After finishing this chapter, you should be able to:

- Provide brief definitions of the following terms: CamelCase, coding, encapsulated methods, integration test, method header, modular development, off-camera, parameter, primitive methods, program development cycle, reusable code, test for correctness, testing shell, top-down design, unit test, user-defined methods, and variable.

- Describe the processes known as top-down design and modular development, including why they are often used together, how organizational charts are involved, and some of the advantages of modular development of new software.

- List and describe the steps in a simple program development cycle.

- Describe the difference between primitive methods and user-created methods in Alice, and show how to run primitive methods to directly position objects and manipulate the camera in an Alice world.

- Create new methods in Alice in a manner that demonstrates good top-down design and modular development.

- Create and demonstrate the use of generic methods in Alice that contain method parameters.
TOP-DOWN DESIGN AND MODULAR DEVELOPMENT

The process of developing methods for objects is mostly a process of developing algorithms because each method is an algorithm. Algorithm development has a long history in the world of mathematics, in which algorithms are step-by-step solutions to well-defined mathematical problems, such as finding the least common denominator of two fractions. Traditional approaches to mathematical problem solving have been applied to many different disciplines that rely on mathematics, such as engineering, the physical and social sciences, and computer programming. Generally, a problem must be clearly specified, so that once a solution has been developed, it can be compared to the specifications to determine its correctness.

It's easier to solve smaller problems than it is to solve big ones, so good mathematicians often start solving a problem by trying to break a large algorithm into parts that are more manageable. This process of decomposition is sometimes called a *divide and conquer* approach to problem solving, in which a problem is broken into parts, those parts are solved individually, and then the smaller solutions are assembled into a big solution. Computer programmers do the same thing when developing new software.

This process is also known as *top-down design* or *top-down development*, starting at the top with one concept or big idea, and then breaking that down into several parts. If those parts can be further broken down, then the process continues. The end result is a collection of small solutions, called modules, which collectively contain the overall solution to the original problem.

Let's look at an example of top-down development. Imagine that you want to write a computer program to process the payroll for a large company. Your big idea is named “Payroll.” What would you need to do if you were going to process a payroll? Generally, you would need to get some data, perform some calculations, and output some results. Figure 2-1 shows Payroll broken down into these three modules.

![Figure 2-1: A payroll program broken down into three modules](image)

You can go further in decomposing the problem. There are two different sets of data—old data, such as a list of employees, and new data, such as the number of hours each person worked during a specific week. The old data will probably be read in from a data file on the hard disk drive, while the new data will be entered into the system for the first time,
so the Get Data module can be broken down further, into Get Old Data and Get New Data. If the Calculations and Output Results modules are broken down in a similar manner, and then the sub-modules of those modules are broken down, and so on, then the final design might look something like the one in Figure 2-2. Such a diagram is called an organizational chart.

Organizational charts show the overall structure of separate units that have been organized to form a single complex entity. The chart shows how the upper levels are broken down into the parts at the lower levels, and conversely, how the parts at the lower levels are layered together to form the upper levels. Organizational charts are most commonly seen describing hierarchies of organizations, such as government agencies. In this case, you can see how the specific modules farther down the chart are combined to form the overall payroll program. A programmer, or a team of programmers working together, can create each of the modules as software methods, and then put them together to build the entire payroll program.
The process of top-down design leads to **modular development**, in which the parts, or modules, that make up a complete solution to a problem are developed individually and then combined to form that complete solution. Modular development of computer software has the following advantages, especially for larger projects:

- Modular development makes a large project more manageable. Smaller and less complex tasks are easier to understand than larger ones and are less demanding of resources.

- Modular development is faster for large projects. You can have different people work on different modules, and then put their work together. This means that different modules can be developed at the same time, which speeds up the overall project.

- Modular development leads to a higher quality product. Programmers with knowledge and skills in a specific area, such as graphics, accounting, or data communications, can be assigned to the parts of the project that require those skills.

- Modular development makes it easier to find and correct errors in computer programs. Often, the hardest part of correcting an error in computer software is finding out exactly what is causing the error. Modular development makes it easier to isolate the part of the software that is causing trouble.

- Most importantly, modular development increases the reusability of your solutions. Solutions to smaller problems are more likely to be useful elsewhere than solutions to bigger problems.

The last point in the preceding list is one of the most important concepts in all of computer programming. Saving solutions to small problems so they can be applied elsewhere creates **reusable code**. In fact, most of the software on a computer system is filled with layers of reusable code because computer programs contain many small tasks that need to be repeated, such as getting a user’s name and password before running a program. Everything from the low-level parts of the operating system that directly control the hardware, to the most complex user applications is filled with layers of short programming modules that are constantly reused in different situations.

Reusable code makes programming easier because you only need to develop the solution to a problem once; then you can call up that code whenever you need it. You can also save the modules you have developed as part of one software development project, and then reuse them later as parts of other projects, modifying them if necessary to fit new situations. Over time, you can build libraries of software modules for different tasks.
Object-oriented programming, by its very nature, encourages such development of reusable code. Methods to perform simple tasks can be reused within an object to form more complicated methods, and entire classes of objects created for one project can be reused as needed for other projects.

