

LUNAR LANDER PROJECT
COMPUTER PROGRAMMING
OPHS SPRING, 2008

and more sophisticated organization of the tasks involved in a development project, including the timing of tasks and the resources needed to complete each task. Gantt charts and PERT charts are not often used by novice programmers, so your development process will be based directly on this organizational chart.

The rest of this chapter has three tutorials—creating the initial scene, designing its events and methods, and then coding the methods and events. Creating the initial scene and coding the methods and events are both hands-on activities in Alice. The design of the events and methods should be completed before the coding begins, so that you as the programmer will understand what you are creating and why. After creating the initial scene, you may want to complete reading the discussion of the design in one session and then do the coding in another, because each of these will take some time. You could, for example, read the discussion of the design on the day or evening before you will actually do the coding. It is a common mistake for novice programmers to try to do too much at once, and to be too eager to jump into coding before they fully understand what the code should do. It also might help if you discussed the design with other students before beginning the coding.

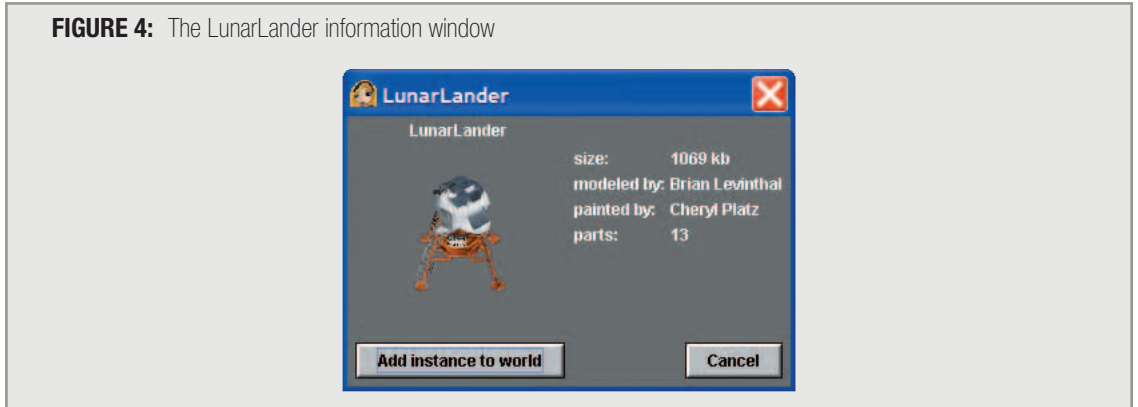
TUTORIAL A—DEVELOPING THE LUNAR LANDING SIMULATOR’S INITIAL SCENE

For this tutorial, the following list of tasks, which is based on the approach shown in the organizational chart above, will walk you through the creation of the Lunar Landing Simulator’s initial scene. For a project of your own, you would need to develop this list based on the specifications for the project.

The simulator will need to be developed from one of the six existing Alice templates. The **space** template seems to be best for this project. To create the initial scene, you need to add two objects to the world: a LEM and something to mark the target zone. The objects will each need to be positioned and renamed, and the camera will also need to be positioned.

1. Open a new Alice world with a space template.
2. There is a *lunarLander* in the Alice *SciFi* folder (not the *Space* folder) in the *Local Object Gallery*. Click the **ADD OBJECTS** button, and then find the *lunarLander* in the Alice *SciFi* folder. Click the **LunarLander** class tile, and you should see the LunarLander information window, as shown in Figure 4. Click the **add instance to world** button. It is important to add each object to the simulation this way (rather than by dragging its class tile into the world), so that the LEM and the target zone will be in the correct starting positions.

FIGURE 4: The LunarLander information window



3. Right-click the **lunarLander** tile in Object tree, and rename it **LEM**. The real Lunar Excursion Module was sometimes referred to as the LEM and sometimes as the LM, although both were pronounced like the name “Lem.”
4. The simulation will need to keep track of the LEM’s altitude and downward velocity. A new property is needed to keep track of the LEM’s downward velocity. You need to create a new LEM variable to store the property. Select the LEM’s **Properties** tab, and then click the **create new variable** button.
5. When the create new variable window appears, type **descentSpeed** as the name and if necessary select **number** as the type. Change the initial value to **0**, and then click the **OK** button. A tile for the new variable will appear near the top of the LEM’s properties tab.

The LEM and the camera need to be moved to their starting positions. The specifications call for the simulation to start with the LEM 100 meters above the surface.

1. Run primitive methods to turn the LEM to **face the camera**, and then **move it up 100 meters**. Remember, you can position an object, turn it, and so on, by right-clicking an object’s tile in the Object tree and then running methods directly. The LEM will move off camera.
2. The camera needs to be positioned. A starting point slightly above the LEM and back from it a little should work. Run the required primitive methods to **move** the camera **up 125 meters**, and then **move it backward 25 meters**.

A target landing zone is required. There is a doughnut-shaped object called a torus in the Shapes folder. This is the red shape that can be seen back in Figure 2.

1. Add a torus to the world by clicking its class tile in the *Shapes* folder and then clicking the **add instance to the world** button.

2. The torus is off-camera. Right-click the **camera's** tile in the Object tree, and then run the **point at** method to point the camera at the torus.
3. The torus is on the surface, but can hardly be seen. It needs to be larger and a more visible color. Set the *color* property on the Torus's properties tab to **red**. Then, right-click the **torus** tile in the Object tree and run a method to **resize** it to be **20** times its initial size. Now it should be more visible on the surface of the moon.
4. The torus needs to be renamed. Right-click the **torus** tile in the Object tree again and rename it **target**.
5. The target needs to be positioned. Since you moved the LEM straight upward, the target is now directly under the LEM. It doesn't matter exactly where it is on the surface, as long as it's not too far away from its starting point and not directly under the LEM. For now, drag it to any corner of the world window.
6. Now the camera needs to be pointed back at the LEM. To point the camera at the LEM, right-click the **camera** tile in the Object tree and run the method to make the camera **point at the LEM, the entire LEM**.
7. The camera needs to be locked onto the LEM so that it will follow it as it descends. To do this, right-click the **camera** tile in the Object tree, and run the primitive method to **set the camera's vehicle** to be the **LEM, the entire LEM**.
8. Now the world is set up in a good starting position with the necessary objects. Click the **DONE** button to leave the Scene Editor mode. Figure 5 shows what the world window should look like with the camera and LEM in their starting positions. The target zone is also in place on the surface, but that is currently off-camera.
9. At this point, you should save the world before continuing. Save the new Alice world with the name **Lunar Landing Simulator** and remember to pay attention to where you save it.

