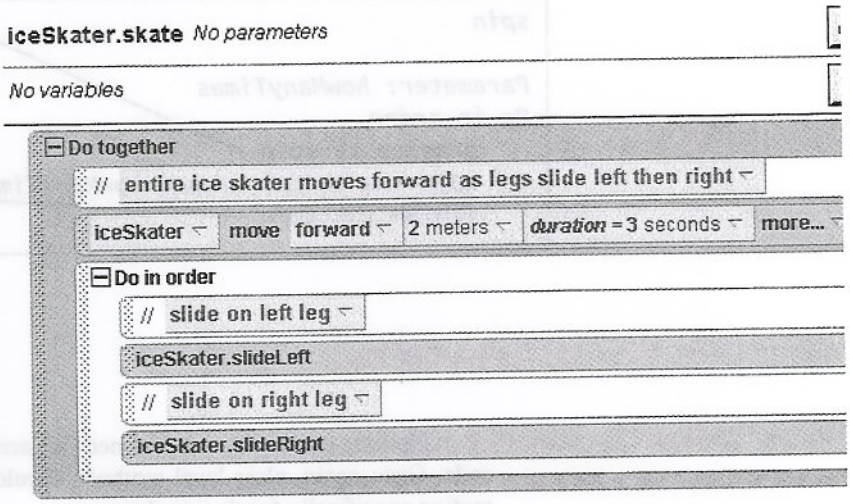


Figure 4-3-5. The *skate* method

A second example—using a parameter

The forward skate motion is truly impressive! Building on this success, let's write a second method to make the ice skater perform a spin. Once again, we will need to write several methods that work together to complete a complex action. In a spin, the ice skater should spin (turn around) several times.

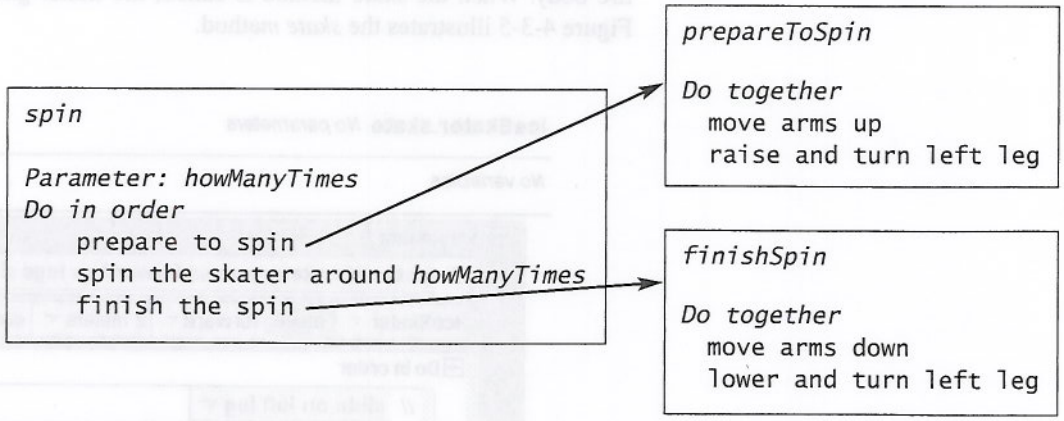
A spin maneuver generally has three parts, the preparation for the spin, the spin itself, and the end of the spin (to finish the spin). In preparation for the spin, the skater's arms and legs change position to provide the strength needed to propel her body around. Then the skater spins around. After the spin, the arms and legs should be repositioned to where they were before the spin. A parameter, *howManyTimes,* is needed to specify the number of times the ice skater will spin around. The storyboard is shown next.

```
spin

Parameter: howManyTimes
Do in order
    prepare to spin
    spin the skater around howManyTimes
    finish the spin
```

We can use stepwise refinement to design the simple steps for each part of the spin. The "prepare to spin" step can be written as a method (*prepareToSpin*) where the skater's arms move up and one leg turns. The "finish spin" step can also be written as a method (*finishSpin*)

to move the arms and legs back to their original positions, prior to the spin. The following diagram illustrates a refinement of the *spin* method.



Nothing else needs to be refined. We are now ready to translate the design into program code. Once again, class-level methods should be used, because we are defining a complex motion specifically for the ice skater.

Figure 4-3-6 illustrates the *prepareToSpin* method, where the ice skater raises her left leg as she lifts her arms.

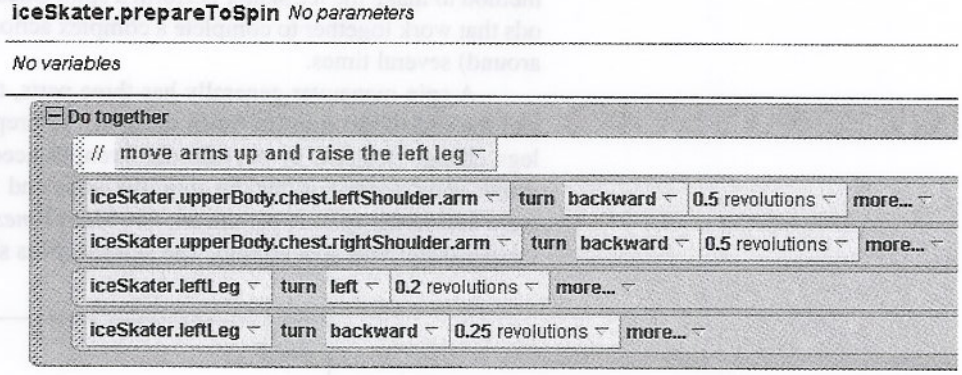


Figure 4-3-6. The *prepareToSpin* method to raise arms and one leg

Figure 4-3-7 presents the *finishSpin* method to reposition the skater's arms and leg to their original positions at the end of her spin.

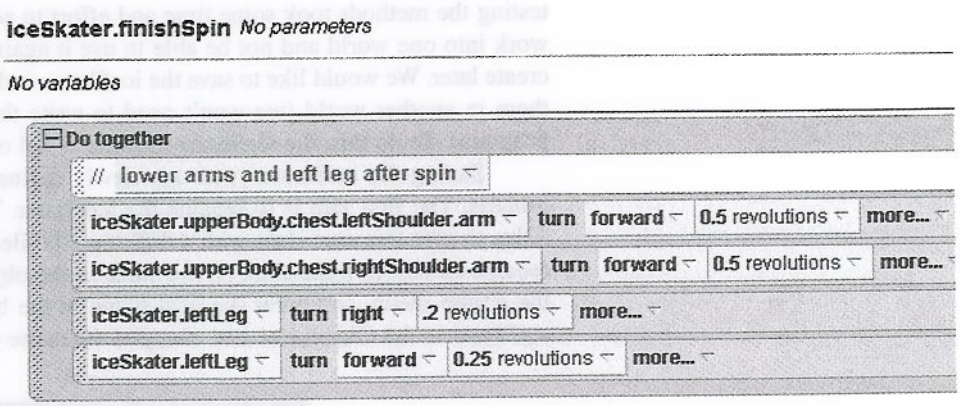


Figure 4-3-7. The *finishSpin* method to lower arms and leg

Now that the *prepareToSpin* and *finishSpin* methods have been written, we can write the *spin* method, as seen in Figure 4-3-8. The *howManySpins* parameter is a number that specifies how many times the skater is to turn around (1 revolution is 1 complete spin around). The order in which the methods are called is important so as to adjust the skater's arms and legs in preparation for the spin and after the spin.

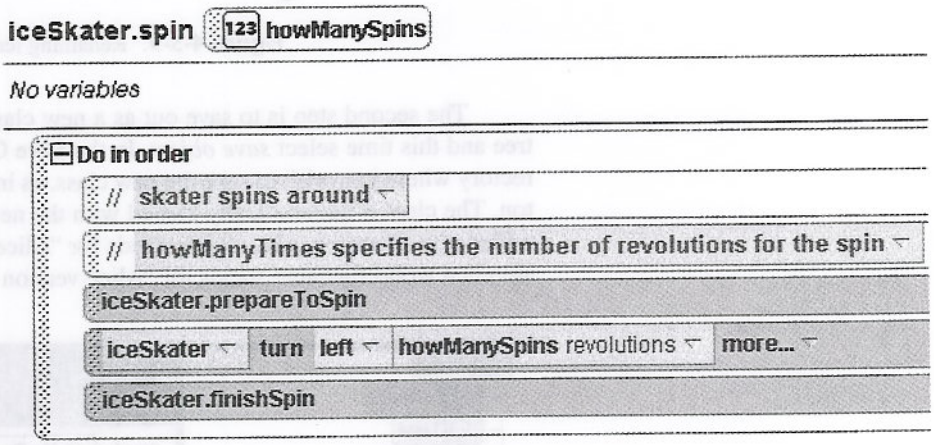


Figure 4-3-8. The *spin* method

The code for the two examples above (the *skate* and *spin* methods) is a bit longer than we have written in previous chapters. It is important that the code is easy to understand, because we have carefully broken down the overall task into smaller methods. The small methods all work together to complete the overall action. Also, the methods have been well documented, with comments that tell us what the method accomplishes. Good design and comments make our code easier to understand as well as easier to write and debug.

Creating a new class

The iceSkater now has two class-level methods, *skate* and *spin*. (She also has several smaller class-level methods that implement small pieces of the *skate* and *spin* methods.) Writing and

testing the methods took some time and effort to achieve. It would be a shame to put all this work into one world and not be able to use it again in another animation program we might create later. We would like to save the iceSkater and her newly defined methods so we can use them in another world (we won't need to write these methods again for another animation program). To do this, the *iceSkater* must be saved out as a new 3D model (class).

Saving the iceSkater (with her newly defined methods) as a new class is a two-step process. The first step is to rename the iceSkater. This is an IMPORTANT STEP! We want Alice to save this new class with a different 3D filename than the original IceSkater class. To rename an object, right-click on the name of the object in the Object tree, select *rename* from the popup menu, and enter the new name in the box. In this example, we right-clicked on iceSkater in the Object tree and changed the name to cleverSkater, as shown in Figure 4-3-9.

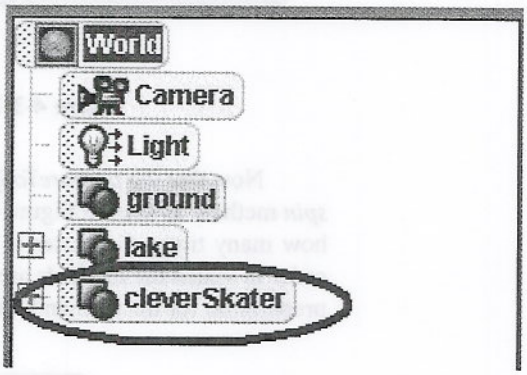


Figure 4-3-9. Renaming iceSkater as cleverSkater

The second step is to save out as a new class: right click on cleverSkater in the Object tree and this time select *save object.* In the Save Object popup box, navigate to the folder/directory where you wish to save the new class, as in Figure 4-3-10, and then click the Save button. The class is automatically named with the new name, beginning with a capital letter and a filename extension .a2c, which stands for "Alice version 2.0 Class" (just as the .a2w extension in a world filename stands for "Alice version 2.0 World").

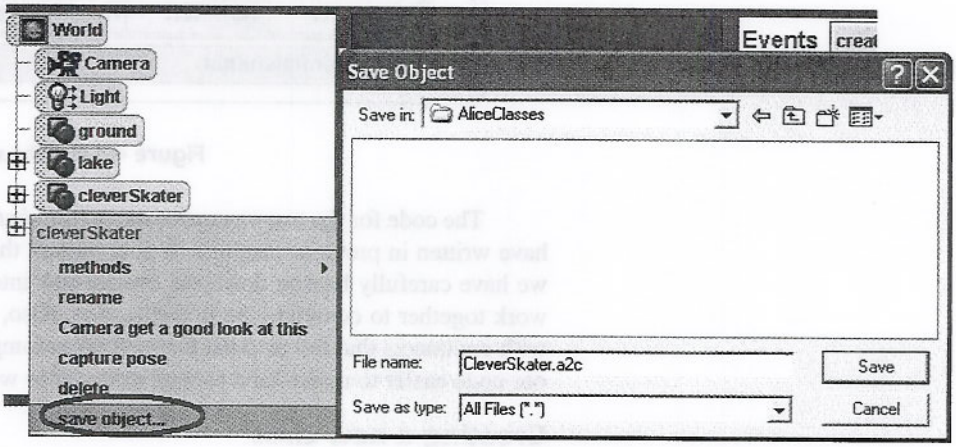


Figure 4-3-10. Save Object dialog box

Once a new class has been created, it can be used in a new world by selecting **Import** from the **File** menu, as illustrated in Figure 4-3-11. When an instance of the CleverSkater

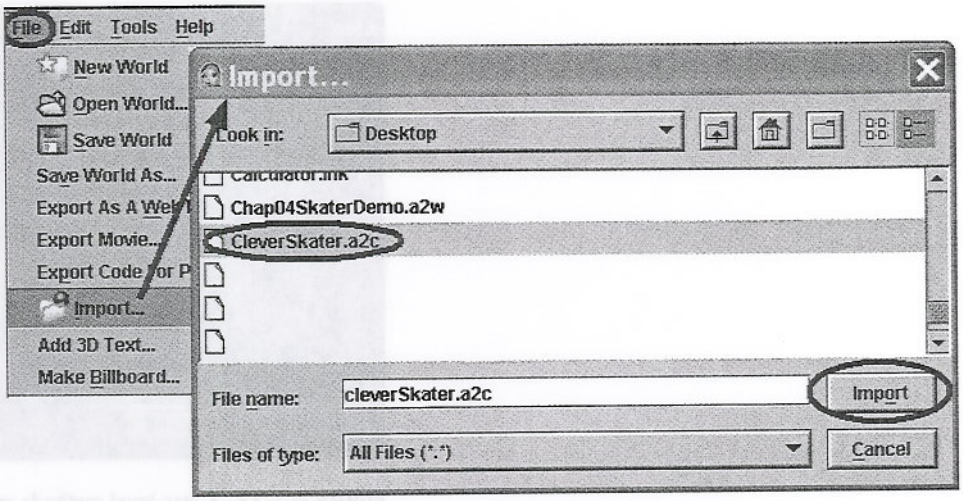


Figure 4-3-11. Importing a new object from a saved-out class

class is added to a world, she will be just like an instance of the IceSkater class, except that a cleverSkater object knows how to *skate* and *spin* in addition to all of the methods an iceSkater object can perform.

Inheritance—benefits

Creating a new class based on a previously defined class is called inheritance. Inheritance in most object-oriented languages is more complicated than in Alice. The basic idea is the same—adding functionality by defining new methods for a new kind of inherited class. Inheritance is considered one of the strengths of object-oriented programming because it allows you to write code once and reuse it in other programs.

Another benefit of creating new classes is the ability to share code with others in team projects. For example, if you are working on an Alice project as a team, each person can write class-level methods for an object in the world. Then, each team member can save out the new class. Objects of the new classes are added to a single team-constructed world for a shared project. This is a benefit we cannot overemphasize. In the "real world," computer professionals generally work on team projects. Cooperatively developed software is often the way professional animation teams at animation studios work.

Guidelines for Writing Class-Level Methods

Class-level methods are a powerful feature of Alice. Of course, with power there is also some danger. To avoid potential misuse of class-level methods, we offer some guidelines.

1. Do create many different class-level methods. They are extremely useful and helpful. Some classes in Alice already have a few class-level methods defined. For example, the Lion class has methods *startStance, walkForward, completeWalk, roar,* and *charge*. Figure 4-3-12 shows a thumbnail image for the Lion class (from the Web gallery), including its class-level methods and sounds.

2. Play a sound in a class-level method **ONLY IF** the sound has been imported for the object (instead of the world). If the sound has been imported for the object and the object is saved out as a new class, the sound is saved out with the object. Then the sound can be played anywhere in any world where an object of this class is added. On the other hand, if the sound is imported for the world, the sound is not saved out with the object and you cannot depend on the sound being available in other worlds.