Figure 2-3 shows part of the methods tab for the iceSkater from the lakeSkater world seen in Chapter 1. Methods to perform simple tasks, such as spin, blinkEyes, and jump are re-used as necessary to form the overall routine in this Alice world. The entire skater object itself can be reused in other worlds where different routines might be assembled from these simple methods. As you create Alice worlds, you should keep this concept in mind and try to write small reusable methods as much as possible. This approach to programming will serve you well in any language you use—Java, Visual Basic, C++, Python—and is one of the driving principles behind much of the modern object-oriented approach to programming. Good computer software is filled with reusable code, and Alice is no exception.

**FIGURE 2-3:** The methods tab for the iceSkater from the lakeSkater world
A PROGRAM DEVELOPMENT CYCLE

Traditional approaches to algorithm development, in which algorithms are developed and tested to see if they meet original specifications, led to the notion of a software development cycle in computer programming. The first step in the creation of software is to design the methods that will make up that software, that is, to plan what the methods will do. Once the methods have been designed, they can then be implemented.

The process of implementing a new method primarily involves entering instructions (and data) on the computer in a particular programming language. This process has traditionally been called coding. After each method has been coded, it needs to be tested to make sure that it works, and then edited as necessary. Whenever a method is edited, it needs to be compared to the original design to be sure that it still performs according to the original specifications. In all, programmers follow a program development cycle, in which they design, code, test and debug methods, as shown in Figure 2-4 and as discussed in the following sections of the chapter.

FIGURE 2-4: A simple program development cycle

DESIGNING METHODS

Many different techniques are used to design methods in new software. Top-down development and modular design have been described previously in the chapter. Other methods, such as the use of flowcharts, pseudo-code, and storyboards, will be discussed in subsequent chapters. For now, a good place to start would be to develop clear specifications for what the new software is supposed to do and to break the problem into smaller parts instead of trying to deal with individual methods that each performs many different actions.

CODING METHODS IN ALICE

One of the unique features of Alice is its drag-and-drop interface, in which tiles containing existing instructions and logic and control structures are dragged into place to create a new method. You should recall from Chapter 1 that tiles for existing methods can be found on
the methods tabs for various objects, while logic and control tiles can be found at the bottom of the Editor area, as shown in Figure 2-5.

In more traditional programming languages, such as Java, C++, or Visual Basic, the coding phase of software development includes translating a software design into a particular language, and then entering that language on the computer. Coding in Alice, however, is a little different. Dragging Alice’s language tiles into place to create new methods eliminates the need for carefully translating and typing individual instructions. Programmers still need to be familiar with the features of the language and how to use them, but they do not need to be so concerned with details like spelling and punctuation. There is still a need to be careful, but Alice programmers can focus their attention on the logic of how new methods will operate rather than the syntax of the language.

TESTING AND DEBUGGING ALICE METHODS

The techniques that professional software developers use to test newly developed methods can be varied and complex, but generally they involve answering a few simple questions, such as the following:

- Most importantly, does the new method do what it is supposed to do? This is known as a test for correctness. Tests for correctness measure whether the program meets the original specifications.
Is the method reasonably efficient? How much time does it take for the method to complete its task, and how much space does it use? If a program takes two hours to sort a list that another program can sort in two seconds, then it is not a very time-efficient program. Similarly, one program could use more or less memory and disk space than another program, but typically time efficiency is more critical than space efficiency as computer memory chips and disk drives become larger and less expensive.

Does the method have any undesirable side effects? For example, if a method to print a long document also erases the hard disk drive, then that would be considered an undesirable side effect. Side effects are related to how methods change stored data, how one program affects the performance of another program, how one method affects another method, or even how two instructions in the same method affect each other.

**Note** Here’s a common example of a side effect: Alice methods to move and turn objects at the same time can occasionally produce unexpected results if the programmer failed to consider that the object’s directions were changing as it turned.

Software testers need to be able to develop a testing plan that examines new software under all possible circumstances. If the program will perform operations on data sets, such as sorting a list, or performing a numerical analysis of the data, they need to develop sample data sets that test the program under all possible circumstances. This can be a rather difficult task. Software testing is a branch of software engineering in which some people specialize.

Software developers perform two general types of tests: unit tests and integration tests. A **unit test** checks to see if the method works as expected all by itself. An **integration test** checks to see if the method works when it is placed into a larger program in combination with other methods.

Sometimes a unit test can be performed by running a newly developed method all by itself, and sometimes it is necessary to create another method, called a **testing shell**. A testing shell is a short method that simulates the environment in which a newly developed method will be used. This is especially important if the new method will be receiving values from other methods, or passing values on to other methods. The testing shell can be written to pass known values into the method being tested, and then the output can be captured and examined for correctness.

Consider the method to calculate state tax in the payroll program mentioned previously. That method receives values as input and sends values as output. A testing shell for the method to calculate state tax would pass it some known values, and then capture the method’s output so it could be examined, as seem in Figure 2-6. By passing in known values and examining the output to see if it is correct, a software tester could determine if the method was performing the correct calculations.
In the lakeSkater demonstration that you have already seen, the iceSkater’s spin method could be tested without a testing shell. The method could be run all by itself, and the tester could observe the action to see if the method performs according to specifications.