FIGURE 5: The world window with the camera and LEM in their starting positions



TUTORIAL B—DESIGNING THE LUNAR LANDING SIMULATOR

Next, the development process calls for the design of events, and then methods needed to handle those events. In this project, you will be given a design for both events and methods. In a project of your own, you would need to spend time creating the design before starting to create any code in the new world. The rest of this section discusses the design for the events, while the next section discusses the design of the methods.

EVENT DESIGN FOR THE LUNAR LANDING SIMULATOR

The simulator needs an event for lunar gravity that will pull the LEM down toward the ground. It also needs events to provide LEM controls for the pilot, and an event to respond to a successful landing or a crash. The gravity event can call a method to pull the LEM downward while it is above the surface, and then call a method to test the landing once it hits the ground. The following events will be needed:

- A gravity event
- The LEM events

The following sections will discuss each in turn.

The Gravity Event

According to the specifications, lunar gravity constantly pulls an object downward, like gravity on earth, but at only 1.6 meters per second squared. This constant, 1.6 mps² is called the *acceleration due to lunar gravity*. Of course, once the LEM hits the ground it will stop moving downward. So, a gravity event needs to move the LEM downward and increase its speed by 1.62 mps for each second until the LEM hits the ground. Once it hits the ground, an event handler can test to see if it crashed.

The gravity event would work like this:

```
While the LEM is above the ground
  Begin: do nothing
  During: LEM fall
  End: test for successful landing
```

The LEM Control Events

The remaining events will provide LEM flight controls for the pilot. These are pretty straightforward and are as follows:

- The spacebar will be used to fire the main engine and provide an upward thrust that will slow the falling LEM by 1 mps. The fire main engine event should work like this:

When the *spacebar is typed*, decrease the LEM's downward speed by 1 mps

- The left and right arrows will turn the LEM left and right. The up and down arrows will move the LEM forward and backward. These are known as horizontal controls.
- The horizontal control events should work like this:

While the *left* arrow is pressed, turn to the left
 While the *right* arrow is pressed, turn to the right
 While the *up* arrow is pressed, move forward
 While the *down* arrow is pressed, move backward

It just so happens that Alice has a single built-in event type, shown in Figure 6, that lets the arrow keys move an object as specified in the pseudocode above.

FIGURE 6: The built-in Alice event to let the arrow keys move an object



The left and right arrow keys turn the object, while the up and down arrow keys move the object forward and backward—just what’s needed here. So, the horizontal controls can be implemented with just this one event, instead of four separate events.

METHOD DESIGN FOR THE LUNAR LANDING SIMULATOR

The gravity event will need some user-designed methods to serve as its event handlers. One method is needed to drop the LEM, and one method is needed to test for a successful landing. The specifications also call for the simulator to “*tell the pilot if the landing was a success or a*

failure, and whether it failed because the LEM was going too fast and crashed, or because it missed the target zone.” So, it looks like you will need the following methods for the LEM:

- The fall method
- The landing methods

The Fall Method

In each second, the fall method needs to increase the LEM’s downward speed by 1.62 mps and then drop the LEM the appropriate distance. The *descentSpeed* property can be used to keep track of the downward speed of the LEM. The method also needs to tell the pilot the LEM’s downward velocity, which is the value stored in the *descentSpeed* property.

The new velocity after each second is simply the old velocity plus 1.62. During each second, the average speed of the LEM, because it is accelerating constantly, is the average of the velocity at the start of the second and the velocity at the end of the second. Reducing the equation and applying this concept, you would find that a simple calculation for the new altitude is the old altitude minus the starting velocity plus .81.

You are not expected to be able to figure this out on your own, nor is it necessary to do so to create the software. The outside experts or clients working with the software developers on a professional project, such as NASA physicists and engineers working on a project like this, would provide the necessary equations. However, a student who has finished a good high school physics course should be able to figure out the equations needed here or find them in a physics book. This shows us two things, first that good software developers need to be able to work with people who have other expertise to create quality software, and second, if you are interested in things like lunar landing simulators, you should probably take a few courses in physics, higher mathematics, or engineering. College students majoring in computer programming or software engineering often choose elective courses, such as science, engineering, business, graphic arts, or entertainment technology, that will better prepare them to understand the business or profession in which they would like to work as a software professional.

Written in pseudocode, the fall method would look like this:

```

Move LEM down [descentSpeed], with a duration of 1 sec, and an
abrupt style
Set new descentSpeed to the old descentSpeed + 1.62 mps
Set new altitude to the old altitude - (descentSpeed +.81)
Display the LEM's new altitude and descentSpeed.

```

The last item in the preceding list of instructions involves creating a message and then displaying the message on the screen. The user will see the term “velocity” instead of