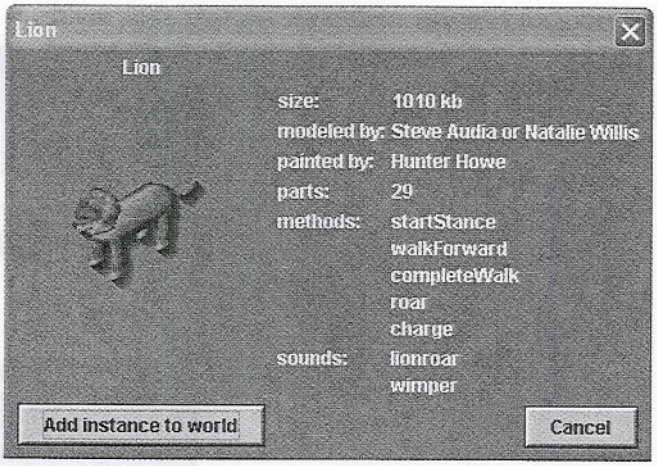


Figure 4-3-12. Class-level methods and sounds for the Lion class

3. Do not call world-level methods from within a class-level method. Figure 4-3-13 illustrates *cleverSkater.kaleidoscope*—a class-level method that calls a world-level method named *World.changeColors*. If the cleverSkater (with the *cleverSkater.kaleidoscope* method) is saved out as a new class and an instance of the CleverSkater class is then added to a later world where the *World.changeColors* method has not been defined, Alice will complain that the *World.changeColors* method cannot be found. Alice stops running your program and opens an Error dialog box with a description of the specific error in your program.

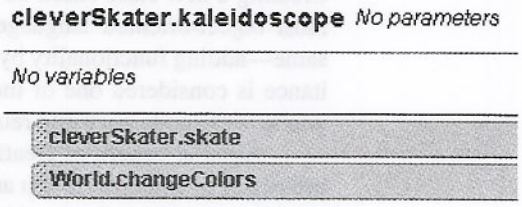


Figure 4-3-13. Bad example: calling a world-level method from a class-level method

4. Do not use instructions for other objects from within a class-level method. Class-level methods are clearly defined for a specific class. We expect to save out the object as a new class and reuse it in a later world. We cannot depend on other objects being present in other programs in other worlds. For example, a penguin (Animals) is added to the winter scene, as in Figure 4-3-14. We write a class-level method named *skateAround*, where the penguin object is

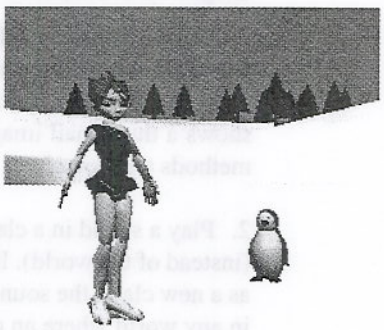


Figure 4-3-14. The skater will skate around the penguin

specifically named in two of the instructions (circled in Figure 4-3-15). If the cleverSkater with the *skateAround* method is saved out as a new class and then a cleverSkater object is added to a later world where no penguin exists, Alice will open an Error dialog box to tell you about a missing object. The error would be that the cleverSkater cannot skate around a penguin that does not exist in the world!

Note: Possible exceptions to guideline #4 are the world and camera objects, which are always present.

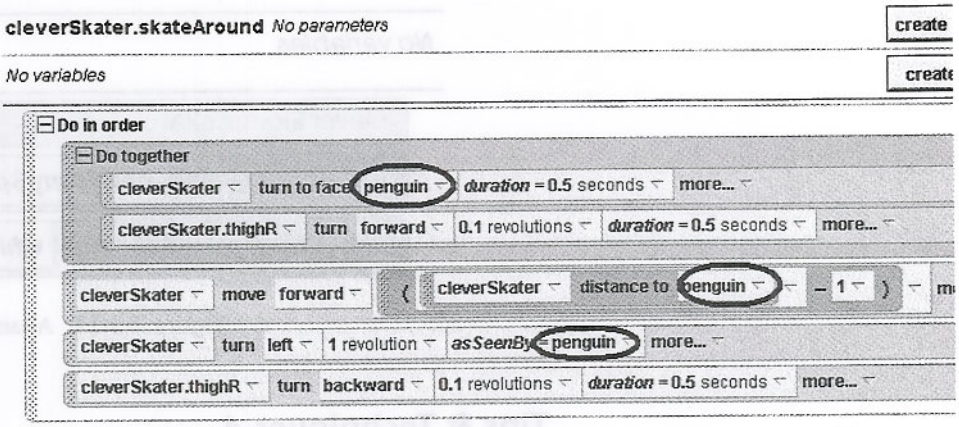


Figure 4-3-15. Bad example: instructions specifying another object in a class-level method

A class-level method with an object parameter

What if you would like to write a class-level method where another object is involved? The solution is to use an object parameter in the class-level method. Let's use the same example as above, where we want a cleverSkater to skate around another object. The *skateAround* method can be modified to use a parameter, arbitrarily named *whichObject*, as shown in Figure 4-3-16. The *whichObject* parameter is only a placeholder, not an actual object, so we do not have to worry about a particular object (like the penguin) having to be in another world. Alice will not

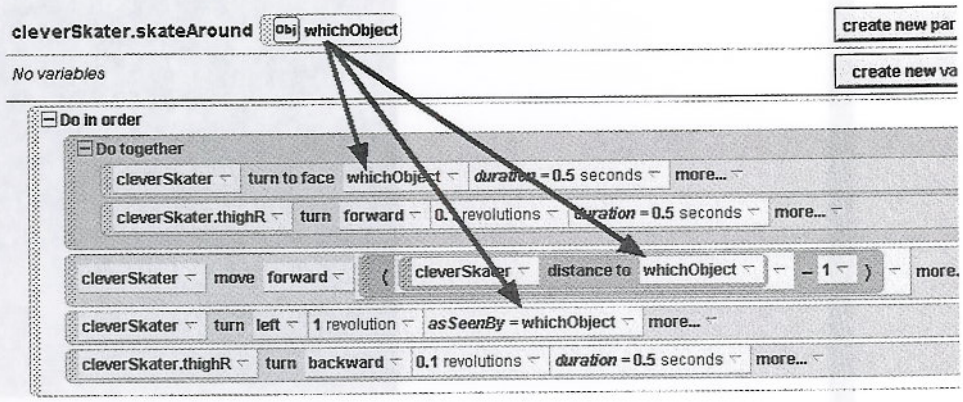


Figure 4-3-16. Using an object parameter in a class-level method

allow the *skateAround* method to be called without passing in an object to the *whichObject* parameter. So, we can be sure that some sort of object will be there to skate around.

Testing

Once you have created and saved out a new class, it should be tested in a new world. The initial scene was shown in Figure 4-3-14. A sample test program is presented in Figure 4-3-17. In this test, we have called the *skate, spin,* and *skateAround* methods to test each method.

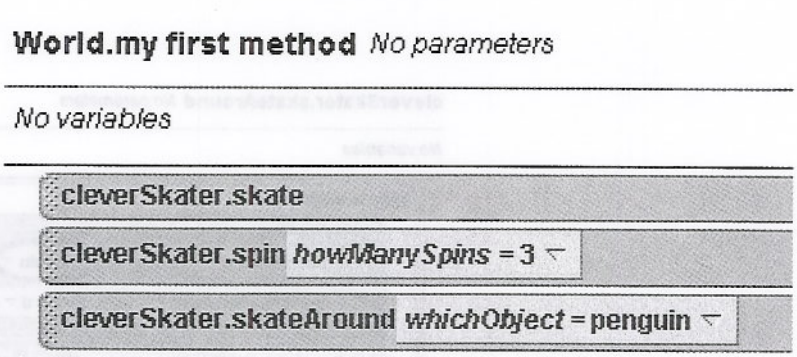


Figure 4-3-17. A sample program

Tips & Techniques 4
Visible and Invisible Objects

Properties of objects are sometimes used in games and simulations to achieve a special effect, such as making an object visible or invisible. In this section we look at techniques and examples of changing the visibility of objects.

The opacity property

The following example changes the opacity of a fish in an ocean world. (Opacity is how opaque something is: how hard it is to see through.) Figure T-4-1 shows an aquatic scene. This world is easily created by adding an oceanFloor (Ocean) and a lilfish (Ocean). (Optional items— seaweed and fireCoral were added from the OceanFloor folder in the CD or Web gallery.)

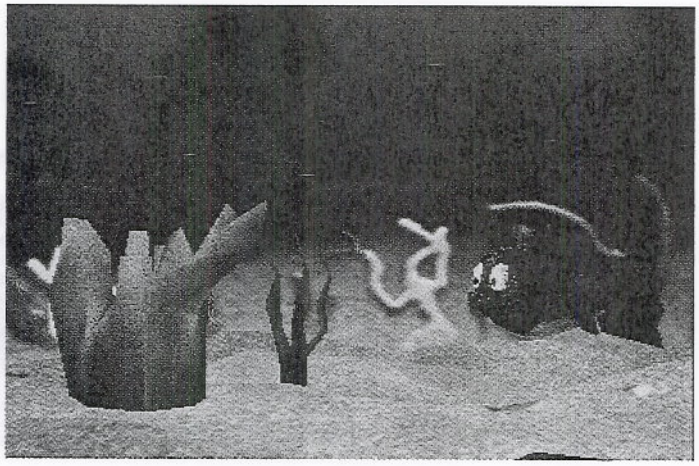


Figure T-4-1. An ocean floor scene with lilfish

The lilfish is swimming out to lunch, and her favorite seafood is seaweed. Instructions to point lilfish at the seaweed and then swim toward it are shown in Figure T-4-2. The *wiggletail* instruction is a method, shown in Figure T-4-2(b), that makes the fish wiggle its tail in a left-right motion.

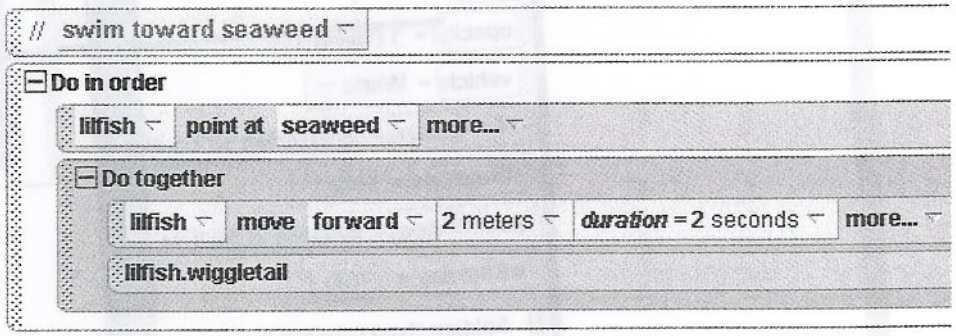


Figure T-4-2(a). Code to make lilfish swim toward the seaweed

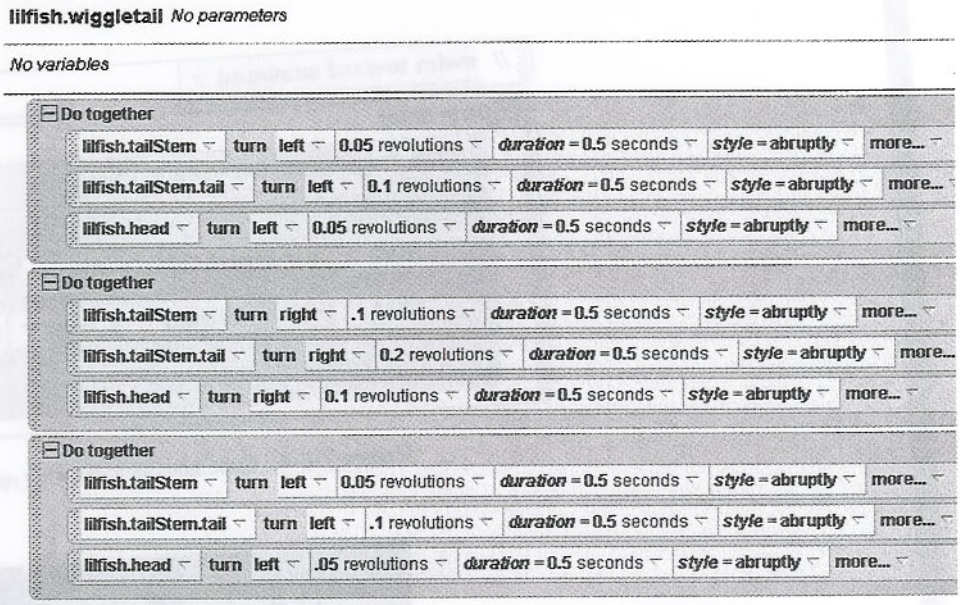


Figure T-4-2(b). The *wiggletail* method

As the fish moves toward the seaweed, she will also move away from the camera. So she should fade, because water blurs our vision of distant objects. We can make lilfish become less visible by changing the opacity property. As opacity is decreased, an object becomes less distinct (more difficult to see). To write an instruction to change the opacity, click on the lilfish's properties tab and drag the *opacity* tile into the editor. From the popup menu, select the *opacity* percentage, as shown in Figure T-4-3.

The resulting code is in Figure T-4-4.

When the world is run, lilfish will become less visible, as shown in Figure T-4-5. At 0% opacity, an object will totally disappear. This does not mean that the object has been deleted; it is still part of the world but is not visible on the screen.

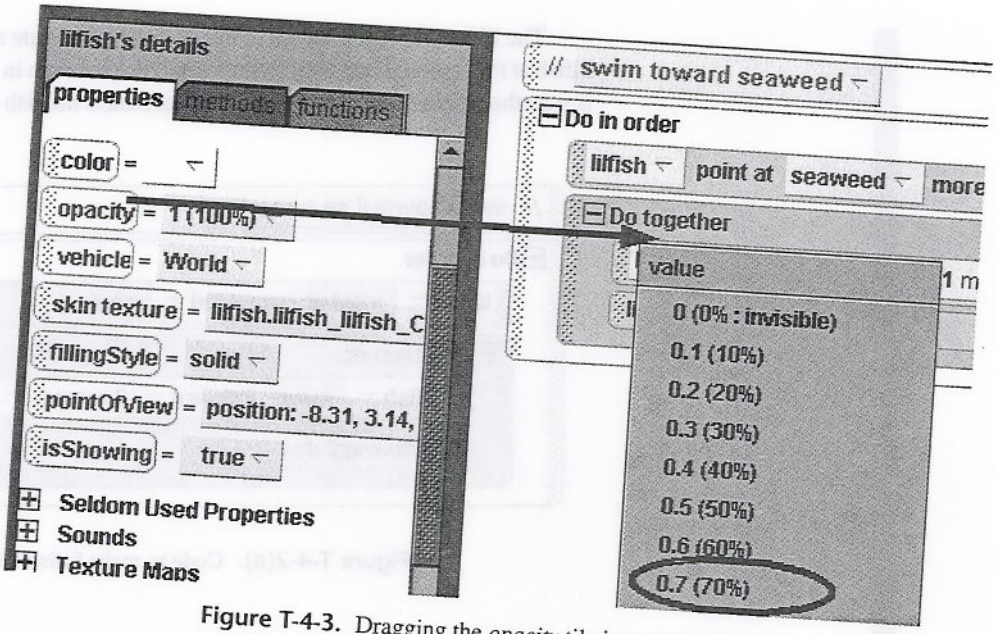


Figure T-4-3. Dragging the *opacity* tile into code editor

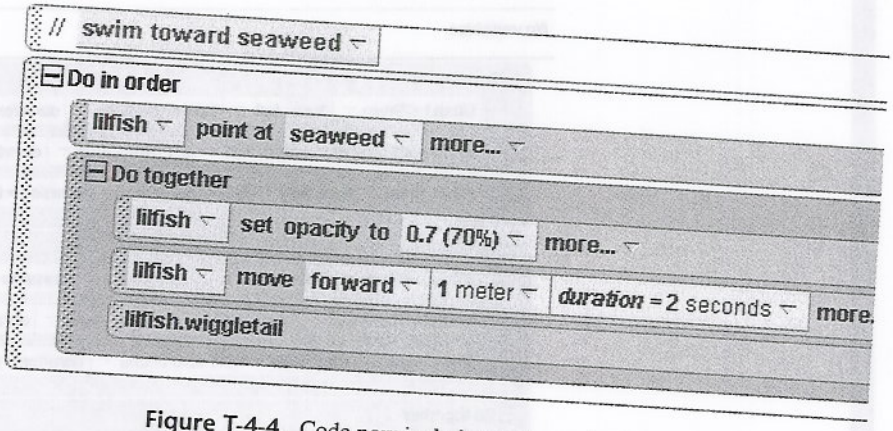


Figure T-4-4. Code now includes a *set opacity* instruction

Figure T-4-5. The lilfish becomes more difficult to see as *opacity* is decreased

The isShowing property

Each object has a property called *isShowing*. At the time an object is first added to a world, the object is made visible in the scene and *isShowing* is set to *true*. Changing the value of this property is especially useful in game like programs where you want to signal the end of a game. Figure T-4-6 illustrates the *isShowing* property as *true* for "You won!" Setting *isShowing* to *false* makes the "You won!" text invisible, as shown in Figure T-4-7. (For this world, we used the bottleThrow object from the Amusement Park folder in the CD or Web gallery.)