Once a method has been unit tested and debugged, it can be integration tested. That is, it then can put in position in the overall program, which can then be run to see if it works with the new methods in place.

This chapter only scratches the surface of testing and debugging, which can be quite complicated and detailed. For now, just try to keep the following questions in mind:

- Does the method do what it supposed to do according to the specifications? Does it work by itself? Does it work as a part of a larger program?
- Is it reasonably efficient? Is there any indication that something is taking too long or using up too much memory?
- Is anything unexpected happening when the program runs?
- Has the program been tested under all possible circumstances? This is especially important if the program has numeric input. For example, what happens if you tell a method to move forward a negative amount? What happens if you tell a method to turn a negative number of revolutions?

If any problems are discovered when the software is tested, then the cause of the problem needs to be isolated. Here unit tests are most helpful, especially when software has been developed as many smaller modules, rather than as a few larger modules. Once you know what is causing the error, you can develop a plan for fixing the problem, modify the necessary method or methods, and then test again.
TUTORIAL 2A—WORKING WITH PRIMITIVE METHODS IN ALICE

Alice objects have two kinds of methods – primitive methods and user-defined methods. Primitive methods are built-in, predefined methods that are part of each Alice object. They provide simple basic behaviors, such as move, turn, and roll, and cannot be edited. User-defined methods are written by people who use Alice, and can be edited. Some of the objects in the Alice object galleries, such as the Penguin class of objects, have user-defined methods that were written by the people who created the object, in addition to the standard set of primitive methods. You can use both primitive methods and user-defined methods to write new methods of your own.

Let’s examine the methods for Penguin class objects:

1. Start Alice, click the Templates tab when the Welcome to Alice! dialog box appears, and begin an Alice world using the snow template.
2. Once the new world is open, click the large green ADD OBJECTS button in the lower-right corner of the World window to enter Scene Editor mode.
3. Now you can see the object galleries at the bottom of the screen. Click the Animals gallery, and then find and click the Class Penguin tile. Click the Add instance to world button.

You should be able to see the Information window for the Penguin class of objects, as shown in Figure 2-7. It contains information about the designers who modeled and painted the Penguin class, tells us how much memory is required for an instance of the object, and lists user-created methods that the designers included with the object. Most Alice objects don’t come with any user-created methods, only a standard set of primitive methods, but here we can see that the designers of the Penguin class included user-defined methods to give the penguin additional behaviors, such as flapping its wings, turning its head, and walking.
Let’s add the penguin to the new world and then look at its methods tab.

1. Click the **Add instance to world** button, and then click the **DONE** button. This will return you to the standard Alice interface.
2. Click the **penguin** tile in the Object tree and the **methods** tab in the Details area. You can now see the methods tab for your instance of the Penguin class of objects, which is shown in **Figure 2-8**.

**FIGURE 2-8:** The methods tab for the Penguin class of objects
Tiles for the penguin’s user-created methods appear above the *create new method* button and are each followed by an *edit* button. Notice that this set of methods matches the list in the Penguin’s Information window. The tiles below the *create new method* button have no *edit* button. These are the primitive methods that come with each Alice object. You cannot edit them. In other words, they are **encapsulated methods**—you can use them, but you cannot access their inner workings. Encapsulation will be discussed further in Chapter Five.

**RUNNING PRIMITIVE METHODS DIRECTLY**

Primitive methods may be used to build more complex methods, or they may be run directly. You can run a primitive method directly by right-clicking an object or an object tile in the object tree, and then selecting the method you wish to run from a list that appears.

One of the penguin’s primitive methods is the *turn to face* method. Let’s run this method directly to make the penguin face the camera.

To directly run a primitive method:

1. Right-click the *penguin* tile in the Object tree, and then click **methods** from the small menu that appears. Now you should see a list of the penguin’s primitive methods, similar to Figure 2-9.
2. One of the methods in the list is *penguin turn to face*. It’s about halfway down the menu. Point to **penguin turn to face**, and then click **camera** from the submenu that appears. On the screen you should see the penguin turn to face the camera.
This is an example of how you can run primitive methods directly, without saving them as part of a program in the Alice world. This is a good technique to use to help manipulate objects when setting the scene for an Alice world, or simply to experiment to learn how certain methods work.

**USING PRIMITIVE CAMERA METHODS TO FIND A LOST OBJECT**

Sometimes in Alice an object is off-camera and you’re not sure exactly where the object is. **Off-camera** means that an object cannot be seen on the screen with the camera in its current position. Often it can be difficult to find an off-camera object by manually manipulating the camera, and the object appears to be lost. There are several primitive methods that you can run directly to help find the lost object.