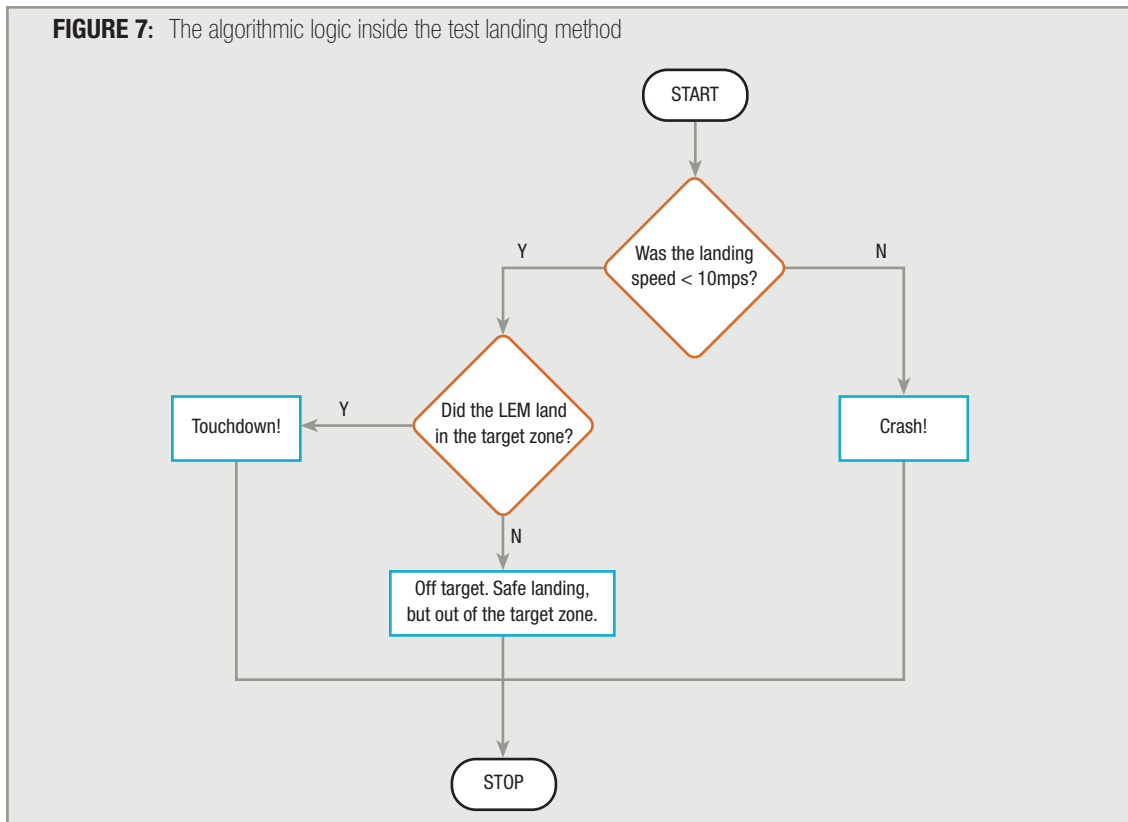
`descentSpeed`, because the `descentSpeed` property stores the craft's downward velocity. We'll make this a separate event, called *display*, which would look like this in pseudocode:

```
Build a string to display "velocity = " descentSpeed
Build a string to display "altitude = " altitude
Display a message that combines the descentSpeed and
altitude strings
```

Building the strings is a bit tedious, so this will probably be one of the more time-consuming methods to develop for this project, and demands a bit of patience on the part of the programmer, especially if the programmer's most common experience with a mouse is playing a video game, because every drop and drag interface does not require the quick reactions of a video game.

The Landing Methods

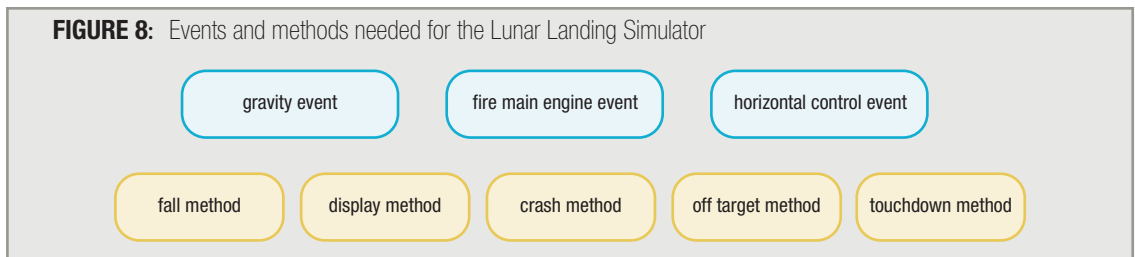
The test landing method will be activated when the LEM hits the ground. It will check to see whether or not the landing was a success, and will call the appropriate method in response to the quality of the landing. Figure 7 shows the logic in the test landing method, and how this method calls the other landing methods.



Consider whether the touchdown, crash, and off target methods really need to be separate methods. If they are very short, their code could be placed inside the test landing method, but creating them as separate methods is good modular design. It will make it much easier to expand these methods later, and has all the benefits of modular design discussed in Chapter 2.

For now, the touchdown, crash, and off target methods are each simple. They will each display the appropriate messages: “Touchdown!”, “Crash!”, or “Off target. Safe landing, but out of the target zone.” In addition, the off target method should show the distance from the center of the target zone.

Figure 8 shows the events and methods that need to be created. As indicated earlier, you’ll create the methods first, then create the events that call those methods.



TUTORIAL C—CODING THE LUNAR LANDING SIMULATOR

Now that your design seems complete, you can start creating the necessary methods and events. As discussed in Tutorial B, the methods should be created before the events that will use those methods.

