3 Methods and Parameters

This chapter introduces the concepts of methods and parameters. Methods and parameters are key concepts in object-oriented programming. A method is a collection of actions that will be carried out by objects in the world when we request that the method be executed. A parameter acts like a basket to send information to a method.

The worlds created as examples and exercises in chapter 2 were relatively small animations constructed to introduce concepts of building and programming animations in virtual worlds. As worlds in exercises and projects grow larger, it will become increasingly important to use many, many methods. Methods provide a number of advantages to the programmer. For example, once a method is written it allows us to think about an overall task instead of all the small actions that were needed to complete the task. This is called abstraction.

In Alice, methods can be character-level (defined as an action for an object acting alone) or world-level (involving the actions of more than one kind of object). In Computer Science, character-level methods are called class-level methods. In programming languages such as Java or C++, world-level methods are often called directly from main – which is called when the program is run.

In section 1, our focus is on learning how to create and run our own world-level methods. To run our own method, it must be called (invoked). Comments are used to document the code.

Section 2 launches a discussion of parameters. A parameter is used to send values and object names to a method – a form of communication between methods. The information that gets sent to a method can be of many different types (e.g., a number value, an object, or some property value such as color).

Note to Instructors:
Every type of object in the Alice gallery has a repertoire of primitive actions it can perform—such as move, turn, roll, and point at. In Chapter 2, we wrote our first program as a collection of primitive instructions (for the snow man and snow woman). In this section, we will learn how to write intricate (and impressive) programs by writing and running new methods.

Methods
A method is a well-designed collection of instructions that will be carried out when requested. Of course we know that, in Alice, instructions are carried out by one or more objects in a virtual world. We also know that when objects carry out instructions, they may be acting alone— that is, not affecting or affected by other objects. On the other hand, when objects carry out instructions they may very well be interacting in some way with another object. When viewed this way, methods can be thought of as prescribing the behavior of objects. Methods that define behaviors for an object acting alone are considered object or character-level. (In object-oriented languages, these are called class-level methods.) Methods that involve actions by more than one object are considered world-level methods. In this section, we focus on world-level methods. Character-level methods are discussed in the next chapter.

Why do you care?
When you created your own animations in the exercises for the last section, no doubt you started to think about more complicated kinds of scenarios with more twists and turns in the storyline, perhaps games or simulations. Naturally, as the storyline becomes more intricate so does the program code for creating the animation. The program code can quickly increase to many, many lines of code— sort of an “explosion” in program size and complexity. Animation programs are not alone in this increase in complexity. Real world software applications can have thousands, even millions, of lines of code.

How does a programmer deal with huge amounts of program code? One technique is to divide a large program into several small modules, where each module performs a specific portion of the functionality of the overall program. In object-oriented programming, modules are used to define classes of objects and methods for those objects. Thus, methods play a major role in structuring the program code. Methods allow the programmer to think about a collection of actions as if it was just one action—this is called abstraction. Furthermore, each individual method can be tested to be sure it works properly. Finding a bug in a few lines of code is much easier than trying to find a bug in hundreds of lines of code. Not only do methods make a program easier to think about, they also make a program easier to read and debug.

Example
In our first program, a snow man tried to meet a snow woman at a winter dance. But, the snow man only tried once to attract the snow woman’s attention. It is perhaps more realistic that the snowman shouldn’t give up so easily. Perhaps he should try to catch the snow woman’s attention several times, speaking to her and making eyes at her repeatedly before she turns to look at him. Clearly, to make this change to the animation we will need to repeat the sequence of attention-catching instructions several times in the program.
Of course, it is possible to copy and paste instructions in the editor wherever needed. But, this becomes particularly tedious if it needs to be done many times. Wouldn't it be nice if these instructions could be grouped together to work as a method? Once the method is defined, Alice can be told to run that method several times, or from several different places in the program, without having to copy the instructions again and again into the editor.

**Creating our own method**

Let's write a method named *catchAttention*. This method will involve both the snowman and the snow woman, so it should be a world-level method. Although the instructions we will use are similar to those presented in the previous section, the animation will be constructed again from scratch, using a new world and a new initial scene. (The purpose of starting from scratch is to illustrate the complete process of writing your own method.)