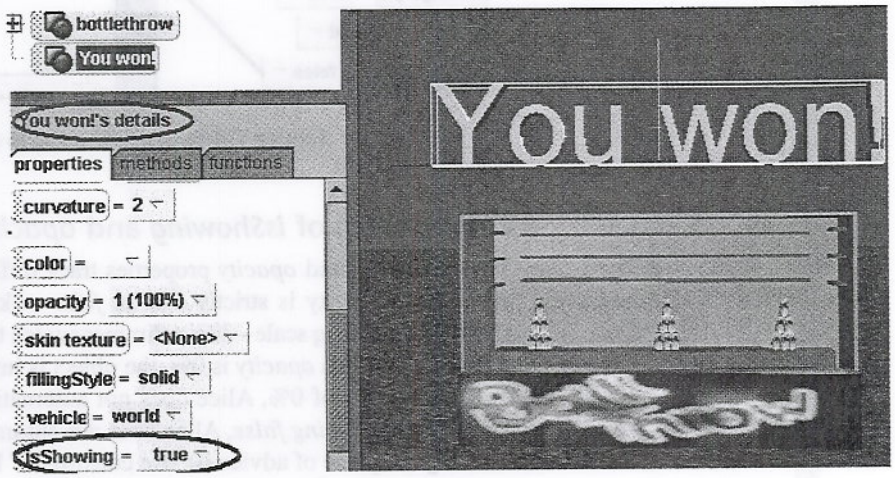


Figure T-4-6. The *isShowing* property is *true* and "You won!" is visible

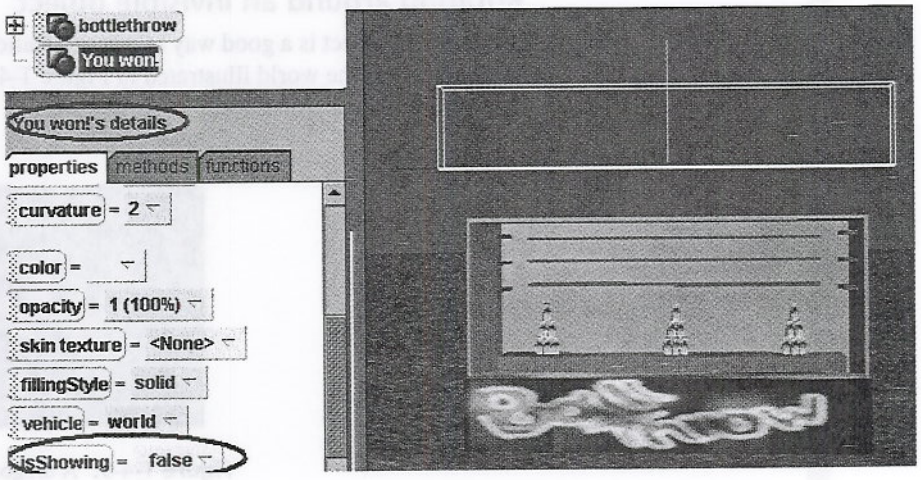


Figure T-4-7. The *isShowing* property is *false* and "You won!" is not visible

When its *isShowing* property is set to *false*, the object is not removed from the world; it is simply not displayed on the screen. The object can be made to "reappear" by setting its *isShowing* property back to *true*.

In this example, we want the text to appear when the player wins the game. To create an instruction that sets the *isShowing* property to *true*, drag the *isShowing* property tile into the world and select *true* from the popup menu. The result is shown in Figure T-4-8.

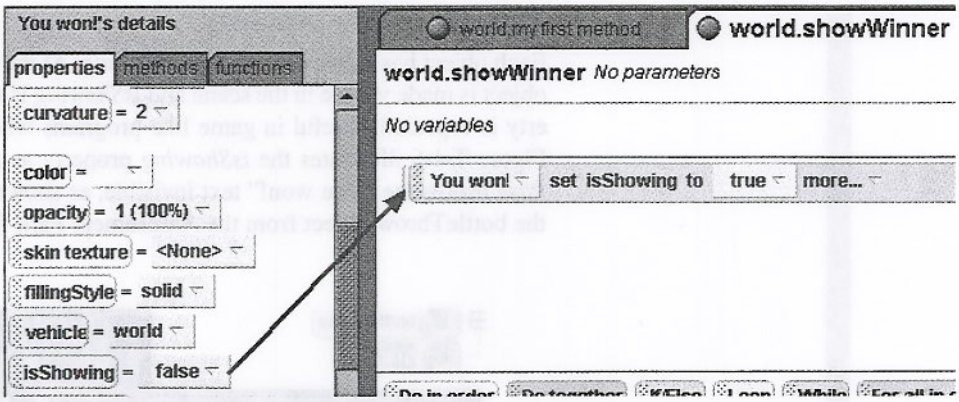


Figure T-4-8. An instruction to set *isShowing* to *true* at runtime

Relationship of *isShowing* and *opacity* properties

The *isShowing* and *opacity* properties track different (though related) states of an object. The *isShowing* property is strictly *true* or *false*—like a light switch that can be either on or off. *Opacity* is a sliding scale—like a dimmer switch that can adjust the brightness of a light. Though it is true that when *opacity* is 0%, the object is invisible, nonetheless when you make an object have an *opacity* of 0%, Alice does not automatically make *isShowing false*. Likewise, when you make *isShowing false*, Alice does not automatically make *opacity* 0%.

A good piece of advice is: "Be consistent." If you are using *isShowing* in your program to set the visibility, then do not use *opacity* to check whether the object is visible. Likewise, if you are using *opacity* to set the visibility, do not use *isShowing* to check whether the object is visible.

Rotating around an invisible object

An invisible object is a good way to set up a stationary reference point for actions by other objects. Consider the world illustrated in Figure T-4-9. We want the pterodactyl (Animals) to fly around the dragon (Medieval).

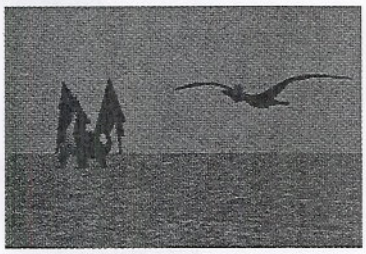
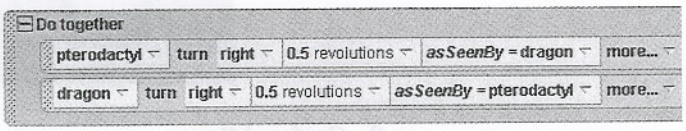


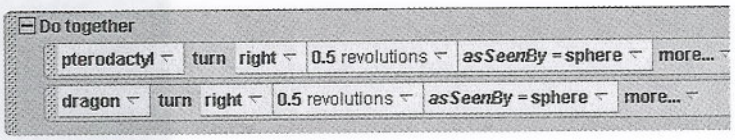
Figure T-4-9. A dragon and a pterodactyl

This is no problem. The pterodactyl will fly around the dragon if we use *asSeenBy = dragon* in a *turn right* instruction for the pterodactyl object. (The *asSeenBy* parameter was described in Tips & Techniques 2.)

Suppose that we want the pterodactyl and the dragon to both fly around in a half-turn relative to each other (facing each other down—sort of a bluffing technique). This would mean that the dragon should end up at the pterodactyl's location (facing the pterodactyl's new location), and the pterodactyl should end up at the dragon's location (facing the dragon's new location). A first attempt might be as follows:



When this program is run, each animal ends up where it started, facing in the opposite direction! The problem is that once each animal has begun to move, its location changes, so that further moves relative to each other lead to unexpected results! What we need is an object that does not move, located somewhere between the dragon and the pterodactyl. Let's add a sphere object between the dragon and the pterodactyl and make it invisible by changing its *isShowing* property to *false*. Now we can write:



When this code is run, the dragon and pterodactyl exchange places, as seen in Figure T-4-10!

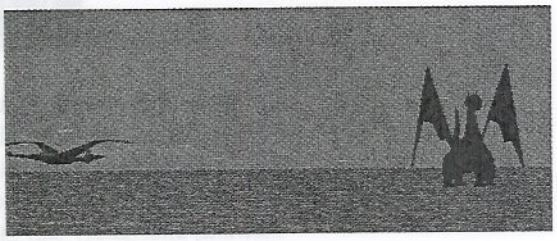


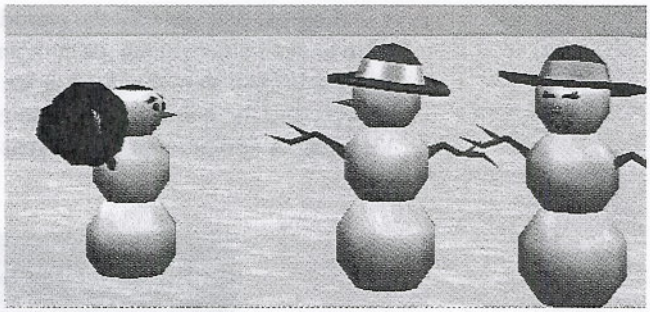
Figure T-4-10. The pterodactyl and dragon change places

Exercises and Projects

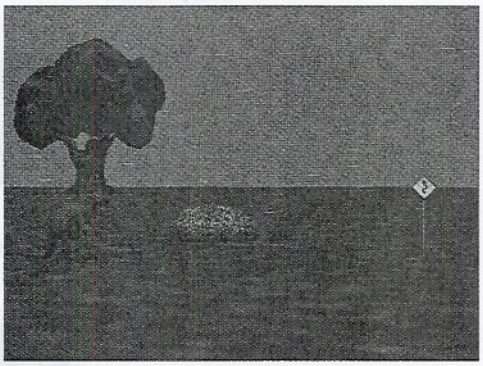
4-1 Exercises

Reminder: Be sure to add comments to your methods to document what each method does and what actions are carried out by sections of code within the method.

1. *Snowpeople Flip Hats*
   Write a new world-level method for the Snowpeople world (from Chapter 2, Exercise 3). The new method, named *flipHats*, will be called when the snowwoman turns her head to look at the snowman. In the *flipHats* method, the snowman uses his right arm to grab his hat and graciously tip his hat to the snowwoman and then returns his hat to his head. After the snowman flips his hat, the snowwoman flips her hat.

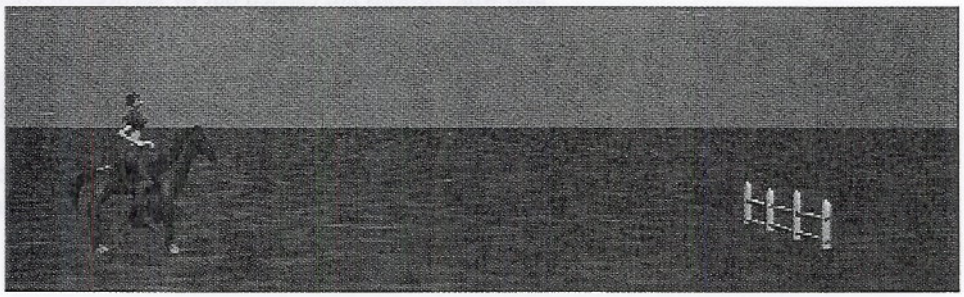


2. *Confused Kanga*
   Scrounging for breakfast on the outback, Kanga (kangaroo from Animals) encounters a rather confusing sign (Roads and Signs folder). Kanga stares at the sign for a few seconds and tilts her head sideways to show that she is confused. Kanga then hops left and turns toward the sign and then hops right and turns toward the sign and then left and then right.



   Create a simulation that implements this comical story. Write methods *hopLeft* (Kanga turns left a small amount and hops, and then turns to face the sign) and *hopRight* (Kanga turns right a small amount and hops, then turns to face the sign). With each hop, Kanga should make some progress toward the sign. In *World.my first method*, alternately call the *hopLeft* and *hopRight* methods (twice) to make Kanga take a zigzag path toward the sign.

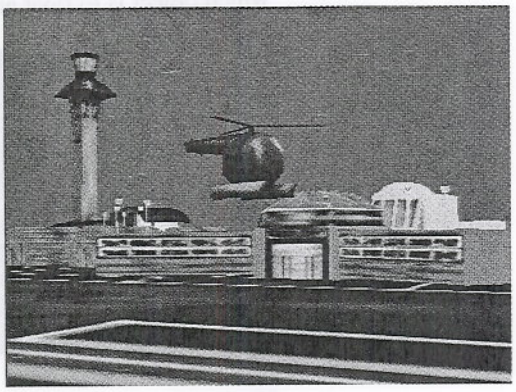
3. *Gallop and Jump*
   Kelly (People) has entered an equestrian show as an amateur jumper. She is somewhat nervous about the competition so she and the horse (Animals) are practicing a jump. Create an initial scene with a horse and rider facing a fence (Buildings), as shown below.



Write two world-level methods, one named *gallop* (horse and rider gallop forward one step) and another named *jump* (horse and rider jump the fence). In the gallop method, the horse's front legs should lift and then go down as the back legs lift and the horse moves forward. Then the back legs should go back down. The jump method should be similar, but the horse should move up far enough to clear the fence in mid-stride. Test each method to be sure it works as expected. You will need to adjust the distance amounts to make each look somewhat realistic.

Hint: If you make the horse a *vehicle* for Kelly (Tips & Techniques 2), you will only need to write an instruction to move the horse and Kelly will go along for the ride.
When you think the gallop and jump methods are both working properly, write instructions in *World.my first method* that call the *gallop* method as many times as needed to move the horse and rider up close to the fence; then call the *jump* method. Use trial-and-error to find out how many times the *gallop* method must be called to make the animation work well.

4. *Helicopter Flight*
Create a world with a helicopter (Vehicles on CD or Web gallery), airport (Buildings), and a control tower (Buildings). Create a *circleTower* method that makes the helicopter fly toward the control tower and then around it. In *my first method*, call the *circleTower* method twice and then make the helicopter land on the airport landing strip.



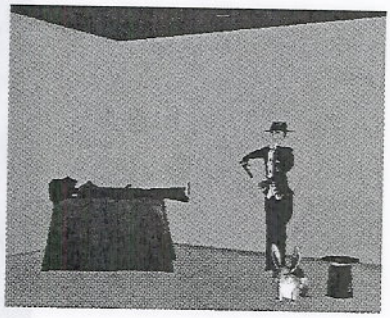
4-2 Exercises

5. *Beetle Band Duet*
In Section 4-2, the Beetle Band example has a method named *solo*, where each member of the band jumps and plays a musical instrument. Recreate the Beetle Band example and write a method named *duet*, where the solo method is called to have two members of the band jump together and play their musical instruments. This exercise can be done with sound (as in Figure 4-2-15) or with lyrics (as in Figure 4-2-17). Parameters must be used to send in the name of the band member and the music to be played or the lyric to be said.

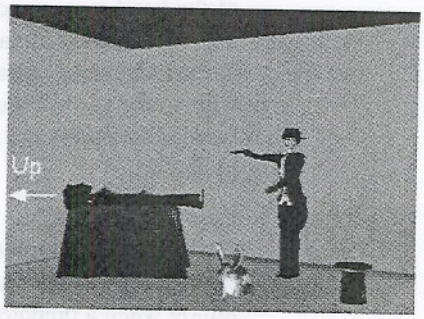
6. *Frog Escape*
At the local lily pond (Environments on CD or Web gallery), the frogs (Animals) enjoy climbing out of the water now and then to warm up in the sun. Of course, they get a bit jumpy when a predator is sighted. On this fine day, a hungry snake (Animals) wanders into the scene. Create a world scene similar to the one below and animate the frogs jumping into the pond when the snake approaches. Write a method that turns the snake toward the frog and slides the snake forward. Then, have the frog turn to the pond and jump in. Your method should use a parameter to specify which frog is escaping.