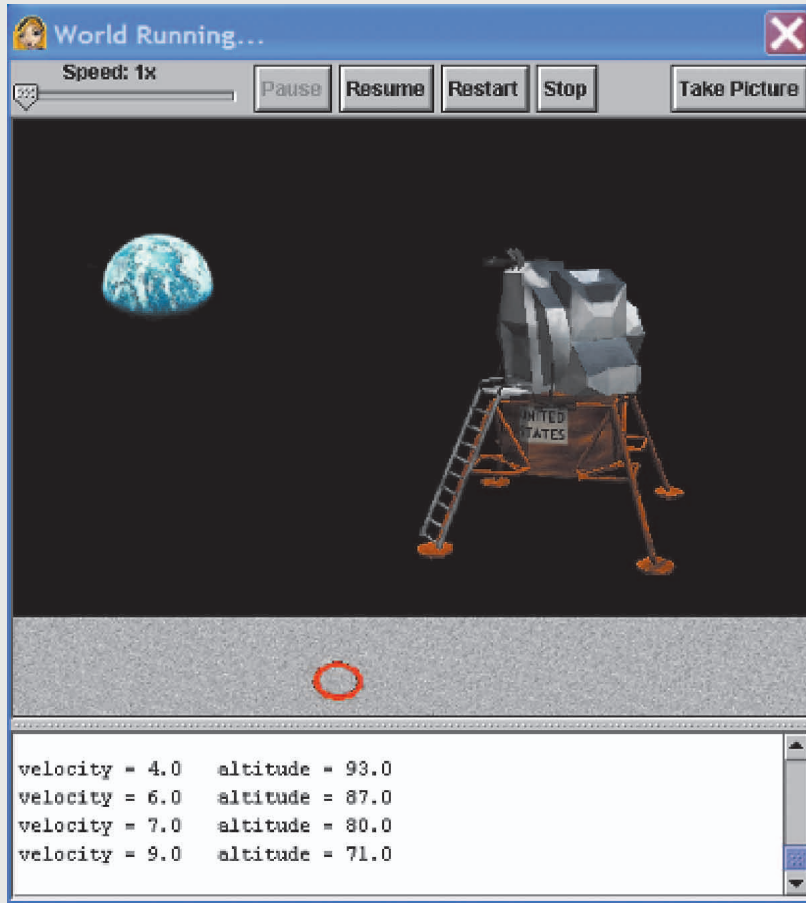
CODING THE SIMULATOR’S METHODS

The LEM.fall method calls the LEM.display method, so you need to have a LEM.display method before you can use it in the LEM.fall method. It makes sense to create the LEM.display method first. This will be the toughest method to create because of the commands needed to create the messages as text strings from numeric data.

Coding the LEM.display Method

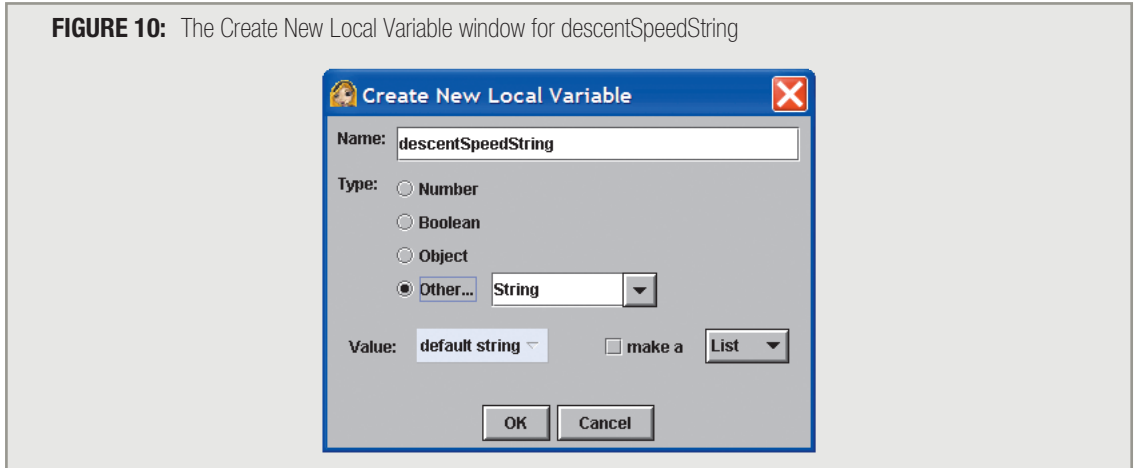
The LEM.display method will assemble a text string to display the LEM’s altitude and descentSpeed. You can use the print instruction to display the information in a text window at the bottom of the running world window, as shown in Figure 9. This method needs to build a text string showing the altitude, build a text string showing the descentSpeed, and then display the two together.

FIGURE 9: The LEM's altitude and descentSpeed are displayed



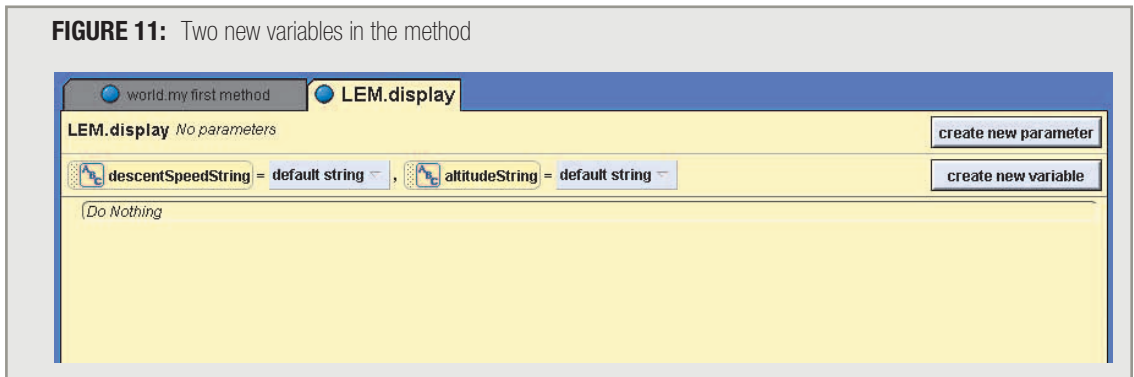
1. If the Lunar Landing Simulator world is not open, then open it now.
2. Click the **LEM** tile in the Object tree and then the **methods** tab in the details area.
3. Click the **create new method** button, type the name **display** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
4. The method needs two string variables to hold the two parts of the message: a `descentSpeed` string and an `altitude` string. Start with the `descentSpeed` string. Click the **create new variable** button on the right side of the `display` method in the Editor area.
5. When a Create New Local Variable window appears, enter `descentSpeedString` as the name and choose **other** and **string**, as shown in Figure 10. Click **OK**.

FIGURE 10: The Create New Local Variable window for descentSpeedString



- Now you can add the altitude string variable. Click the **create new variable** button on the right side of the method in the Editor area. When the Create New Local Variable dialog box appears this time, enter **altitudeString** as the name, choose **other** and **string**, and then click **OK**. You should be able to see tiles for the two new variables in the method, as shown in Figure 11.

FIGURE 11: Two new variables in the method

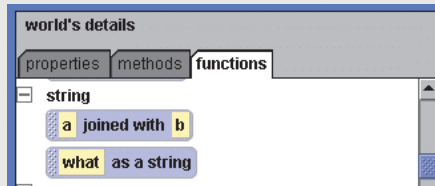


The method now has the variables it needs, but three instructions are needed. The first assembles the `descentSpeedString`, the second assembles the `altitudeString`, and the third displays them together. These instructions will need to concatenate strings. To **concatenate** two strings means to join them together to form one longer string. For example, the strings “Rumple” and “stiltskin” can be concatenated to form “Rumplestiltskin”. Alice has a string function of the form `[a joined with b]` on the world’s functions tab that can do this.