In the object tree, the *World* object is selected and then the *methods* tab in the Details pane (located in the lower left of the screen). Then, the *create new method* tile (in the methods detail pane) is clicked. Figure 3-1-1 illustrates the *World, methods, and create new method* selections.

![Figure 3-1-1. Selecting World's "create new method"](image)
When the *create new method* tile is clicked, a new tile automatically appears in the method detail pane. Alice automatically names the new method with the simple term "method," as seen in Figure 3-1-2.

Replace the word "method" with a new name by highlighting the word "method" in the box and using the keyboard to enter the name "catchAttention", as shown in Figure 3-1-3.

![Figure 3-1-2. The Name Tile](image1)

![Figure 3-1-3. Renaming the New Method](image2)

Figure 3-1-4. Selecting *World.catchAttention* for editing

Clicking on the *edit* button to the right of the *catchAttention* tile in the details opens a new pane in the editor to allow code to be written for the method. (See Figure 3-1-4.) Note that the particular method being edited has its tab colored yellow, and that all other method tabs are grayed out. The instructions for the *World.catchAttention* method are illustrated in Figure 3-1-5.
Warning: If the PLAY button is clicked at this time, the animation will NOT run. This is because the catchAttention method has been defined but Alice has not been told to execute the method. That is, the method has not been called into action. Computer scientists would say the method has not been invoked.

Calling (Invoking) a Method
So, how do we call (invoke) our new method? Well, in the previous section of this chapter, we learned that when you (as the human “user”) click on the Play button, Alice automatically executes World.my first method. We can see why this happens by looking carefully at the Events editor, located in the top right of the Alice interface as seen in Figure 3-1-6. The instruction in this editor tells Alice When the world starts, do World.my first method. We didn't put this instruction here – the Alice interface is automatically programmed this way. So, when the user clicks on the Play button the world starts and myFirstMethod is invoked.
Let’s take advantage of this arrangement. All we have to do is drag the `World.catchAttention` method from where it is listed in the methods tab of the details pane into `World.my first method`, as illustrated in Figure 3-1-7. Now, whenever the Play button is clicked, `my first method` will run and the `World.catchAttention` method will be invoked.

Figure 3-1-7. Dragging `catchAttention` into `my first method`

Figure 3-1-8. Calling `catchAttention` from `my first method`
Of course, we want other actions to take place as well. So, other instructions are added to *World.my first method*, as shown in Figure 3-1-8. This new program is somewhat different from the first program code written in Chapter 2. In the new program, the snowman tries to catch the snowwoman’s attention twice, moving closer to her (so she can hear him) after the first time.

**Comments**

Now that we have written our own method, it is time to look at a useful component to programs and methods – comments. Comments are not executable instructions. This means that Alice can ignore comments when running a program. However, comments are considered good programming “style” and are extremely useful for humans who are reading a program. Comments help the human reader to understand what a program does -- particularly helpful when someone else (for example, your instructor) has to read your program code.

Comments in Alice are created by dragging the green `//` tile into a program and then writing a description of what a sequence of code is intended to do. Figure 3-1-9 illustrates the program with comments added.

![Figure 3-1-9. Comments in *World.my first method*](image)

In methods where it is not obvious, a comment should be included at the beginning of a method to explain what the method does. In addition, small sections of several lines of code that collectively perform some action should be documented using a comment. The first comment in Figure 3-1-9 provides a general statement about what the program does: “…simulates a
snowman trying to meet a snowwoman.” The second comment describes the lines of code that animate the snowman trying to catch the snow woman’s attention and the third comment documents the snow woman’s responding actions.

**Technical Note**
The Events editor will be described in further detail in Chapter 4. But, it is worthwhile mentioning at this time that the Events editor can be used to modify the *When the world starts* instruction. Modifying this instruction allows you to call a new method rather than *World.my first method* when the user clicks on the Play button.

To modify the instruction in the Events editor, click on the image to the right of *World.my first method* in the Events editor. Then, select *catchAttention* from the drop down list that appears, as seen in Figure 3-1-10. Now, when the world starts, the *World.catchAttention* method will run instead of *World.my first method*.