To find an object that is off-camera:

1. First, using the arrow controls at the bottom of the window world, manipulate the camera so that you cannot see the penguin anymore. Now the penguin is off-camera.
2. Now you can find the penguin. Right-click the **camera** tile in the Object tree, select **methods** on the menu that appears, then select **camera point at**. On the target menu that appears, select **penguin**, and then the entire penguin, as shown in **Figure 2-10**. On your screen you will see that the camera is now pointing at the penguin. Can you see how this might be a useful method to run directly?
3. Using the arrows, move the camera again so that the penguin is off-camera.
4. This time, run the camera get a good look at method on the entire penguin. Do you see how this is a little different than the camera point at method?
5. Try experimenting by running a few more of the penguin’s methods directly before moving on to the next tutorial. For example, can you determine the difference between the turn and roll methods by running them
directly and watching what happens? Some of the methods, such as `resize`, provide more control in manipulating objects than the object manipulation tools in the Scene Editor. If you run into trouble, you can recover with the Undo button, which is located near the upper-left corner of your screen.

6. There is no need to save this experimental world, so when you are finished, close the Alice software before continuing. You will open Alice again in the next tutorial, but often it is a good idea to close Alice and reopen it between worlds. It isn’t always necessary, but it could help to avoid side effects, such as memory overflows, which can sometimes happen when loading one large Alice world after another.

**TUTORIAL 2B – CREATING NEW METHODS IN ALICE**

In this tutorial, you will practice some of the techniques discussed earlier in the chapter to create an Alice world with several new methods. You will create a world with three objects from Lewis Carroll’s *Alice’s Adventures in Wonderland* – the Cheshire Cat, the White Rabbit, and Alice, and you will write methods to make them jump up and down. The Alice software really has no connection to Lewis Carroll’s story except for the name Alice, but it might be more interesting to use these characters in this exercise.

It’s easy to find objects for Alice and the Cheshire Cat in the object galleries, but you need to be careful with the White Rabbit – there are classes for Bunnies, Rabbits, and Hares in addition to the WhiteRabbit class. You want the White Rabbit with the waist coat and pocket watch on a chain, as described by Lewis Carroll in *Alice’s Adventure’s in Wonderland*.

---

Alice started to her feet, for it flashed across her mind that she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it . . .

—*Alice’s Adventures in Wonderland*

by Lewis Carroll, 1865
SETTING THE SCENE

The first step in creating a new world is to set the scene, by opening a template, adding necessary objects, and positioning things. In long programs, some naming convention is needed to make it easier to read the names of the many objects, methods, and variables used within the program. As you work with Alice, you will notice that the names of classes and objects in Alice follow a pattern called the CamelCase naming convention, which is used with most programming languages and is recommended by many software developers, such as Microsoft.

**CamelCase** is the practice of writing compound names without using blank spaces, but capitalizing the first letter of each name that forms the compound name, like the name CamelCase itself. In Lewis Carrol’s *Alice in Wonderland*, the proper name of the character is “Cheshire Cat,” while the name of the class of characters in the object gallery is “CheshireCat.” You will also notice that the very first letter of object class names begin with capital letters, whereas the names of instances of a class begin with lowercase letters. The class name is CheshireCat, with a capital “C” at the beginning; the first instance of the class is named cheshireCat with a lowercase “c.”

To set the scene for your new world:

1. Start the Alice software, click the **Template** tab, and select the **grass** template from the templates tab. Next, click the **ADD OBJECTS** button to enter Scene Editor mode.

2. You need to add the three characters to this Alice world and position them as seen in **Figure 2-11**. Start with the Cheshire Cat. Click **Local Gallery**, and then click **Animals**. Click the **cheshireCat** tile, and then click the **Add instance to the world** button. A cheshireCat should appear near the center of the world window and a cheshireCat tile should appear in the Object tree.
3. Place the cheshireCat on the left, as seen by the camera, and have him face the camera. Drag the cheshireCat over to the left side of the World window.

4. Next, you will make the cat turn to face the camera. Right-click the cheshireCat tile in the Object tree, and then point to methods from the small menu that appears. Now you should see a list of the cheshireCat’s primitive methods, similar to Figure 2-12.
5. About halfway down the list, find and point to the `cheshireCat turn to face` method, and then click `camera` as the target object. On the screen you should see the cheshireCat turn to face the camera.

6. The WhiteRabbit class is in the same animal gallery as the CheshireCat class. Note that you will have to scroll to the far right in the Animals gallery to see it. Add an instance of the WhiteRabbit class to your world.
7. After you have added an instance of the WhiteRabbit class to the world, you will need to position it on the right side of the world window, facing the camera, just as you positioned the cheshireCat on the left. To do this, drag the whiteRabbit over to the right, and then run the method to make it face the camera as you did with the cheshireCat.

8. Finally, add an instance of the AliceLiddell class to your world. Leave Alice in the center of the scene, and make her turn to face the camera. You are finished setting the scene, so exit the Scene Editor Mode by clicking the large green DONE button.

DESIGNING A METHOD

The design of a new method begins with specifications. In the simple case in this tutorial, the specifications are given: the three characters will first appear on a grass background, and then they will jump. Alice jumps, the whiteRabbit jumps, and then the cheshireCat jumps. Alice’s programming language has no primitive instruction to make an object jump up and down, so you will need to use the move up and move down methods to make each character jump. Figure 2-13 contains the organizational chart for the program.