7. *Magic Act*

A magician is performing a levitation illusion, in which objects seem to rise magically into the air. The magician (People) points a magic wand (Objects) at his assistant (Girl from People folder), and she gently rises into the air and then floats back down to her original position on the table (Furniture on CD or Web gallery). Then the magician performs the same trick with the rabbit (Animals). The rabbit, being a lighter object, floats up higher than the magician's assistant. Because the magician's assistant and rabbit are each to levitate in the same way, use a single method and use parameters to communicate which object is to float and the distance the object is to move upward (and back down).



Hint: The magician's assistant is reclining on the surface of the table. A *move up* instruction will cause the assistant to move upward from her point of view, as shown below. Use the *asSeenBy* argument to make the magician's assistant move upward as seen by the ground.



8. *Dragons*

Legend has it that dragons are distant relatives of chickens. We are not surprised, then, that a favorite pastime of dragons was a game of "chicken." The scene below shows a world with four dragons (Medieval) carefully placed in a diamond like pattern (similar

to baseball players at the four bases). Create a simulation of a game of chicken where any two dragons face each other and fly upward to a slightly different height above the ground. Then the dragons fly toward each other, nearly missing one another. Each dragon should land in the position where the other dragon was located. That is, the two dragons trade places. Your simulation should use a method named *dragonFlight* that has four parameters—the two dragons that will face off in a game of chicken and the height for each dragon's flight.

4-3 Exercises

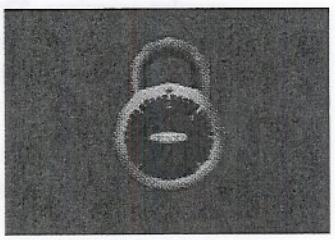
9. *Enhanced cleverSkater*
Create an even better cleverSkater than the one presented in Section 4-3. In addition to the *skateForward, spin* and *skateAround* methods, create *skateBackward* and *jump* class-level methods. In *skateBackward*, the skater should perform similar actions to those in the *skateForward* method, but slide backward instead of forward. In a *jump* method, the skater should move forward, lift one leg, then move upward (in the air) and back down to land gracefully on the ice and lower her leg back to its starting position. Save out your enhanced skater as EnhancedCleverSkater.
Test your newly defined class by starting a new world with a frozen lake. Add an enhancedCleverSkater to the world. Also, add a penguin and a duck.

   (a) Call each of the methods you have written.

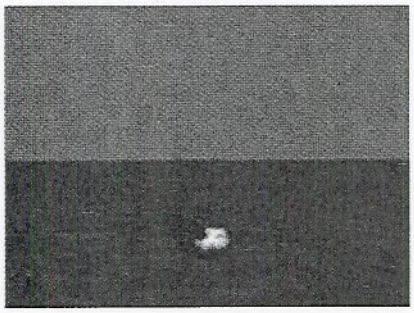
   (b) Then call the *skateAround* method—to make the skater skate around the penguin and then the duck. (This will require two calls to the *skateAround* method.)

10. *Lock Combination*
Create a world with a comboLock (Objects folder). Create four class-level methods—*leftOne, rightOne, leftRevolution,* and *rightRevolution*— that turn the dial 1 number left, 1 number right, 1 revolution left, and 1 revolution right, respectively. Then, create a class-level method named *open* that opens and another named *close* that closes the lock.

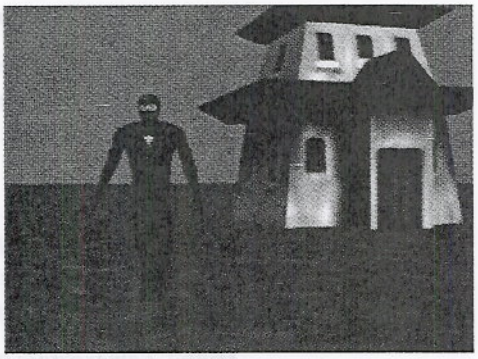
Hints: One position on the dial is actually 1/40 of a revolution. Use the *endGently* style to make the motion more realistic.) Rename comboLock as TurningComboLock and save it as a new class.



11. *Funky Chicken Dance*
Starting with a basic chicken, create a class-level method *walk* that will have the chicken perform a realistic stepping motion consisting of a left step followed by a right step. Create a second method to make the chicken perform a *funkyChicken* dance, where the chicken walks and twists its body in all different directions! Save the chicken as a new class named CoolChicken. Create a new world and add a coolChicken to the world. In *my first method*, call the *walk* and *funkyChicken* methods. Play a sound file or use a *say* instruction to accompany the funky chicken dance animation.



12. *Ninja Practice*
Create a world with an evilNinja (People) and write class methods for traditional Ninja moves. For example, you can write *rightJab* and *leftJab* (where the Ninja jabs his hand upward with the appropriate hand), *kickLeft* and *kickRight* (where he kicks with the appropriate leg), and *leftSpin* and *rightSpin* (where he does a spin in the appropriate direction). Each method must contain more than one instruction. For example, in the *kickLeft* method, the left lower leg should turn and the foot twist at the same time as the entire leg kicks out to the left. Save the Ninja as a new class named TrainedNinja. Start a new world and add two trainedNinja objects. Create an animation where the two trainedNinja objects practice their moves, facing one another.



Projects

We are using the term project to describe advanced exercises that are more challenging than regular exercises. The projects in this chapter involve motion of human body parts. Professional animators spend many hours mastering the art of making these movements look realistic. Our focus is on mastering the art (and science) of writing methods in a program.

1. *Dance*
Technical Note: To assist you in learning how to animate human body parts, this first project includes some explanations and coding suggestions. The goal of this animation

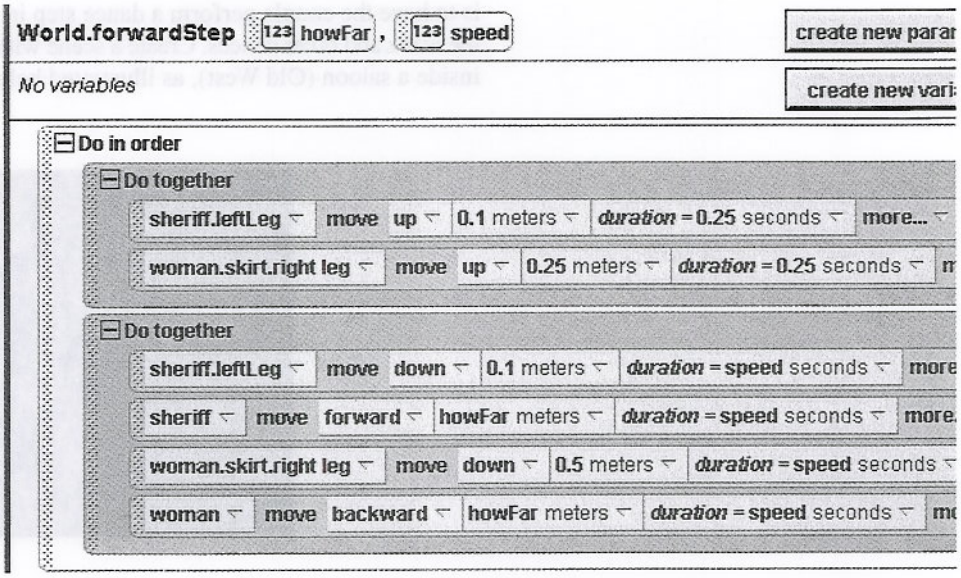
is to have the couple perform a dance step in a traditional box (square) figure as used in the waltz and other dances. Create a scene with a sheriff (Old West) and a woman (People) inside a saloon (Old West), as illustrated below.

In the first step of a box figure, the sheriff takes a step forward, leading with his left leg and (at the same time) the woman takes a step backwards, leading with her right leg. This is not as simple as it sounds. One way to make an object with legs appear to take a "step" is to have the object raise one leg some small amount and then move forward as the leg moves back down. Then, the other leg performs a similar action. Thus, to make two objects appear to dance together requires coordinated leg lift, move and drop actions for both objects. The easiest way to do this is to write a method, perhaps named *forwardStep*. The *forwardStep* method will need two parameters: *howFar* (the distance forward), and *howLongToTake* (the time it takes). A possible storyboard is:

```
forwardStep

Parameters: howFar, howLongToTake
Do in order
        Do together
            sheriff's left leg moves up
            woman's right leg moves up
        Do together
            sheriff moves forward howFar
            sheriff's left leg moves down
            woman moves backward howFar
            woman's right leg moves down
```

To help you get started, an example of the code for the *forwardStep* method is shown below. The distances used in this code worked well for us in our example. You may need to experiment with the amount to move the legs up (and down), and with the size of step forward, backward, and sideways the couple is to take. The distances depend on the size of the models in your world.



Other methods you will need are:

(a) *rightStep,* where the man and woman take a step sideways (his right, her left)

(b) *backStep,* where the man takes a step backward, leading with his left leg, and the woman takes a step forward, leading with her right leg

(c) *leftStep,* where the couple takes a step sideways (his left, her right)

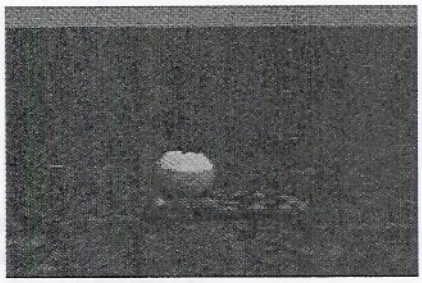
(d) *spin,* which has the man spin the woman around

If *forwardStep, rightStep, backStep*, and *leftStep* are properly performed in sequence, the couple should move in a squarelike pattern on the dance floor. Create a method to call all the methods in order so the couple performs a box figure followed by a spin for a dance. Then, create a second method to animate a different dance figure—calling the dance steps in a different order.

2. *Hand Ball*

Create a world with a right hand (People) holding a toy ball (Sports). Have the fingers close to grasp the ball. Then, throw the ball into the air while opening the fingers of the hand. Finally, make the hand catch the ball as the hand re-closes its fingers.

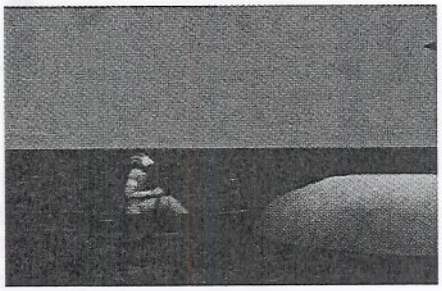
Hint: Tips & Techniques 2 tells how to use the *vehicle* property to make the ball move as the hand moves.



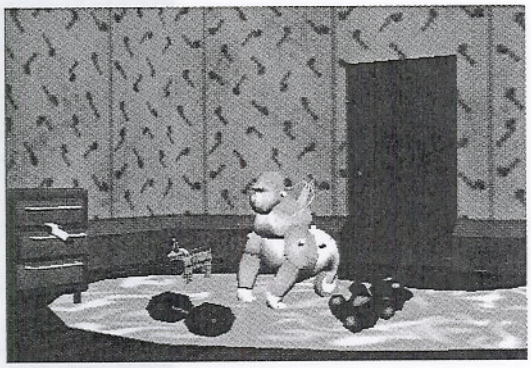
3. *Ra Row Your Boat*

Create a world with a boat, a person sitting in the boat, an island, and a pier located 25 meters from the island. In the world shown below, Ra (Egypt) is sitting in a rowboat

(Vehicles). Create a method to make the Ra object row the boat 25 meters from the island to a pier (Beach). One suggested way to do this would be to create the methods: *rowLeft* and *rowRight* (to control the arms' motions), *controlTorsoAndHead* (to control back and head motions), and *startRow* and *stopRow* (to put Ra's body in and out of the rowing position).

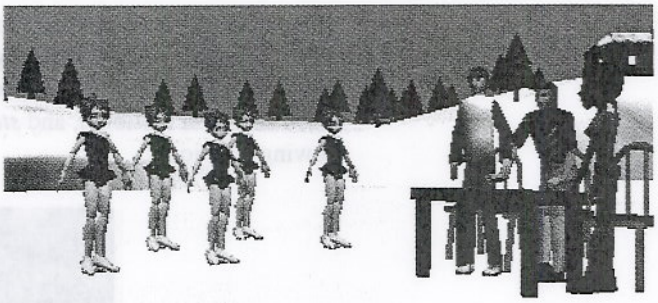


4. *Cleanup Robot*
The scene below shows a child's room with toys scattered around on the floor. The gorilla robot (SciFi) can be programmed pick up things in the room and put them behind the door in the closetRoom (Environments on CD or Web gallery). In the initial scene for this animation, the robot is standing in the middle of the room near several objects scattered around the floor (for example, a barbell, a pinata, and a teddy bear from the Objects folder).



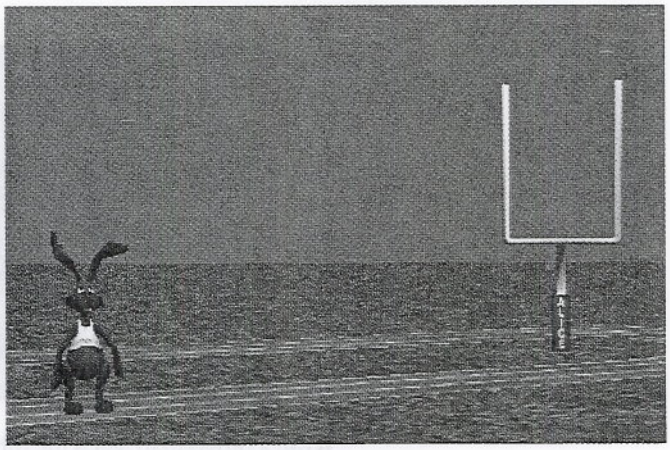
Write a program to teach the robot to pick up one object at a time and put it in the closet. Write three methods, named *pickup*, *putdown*, and *putInCloset*. The methods should have one parameter identifying the object to be picked up or put down. The *pickUp* method should make the robot pick up an object in its hand. The *putDown* method should have the robot put the object down. The *putInCloset* method should make the robot turn one-half revolution and move to the door. The door opens and the robot puts down the object. When these methods have been written, then write instructions in *my First Method* to make the robot pick up an object and put it down in the closet.

5. *Skater Competition*
Add five enhancedCleverSkater objects to a world with a lake (Environments) scene. See Exercise 9 for a description of the EnhancedCleverSkater. Also, add three people to act as judges of a skating competition. Write a method for each skater that has her perform a skating routine (each skater should perform some combination of *skate, spin, jump* and other methods). Each skater (one at a time) will perform her skating routine; then have the three judges say a score. Scores range from 1 to 9. Then the skater that has just performed will move out of the way and the next skater will perform.



6. *Walking and Jogging Hare*
   Add an instance of the Hare (Animals) to a new world. Write a class-level method to make the hare *walk* forward and a second method to make the hare *jog*. (A jog is similar to a walk, but the hare moves faster and the entire body moves up and down with each step.) Save the enhanced hare as a new class named AthleticHare.

   Now, start a new world with a grassy scene. Add an instance of the AthleticHare to the world. Add a goalpost (Sports) and highway objects to the scene to create a track around the goalpost. Write a world-level method that calls the *walk* method three times (to get the hare started down the track); then call the *jog* method and turn the hare to have him jog around the goalpost.



7. *Your Own Creation (Open-ended)*
   Choose an animal or a person from one of the galleries. The object selected must have at least two legs, arms, and/or wings that can move, turn, and roll. Write three class-level methods that substantially add functionality to what the objects of the class know how to do. Use Save Object to create a new class with a different name. Add an instance of your new class to a new world. Then write an animation program to demonstrate the new methods.

Summary

In this chapter we looked at how to write our own methods and how to use parameters to send information to a method when it is called. An advantage of using methods is that the programmer can think about a collection of instructions as all one action—abstraction. Also, methods make it easier to debug our code.

Two different kinds of methods can be written: world-level methods that involve two or more objects interacting in some way, and class-level methods that define a complex action carried out by a single object acting alone.

Parameters are used to communicate values from one method to another. In a method, a parameter acts as a placeholder for a value of a particular type. The values sent in to a method are known as arguments. When an argument is sent in to a method, the parameter represents that argument in the instructions in the method. Examples presented in this chapter included object, sound, string, and number parameters. Parameters allow you to write one method but use it several times with different objects, sounds, numbers, and other types of values.