![Figure 3-1-10. Modifying "When the world starts" event](image)
3-1 Exercises

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

1. Confused Kanga
Scrounging for breakfast on the outback, Kanga encounters a rather confusing sign (Roads and Signs folder.) Kanga, stares at the sign for a few seconds and then hops left and turns toward the sign and then hops right and turns toward the sign and then left and then right….

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

2) Gallop and Jump.
Kelly has entered an equestrian show as an amateur jumper. She is somewhat nervous about the competition so she and the horse are practicing a jump. Create an initial scene with a horse and rider facing a fence, as shown below.
Write two world-level methods, one named *gallop* (horse and rider gallop forward one step) and another named *jump* (the horse and rider jump the fence.) In the gallop, the horse’s front legs should lift and then go down as the back legs lift and the horse moves forward. Then the back legs should go back down. The jump should be similar but the horse should move up far enough to clear the fence in mid-stride. Test each method to be sure they work as expected. You will need to adjust the distance amounts to make each look somewhat realistic.

*Hint:* If you make the horse the vehicle for Kelly (see Tips & Techniques 1 for details on the vehicle property), you will only need to write an instruction to move the horse and Kelly will go along for the ride.

When you think the gallop and jump methods are both working properly, write instructions in *World.my first method* that call the *gallop* method as many times as needed to move the horse and rider up close to the fence and then call the *jump* method. Use trial and error to find out how many times the *gallop* method must be called to make the animation work well.

### 3. Rabbit Maze.

Create a world with the WhiteRabbit object (Animals folder) standing at the edge of a maze. The maze is created using blocks (Shapes folder.) Create walking methods for the rabbit to walk forward and backward. (Hint: Just reverse the forward walk for the backward walk.) Also create *TurnLeft* and *TurnRight* methods. In *my first method*, place these four methods in an order that will guide the rabbit successfully through the maze and then rewind through the maze (or just go through the maze backwards).
3-2 Parameters

It is clear from the example and exercises in the previous section that one program may be made up of several methods. Each method is its own small block of instructions, designed to perform a specific task when requested. It is possible, if not probable, that objects may be performing actions in more than one world-level method. We can appreciate that some communication might need to occur between methods. In this section, we look at parameters. Parameters are used for communication between methods. We arrange to transfer values (e.g., a number or a color) or names of objects from one method to another by using parameters in our methods.

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

![Figure 3-2-1. Bug Band on Concert Stage](image)

Setting the Stage
Figure 3-2-1 shows the initial scene for the animation. The world is made up of a concert stage (Sets), queen bee, mantis, worker bee, ant (Animals), bass, saxophone, drums, and guitar (Musical Instruments). A spotlight will be used to highlight one band member at a time for their solo. To make the spotlight more dramatic, the light in the scene should be less bright. To dim the lights on stage, Light is selected from the Object tree and then the brightness property of the Light is changed from 1 to 0.5, as seen in Figure 3-2-2. A spotlight is added to the scene and a one-shot instruction is used to point the spotlight at the crowd. Figure 3-2-3 shows the dimmed scene with the spotlight shining on the crowd. (The spotlight is circled in Figure 3-2-3 for the purpose of showing the spotlight pointed at the crowd.)
Storyboard with a Parameter
The storyline for this animation is that each band member will be highlighted by a spotlight and will perform a short solo (play a musical instrument.) Let’s create a storyboard for the spotlight focus.
and movement of each band member during a solo. Because we have four band members, four textual storyboards are composed:

<table>
<thead>
<tr>
<th>Do in order</th>
<th>Do in order</th>
</tr>
</thead>
<tbody>
<tr>
<td>point spotlight at queen bee</td>
<td>point spotlight at mantis</td>
</tr>
<tr>
<td>queen bee move and play instrument</td>
<td>mantis move and play instrument</td>
</tr>
<tr>
<td>point spotlight at crowd</td>
<td>point spotlight at crowd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Do in order</th>
<th>Do in order</th>
</tr>
</thead>
<tbody>
<tr>
<td>point spotlight at worker bee</td>
<td>point spotlight at ant</td>
</tr>
<tr>
<td>mantis move and play instrument</td>
<td>ant move and play instrument</td>
</tr>
<tr>
<td>point spotlight at crowd</td>
<td>point spotlight at crowd</td>
</tr>
</tbody>
</table>