![Figure 2-13: An organizational chart for the new jumping program](image)

The outline for the program would resemble Figure 2-14.

![Figure 2-14: A first outline for the jumping program](image)

**Triple Jump Program**
1. whiteRabbit jump
2. cheshireCat jump
3. aliceLidell jump
And finally, an expanded outline for the program would resemble Figure 2-15.

| FIGURE 2-15: An expanded outline for the jumping program showing more details |
|--------------------------------|------------------|------------------|
| **Triple Jump Program**      | **1. whiteRabbit jump** | **2. cheshireCat jump** |
|                              | a. whiteRabbit move up | a. cheshireCat move up     |
|                              | b. whiteRabbit move down | b. cheshireCat move down |
|                              | **3. aliceLidell jump** |                     |
|                              | a. aliceLidell move up |                     |
|                              | b. aliceLidell move down |                     |

Top-down design and modular development suggest that separate jump methods should be written for each object rather than placing all of the code into a single method. By using proper programming techniques as you begin to write short programs like this one, you will develop good habits that will help you when you write larger programs, whether in Alice, Java, Visual Basic.NET, or some other language.

**CODING THE JUMP METHODS**

To make an object jump, you are simply going to move the character up one meter and then move the character down one meter.

To make the object jump:

1. Let’s start by creating a jump method for the whiteRabbit. Click the **whiteRabbit** tile in the Object tree, and then the methods tab in the Details area.

2. Click the create new method button, and a small New Method dialog box will appear asking you to name the method. Type **jump** as the name for the new method, and then click the OK button. In the Editor area, a new tab appears for a blank method with the name **whiteRabbit.jump**, as seen in Figure 2-16.
The new blank method in the Editor area has several components. You see the name of the method on a tab at the top of the method, and again on the method’s first line of text, followed by the phrase *No parameters*. To the right of this you see a *create new parameter* button. On the next line of text you see the phrase *No variables*, with a *create new variable* button on the right. Your method will not have any variables or parameters, so for the moment, you can ignore these buttons. (You will learn more about variables and parameters in the next part of this chapter.) The lines of text at the top of the method can be considered a **header** for a method, which gives us information about how the method works. From the header information, you can see that you can run this method without any parameters, and that there are no variables in the method.

You can see a box with rounded corners containing the phrase *Do Nothing* below the header for the method. This part of your method is sometimes called the **instruction zone**. Right now there are no instructions in the method, so the zone has the text *Do Nothing*.

You now want your method to include an instruction to make the whiteRabbit move up one meter followed by an instruction to make the whiteRabbit move down one meter, as shown by the specifications in Figure 2-17.

**FIGURE 2-16:** A new method tab with the name *whiteRabbit.jump*

**FIGURE 2-17:** Specifications for a generic jump method

<table>
<thead>
<tr>
<th>Jump <em>(who)</em> is passed in as an object parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(who)</em> moves up 1 meter</td>
</tr>
<tr>
<td><em>(who)</em> moves down 1 meter</td>
</tr>
</tbody>
</table>
To make the method move the whiteRabbit as desired:

1. Drag the **whiteRabbit move** tile from the left side of the screen and drop it on the *Do Nothing* phrase. The tile will be surrounded by a red border as you drag it. The border will turn green when you reach the right place.

2. When you drop the tile into the instruction zone, a menu will appear asking you the direction in which you want to move the object. Select **up** for the direction, and then select **1 meter** for the amount. Direction and amount are two parameters, or pieces of information, that you must give a method whenever you wish to use the method.

3. Next, drag another copy of the **whiteRabbit move** tile from the left into the instruction zone below your first instruction. This time set the direction to **down**, and the distance again to **1 meter**.

You are now finished with your method to make the whiteRabbit jump. Now you need to create a **jump** method for the cheshireCat, just as you did for the whiteRabbit.

To create the required **jump** method for the cheshireCat:

1. Select the **cheshireCat** tile in the Object tree, and then click the **methods** tab in the Details area.

2. Click the **create a new method** button, and then name the method **jump**.

3. Add an instruction to make the cheshireCat move up **1 meter**.

4. Add an instruction to make the cheshireCat move down **1 meter**.

5. Create a **jump** method for aliceLiddell in a similar manner. The aliceLiddell object can be found in the People gallery. Alice Liddell was the young daughter of a friend of Lewis Carroll’s for whom the story *Alice’s Adventures in Wonderland* was named.

**CREATING THE MAIN METHOD**

Now you need a method that calls all three jump methods in the order in which you want the characters to jump. This method will be your main method, at the top of the organizational chart. *World.my first method* is often used as this top-level method, which is what you will do here. This is a world-level method, associated with the world itself rather than with any particular object in the Alice world.

To create the required top-level method:

1. Start by clicking the **world** tile in the Object tree, and then click the **methods** tab, if necessary, to see the methods in the Details area. Things look a little different here than when you looked at an object’s methods. There are no
primitive methods listed below the create new method button, but my first method already exists as a user-created method.