In a way, class-level methods can be thought of as extending an object's behavior. Once new class-level methods are defined, a new class can be saved out. The new class has a different name and has all the new methods (and also the old methods) as available actions. It inherits the properties and actions of the original class but defines more things than the original class. A major benefit is that you can use objects of the new class over and over again in new worlds. This allows you to take advantage of the methods you have written without having to write them again.

Some guidelines were provided for writing class-level methods: only sounds imported for the class should be played in a class-level method; world-level methods should not be called; and instructions involving other objects should not be used. Following these guidelines will ensure that objects of your newly defined classes can safely be used in other worlds.

Stepwise refinement is a design technique where a complex task is broken down into small pieces and then each piece is broken down further—until the entire task is completely defined by simple actions. The simple actions all work together to carry out the complex task.

Important concepts in this chapter

- To run (or execute) a method, the method must be called.
- Parameters are used for communication with a method.
- In a call to a method, a value sent in to a method parameter is an argument.
- A parameter must be declared to represent a value of a particular type. Types of values for parameters include object, Boolean ("true" or "false"), number, sound, color, string, and others.
- A new class can be created by defining class-level methods and then saving out the class with a new name.
- Inheritance is an object-oriented concept where a new class is defined based on an existing class.
- Class-level methods can be written that accept object parameters. This allows you to write a class-level method and pass in another object. Then, the object performing the class-level method can interact with the parameterized object.

Chapter 5
Interaction: Events and Event Handling

Alice laughed, "There's no use trying," she said, "one can't believe impossible things." "I daresay you haven't had much practice," said the Queen. "When I was your age, I always did it for half-an-hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."

The real world around us is interactive. A conversation, as between Alice and the Queen above, is a "give and take" form of interaction. As we interact with objects in our world, we often give directions to them. For example, we change the channel on a television set by sending a signal from a remote control. We press a button on a game controller to make a character in a video game jump out of the way of danger.

We have concentrated on writing programs that were not interactive—we watched the objects perform actions in a movie-style animation. It's time we looked at how to create interactive programs in Alice—where the objects in the scenes respond to mouse clicks and key presses. In this chapter we will see how programs can be made interactive.

Much of computer programming (and the movie-style animations seen earlier) is computer-centric. That is, the computer program basically runs as the programmer has intended it. The programmer sets the order of actions and controls the program flow. However, many computer programs today are user-centric. In other words, it is the computer user (rather than the programmer) who determines the order of actions. The user clicks the mouse or presses a key on the keyboard to send a signal to Alice about what to do next. The mouse click or key press is an event. An event is something that happens. In response to an event, an action (or a sequence of many actions) is carried out. We say the event triggers a response.

Section 5-1 focuses on the mechanics of how the user creates an event and how the program responds to the event. Naturally, all of this takes a bit of planning and arrangement. We need to tell Alice to listen for a particular kind of event and then what to do when the event happens. This means we need to write methods that describe the actions objects should take in response to an event.

Section 5-2 describes how to pass parameters to event handling methods. In some programming languages, arranging events and writing event handling methods is a rather complex kind of programming. We hope you will find that learning to use events and event handling methods is easy in Alice.

5-1 Interactive programming

Control of flow

Writing an interactive program has one major difference from writing a non-interactive one (like the movies we wrote in the previous chapter). The difference is in how the sequence of actions is controlled. In a non-interactive program, the sequence of actions is pre-determined

by the programmer. The programmer designs a complete storyboard and then writes the program code for the animated actions. Once the program is constructed and tested, then every time the program runs, the same sequence of actions will occur. In an interactive program the sequence of actions is determined at runtime, when:

- The user clicks the mouse or presses a key on the keyboard.
- Objects in the scene move (randomly or guided by the user) to create some condition, such as a collision.

Events

Each time the user clicks the mouse or presses a key, an event is generated that triggers a response. Objects in the scene may move to positions that trigger a response. Each time the program runs, different user interactions or different object actions may occur and the overall animation sequence may be different from some previous execution of the program. For example, in a video game that simulates a car race, where the player is "driving" a race car, the sequence of scenes is determined by whether the player is skillful in steering the car to stay on the road through twists, turns, and hazards that suddenly appear in the scene.

Event handling methods

How do events affect what you do as an animation programmer? You must think about all possible events and make plans for what should happen—responses to the events. Animation methods are then written to carry out responses. Finally, the event must be linked to the responding method. The method is now said to be an event handling method.

When an event occurs and an event handling method is called, the location of objects in the scene may or may not be the same as the last time. This is because the user's actions may change the scene and the location of objects between calls to the event handling method.

Keyboard-control example

We begin with an acrobatic air-show flight simulator. The initial scene, as illustrated in Figure 5-1-1, consists of the biplane (Vehicles) in midair and some objects on the ground (house, barn, and so forth from the Buildings and Farm folders). A guidance system will allow

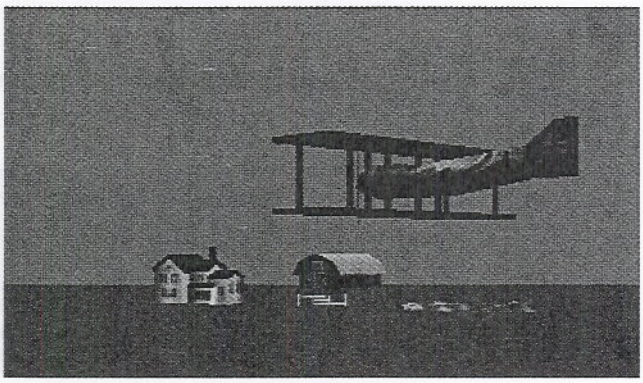


Figure 5-1-1. Initial scene

the user to be the pilot. The biplane has controls that allow the pilot to maneuver the plane forward, left, and right. We want to program the biplane to perform a popular show stunt—a barrel turn. In the exercises at the end of this Chapter, other stunts can be added.

Input

The whole idea of a flight simulator is to allow the user to interact with the biplane. The user provides input that sends a signal to animate a particular motion, perhaps by pressing a set of keys on the keyboard. For example, arrow keys can be used, each corresponding to a given direction of movement. Of course, input can also be obtained from mouse clicks, the movement of a trackball, or the use of a game stick controller. In this text, we will rely on the keyboard and mouse to provide user input for interaction with the animations.

In our flight simulator, the arrow keys and spacebar will be used to provide input from the user. If the user presses the up arrow key, the biplane will move forward. If the user presses the left or right arrow keys, the biplane will turn left or right. For the acrobatic barrel turn, we will use the spacebar. The selection of these keys is arbitrary—other keys could easily be used.

Design—storyboards

We are ready to design the flight simulator program—the set of instructions that tell Alice how to perform the animations. Each time the user presses an arrow key or the spacebar, an event is generated. The animation program consists of methods to respond to these events. To simplify the discussion, let's concentrate on two possible events: the spacebar press for the barrel turn and the up arrow key to move the biplane forward. Two storyboards are needed, as shown below. Note that sound is optional and can be omitted.

```
Event: Spacebar press

Response:
  Do together
    roll biplane a full revolution
    play biplane engine sound
```

```
Event: Up arrow key press

Response:
  Do together
    move biplane forward
    play biplane engine sound
```

Methods to respond to the events

The only object affected by key-press events is the biplane, so the methods can be class-level methods for the biplane. Two methods will be written, *flyForward* and *barrel*. The *flyForward* method will handle an up arrow key-press event by moving the biplane forward as illustrated in Figure 5-1-2. The barrel method will handle a spacebar-press event by rolling the biplane

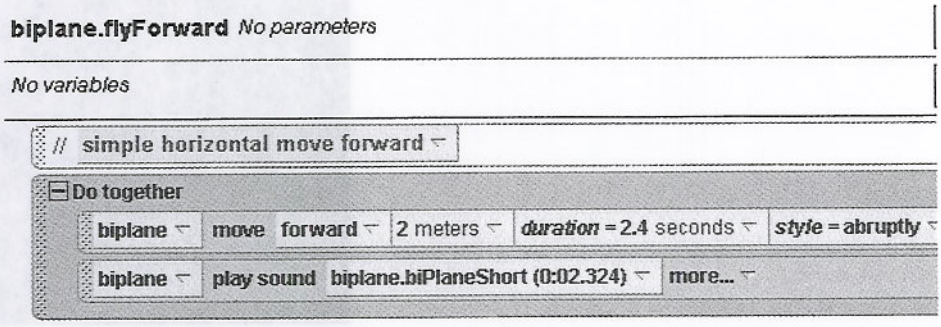


Figure 5-1-2. The *flyForward* method

one complete revolution, illustrated in Figure 5-1-3. In the methods shown here, a sound is played simultaneously with the movement. The duration of the biplane movement is set to be approximately the same as the length of the sound (in seconds). As noted previously, sound is a nice feature but can be omitted. If sound is used, the sound should be imported for the biplane. (Importing a sound file was introduced in Chapter 4, Section 2.)

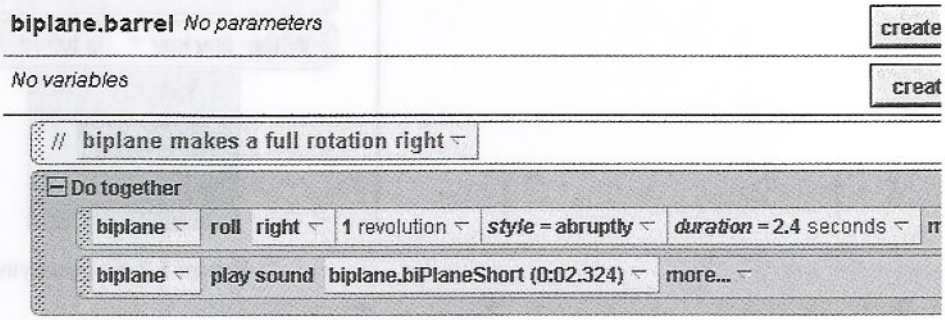


Figure 5-1-3. The *barrel* method

Link events to methods

Each method must be linked to the event that will be used to trigger the method as a response. The Events editor is where links are created. The Events editor is shown in Figure 5-1-4. As you know, Alice creates a link between *When the world starts* (an event) and *World.my first method*, as shown in Figure 5-1-4.

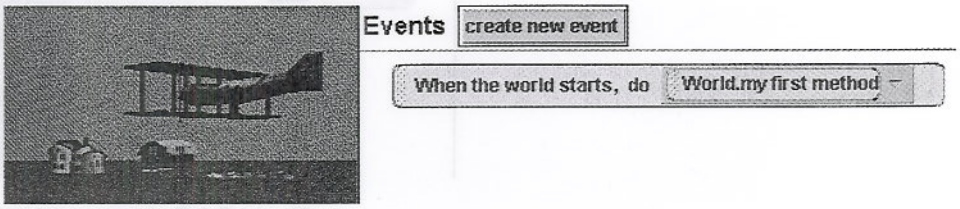


Figure 5-1-4. Event editor

In the flight simulator, two events (the up arrow key press and the spacebar key press) need to be linked to their corresponding method (*flyForward* and *barrel*). First, create an event by clicking the **create new event** button and then selecting the event from the popup menu. In Figure 5-1-5, the *When a key is typed* event is selected.

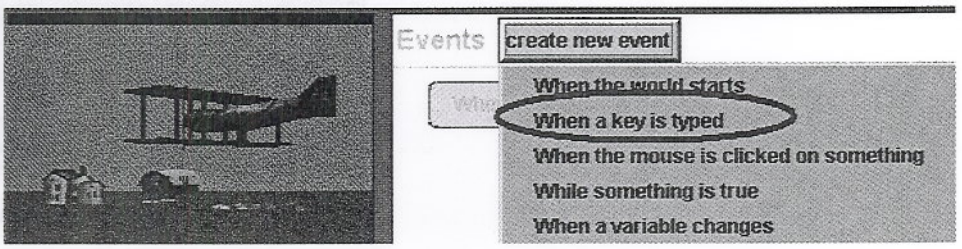


Figure 5-1-5. Creating a key-press event

In Figure 5-1-6, an event for *any key* pressed has been added to the Events editor. The "*any key*" and "*Nothing*" tiles are placeholders that need to be replaced. To tell Alice

that we want to use the up arrow key, clicking on the *any key* tile and select *Up* from the popup menu.

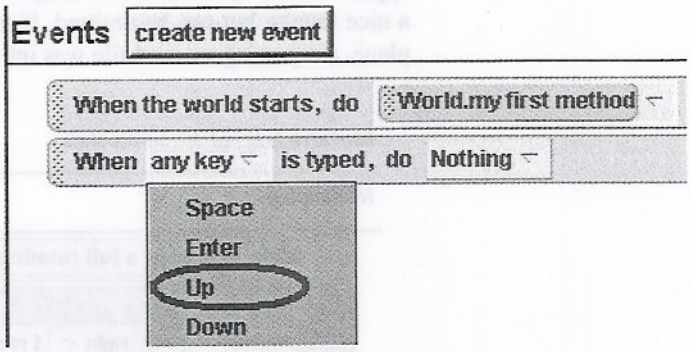


Figure 5-1-6. Specifying the up arrow key

Now that Alice has been notified that an up arrow key event may occur, we need to tell Alice what to do when the event happens. As shown in Figure 5-1-7, click on the *Nothing* tile and then select *biplane* and *flyForward* from the popup menu.

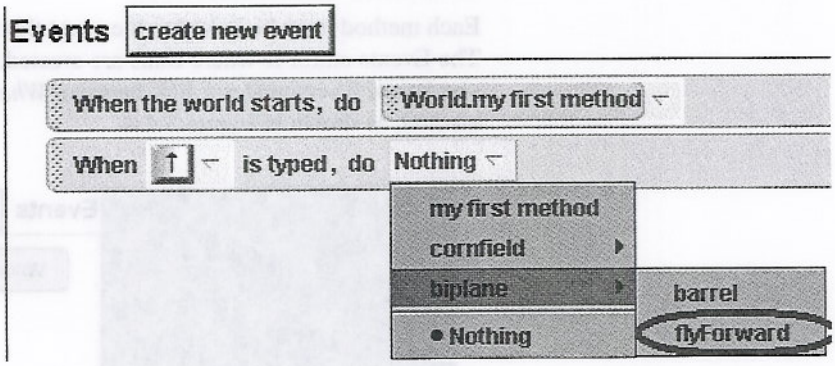


Figure 5-1-7. Link event-handling method to an event

The process is repeated to link the spacebar to the barrel method. Figure 5-1-8 shows the Events editor with both links completed.

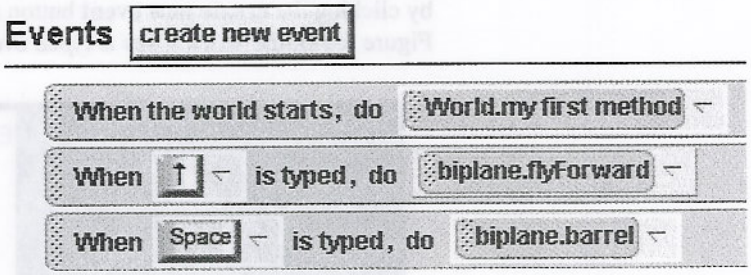


Figure 5-1-8. Links completed

Testing

Now the world should be tested. To test the flight simulator, just save the world and press the **Play** button. Nothing happens until the up arrow is pressed, which causes the biplane to call its *flyForward* method.

Events and methods could be created for the left and right arrow keys, and other acrobatic stunts could be written. (See exercise 1.) However, it is important to test event handling methods as they are developed. Write a method and test it, write a method and test it, until the program is completed. This is a recommended program development strategy called incremental development. Its advantage is in making it easier to debug your program. When something isn't working, it can be fixed before it causes problems elsewhere.

Note: An interactive world such as a flight simulator requires that the user know what keys to press to make the simulation work properly. A startup method could be written in *World.my first method* to display 3D text or a billboard for a quick explanation of the guidance system. After a few seconds, the 3D text (or billboard) can be made to disappear (by setting its *isShowing* property to *false*), and then the simulation can begin. 3D text and billboards were described in Tips & Techniques 2.

Events are world-level

In this example, the events were associated with the world. In Alice, we say that all events are world-level. Think about it this way: at all times, the Alice world is "listening" for an event to happen. When it happens, a method is called to respond to the event. With each new world, you can add events (as needed) in that world.