Of course, we could write four methods – one for each solo storyboard. But, it is quite clear that the four storyboards are strikingly similar. The major difference is which band member will perform. This is where parameters come in. Let’s collapse the four storyboards into one storyboard and use a parameter to communicate which band member will perform the solo. The storyboard with a parameter is:

<table>
<thead>
<tr>
<th>Parameter: bandMember</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do in order</td>
</tr>
<tr>
<td>point spotlight at bandMember</td>
</tr>
<tr>
<td>bandMember move and play</td>
</tr>
<tr>
<td>point spotlight at crowd</td>
</tr>
</tbody>
</table>

The *bandMember* parameter name (an arbitrary name) is taking the place of the name of the specific object that will perform the solo. You can think of a parameter as acting like someone who stands in a cafeteria line for you until you arrive – sort of a *place-holder*. If a parameter is created, we can write just one method use the parameter to communicate which band member is to perform the solo when the method is called.

**Creating an Object Parameter**

To implement the animation, a new world-level method, named *solo*, is created. The editor creates a new method pane, as seen in Figure 3-2-4. A “*create new parameter*” tile automatically appears in the upper right of the method’s tabbed pane in the editor. When the create new parameter tile is clicked, a dialog box pops up as shown in Figure 3-2-5. The name of the parameter is entered and its type (Number, Boolean, Object, or Other) is selected. In this example, the name of the parameter is *bandMember* and its type is *Object*. When completed, the parameter name is in the upper left of the method pane, as shown in Figure 3-2-6. Note the
type label that appears immediately before the parameter name. This means that any method that calls the `World.solo` method must supply a value that is the name of the object.

![Figure 3-2-4. World.solo method pane](image)

![Figure 3-2-5. Name and select type for parameter declaration](image)

![Figure3-2-6. Resulting parameter](image)

**Using an Object Parameter**

In this example, two different ways to use an object parameter will be illustrated: (1) to specify a target of an action, and (2) to specify the object that performs an action.

*(1) Parameter is a target*

The first step in the storyboard above is “point spotlight at bandMember.” The `SpotLight` is selected in the Object tree and its `point at` method is dragged into the editor. The target of the spotlight is selected from the cascading menu, as shown in Figure 3-2-7. The list of possible targets for a `point at` instruction is composed of all the objects in the Object tree. But,
bandMember is a parameter, not an object in the Object tree. Instead, bandMember is considered an expression and is listed at the bottom of the cascading menu. The resulting code, illustrated in Figure 3-2-8, shows the object parameter bandMember used as a target for the point at instruction.

(2) Parameter performs an action
The second step in the storyboard is “bandMember move and play.” Clearly, in this action, the bandMember will perform an action, not be the target of an action. This is the second way a parameter can be used in a method. Intuitively, we look at the Object tree to find bandMember so its move instruction can be dragged into the editor. But, of course, bandMember is not in the Object tree. (See Figure 3-2-9) This makes sense because, as mentioned earlier, bandMember is not an object – it is acting as a place-holder for an object.
Because *bandMember* is a parameter and not an object, two steps are needed to create instructions where an object parameter is to perform an action. The first step is to select one of the objects (Bee, mantis, ant, or QueenBee) and create instructions for that object. Let’s use the QueenBee (an arbitrary choice) for move instructions, as shown in Figure 3-2-10.

![Figure 3-2-10. Instructions for an arbitrary object](image)

The second step is to modify the move instructions to allow the parameter to act as a place-holder for the object. The *bandMember* parameter is dragged toward a move instruction as illustrated in Figure 3-2-11. In this example, *bandMember* will be used to take the place of *QueenBee* in both move instructions.

**Importance of Parameter Type**

We wish to draw your attention to the appearance of the instructions in Figure 3-2-11. When a parameter tile is dragged over instructions, all the tile locations where the parameter tile can be dropped are immediately enclosed by a green border. The color-change is a visual hint to allow you to select an appropriate location where the parameter can be used. In this example, *bandMember* is to be used as a place-holder for an *object* in a move instruction. But,
bandMember should not used as a place-holder for distance (0.25 meters) or duration (0.5 seconds) because these are number values, not objects.

![Figure 3-2-11. Drag parameter to object position](image)

The last step in the storyboard is to turn the spotlight back on the crowd (that is, away from the band member who just performed a solo). A second spotlight point at instruction is added at the end of the World.solo method. In this instruction, concert_stage.crowd is the target for the point at instruction. The resulting code is shown in Figure 3-2-12.