You will also need to convert a numeric value to a string. Alice also has a string function on the world’s functions tab to do this. Both of these string functions are shown in Figure 12.

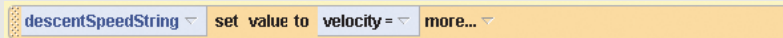
You will use these in creating this method. This is one of the more detailed parts of this tutorial, so take your time and be careful. If you run into trouble, back up and try again.

FIGURE 12: String functions on the world's functions tab



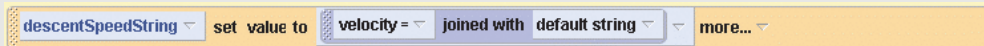
1. There is a *descentSpeedString* variable tile near the top of the method. Drag and drop a copy of it in place of *do nothing further* down in the method. When the value menu appears, choose other and then type “**velocity =**” as the value. Be sure to include the blank spaces before and after the equal sign, but do not type the quotes. Click the **OK** button. The instruction should look Figure 13.

FIGURE 13: Assigning a value to *descentSpeedString*

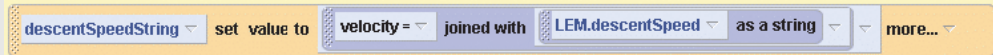


2. Now you need to add the concatenation function to the instruction. Click the **world** tile in the Object tree, then the **functions** tab, and drag and drop **a joined with b** into the *descentSpeed set value to [velocity =]* tile in place of *[velocity =]*. Choose **default string** as the value for *b*. The instruction should now look like Figure 14.

FIGURE 14: Assigning a value to *descentSpeedString*



3. Next you need to put a string with *descentSpeed* in place of the default string. However, *descentSpeed* is a number variable that needs to be converted to a string, so start by dragging a copy of the world-level function **[what] as a string** into the method in place of *[the default string]*.
4. When the menu appears, select **expressions** and then **LEM.descentSpeed** as the value. Your instruction should now look like Figure 15.

FIGURE 15: A further modification to the *descentSpeedString* instruction

5. Finally, it would be good to round off the value of *descentSpeed*; otherwise, the computer will display too many digits on the screen. Find the **round** function on the world's functions tab and drag and drop it into the instruction in place of *[LEM.descentSpeed]*. Choose **expressions** and **LEM.descentSpeed** as the value to be rounded. Your completed expression should now look like Figure 16.

FIGURE 16: The finished *descentSpeedString* assignment instruction

6. If your new instruction matches Figure 16, then this would be a good time to save the method with the same name as before. If your new instruction does not match Figure 16, then you can delete it and try the set of steps again before proceeding.

The method also needs an instruction to build the altitude string. The easiest way to create this will be to copy the instruction you just created, and modify it, as shown in Figure 17. You will create this instruction in the following step sequence.

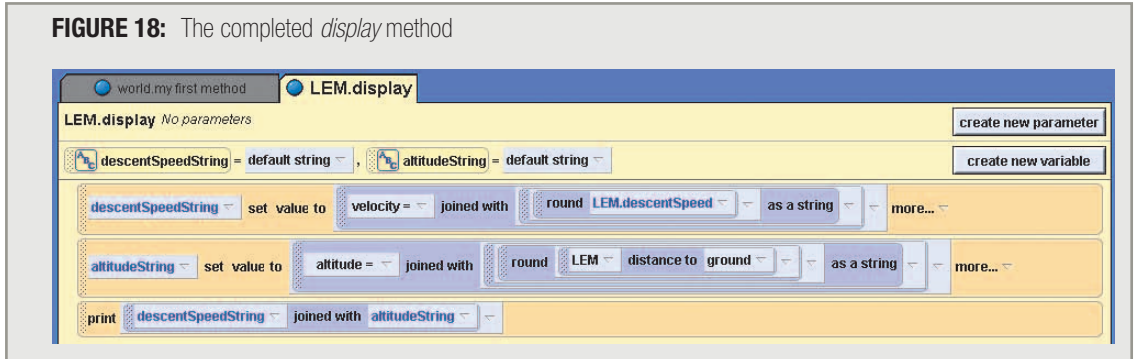
FIGURE 17: The *altitudeString* assignment instruction

1. Right-click the **descentSpeedString set value** tile and select **make copy** from the short menu that appears. Make sure that you click the tile and not one of the parameters in the tile. A copy of the instruction will appear just below the original.
2. Now you can modify the parameters in the copy to turn it into an *altitudeString set value* instruction. Click the first parameter, **descentSpeedString**, and select **LEM.display.altitudeString** from the menu that appears. This will replace *descentSpeedString* with *altitudeString*.

3. Next, click the second parameter, [**velocity =**], select **other** from the menu that appears, and then enter “ **altitude =** ”. Be sure to type two blank spaces before the word altitude, as well as blank spaces before and after the equals sign.
4. The last parameter will be a little harder to change because there is no variable for altitude like there is for *descentSpeed*. You need to put the *LEM distance above ground* function in place of *descentSpeed* inside the *round* function. Click the **LEM** tile in the Object tree, and then the **functions** tab in the details area.
5. Find the *LEM distance above* function and drag and drop it in place of *LEM.descentSpeed* inside the *round* function. Choose **ground** from the object menu that appears. Your instruction should now look like Figure 17.

The last instruction needed for this method is the instruction to display the two strings together. You will use the print instruction to do this.

1. Drag and drop a copy of the **print** instruction tile from the bottom of the Editor area to a spot just below the last instruction in the method. When the menu appears, select **text string ...**, and then type **XXX**. This is just a placeholder until the next step.
2. Next, you need to put a copy of the *[a] joined with [b]* string function in the print instruction tile in place of **XXX**. Click the **world** tile in the Object tree, and then the **functions** tab in the details area. Find the **[a] joined with [b]** function and drag and drop a copy of it into the print instruction in place of **XXX**. When the menu appears, choose **altitudeString** as the *b* parameter.
3. The instruction still has **XXX** as the first parameter for the join function. Drag a copy of the **descentSpeedString** tile and drop it in place of **XXX**. The print instruction and the method are now complete. The method should look like Figure 18, and the print instruction should look like the last instruction in the method.