2. Click the edit button. This will open up the method for editing in the Editor area. It begins as a blank method with no parameters, variables, or instructions. Your plan calls for aliceLiddell to jump, followed by the whiteRabbit, and then the cheshireCat.

3. First you want aliceLiddell to jump. Click the aliceLiddell tile in the Object tree, and then click the methods tab, if necessary, in the Details area. Note that there will be a tile for the jump method above the create new method button because this is a user-created method that can be edited.

4. Click the jump tile and drag a copy of it into the instruction zone of world.my first method.

5. Next you need to add a whiteRabbit.jump method. Click the whiteRabbit in the Object tree and drag and drop the whiteRabbit’s jump method into world.my first method. Make sure that you drop it just below the whiteRabbit.jump instructions in the desired sequence.

6. Finally, you need to add the cheshireCat.jump method by clicking the cheshireCat in the Object tree and then dragging and dropping its jump method below the whiteRabbit.jump instruction. Your screen should resemble Figure 2-18.

TESTING (AND DEBUGGING) YOUR FINISHED SOFTWARE

If you have done everything correctly, then your Alice world should be ready to go. Let’s find out.

To test your world:

1. Click the Play button to play the Alice world
2. Watch what happens. Did it perform according to your original specifications? Did Alice, then the whiteRabbit, and then the cheshireCat, each jump up and down? If not, then you need to look at each of your methods, find the error, and fix it.

3. Before you move on to the next tutorial, you should save your Alice world as triple jump. (You can do this through the File menu, just as you can with most Windows programs.) Remember to pay attention to where the file is saved, as discussed in Chapter 1.

You can think of each method as performing a single task. If an object is doing something wrong, then you should ask which of your methods performs that task. If you have an error, then it could be in either the main method or one of the jump methods. If the three objects are jumping in the wrong order, then the main method is probably where you have an error. If one of the objects is not jumping properly, then that object’s jump method is where you can expect the error to be. By writing small methods for each individual task, you not only have all the benefits of modularity listed earlier, but there is another benefit—it is much easier to debug software that is written as small modules, each performing an individual task.

TUTORIAL 2C – GENERIC METHODS AND PARAMETERS

In the previous tutorial, you created an Alice world with specific methods to make each of your three objects jump. You created an aliceLiddell.jump method, a whiteRabbit.jump method, and a cheshireCat.jump method. In this part of the chapter, you will create a generic jump method that will work for any object.

Your new method will have a variable parameter. A variable is a name for a memory location that temporarily stores a value while a method is running. Variables are a lot like the properties of objects. They have data types, just like properties, and they are stored in the memory of the computer, just like properties. However, properties are associated with an object, and their values are maintained as long as the object exists, whereas variables are associated with a particular method, and their values exist only inside the method. Once the method stops running, their values are gone, unless, of course, you had saved them somewhere else while the method was running.

A parameter is a variable whose value is passed from one method to another—just like a baton is passed from one runner to another in a relay race. Thus, a variable is a value that can change inside a method, and a parameter is a variable whose value is passed from one method to another. You had to tell the cheshireCat.move method that you used inside your cheshireCat.jump method in the last exercise the direction and amount you wanted the cheshireCat to move. These values are parameters for the primitive cheshireCat.move
method. Your `cheshireCat.jump` method passed the values up (or down) and 1 meter to the `move` method when the `move` method was called to perform its task.

**DESIGNING A GENERIC METHOD**

Methods that are associated with a particular object are called **object-level** methods, or **class-level** methods if they work with any object in a particular class. Methods that are associated with the entire world are called **world-level** methods in Alice.

You can build a generic world-level method that will make any object jump, very similar to the object-level jump methods you previously created. To do this, you need to use a variable. You can design a generic jump method, based on your `aliceLiddell.jump` method.

You want to create a method to do the same thing, but instead of moving the cheshireCat, you want to add a variable to your method that will allow any object to jump. However, how will the new method find out which object should jump each time the method is used? You want the user to specify which object will jump whenever this new method is called. Thus, the variable needs to be a parameter because its value needs to be passed from the calling method.

Now that you have a design in mind for the method, let’s write the code.

**CREATING A PARAMETER**

As you work through this exercise, you’ll see that it’s the same thing you did to create a `cheshireCat.jump` method, except now you are creating a `jump` method for the world. You’ll find that you need to give your parameter a name, and declare what data type the parameter will be. Remember, that the computer stores different data types in different formats, so you need to tell the system what data type to use for this new parameter. The data type can be an Alice object, a number, a Boolean value (true or false), or any one of more than a dozen types built into Alice, such as a string of text, a color, a sound file, a direction, and so on.

To create a parameter:

1. First, start a blank Alice world with the **grass** template.
2. Next, select the **world** tile in the Object tree, and then the **methods** tab in the Details area down below. Click the **create new method** button, and a small New Method dialog box will appear asking you to name the method.
3. Type `jump` as the name for the new method, and then click the **OK** button. In the Editor area, a new tab appears for a blank method with the name `World.jump`.
4. You know that your method needs a parameter. Click the **create new parameter button** on the right side of the `jump` method’s header information, and a **Create New Parameter** dialog box appears, as seen in Figure 2-19.
5. Type **who** as the name of your new parameter and select **object** as the data type. Make sure that the list box is not checked, and then click the **OK** button. Now you should see a tile appear after the name of the method showing us that this method has an object parameter. The **who** parameter may now be used like a variable within the method, and its value must be specified whenever this method is called from another method.