![Figure 3-2-12. Resulting solo code](image)

Test with Arguments
Let’s pause to save and test the code. To test the World.solo method, the solo method is called from my first method. When solo is dragged into my first method, cascading menus (Figure 3-2-13) allow the selection of an object that bandMember will represent for that call of the method.

To be certain the solo method works for each Bug Band musician, four statements are written, as seen in Figure 3-2-14. In the first call, the solo method will be performed with bandMember representing QueenBee, in the second call bandMember will represent mantis, in the third the worker Bee, and in the fourth the ant. In Computer Science terminology, the value communicated to a method when called is known as an argument. In this example, the QueenBee, mantis, worker Bee, and ant are each used as an argument in a call to the solo
method. In this way, the solo method is somewhat *generic* – it can be used with different arguments to carry out the same task with different objects.

### Multiple Parameters

You may have noticed that the above code does not yet complete the animation. In each solo, the band member should not only move but should also play a musical instrument. (If your computer does not have a sound card, the “say” instruction can be used to display song lyrics instead of
An instruction is needed in the solo method to play a sound. But, each musical instrument should have a different sound. This means that the sound of the bass should be played for the QueenBee, a saxophone for the mantis, drums for the worker Bee, and guitar for the ant’s performance. Evidently, more than one parameter would be useful in this situation. To illustrate the use of multiple parameters, a second parameter can be added to the solo method. The type of this parameter will be Sound, as shown in Figure 3-2-15. The necessary sound files are then imported (see Tips & Techniques 2 for details on importing and using sound files). As with object parameters used to perform an action, an instruction to play a specific sound (bassMix) is dragged into the editor, as illustrated in Figure 3-2-16.

Figure 3-2-15. Sound parameter
Finally, the *music* parameter is used to act as a place-holder for a sound. The completed solo method is shown in Figure 3-2-17. Calls to the solo method are revised in *my first method* to pass in two arguments, an object and a sound. The revised code is shown in Figure 3-2-18.
As mentioned above, this animation can be completed without the use of sound. An alternate version of the solo method is shown in Figure 3-2-19. A String parameter, songLyric, is used instead of a Sound parameter. The string is displayed using a say instruction.
Other types of parameters
The Bug Band example illustrated two kinds of parameters, objects and sound. A parameter can also be a number, a Boolean (true or false) value, a color (red, blue, green, etc.), or any of several other types. (See Figures 3-2-5 and 3-2-15 for examples of parameter type selection.) Each of these types of values contributes to a rich environment for programming. Number values play an important role in many programming languages. So, before leaving our discussion of parameters, let’s take a look at a simple world in which a number parameter is illustrated.

Figure 3-2-19. Magic Act Initial Scene

Example
A magician is performing a levitation illusion. In a levitation illusion, objects seem to rise magically into the air. The magician points a magic wand at his assistant and she gently rises into the air and then floats back down to her original position on the table. Then, the magician performs the same trick with the rabbit. The rabbit, being a lighter object, will float into the air higher than the magician’s assistant. The initial scene is illustrated in Figure 3-2-19.

Because the magician’s assistant and rabbit are each to levitate in the same way, the animation can be implemented as a single method if we use a parameter to communicate which object is to float. The height the object floats could also be passed as an argument to the method. So two parameters are needed, one an object (floatingObject) and the second a number (height). A possible storyboard is shown below.

Parameters: floatingObject, height
Do in order
  Magician right arm point at floatingObject
  floatingObject move upward height meters
  floatingObject move downward height meters
Number parameter
An object parameter named `floatingObject` and a number parameter named `height` are created and then lines of code are written to float the object up and back down the given height. The resulting method is shown in Figure 3-2-20.

![Figure 3-2-20. Object and Number parameters](image)

Calls to the levitate method are written in my first method, as illustrated in 3-2-21 and then the program is run to test the method.