FIGURE 18: The completed *display* method

Coding the LEM.fall Method

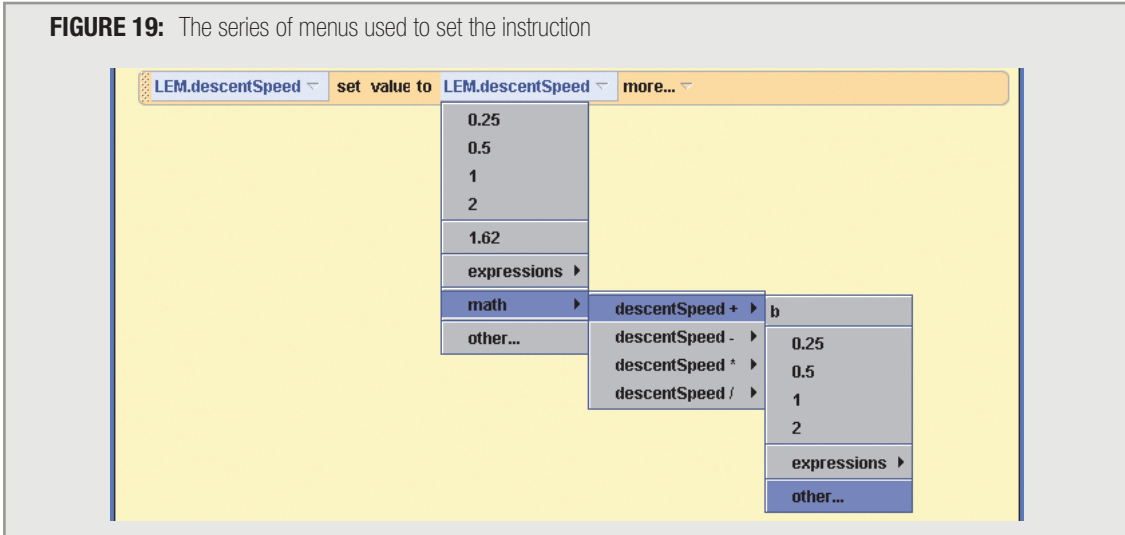
Now that the LEM.display method exists, you can use it to create the LEM.fall method. Based on the design that has been established in this chapter, the LEM.fall method should work like this:

```
Move LEM down [descentSpeed], duration = 1 sec, style = abrupt
Set new descentSpeed to descentSpeed + 1.62
LEM.display
```

1. Click the **LEM** tile in the Object tree and then the **methods** tab in the details area.
2. Click the **create new method** button, type the name **fall** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
3. You need to add the instruction to move the LEM downward. Drag and drop the **LEM move** tile into the new method in place of *do nothing*. For the direction, select **down**; for the amount, select **expressions**, and then **LEM.descentSpeed**.
4. To complete coding the instruction, click **more** and make sure the *duration* is **1 second**, then click **more** again, and make sure the *style* is **abruptly**.
5. Next, you need to add the instruction to increase the *descentSpeed*. Click the **properties** tab in the details area, then drag and drop the **descentSpeed** tile into the *LEM.fall* method below the first instruction. A *set value* menu will appear. Select **expressions**, and then **LEM.descentSpeed** itself as the value.
6. The instruction now says *LEM.DescentSpeed set value to LEM.DescentSpeed*, but it should be *LEM.descentSpeed set value to LEM.descentSpeed + 1.62*. Click the second occurrence of **LEM.descentSpeed**. Select **math** from the menu that appears, then **LEM.descentSpeed +**, then **other**, enter the value

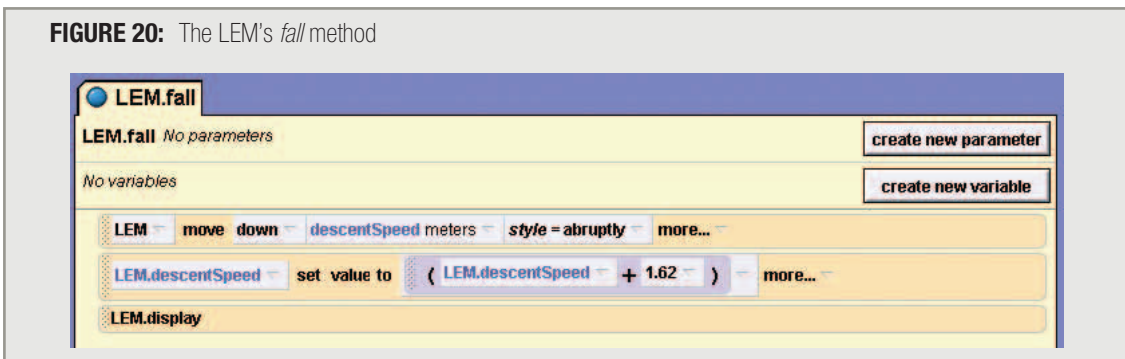
1.62, and then click **Okay**. Figure 19 shows what this sequence of menu selections should look like. When you are done, the instruction should say *LEM.descentSpeed set value to LEM.descentSpeed + 1.62*.

FIGURE 19: The series of menus used to set the instruction



7. Finally, you can add the display instruction to the method. Click the **LEM** tile in the Object tree, and then the **methods** tab in the details area. Drag and drop a copy of the **LEM.display** method tile into the *fall* method below the other instructions. Your method should now look like Figure 20.

FIGURE 20: The LEM's *fall* method



Coding the Landing Methods

The last methods that need to be coded are the landing methods. The `LEM.testlanding` method will call each of the other three methods and will contain logic like that shown in the flowchart back in Figure 7. For now, the `LEM.touchdown`, `LEM.crash`, and `LEM.offTarget`

methods will just display a message and some information. These methods will be easier to create than the string handling method you just finished. Because the `LEM.testLanding` method uses the other three, it makes sense to create the `LEM.touchdown`, `LEM.crash`, and `LEM.offTarget` methods first.

1. Click the **LEM** tile in the Object tree and the **methods** tab in the details area.
2. Click the **create new method** button, type the name **touchdown** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
3. Drag a copy of the **print** instruction tile from the bottom of the Editor area and drop it into the new method in place of *do nothing*. Select **text string ...** from the short menu that appears, and enter **Touchdown!**
4. Drag another copy of the **print** instruction tile from the bottom of the Editor area and drop it into the new method below the *[print Touchdown!]* tile. Again, select **text string ...** from the short menu that appears, and this time enter **Congratulations! Successful landing in the target zone**, and then click **Okay**.