**CODING A GENERIC INSTRUCTION**

You need to use an object in your world to gain access to the primitive methods that are associated with individual objects. To do this, you are going to use the **ground** object.

To code a generic instruction:

1. Select the **ground** tile in the Object tree and then, if necessary, click the **methods** tab in the Details area.
2. First, drag and drop the ground’s **move** method into your new method in the Editor area in place of **Do Nothing**. Select **up** for the direction and **1 meter** for the amount.
3. Next, drag the **who** tile from the method header and drop it to replace **ground** in the instruction you just added. If you drop it in the wrong place (which could happen), click the **Undo** button and try again.

**COPYING AN INSTRUCTION**

Instead of adding and modifying another instruction from the **ground** object, as you just did in the previous section of the chapter, this time you’re going to copy the **who move up 1 meter** tile, and change the direction to **down**. To copy an instruction, you can drag a copy of the instruction to the clipboard, found in the upper-right corner of the standard Alice interface, and then drag it to its new location.
To copy an instruction:

1. First, drag the **who move up 1 meter** tile from your method and drop it on the clipboard. A copy is placed on the clipboard and the original instruction is still where it belongs.
2. Next, drag the clipboard and drop it in your new method just below the existing **who move up 1 meter** tile. Now you should see two copies of the **who move up 1 meter** instruction in your method.
3. Finally, click the parameter box that says **up** in the bottom copy of the instruction, and change the direction to **down**.

If you have done everything correctly, then your method should now match the specifications, and you are ready to see if it works.

**TESTING (AND DEBUGGING) A GENERIC METHOD**

To test your new generic method you need to perform two steps. First, you need to see if the method makes an object move up and down as specified. Then, you need to make sure that the method works with a variety of objects.

To test the generic method:

1. You need to add a few objects to the new method so that you can use them to test your new generic method. First click the large green **ADD OBJECTS** button to enter Scene Editor mode, so you can access the object galleries and tools to position the objects.
2. Let’s add one instance each of the aliceLiddell, whiteRabbit, and cheshireCat classes and position them as you did at the start of Tutorial 2B. Then click the **DONE** button to return to the standard Alice interface.
3. Select the **world** tile in the Object tree and then the **methods** tab in the Details area.
4. In the Details area, you can see tiles for two methods that can be edited: **world.myfirst method** and **world.jump who**. Click the **edit** button to access **world.myfirst method** in the Editor area.
5. You next need to add your generic jump instruction to **world.myfirst method**. Drag and drop the tile for **jump who** from the Details area to replace **Do nothing** in the Editor area. When you do this, you will be asked to select a value for the **who** parameter from a list of all the objects in the current world. If you follow the same pattern as the last tutorial, then Alice should jump first, so select **aliceLiddell, the entire aliceLiddell**.
6. Before continuing, play the world to see what happens. Did it perform as expected? Did Alice jump up and then down? If not, it might be time for the debug step in the program development cycle – design, code, test, and debug.

7. Once the method works properly with alice.liddell, add instructions to world.my first method to make the whiteRabbit jump. You do this by adding a copy of the world.jump method after the instruction to make Alice jump, and then choosing whiteRabbit as the value of the who parameter.

8. Do the same actions for the cheshireCat, which should jump after the whiteRabbit does so.

9. Now test your method again. If everything is correct, Alice should jump, then the whiteRabbit, and then the cheshireCat. When you play the world you should not be able to see the difference between a world with a generic jump method and one with three separate object-level jump methods.

10. This is a world that you should save for use in future tutorials. Save it with the name generic triple jump and remember where you saved it.
CHAPTER SUMMARY

This chapter consisted of discussions of top-down development, modular design, and a simple software development cycle, followed by three hands-on tutorials involving Alice methods. The discussion of top-down design and modular development included the following:

- The process of developing methods for objects is mostly a process of developing algorithms, since each method is an algorithm.

- Algorithm development has a long history in the world of mathematics, where algorithms are step-by-step solutions to well-defined mathematical problems.

- Good mathematicians often start solving a problem by trying to break a large algorithm into parts that are more manageable. This process is also known as top-down design or top-down development, starting at the top with one concept or big idea, and then breaking that down into several parts.

- The process of top-down design leads to modular development, in which the parts, or modules, that make up a complete solution to a problem are developed individually, and then combined to form that complete solution.

- Modular development makes a large project more manageable, is faster for large projects, leads to a higher quality product, makes it easier to find and correct errors, and increases the reusability of your solutions.

- Saving solutions to small problems so they can be applied elsewhere creates reusable code. Object-oriented programming encourages the development of reusable code.

The discussion of a software development cycle included the following:

- Programmers follow a software development cycle, in which they design, code, test, and debug methods.

- Software testers need to be able to develop a testing plan that examines new software under all possible circumstances. They examine the correctness of a method, its time and space efficiency, and whether or not it has any undesirable side effects.