![Figure 3-2-21. Calling levitate method](image)

Technical Note
When this animation is run, the result is rather surprising. The rabbit floats up and down just as expected. But, the magician’s assistant moves horizontally instead of floating upward in the air! Of course, the magician’s assistant is reclining on the table and so her “up” (see Figure 3-2-22) is not exactly what we had in mind. To solve this problem of orientation, the `move` instructions in the `levitate` method are revised to use the `as seen by` option. As seen by allows us to select a direction based on the perspective of a different object. In this example, we selected the Ground as a reference, as shown in Figure 3-2-23. The revised code is presented in Figure 3-2-24. Now the objects will move upward with reference to the Ground. (See Tips & Techniques 2 for more detail on `asSeenBy`.)
Figure 3-2-22. Up – assistant’s orientation

Figure 3-2-23. Selecting up asSeenBy Ground

3-2-24. Revised Levitate
3-2 Exercises

1. FrogEscape
At the local lily pond, the frogs enjoy climbing out of the water now and then to warm up in the sun. Of course, the frogs get a bit jumpy when a predator is sighted. On this fine day, a hungry snake wanders into the scene. Create a world scene similar to the one below and animate the frogs jumping into the pond when the snake approaches. Write a method that points the snake at a frog and slides the snake toward the frog. Then, have the frog turn to the pond and jump in. Your method should use a parameter to specify which frog is escaping.

2. FunnyRace
This is a funny kind of race because you will know who is going to win before the race begins. (But, this is a good exercise to gain some experience in race simulations as a preparation for more complicated worlds later in the book.) Create a world with three characters lined up as if they were at the starting line of a race. Add another object to the world that can serve as a start and finish line. (In the scene below, we used the solorail from the Skate Park collection.) Create a method that simultaneously makes each character move forward to pass the finish line. Use three parameters (one for each character) that specify the amount of time (duration) for each of the three characters to complete the race.
3 Dragons
Legend has it that dragons are distant relatives of chickens. So, we are not surprised that a favorite pastime of dragons was a game of “chicken.” The scene below shows a world with four dragons, carefully placed in a diamond-like pattern so any dragon is an equal distance from each of the other dragons (similar to baseball players at the four bases of a baseball field). Create a simulation of a game of chicken where any two dragons face each other and fly upward to a slightly different height above the ground. Then, the dragons fly towards each other, nearly missing one another. The dragons should each land in the position where the other dragon was located. That is, the two dragons trade places. Your simulation should use a method named `dragonFlight` that has four parameters – the two dragons that will face off in a game of chicken and the heights for each of the dragon’s flight.

4 WheelMotion
This exercise is to allow you to work with motor vehicles that have wheels. The challenge here is that the motion of the wheels (rotation) is different that the overall motion of the motor vehicle. The wheels must rotate while the car itself moves forward. To gain some appreciation for this kind of animation, create a simple world with a car and a dump truck. Create 2 methods `wheelRoll` and `vehicleMove`. `wheelRoll` should receive a parameter that is one wheel and turn it one revolution forward. The `vehicleMove` method should receive a parameter that is a motor vehicle and make it move forward while the wheels roll (by calling `wheelRoll` for each wheel of the vehicle).
3 Summary

In this chapter, we looked at how to write our own methods and how to use parameters to communicate between methods. In particular, we focused on world-level methods that involve two or more objects interacting some way. An advantage of using methods is that the programmer can think about a collection of actions as if it was just one action—abstraction. Also, methods make it easier to debug our code. Comments were used to document methods, where the purpose of the method or a segment of a method is not immediately obvious. Comments are considered good programming “style.”

We used parameters to arrange a transfer of values from one method to another. In a method, a parameter acts as a place-holder for a value of a particular type. Examples presented in this chapter included object, sound, string, and numeric parameters. Parameters allow methods to be written in a generic manner. The method can be called with different arguments to carry out the same task with different values.

**Important concepts in this chapter**

- A method is a well-designed collection of instructions that will be carried out when requested.
- Methods that define behaviors for an object acting alone are considered object or character-level.
- Methods that involve actions by more than one object have a more global perspective and are considered world-level methods.
- To run (or execute) a method, the method must be called (invoked).
- Parameters are used for communication between methods.
- A parameter can be declared to represent a value of a particular type.
- Types of values for parameters include object, Boolean, number, sound, color, string, and others.
- In a call to a method, a value sent in to a method parameter is an argument.
3 Projects
Projects are more challenging than exercises in a chapter. The projects in this chapter involve motion of human body parts. Professional animators spend many, many hours mastering the art of making these movements look realistic. But, our focus is on mastering the art (and science) of writing methods in a program.