The `LEM.touchdown` method is now complete. Now you can create the `LEM.crash` method.

1. Click the **LEM** tile in the Object tree and the **methods** tab in the details area.
2. Click the **create new method** button, type the name **crash** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
3. Drag a copy of the **print** instruction tile from the bottom of the Editor area and drop it into the new method in place of *do nothing*. Select **text string ...** from the short menu that appears, enter **Crash!**, and then click **Okay**.

That's all this method will do for now. Once the simulation works, you might want to come back to this method to add some action to the crash landing. Next, you can create the `LEM.offTarget` method.

1. Click the **LEM** tile in the Object tree and the **methods** tab in the details area.
2. Click the **create new method** button, type the name **offTarget** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
3. Drag a copy of the **print** instruction tile from the bottom of the Editor area and drop it into the new method in place of *do nothing*. Select **text string ...** from the short menu that appears, enter **Off target!**, and then click **Okay**.

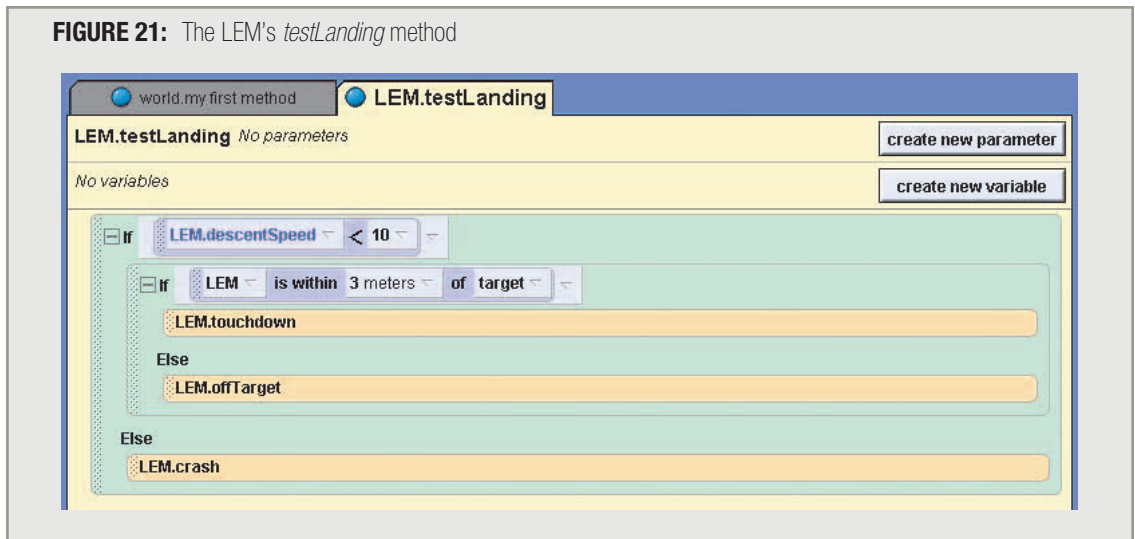
4. The method needs an instruction to tell the pilot how far off target. This will be done with two separate print instructions. First, right-click the **[print Off target!]** instruction tile and select make a copy. Then click **[Off Target!]** in the bottom copy and change the text string to **Distance to center of target zone:**.
5. Now make a copy of the **print [Distance to center of target zone:]** tile. This will be modified to print the actual distance, which is a number that needs to be converted into a string. Click the **world** tile in the *object tree*, and then the **functions** tab in the editor area. Drag a copy of the **[what] as a string** function and drop it in the bottom *print* instruction tile in the *LEM.offTarget* method in place of *[Distance to center of target zone:]*. Select **target** from the menu that appears.
6. Now you can modify this instruction to display the distance from the LEM to the target. Click the **LEM** tile in the Object tree and then the **functions** tab in the details area. Drag a copy of the **LEM distance to** function tile and drop it in the *print [target] as a string* instruction tile in place of the parameter *target*. Select **target** from the menu that appears, and the instruction is complete.

The last method that needs to be created is the `LEM.testLanding` method, which will include the logic to determine which message should be displayed when the LEM hits the ground. The logic, shown back in Figure 7, calls for a nested If/Else structure.

1. Click the **LEM** tile in the Object tree and the **methods** tab in the details area.
2. Click the **create new method** button, type the name **testLanding** in the dialog box that appears, and then click the **OK** button. The new method will appear in the Editor area.
3. Drag a copy of the **IF/Else** tile from the bottom of the Editor area and drop it in the method in place of *do nothing*. Select **true** from the menu that appears.
4. Click the **LEM** tile in the Object tree and then the **properties** tab. Drag and drop a copy of the LEM's **descentSpeed** property into the *If/Else* command in place of *true*. When the menu appears, select **descentSpeed < , then other**, enter the number **10**, and then click **Okay**.
5. Next, click the **methods** tab to see the LEM's methods. Drag a copy of the **crash** method tile and drop it into the *If/Else* instruction tile in place of *Do Nothing* below *Else*.
6. Now drag a second copy of the **If/Else** tile from the bottom of the Editor area and drop it the existing *If/Else* instruction in place of *Do Nothing*

between *If* and *Else*. Select **true** from the menu that appears. This will create a set of nested if commands like those in Figure 21.

7. You need to create the condition for the inner if command. The LEM needs to be within 3 meters of the center of the target zone. Click the **functions** tab in the details area and then drag a copy of the **LEM is within threshold of object** function tile into the instruction in place of *true*. When the menu appears, select **1 meter** as the *threshold* and **target** as the *object*.
8. You must replace *1 meter* with 3 meters. Click the **[1 meter]** parameter and change it to **3 meters**. The LEM will be off target if it is more than 3 meters away from the center of the target zone.
9. Now click the **methods** tab, and drag and drop a copy of the LEM's **touchdown** method into the inner *If/Else* tile in place of *Do Nothing* between *If* and *Else*.
10. Finally, drag and drop a copy of the LEM's **offTarget** method into the inner *If/Else* tile in place of *Do Nothing* below *Else*. The method is now complete, and should look like Figure 21.