- Unit tests check to see if methods work as expected individually. Sometimes a method, called a testing shell, is used in unit testing to simulate the environment in which a method will operate.

- Integration tests check to see if larger methods work after newer methods are incorporated into them.

In Tutorial 2A you explored the primitive methods that come with every Alice object. Sometimes it is desirable to run primitive methods directly, such as when positioning objects to set up a scene, or to find a lost object. You also saw that some methods come with user-created methods to provide them with additional behaviors, such as the walk method in the Penguin class.

In Tutorial 2B you applied some of the methods discussed earlier in the chapter to create an Alice world with a top-level method and separate sub-level methods to make three characters from Alice’s Adventures in Wonderland jump up and down on the screen.
In Tutorial 2C you learned to create a generic method to make any object jump up and down. You learned that a parameter can be used to pass a value to a method when the method is called.

**REVIEW QUESTIONS**

1. **Define the following terms:**
   - CamelCase
   - coding
   - encapsulated methods
   - integration test
   - method header
   - modular development
   - off-camera
   - parameter
   - primitive methods
   - program development cycle
   - reusable code
   - test for correctness
   - top-down design
   - unit test
   - user-defined methods
   - variable

2. **Describe the processes known as top-down design and modular development, and how they are used together.**

3. **How do organizational charts help with top-down design and modular development?**

4. **List and describe the advantages of using modular development.**

5. **How does the practice of object-oriented programming encourage the development of reusable code?**

6. **List and describe the steps in a simple program development cycle.**

7. **What is the difference between a unit test and an integration test? Why are they both used in software development?**

8. **What are the differences between primitive methods and user-defined methods in Alice? Which of these are encapsulated methods and what does that mean?**

9. **Describe two different primitive methods that can be used to find objects that are off-camera. What are the differences between using these methods?**

10. **Describe what parameters are, and how they are used in Alice methods.**

**EXERCISES**

1. **Create an organizational chart showing a top-down, modular design for each of the following:**
   a. Baking a cake.
   b. Traveling from your house to Buckingham Palace in London.
   c. Sorting a list of names alphabetically by last name.
   d. The plot of your favorite film or episode of a television show.
   e. The solution to your favorite math problem.

2. **Create a modified jump method, similar to the one you created in the generic triple jump world in Tutorial 2C, to include a parameter for how high the object should jump.**
3. Open an Alice world with a bunny in it. Create a `hop` method for the bunny that has the object move forward while jumping up and down. (Hint: Use the `do together` logic and control tile.) Create and test a generic `hop` method that has three parameters — `who`, `how high`, and `how far`.

4. Modify the generic triple jump world from Tutorial 2C, so that it functions as follows:
   a. The three characters appear on the screen in position as before.
   b. Alice says “Simon says, Jump!”
   c. Alice jumps.
   d. The White Rabbit jumps.
   e. The Cheshire cat jumps.
   f. The Cheshire disappears, but his smile stays visible. (Hint: There is a property named `opacity`. Dragging a property into a method will create an instruction to set the property to a new value.)
   g. Alice and the White Rabbit both turn their heads to look at the Cheshire cat.

   Save your new world with a different name so that the generic triple jump world is preserved.

5. Begin a new Alice world and add instances of a chicken, a horse, and a cow to the world. Look at the Object tree. Can you see which object violates one of the naming conventions in Alice?

6. Some Alice method names do not follow the CamelCase naming convention. How should `world.my first method` be renamed so that it is in CamelCase? What about the names of user-created methods that are supplied with the Penguin class of objects?

7. Examine the code in the lakeSkater world that is provided with the Alice software as an example world and that was used in Chapter 1.
   a. Explain how the organization of the iceSkater’s object-level methods are related to the concept of reusable code.
   b. While the world exhibits good modular design in many of its methods, the method `world.my first animation` could be improved. Create an organizational chart showing an improved design for `world.my first animation` that exhibits better modular design.

8. The American Film Institute’s list of *The 100 Greatest Movie Quotes Of All Time* is on the web at [http://www.afi.com/tvevents/100years/quotes.aspx#list](http://www.afi.com/tvevents/100years/quotes.aspx#list). Pick one of the quotes and create a short Alice world in which one of the characters uses the quote. Make sure that your world exhibits good modular design.

9. On the surface of the moon there is a crater, named al-Khwarizmi, which sounds a lot like algorithm. It was named after Abu Ja’far Muhammad ibn Musa al-Khwarizmi, who lived in the Ninth Century. See if you can find out why he is important enough in the history of mathematics to have a crater on the moon named after him. What does his work have to do with computer programming and with ancient Greek mathematicians?

10. Select one of the themed folders in the Alice galleries, such as the Japan, Egypt, or Old West folders. Select several objects from the folder and develop an outline for a short story using those objects. Create an organizational chart that shows the modules in the story, and develop a list of the Alice methods that you would need to write to implement your story as an Alice world. Which of these methods do you think would be reusable for other stories? If you have high speed Internet access, you may want to look in the Web Gallery, which is more extensive than the Local Gallery.