5-2 Parameters and event handling methods

As you have seen in Chapter 4, parameters are powerful tools. They allow us to customize methods to work with different objects and different numeric values. They are useful in building either world-level or class-level methods. In this section we will look at how to use parameters with events and event handling methods in interactive programs.

Once again, examples will provide a context for presenting the concepts of interactive programming. The first example illustrates how to use parameters in event handling methods. The second example illustrates how to allow the user to click on an object and then pass that object to an event handling method. Mouse-click selection is an important technique used in game programs and simulations.

A simple example

A firetruck (Vehicles) has been called to an emergency in a burning building (Buildings). A person and a fire object has been placed on each floor. The truck will need to extend its ladder so that each person can climb down to safety. The initial scene is illustrated in Figure 5-2-1.

Figure 5-2-1. Burning-building initial scene

Design—storyboard

To design an interactive program storyboard, we must give some thought to what events will occur and what event handling methods are needed. Let's allow the user to select the person to which the ladder should be extended. A textual storyboard is shown below.

```
Event: Click on guy1

Responding Method:
  Save guy on the first floor
```

```
Event: Click on girl2

Responding Method:
  Save girl on the second floor
```

```
Event: Click on girl3

Responding Method:
  Save girl on the third floor
```

Three events, one event handling method

Three events are possible, and three event handling methods could be written (one to respond to each event). Notice, however, that all three responses in the storyboards are exactly the same—save the person by extending the ladder and having the person slide down the ladder. Writing three event handling methods is unnecessary. A better solution is to write just one and send in the information needed to perform the action.

To simulate a rescue, the ladder must be aimed toward the floor and then extended to the window where the person is located. Then the person can slide down the ladder to the firetruck. Finally, the ladder should retract to prepare for saving another person. We will write one event handling method, named *savePerson*. A decision we have to make is whether to write the *savePerson* method as a world-level or as a class-level method. It makes sense to construct a class-level method for the firetruck because it is the object performing all the actions. On the other hand, other objects (guy1, girl2, and girl3) are the targets of the actions. We decided to create the *savePerson* method as a class-level method for the firetruck, passing in a target object to a parameter, named *whichPerson*. (This technique was previously described in Chapter 4, Section 3.) Using a class-level method will allow us to save out the firetruck with its *savePerson* method for reuse in other worlds.

In addition to the *whichPerson* parameter, two other parameters are needed: which floor the person is on (so the ladder can be made to point toward the right floor) and how far the ladder will need to be extended. The *whichFloor* and *whichPerson* parameters are of type *Object*. The third parameter, *howFar*, is a distance (for extending the ladder) and will be a *Number*. A possible storyboard for the *savePerson* method is:

```
savePerson

parameters: whichFloor, whichPerson, howFar
Do in order
  point ladder at whichFloor
  extend the ladder howFar meters
  whichPerson slides down the ladder to the firetruck
  pull the ladder back howFar meters
```

The code is presented in Figure 5-2-2. The swivel at the base of the ladder is pointed at *whichFloor* (the floor where the person is located). Then the ladder is extended *(smallLadder move forward) howFar* meters to reach the floor. The person slides down the ladder (*whichPerson move to*) to the firetruck. The ladder retracts backward the same distance (*howFar)* it was previously extended.

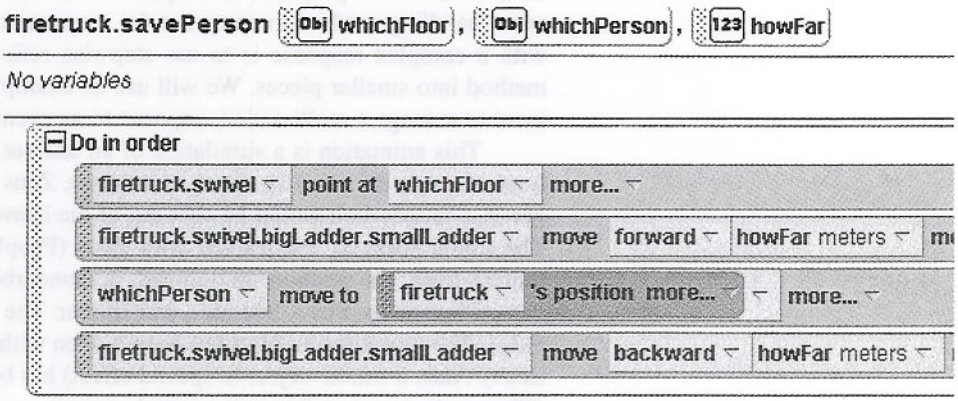


Figure 5-2-2. The code for the *savePerson* method

Link events to event handling method

Three events are possible, so three events are created in the Events editor, one for each person that can be selected by a mouse click, as shown in Figure 5-2-3. For each event, the same event handling method is called (*firetruck.savePerson*). The arguments sent to the parameters depend on which person was selected. For example, if randomGirl3 was clicked (on the third floor), *whichFloor* is sent *burningBuilding.thirdFloor*, *whichPerson* is sent randomGirl3, and *howFar* is sent 3 meters (the distance of the ladder from the third floor). (In setting up our world, we positioned the burning building and the firetruck so the distance of the ladder from the third floor is 3 meters, the second floor is 2 meters, and the first floor is 1 meter.)

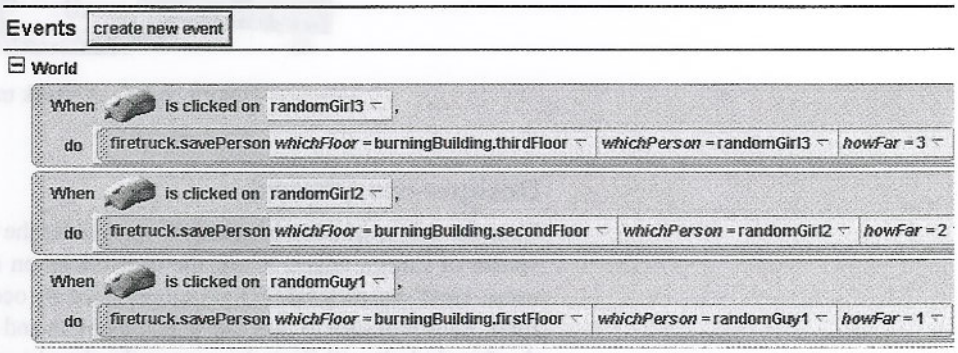


Figure 5-2-3. Three events: one for each object that can be clicked

Testing

When parameters are used in event-driven programming, it is especially important to run the animation several times, each time creating different events to be sure that each possible parameter value works as expected. A well-known guideline for testing numeric parameters is to

try a small value, a large value, and perhaps even a negative value—just to be sure the program works with a range of parameter values. In this example no negative value is used, but we could put one in just to see what would happen. We recommend you try out these tests as you are reading this section, if you have a computer nearby.

A more complex example

In an event-driven program, the response to an event may involve multiple actions. Writing an event handling method to carry out the response can become a bit messy. One way to deal with a complex response is to use stepwise refinement to break down the event handling method into smaller pieces. We will use an example to illustrate how to use stepwise refinement to manage a multi-action response to an event.

This animation is a simulation of an ancient Greek tragedy. (The ancient Greeks were fond of tragic dramas.) In Greek mythology, Zeus was an all-powerful god. If Zeus was angered, a thunderbolt would be shot out of the heavens and strike anyone who got in the way. The initial scene is constructed with Zeus (People) overlooking a temple scene (Environments) from his position on a cloud, a thunderbolt object, and some Greek philosophers named Euripides, Plato, Socrates, and Homer. The initial temple scene is illustrated in Figure 5-2-4. The thunderbolt object has been hidden within a cloud (the one immediately in front of Zeus). Also, a smoke object (a special effect) has been positioned 5 meters below the ground. (The smoke is not visible in the initial scene.)

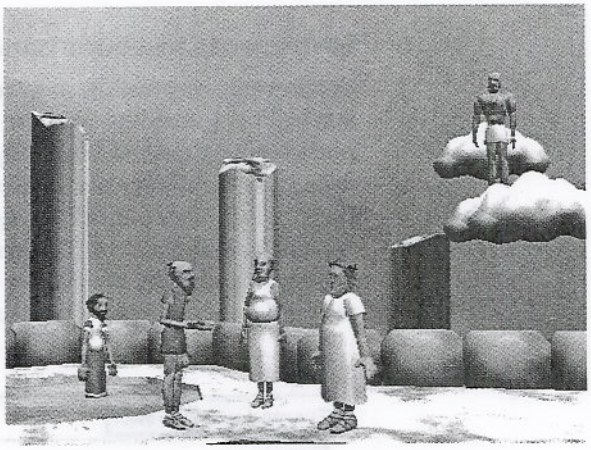


Figure 5-2-4. A Greek tragedy initial scene

Design—storyboard

To make this animation interactive, we will let the user choose the object that will be the next victim of Zeus's anger. When the user clicks on an object, the object will be passed to the event handling method. What actions need to occur in response to the mouse-click event? First, Zeus will turn to face the selected object and the thunderbolt will be made visible. Then, the thunderbolt will flash down to strike the object. Smoke will be used as a special effect to make the object appear to meet a sad fate. Then, the lightning bolt must be repositioned to prepare for another lightning strike. (In interactive worlds, the user can click on more than one object.)

Clearly, the event handling method will involve many different actions. To organize an event handling method (named *shootBolt*), let's begin with a storyboard that summarizes the overall actions. A parameter (arbitrarily named *who*) is needed to send in the object that was clicked.

```
Event: An object is mouse-clicked

Event handler: shootBolt

Parameter: who-the object that was clicked

Do in order
    prepare to strike the object that was clicked
    thunder plays and lightning strikes the object that was clicked
    lightning is repositioned for the next strike
```

In this storyboard, the first two steps in the *Do in order* block are actually composed of many actions. If we were to translate these into code in a single method, it would be many, many lines of code. A long method (consisting of many lines of code) is often difficult to read and debug. Let's use stepwise refinement to break the design down into smaller pieces. We could write a method, *prepareToShoot,* for the first step and a second method, *lightningAndThunder,* for the second step. In the third step, repositioning the lightning bolt for another strike can be performed as a single *move to* instruction. The revised storyboard would look like this:

```
Event: An object is mouse-clicked

Event handler: shootBolt

Parameter: who-the object that was clicked

Do in order
    call prepareToShoot method-send who as the target
    call lightningAndThunder method-send who as the target
    lightning move to cloud's position
```

Now the *shootBolt* method will be easy to write because all it does is call two other methods and then reposition the lightning bolt behind the cloud. A method that does very little other than organizing the calls to other methods is known as a driver. The driver organizes and calls the methods. The called methods do almost all the work. An important role played by the *shootBolt* method is to pass along (to the called methods) the object that was clicked (*who*). Each of the called methods will use the object that was clicked as the target of its own actions.

For now, let's assume that the *prepareToShoot* and *lightningAndThunder* methods have been written (we will write them later). Then, we can write the *shootBolt* method, as in Figure 5-2-5.

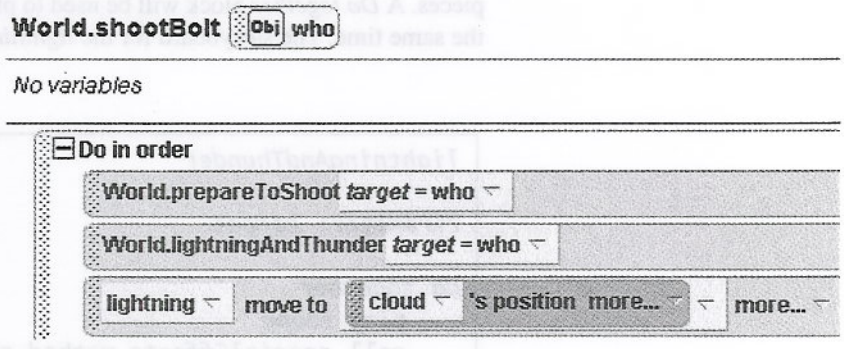


Figure 5-2-5. The *shootBolt* event handling method calls other methods

Now that you have the overall idea of how an event handling method can act as a driver to call other methods, let's write the methods that are called. First, the *prepareToShoot* method prepares Zeus and the lightning bolt for shooting at the object that was clicked. A simple storyboard is:

```
prepareToShoot

Parameter: target
Do together
  turn Zeus to face the target
  make the lightning bolt visible
```

A *turn to face* instruction will make Zeus look at the target. In setting up the world, the lightning bolt was made invisible by setting its opacity to 0 (0%). To make the lightning bolt visible (so we can see it flash across the scene), we write an instruction to set its opacity to 1 (100%). The code for the *prepareToShoot* method is shown in Figure 5-2-6.

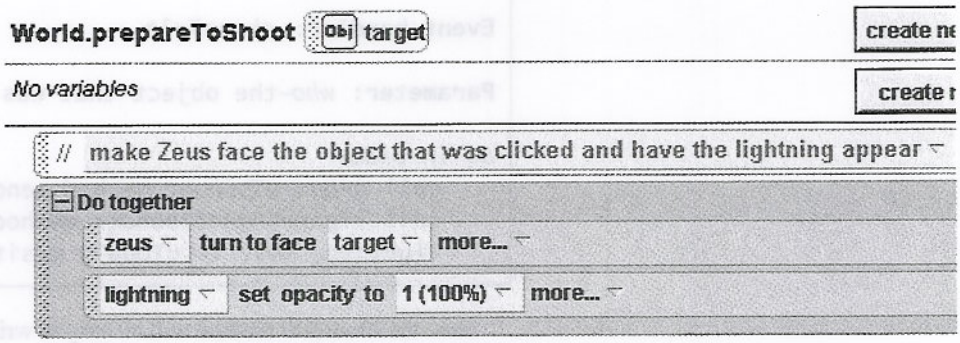


Figure 5-2-6. The *prepareToShoot* method

In the second method, *lightningAndThunder,* a sound (thunder) plays at the same time as lightning strikes the targeted philosopher. Of course, the lightning strike should have some special effects (smoke, color change, philosopher jumps). The special effects involve many actions. Once again, we will use stepwise refinement to break our code down into smaller pieces. A *Do together* block will be used to play the sound and call a *specialEffects* method at the same time. The storyboard for the *lightningAndThunder* method is:

```
lightningAndThunder

Parameter: target

Do together
    play sound
    call specialEffects method-send target as the victim
```

Now that we have a design, the next step is to translate it into program code. Let's pretend that the *specialEffects* method is already written. Then, the *lightningAndThunder* method could be written, as shown in Figure 5-2-7. Playing a sound at the same time as other actions requires that the amount of time the sound plays must be synchronized with the amount of time needed for the special effects. In this program, we found it worked best if we inserted a *Wait* instruction with the *play sound* instruction (in a *Do in order*) to create a short delay. The call to the *specialEffects* method passes along *target* as the victim of the lightning strike.

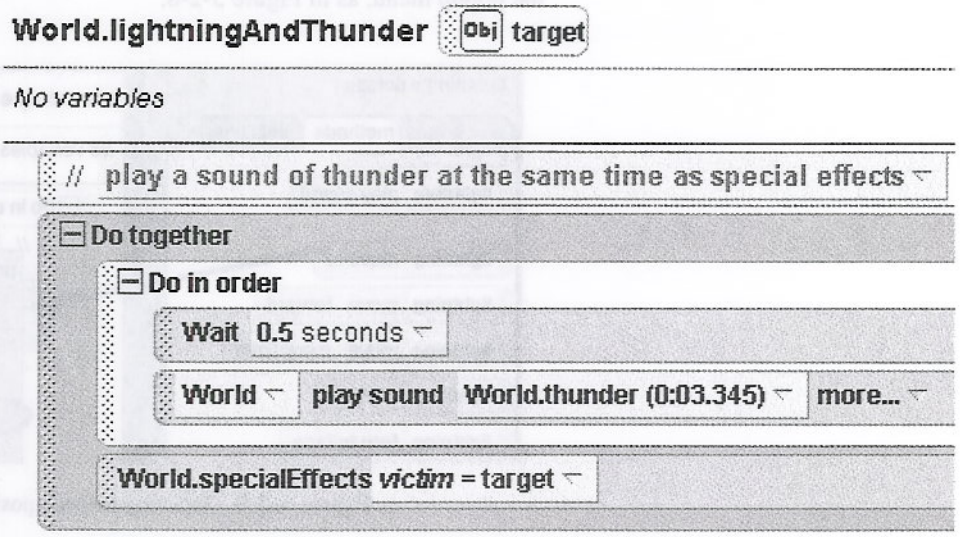


Figure 5-2-7. Code for the *lightningAndThunder* method

Finally, it is time to write the *specialEffects* method. First the lightning should strike, and then smoke should appear around the targeted object (*victim*). The victim should show the effects of a lightning strike—change color and jump up and down. A storyboard for *special Effects* is:

```
specialEffects

Parameter: victim

Do in order
  Do together
      lightning bolt move to victim
      smoke move to victim
  Do together
      set smoke to visible
      set lightning to invisible
      call smoke cycle-built-in method
      set victim's color to black
      move target up and down
  Do together
      set smoke's opacity to 1
      smoke move down 5 meters
```

Using an object parameter with *move to*

We are now ready to translate the storyboard for *specialEffects* into program code. The only troublesome instructions are "lightning bolt move to victim" and "smoke move to victim." A *move to* instruction needs to know a position (location in the world) to which an object will be moved. This is no problem when *move to* is targeted at the position of another object and that object is listed in the Object tree (and thereby in the popup menu for the *move to* instruction). For example, moving the lightning from the *cloud* to *homer* is easy to do. Just drag the *lightning move to* tile into the editor and select *homer's* position as the targeted victim from the popup menu, as in Figure 5-2-8.