CODING THE SIMULATOR'S EVENTS

Now that all of the required methods are in place, we can create the events for which they will be event handlers. According to the revised specifications, as summarized back in Figure 8, the following three methods are needed:

- The gravity event, as follows:

```
While LEM is above the ground
  Begin: Nothing
  During: LEM.fall
  End: LEM.testLanding
```

- The fire main engine event, as follows:

```
When the spacebar is typed, descentSpeed = descentSpeed -1
```

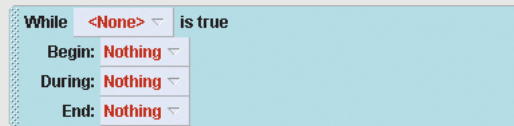
- The horizontal controls event, as follows:

```
Let the arrow keys move the LEM
```

To create the gravity event:

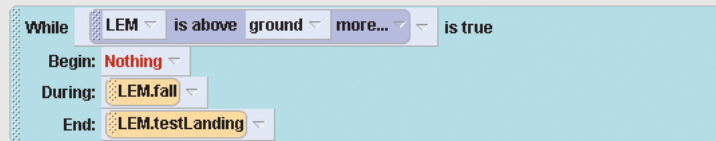
1. Click the **create new event** button in the Events area, and then select **While something is true** from the menu that appears. You should see a new blank method similar to Figure 22.

FIGURE 22: A blank *While something is true* method



2. You need to replace *<None>* with the condition *LEM is above the ground*. Click the **LEM** tile in the Object tree and then the **functions** tab. Scroll down through the functions and drag and drop the **LEM is above** function to replace the word *<None>*. Select **ground** from the menu that appears.
3. Next, you need to put copies of three different method tiles in the new event following *Begin:*, *During:*, and *End:*. Select the **LEM** tile in the *object tree* and the **methods** tab in the *details area*. Drag a copy of the **fall** method tile from the details area and drop it after *During:*.
4. Finally, drag a copy of the **testLanding** method and drop it after *End:*. Your new event is complete and should look like Figure 23.

FIGURE 23: The finished event



Next, you need to create the fire main engine event.

1. Click the **create new event** button in the Events area, and select **When a key is typed** from the menu that appears.
2. Click the **any key** parameter in the new event tile and select **Space** from the menu that appears.
3. Select the **LEM** tile in the *object tree* and the **properties** tab in the *details area*. Drag a copy of the **descentSpeed** tile and drop it in the new event in place of *Nothing*. When the menu appears, select **decrement LEM.descentSpeed by 1**. This will perform the same instruction as $descentSpeed = descentSpeed - 1$. The event should now look like Figure 24.

FIGURE 24: The fire main engine event



Now the horizontal controls event needs to be created.

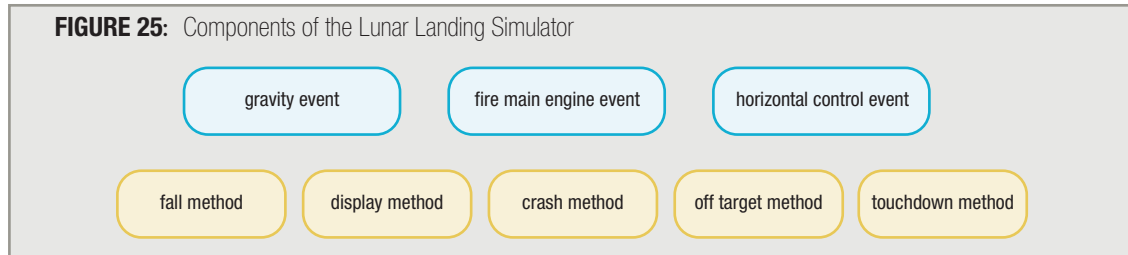
1. Click the **create new event** button in the Events area, then select **Let the arrow keys move <subject>** from the menu that appears.
2. Click the **<camera>** parameter in the new event tile and select **LEM**, then **the entire LEM**, from the menu that appears.

The code for the Lunar Landing Simulator is now complete. Make sure to save your world and remember where you save it and what name you use. You may want to save a backup copy in another location. It is estimated that the finished world should be between 1 and 2 Mb in length.

TESTING THE SIMULATOR

A programmer who finishes coding the software is like an author who has finished writing a first draft of a document. It's now time to test the software and find out if any modifications are necessary. The primary purpose of testing is to make sure that the software functions according to its specifications, but you also need to watch for unexpected problems, such as the side effects mentioned in Chapter 2, and for awkward things that could be slightly improved, such as a camera angle in a particular scene, or the way in which a particular button works.

Figure 25 shows the software components contained in the Lunar Landing Simulator. There are three events and four methods that need to be tested. You saw in Chapter 2 that this involves unit tests, in which each software module is tested by itself to see if it performs its function properly, and integration tests to see if a module works when included in a software package with other modules.



In a professional software development project, the client with outside expertise would help develop a testing plan to see if the software is functioning in a technically correct manner. For example, the client would determine if the gravity event was properly simulating lunar gravity. Further, engineers would check the math in the simulation software and time the activities of the simulation to see that everything is working correctly. Of course, each of the controls, such as the spacebar to fire the main engine, and the arrow keys to fire the horizontal control thrusters would be checked to see if they are working properly.

To simulate such a testing environment, you will try the simulator to see if it works.

1. Play the world and first look to see if the velocity is increasing and the altitude is decreasing while the LEM falls. When the LEM hits the ground, the *Crash!* message should be displayed.
2. Next, play the world again and try to slow the LEM's descent by using the spacebar to fire the main engine. Do not use the arrow keys. See if you can land with a velocity of less than 10 mps. You might need to try several times. If you thrust the engine too much, the LEM will start to move upward, with a negative descent velocity. When you land softly enough, the *Off target!* message should be displayed rather than the *Crash!* method.
3. Finally, play the world again and try to steer the LEM and move it forward and backward while it is falling by using the arrow keys. The camera's position is locked to the LEM, so as you move and turn, the LEM will appear stationary and the world and target area will appear to move. If you can use the arrows to control the horizontal thrusters so that you land in the target zone, and use the main engine so that you are going less than 10 mps when you hit the ground, then you will get the *Congratulations! Successful landing . . .* message.