1. Dance
Note: To assist you in learning how to animate human body parts, this first project includes some explanations and coding suggestions. The goal of this animation is to have the couple perform a dance step in a traditional box (square) figure as used in a waltz and other dances. Create a scene with a man and a woman character (People) inside a saloon (Old West), as illustrated below.

In the first step of a box figure, the man takes a step forward, leading with his left leg and (at the same time) the woman takes a step backwards, leading with her right leg. This is not as simple as it sounds. One way to make a character with legs appear to take a “step” is to have the character raise one leg some small amount and then move forward as the leg moves back down. Then, the other leg performs a similar action. Thus, to make two characters appear to dance with each other requires a coordinated leg lift, move and leg drop action for both characters. A potential storyboard for the first step of the box dance is illustrated below.

<table>
<thead>
<tr>
<th>Parameters: howFar, speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do in order</td>
</tr>
<tr>
<td>1) Do together</td>
</tr>
<tr>
<td>Man's left leg move up</td>
</tr>
<tr>
<td>Woman's right leg move up</td>
</tr>
<tr>
<td>2) Do together</td>
</tr>
<tr>
<td>Man move forward howFar</td>
</tr>
<tr>
<td>Man's left leg move down</td>
</tr>
<tr>
<td>Woman move backward howFar</td>
</tr>
<tr>
<td>Woman's right leg move down</td>
</tr>
</tbody>
</table>
Write a method named `forwardStep`. The `forwardStep` method has two parameters -- `howFar` which specifies the distance of the move forward, and `speed` which specifies the duration for the forward motion. An example of a `forwardStep` method is shown below.

After the `forwardStep`, three steps remain: (1) `rightStep`, where the man and woman take a step sideways (his right, her left), (2) `backStep`, where the man takes a step backward, leading with his left leg and the woman takes a step forwards, leading with her right leg, and (3) `leftStep`, where the couple takes a step sideways (his left, her right). You will need to experiment with the amount to move the legs up (and down), and with the size of step forward, backward, and sideways the couple is to take.

Then, create a method named `spin`, which has the man spin the woman around. The code for a spin method might appear as follows:
If the four steps are properly performed in sequence, the couple moves in a square-like pattern on the dance floor. Create a method to call all the methods in order so as to animate the couple performing a box figure followed by a spin for a dance. Then, create a second method to animate the performance of a different dance figure – calling the dance steps in a different order.

2. HandBall
Create a world with a RightHand holding a ball. Have the fingers close to grasp the ball. Then, throw the ball into the air while opening the fingers of the hand. Finally, make the hand catch the ball as the hand re-closes its fingers. (Hint: Consult Tips & Techniques 1 for how to use the vehicle property. The vehicle property will help in making the ball move as the hand moves.)

3. RaRowYourBoat
Create a world with a boat, a person sitting in the boat, an island, and a pier located 25 meters away from the island. In the world shown below, Ra (Egypt) is sitting in a rowboat (Vehicles). (Note: this world takes some time to set up because getting a person object to sit down requires patient rearrangement of the body parts in the scene editor. If you wish, you can use the world provided on the CD.) Create a method to make the Ra row the boat 25 meters from an island to a pier. One suggested way to do this would be to create the methods: RowLeft and RowRight (to control the arms motions), ControlTorsoAndHead (to control back and head motions), and StartRow and StopRow (to put Ra’s body in and out of the rowing position).
4. RearrangingRobot

Alice received a gorilla robot (Animals) as a gift. She is trying to figure out how to program the robot to help her pick up things in her room (a chore she has to do every Saturday morning). In the initial scene for this animation, the robot is standing in the middle of Alice’s room near several objects scattered around the floor (for example, a barbell, a pinata, and a teddy bear). The robot should be located between all the objects so it can easily be turned during the animation – not moving away from any of the objects.