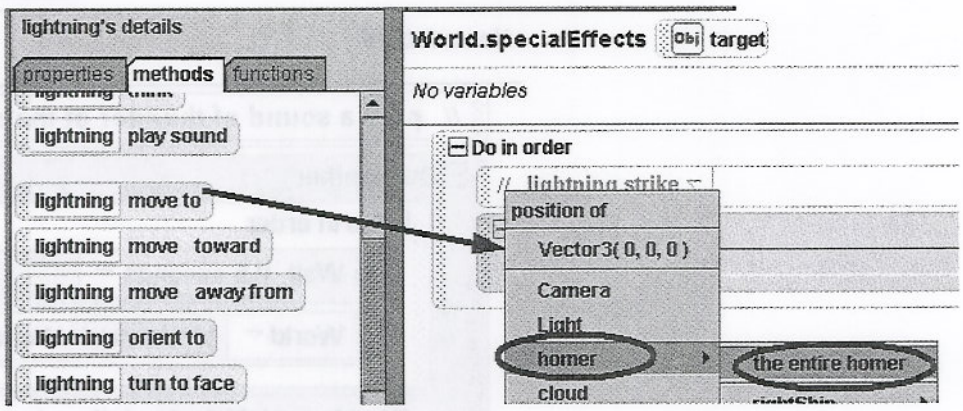


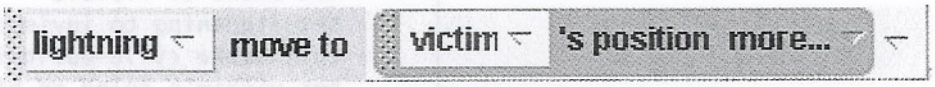
Figure 5-2-8. Selecting homer's position for a *move to* instruction

What about a selected parameter's position in the popup menu for a *move to* instruction? The parameter (*victim)* does not appear in the popup menu of available positions. This is because *victim* is just a placeholder (is not a real object) and does not appear in the Object tree in the world. It is not reasonable to expect that Alice would put a non-existing object in a menu of available positions for the *move to* instruction.

Not to worry—a two-step process can be used to put the *victim* parameter in a *move to* instruction:

1. From the popup menu of available positions, select a position (we chose homer arbitrarily, but any object is okay to select). The result should look something like this:

2. Drag the *victim* parameter tile into the editor and drop it onto the position tile. The result is:



An instruction to move the smoke to the position of the victim is created in a similar manner. We realize that this is a bit of a subterfuge technique. It is, though, a reasonable way to handle the fact that an object parameter (in this example, *victim*) is a nonexistent object and cannot appear in the popup menu as a targeted victim for the *move to* instruction.

After the lightning strikes, we need to make the lightning bolt invisible and the smoke visible. This is accomplished using *set opacity* instructions. Figure 5-2-9 shows the complete implementation of the *specialEffects* method.

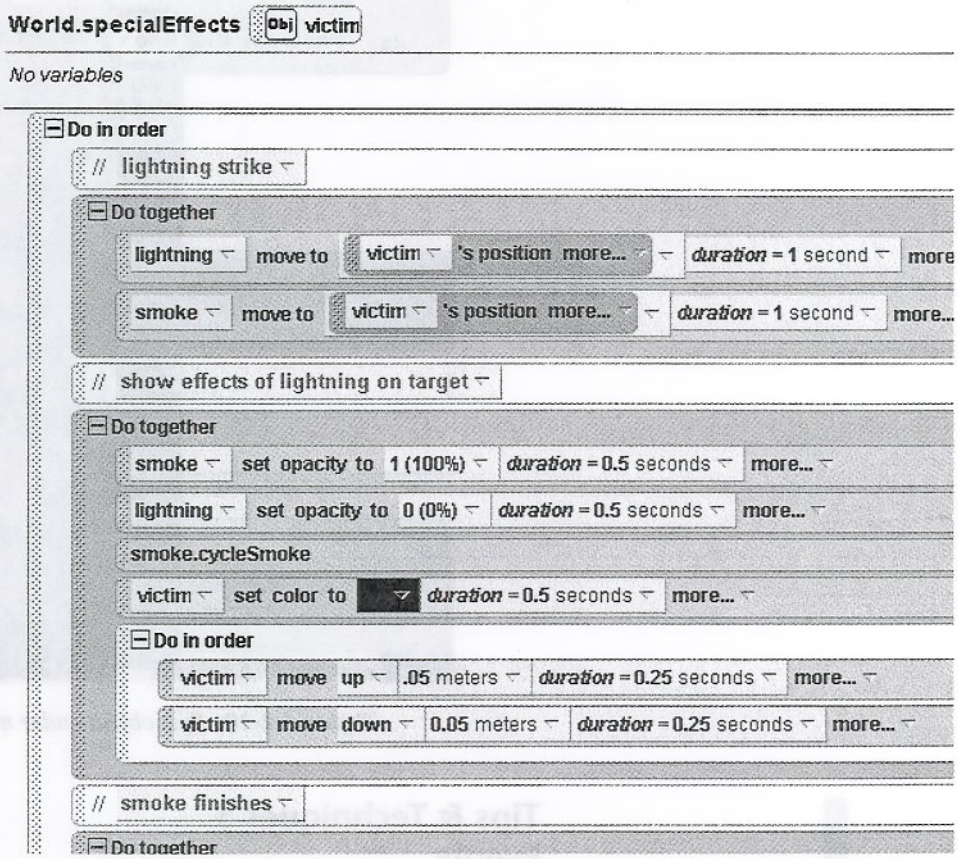


Figure 5-2-9. The *specialEffects* method

In the *specialEffects* method, we took advantage of a built-in method for smoke (*cycleSmoke*) to create a smoke animation. To animate the effects of the lightning strike, the target color is set to black and made to move up and down.

Link the event to the event handling method

All that is left to do is link the mouse-click event to the *shootBolt* event handling method. In the Events editor, select *when the mouse is clicked on something*. Then drag the *shootBolt* method tile into the link. Of course, *shootBolt* expects to be passed an argument to identify the object that was clicked. Select *expressions* and then *object under mouse cursor*, as shown in Figure 5-2-10.

Testing the program

The Zeus world is now complete. Naturally, we should test the program by running it and having Zeus shoot thunderbolts. When we tested this program, we clicked on each of the philosophers, to make sure the thunderbolt properly hit the target. But when we clicked on the clouds, the thunderbolt struck the clouds, turning them black. And when we clicked on the scene itself, the whole scene was turned black! And, clicking on Zeus causes Alice to complain. (Zeus can't zap himself with lightning!) This is not the behavior we wanted or expected. Another

problem with the animation is that the user can click on an object that has already been zapped with lightning. A solution to these problems will be presented in the next chapter.

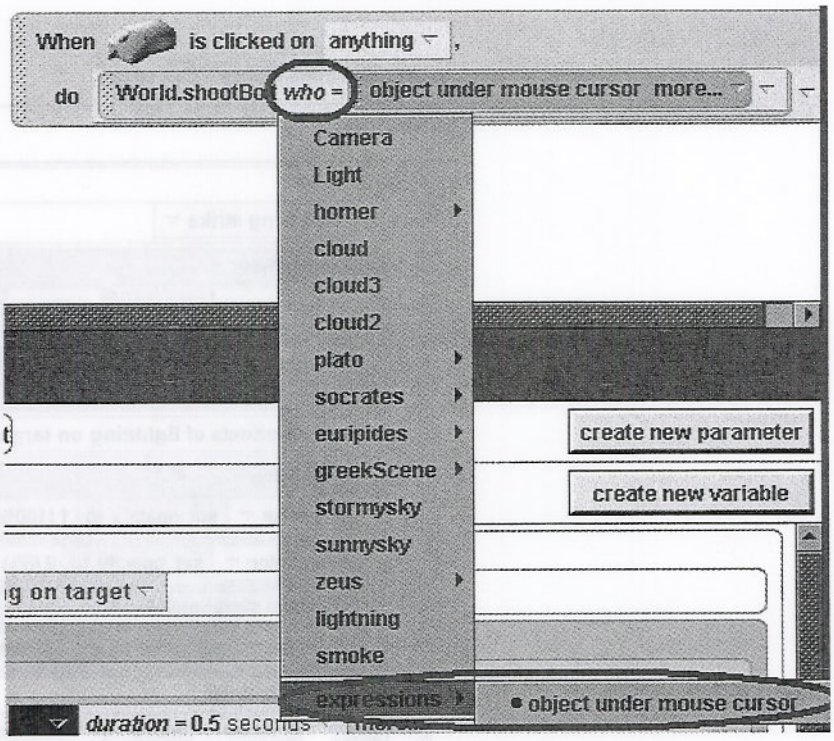


Figure 5-2-10. Pass *object under mouse cursor* as the parameter

Tips & Techniques 5
Events

A Quick Reference to Events

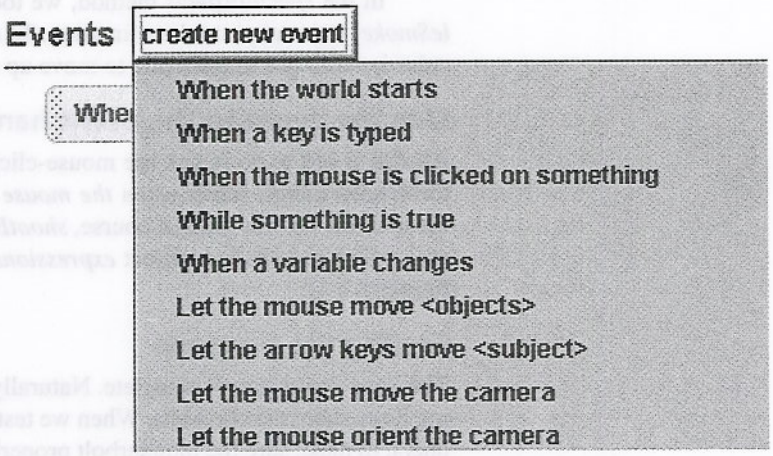


Figure T-5-1. Possible events in Alice

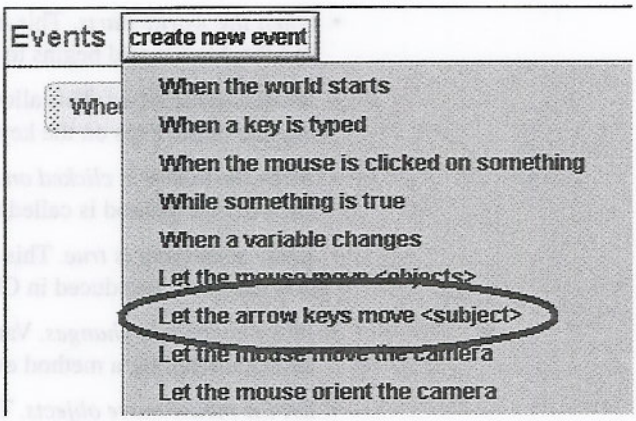
- *When the world starts*. This event happens once, when the Play button is first pressed and the Alice world begins to run.
- *When a key is typed.* This allows for a method to be called in response to the user pressing one of the keys on the keyboard.
- *When the mouse is clicked on something.* Each time the mouse is clicked on an object in the world, a method is called to handle the event.
- *While something is true.* This is a sophisticated event that will be presented after repetition has been introduced in Chapter 7.
- *When a variable changes*. Variables will not be introduced until Chapter 10. This event allows for calling a method every time a specified variable changes value.
- *Let the mouse move objects*. This event automatically calls an internal Alice method that moves the object in a drag-and-drop manner.
- *Let the arrow keys move <subject>*. This allows the user to move a specified object by pressing the arrow keys. The up arrow moves the object forward, down arrow backward, the right and left arrows move the object right and left.
- *Let the mouse move the camera.* This allows the user to "steer" the camera with the mouse during an animation. Note that it is possible to point the camera away from the animation. If this happens, you will no longer see what is going on.
- *Let the mouse orient the camera*. Like the preceding event, this one must be used with caution, as you can easily orient the camera to point into space and then miss the rest of the animation that is running.

Exercises and Projects

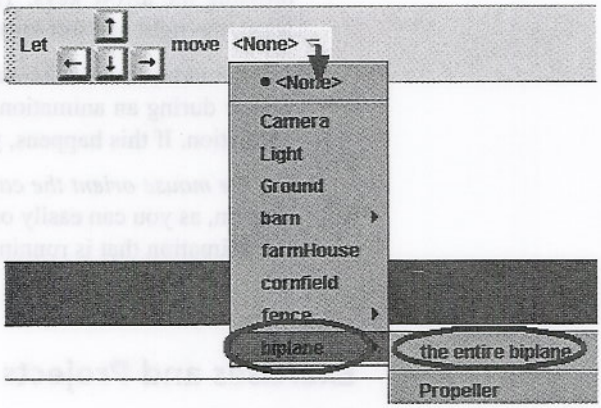
5-1 Exercises

1. *Flight Simulator Completion*
   (a) Create the world for the biplane acrobatic air-show example as presented in this section. Implement the *flyForward* and *barrel* event handling methods and link them to the corresponding events. Make the move and roll actions have an abrupt style to reduce the pause in the animation between key presses. If your computer has sound, use a biplane sound to make the animation more realistic.
   (b) When you have the *flyForward* and *barrel* methods working, add *flyLeft* and *flyRight* event handling methods for the left and right arrow keys to steer the biplane left or right.
   (c) Add a *forwardLoop* stunt that works when the user presses the Enter key.

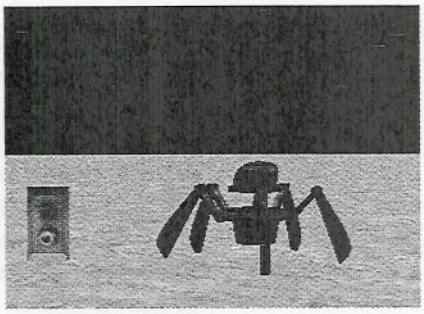
2. *Flight Simulator—Alternate Version*
   The arrow key-press events work when the user releases the key. Of course, this means that multiple key-press/release events are needed to keep the biplane moving. In this exercise, you can experiment with a different kind of event. Create a second version of the biplane world (use **File → SaveAs** to save the world with a different name). In the second version of the world, remove the events that link the arrow keys to the *flyForward*, *flyLeft* and *flyRight* event handling methods. In the Events editor, create a new event by selecting *let the arrow keys move <subject>*, as shown below.



Then, link the biplane, as shown below. Run the flight simulator again to see the effect.



3. *Robot Remote Control*
The world for this exercise is similar to the FirstEncounter world in Chapters 2 through 4. In this world, however, we want to allow the user to control the robot using some sort of remote control. One possibility is to use the TwoButton switch (Controls) to simulate a robot control system. When the user clicks the green button on the switch, the robot should move forward, with two of its legs providing a walking motion. When the user clicks the red button on the switch, the robot should move backward, with two different legs walking in the opposite direction. Use the world-level function *ask the user for a number* to allow the user to determine how many meters the robot moves forward or backward.