Write a program to teach the robot to pick up one object at a time and rearrange it. Write two methods, named `pickUp` and `putDown`. Each of these methods should have one parameter identifying the name of the object to be picked up or put down. The `pickUp` method should make the robot pick up an object in its hand. The `putDown` method should have the robot put the object down somewhere other than where it started. (Hint: have the robot turn 1/4 left or right before putting the object down.) When these methods have been written, then write instructions in `myFirstMethod` to make the robot pick up each object, and place it somewhere else.
Renaming an Object
In some animations several objects of the same type of character may be in the scene at the same time. Consider the scene in Figure T-3-1. Four chickens have been added to the world. Alice automatically names the chicken objects as Chicken, Chicken2, Chicken3, and Chicken4. Names serve to identify an object as being different from other objects in the world. If the names of particular objects do not suit your purpose, the names can be changed.

Figure T-3-1. A scene with multiple objects of the same model

To rename an object, right-click on the object name in the object tree, and select the rename option from the popup menu, as shown in Figure T-3-2. Then, enter the new name in the dialog box that pops up. Be sure to press the enter key after typing the new name.

Figure T-3-2. Renaming an object

Color Property as Identification
After several operations to arrange objects in the world, you likely will not remember which chicken is which – regardless of what the chickens are named! The simple problem is all four
chickens look identical. For example, just by looking at the chickens in the scene, it is impossible to determine which one is Chicken2.

The solution is to change the appearance of each chicken in some way, so that each chicken can easily be identified. One possible change is the color property of each chicken’s Upper Lip. (Do chickens have lips?) This is a quick and easy change -- click the chicken object in the Object-tree and successively work your way down the tree to eventually select the Upper Lip, as seen in Figure T-3-3.

![Figure T-3-3. Chicken's UpperLip](image)

Then, change the color of the chicken's Upper Lip to a different color. This is done by right-clicking on the name of the object in the object tree, selecting *one shot*. When *one shot* is selected, cascading menus popup one at a time. In Figure T-3-4, the menu selections are *one-shot* → *property animations* → *Chicken.Neck.Head.UpperLip set color to* → *value* (where the value is a color).

![Figure T-3-4. Setting the color of chicken's UpperLip](image)
If all four chickens have a different color upperLip, the chickens will now be easily identifiable, as seen in Figure T-3-5.

![Figure T-3-5. Identifiable objects](image)

This identification technique will be quite useful in future worlds. If we wish to know that something happened during the course of running an animation, we can just change some property of that object, such as its color!

**Printing your program code**

It is often useful to print the code from one of your methods. (Some instructors wish to have students hand in a printed copy of their program code.) To print a method, click on File, and then select Print. Select the name of the method to be printed, as shown in Figure T-3-6.

![Figure T-3-6. Printing a method](image)

**Orient to**

Surprising results can occur when two objects must move together but have different orientations. One technique is to make one object the vehicle of another. But, sometimes we do
not want to make one object be the vehicle of another. In the world shown in Figure T-3-7, the monkey runs and jumps on the ball. Then, both the monkey and the ball are supposed to roll forward together.

We wanted the monkey to stay on top of the ball, moving along with it for a short distance and then jump off and run away. The code we wrote to make the monkey and ball move forward together was:

```
Do together
```

```
  toyball2 move forward 1 meter more...
  monkey move forward 1 meter more...
```

Imagine our surprise when the ball rolled in one direction and the monkey moved in a different direction, ending up well away from the ball, sort of suspended in midair. (No doubt, the monkey was a bit startled, as well!)
Why did this happen? Well, the ball is an example of an object for which we can’t tell (just by looking at it) which direction is forward and which direction is backward. Evidently, in moving the ball and the monkey around in the scene, we positioned the ball so its forward direction was not the same as the forward direction for the monkey.

The way to solve this problem is to synchronize the orientation of the two objects when the initial world is being created. First add the objects to the world scene. Then use a one-shot orient to $(0,0,0)$ instruction on each object. The orient to $(0,0,0)$ instruction may seem a bit weird, but it simply tells Alice that the object should take on the same sense of the center of the world, as viewed by the camera. So, if we orient two objects to same orientation as the world, then the two objects are synchronized – they have the same sense of up, down, left, right, forward, and backward.

The expanded menu selection for orient to is illustrated in Figure T-3-9. This shows the monkey being oriented to the world center. Of course, the same must be done for the ball. Then, the two objects will move in the same direction when a move forward instruction is given to each.
Figure T-3-9. Cascading menus for one-shot *orient to*