7. *Ninja Motion*
   A ninja (EvilNinja in People folder) is trying out for a karate movie, and he needs a little practice. Create a world with a ninja object in a dojo. The motions the ninja needs to practice are: jump, duck, chop, and kick. If you have not already done so, (see Exercise 12 in Chapter 4), write motion methods for the ninja that include the following:

   kickRight, *kickLeft:* allows the ninja to kick his right/left leg, including all appropriate movements (e.g., foot turning, etc.)

   rightJab, *leftJab:* allows the ninja to do a jabbing motion with his right/left arms

   Create events and event handling methods that provide the user with controls to make the ninja jump, duck, jab and kick.

8. *The Cheshire Cat*
   Consider the Cheshire cat (Animals) from the *Alice in Wonderland* books. Sometimes, the cat would disappear, leaving only his grin. At other times, the cat would reappear. Create such a world, where the cat (except for its smile) disappears when the red button of the switch (Controls) is clicked, then reappears when the green button is clicked. (The tree is found in the Nature folder.)

5-2 Exercises

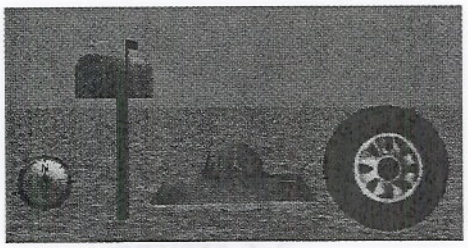
9. *Zeus Modification*
   Recreate the Zeus world presented in Section 5-2. In this world, a philosopher that has been zapped by lightning and is scorched to show he has met a tragic ending. The philosopher is still in the scene, however, and the user can zap him again. Revise the program to make the zapped philosopher fall down below the ground, where he can't be mouse-clicked again.

10. *Furniture Arrangement*
    Create a world where the user will rearrange the furniture in a room. (A variety of furniture items can be found in the Furniture folder on the CD and Web galleries.) An example

4. *Typing Tutor*
   Learning to type rapidly (without looking at the keyboard) is a skill requiring much practice. In this exercise, you are to create a typing tutor that encourages the beginning typist to type a specific set of letters. Use 3D text letters (3D Text folder) to create a word in the world, (for example, you could create the word ALICE with the letters A, L, I, C, and E) and create a method for each letter that spins the letter two times. When the user types a letter key on the keyboard that matches the letter on the screen, the letter on the screen should perform its spin method. Also include an additional method, *spinWord*, which spins all the letters when the user presses the spacebar.

   Hint: Use *asSeenBy* to spin the word.

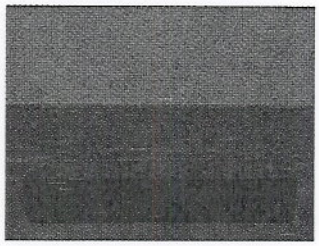
5. *Rotational Motion*
   A popular topic in Physics is the study of rotational motion. Create a world with at least four objects (such as a compass, mailbox, mantleClock and tire.) Create a realistic rotation method for each object that has the object rotate 1 full revolution and then perform some other action. For example, if one of your objects is a compass, make the compass needle spin around quickly in one direction and then spin around again in the opposite direction. Add events that will call the rotational motion method for an object when the object is clicked. (If the world has four objects, four events will be needed.)



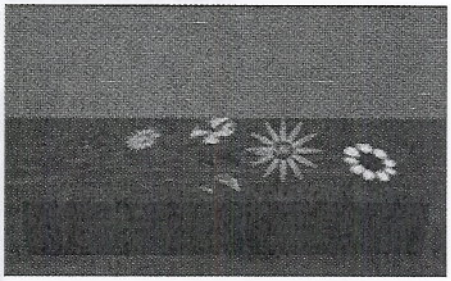
6. *Mad Scientist Magic*
   Who says that a mad scientist has no magic skills? Create a world using a mad scientist (People) behind a table (Furniture). Place three objects on the tabletop—for example a blender and mug (Kitchen). The point of the exercise is that when the mouse clicks any one of the objects on the tabletop, the mad scientist will turn to face that object, raise his arm (like he is casting a spell of some sort), and have the object spin or turn in any way you wish. The mug has some kind of liquid in it. Make the liquid disappear (by setting its *isShowing* property to *false*) when the mad scientist performs his "spell" on it.

13. *Flowerbox*

It's spring and you are anxiously waiting for flowers to grow. You decide to give them a little help. Create an initial scene of a flowerbox (Box from Shapes, change color to red) with five flowers (of your choosing) in it. (Use instructions in the initial scene to move the flowers down out of sight.)



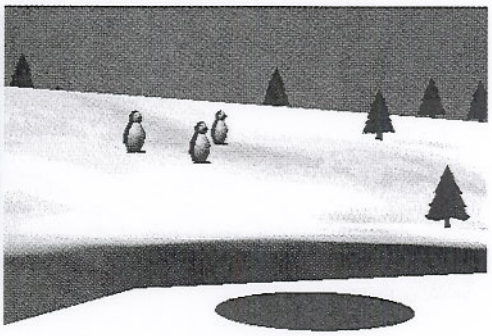
Write one method to grow a flower in the box (move the flower up into view). The flower that grows in the box depends on which key the user presses on the keyboard. For example, if the user presses "S" key, the sunflower will grow but if the user presses the "D" key, the daisy will grow. To grow the flowers, create a "*When <key> is typed*" event for each key selected to represent a specific flower. Link the key-press event to the *growFlower* event handling method, using the particular flower represented by that key as its parameter. When all the flowers are grown, the flowerbox will look something like the following:



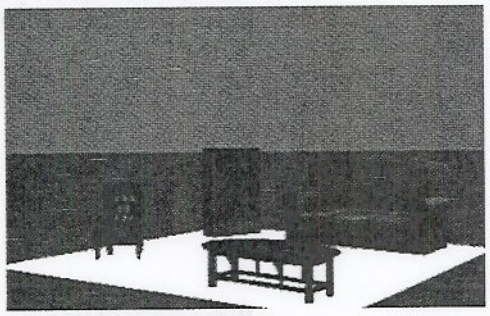
14. *Penguin Slide*

A favorite activity of penguins in the local zoo is to slide down an icy slope into a pool of water in the pond. Create a world with a lake scene (Environments) and three penguins (Animals) on the slope, as shown below. Make the program event-driven. Allow the user to click on the next penguin to slide down the slope into the pool of water. Each penguin slides on its belly and spins around as it slides. Each penguin should spin some number of times. When the penguin reaches the pond, move the penguin down 5 meters so it disappears below the water. Write only one event handling method. When the penguin is mouse-clicked, pass the penguin object that was clicked and the number of times the penguin is to spin around as it slides down the slope.

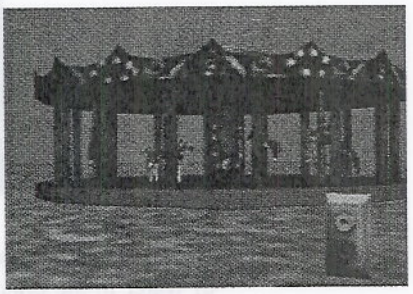
Optional: Add a water-splash sound as the penguin hits the water.



of a room arrangement is shown below. (We used a rectangle Shape to represent the floor of the room.) To allow the user to move the furniture around, create a *Let the mouse move objects* event.

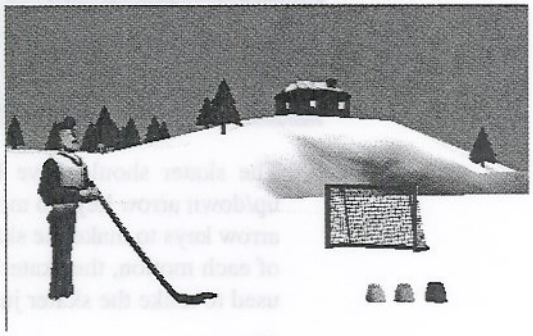


11. *Carousel Go-round*
Create an amusement park scene with a carousel. (Amusement Park) In this animation, the carousel is to have at least four animals. (You can use the animals that come with the carousel or add your own.) Add a two-way switch (Controls) to the initial scene. Create two event handling methods—one to make the carousel rotate clockwise and one to make it rotate counterclockwise. When the green button is clicked, the carousel should rotate clockwise, and a click on the red button should make the carousel rotate counterclockwise. Optional: Import a sound file and have it play as the carousel goes around.



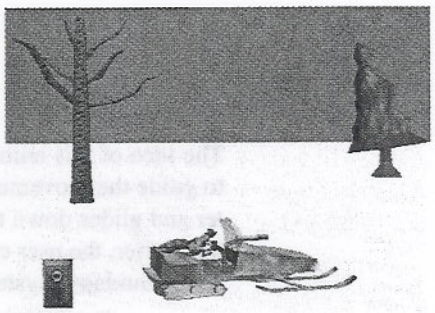
12. *Snow Festival*
Your team has created a snowman as the centerpiece of an entry in the Winter Snow Festival competition. To attract attraction to your display, you have set up colored spotlights that will turn the color of the snowman any one of four different colors. Create an initial world with four spotlights (spheres from the Shapes folder, of four different colors) and a snowman, as shown below. Write only one method to change the color of the snowman. When the user clicks on a spotlight pass the color of the spotlight to the method and make the snowman change to be that color.

15. *Hockey*
Jack (Jock from High School/Students and Teachers folder on the CD or Web gallery) is planning to try out for the school hockey team this fall. As a successful athlete, Jack knows that "practice is the name of the game." Jack has set up a hockey net (Sports) on the lake and is going to practice his aim with the hockey stick (Sports) to improve his chances of making the team.



This animation could be the first phase of developing an interactive ice-hockey game. To design an interactive program storyboard, some thought must be given to what events will occur and what event handling methods are needed. Let's allow the user to select the power factor behind Jack's swing of the hockey stick. The power factor will determine how fast Jack swings the stick and how far the hockey puck travels when hit by the stick. The power factor will be selected by a mouse-click on one of the power buttons (GumDrops from Kitchen/Food folder) in the lower right of the scene. The yellow button will select low, green will select average, and red will select a high power factor.

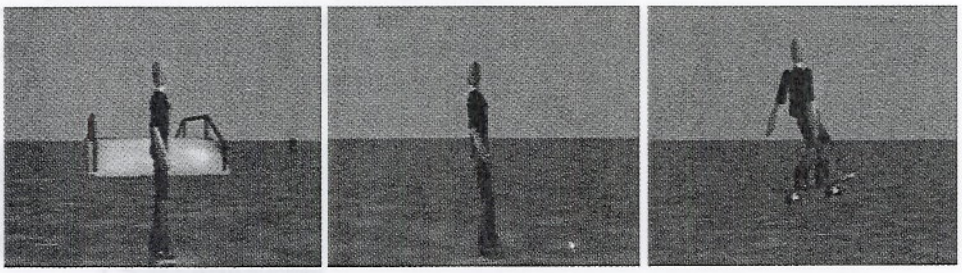
16. *Slappy*
Slappy, an adventuresome squirrel (Animals on CD or Web gallery), has just gotten her own squirrel-sized snowmobile (Vehicles on CD or Web gallery). Create a program to animate Slappy's first ride on the snowmobile. The user controls the forward and reverse motion of the snowmobile, using a two switch box (Controls). When the user clicks the green button on the switch, the snowmobile and Slappy move forward and Slappy screams something like "wahoo!" When the user clicks the red button on the switch, the snowmobile and Slappy move in reverse and Slappy looks at the camera and says something like "!oohaw."



Projects

1. *Skater World*
The goal of this world is to simulate a person doing various skating movements on a skate board. Create a world with a skater girl (People) on a skateboard (SkatePark). Add

one or two objects she can jump over. We used a rail ramp from the SkatePark folder on the CD or Web gallery. A sample scene is shown below.



The skater should have both *jump* and *spin* methods. Allow the user to press the up/down arrow keys to move the camera and skater forward/backward and the left/right arrow keys to make the skater lean left/right while the camera goes left/right. At the end of each motion, the skater should lean back to her original position. The "**j**" key can be used to make the skater jump and the "**s**" key to make her spin.

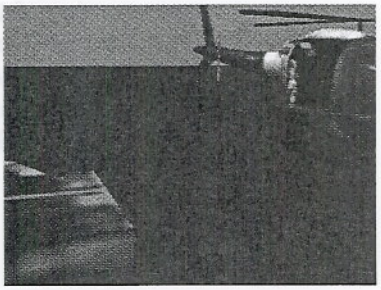
Hints:

(a) If you are having problems with the skater girl moving where you want her to move, trying using *asSeenBy* camera or *asSeenBy* skateboard.

(b) One way to make the camera follow the action is to position it behind the skater and make the skater its vehicle.

2. *Skydiving Guidance System*

Alice Liddell (People) has taken up a new hobby: skydiving. She is on a helicopter (Vehicle), wearing a parachute (Objects). She is to jump to the carrier (Vehicle), which is a little way in front of her. In the world shown below, we added a half cylinder (Shapes) inverted and connected to the helicopter to be used as a jump platform. A torus (Shapes) was used to create a harness for Alice Liddell. (Although this isn't absolutely necessary, it is helpful due to Alice Liddell's small waist compared to the parachute's cords.)



The idea of this animation is to provide a skydiving guidance system to allow the user to guide the movement of Alice Liddell as she jumps from the platform of the helicopter and glides down to the carrier. When the user thinks Alice Liddell has hit the top of the carrier, the user can press the Enter key to have Alice Liddell drop her chute.

Guidance system methods (as smooth and lifelike as possible):

jump: jump from the helicopter's platform

glideForward, glideBack, glideRight, glideLeft: glide in the appropriate direction

swingLegs: legs swing a bit when gliding or jumping

dropChute: get rid of parachute (to simplify things, just have the chute rotate as if it were falling and make it disappear)

Keyboard controls:

Space bar—jump off the platform
Up/Down/right/left arrow—glide forward/back/right/left
Enter key—parachute drops

Remember that Alice Liddell should first jump off the platform prior to gliding and should not drop her chute until she hits the carrier.

3. *Turtle Motion Control*

In this project, you are to create a turtle motion controller to help the turtle (Animals) perform exercises for his upcoming race with the rabbit. Create a world that contains only a turtle and then create motion control methods for the turtle:

headBob: allows the turtle's head to bob a little

tailWag: allows the turtle's tail to wag

oneStep: allows the turtle to move forward one step; his legs should move while he is taking that one step

walkForward: combines the above three methods, to make a realistic step; all movements should take the same amount of time and occur at the same time

turnAround: turns the turtle 180 degrees; he should be walking while turning

turnLeft, turnRight: turns the turtle left/right, walking while he is turning

hide: allows the turtle to hide in his shell (you may assume that the turtle is currently outside of his shell); remember not to leave the shell hanging in midair

reappear: allows the turtle to reappear from his shell (you may assume that the turtle is currently hidden)

talk: has the turtle look at the camera and say "hello" (or something different, if you wish) to the user

Create keyboard controls:

When the up arrow key is pressed, the turtle is to walk forward.

When the down arrow key is pressed, the turtle is to turn around.

When the left arrow key is pressed, the turtle is to turn left.

When the right arrow key is pressed, the turtle is to turn right.

When the letter "H" is pressed, the turtle is to hide in his shell.

When the letter "R" is pressed, the turtle is to reappear from his shell.

When the letter "T" is pressed, the turtle is to talk to the user.

Test the turtle motion control system by running your world and trying all the interactions at least once. Be sure to hide the turtle only when he is already out of his shell and have him reappear only when he is hiding.

Summary

The focus of this chapter was the creation of interactive (event-driven) worlds. Creating worlds with events will allow you to build significantly more interesting worlds such as game-like animations and simulations. In many object-oriented programming languages, event-driven programming requires knowledge of advanced topics. The Events editor allows you to create events and link them to event handling methods. The event handling method has the responsibility of taking action each time the event occurs. The Events editor handles many of the messy details of event-driven programming.

Important concepts in this chapter

- An event is something that happens.
- An event is created by user input (keyboard press, mouse click, joystick movement).
- An event is linked to an event handling method.
- Each time an event occurs, its corresponding event handling method is called. This is what is meant by event-driven programming.
- The event handling method contains instructions to carry out a response to the event.
- A parameter can be passed to an event handling method when an event occurs.
- Parameters allow us to write one method that can handle several related events.
- Incremental development means that you write and test a small piece of your program, then write and test the next small piece, and so forth, until the entire program is completed. Incremental development is another technique that makes it easier to debug your programs.