After finishing this chapter, you should be able to:

- Provide brief definitions of the following terms: array, Array Visualization Object, binary tree, data structure, index value, iterate a list, list, matrix, node, queue, root node, and vector
- Describe what a data structure is, and give several examples of data structures
- Generally describe why there are so many different data structures, and how programmers decide what data structures to use for different programs
- Describe the simple data structure known as a list, and how it is implemented in Alice
- Describe the data structure known as an array, how it differs from a list, and how it is implemented in Alice
- Create a list of objects in an Alice world and methods that perform operations on the list items one at a time and all at once
- Create methods in Alice that can manipulate the parts of objects contained in a list
- Describe the purpose of the Array Visualization Object in Alice
DATA STRUCTURES IN ALICE

A data structure is a scheme for organizing data in the memory of a computer. A set of names, addresses, and phone numbers stored as a table of data is an example of a data structure. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs. They can be simple, or they can become quite complex.

THE NEED FOR DIFFERENT DATA STRUCTURES

Computer programmers decide which data structures to use based on the nature of the data and the operations that need to be performed on that data because the way in which the data is organized affects the performance of a program for different tasks. As an example of this, let’s take a look at two data structures, a queue and a binary tree, and see how the differences between the two affect how computer programmers use them.

A queue is a set of data items with a beginning and end, called the front and back of the queue. Data enters the queue at one end and leaves at the other. Because of this, data exits the queue in the same order in which it entered the queue, like people in a checkout line at a supermarket. A queue has many uses in the world of computers—it would, for example, be a good data structure to use in a program for keeping track of documents waiting to be printed on a network printer, as shown in Figure 8-1.

A binary tree is a data structure that looks like an upside down tree. Each spot on the tree, called a node, holds an item of data, along with a left pointer and a right pointer, as shown in Figure 8-2. The pointers are lined up so that the structure forms the upside down tree, with a single node at the top, called the root node, and branches increasing on the left and right as you go down the tree. The nodes at the bottom of each branch, with two empty pointers, are called the leaf nodes.
A binary tree is used when it is necessary to keep a set of data items sorted in a particular order and quickly find items in the set. The middle item from the set of data is put in the root node, with anything before the middle item stored in the left branch of the tree, and anything after it stored in the right branch of the tree, as shown in Figure 8-3. It is good for storing information that needs to be searched quickly, such as a dictionary or phone directory.

By comparing the queue with the binary tree, you can see how the structure of the data—in other words, the way in which the data is organized—affects what can be done efficiently with the data. The queue works well for keeping track of documents waiting to be printed.
on a networked printer because it will keep them in the order in which they were sent to
the printer. The first document sent to the print queue will be the first document to leave
the queue and be printed, the second document will be printed second, and so on.

On the other hand, the queue is not a good data structure to use for a set of objects that needs
to be searched quickly. To find an item in the queue, you need to start at one end of the
queue and go through each item in the queue one at a time until you find the item for which
you are searching. If the object of your search is the first item in a queue of 1,000 items, then
you will only need to look at one item to find it; whereas if it happens to be the last item,
then you will need to look at all 1,000 items before finding it. On the average, you will need
to look at 500 items to find a single object in a queue with 1,000 items.

Now let’s look at searching a binary tree. You always start searching the binary tree by look-
ing at the root node. If it does not contain the data item you want, then you follow one of
the pointers to either the node to the left of the root node, or the node to the right of the root
node, depending on how the item for which you are searching compares to the item in the
root node. For example, if you were searching by name for Alice, and the root node had an
object named Cheshire Cat, then you would know to go to the left branch of the tree to find
Alice since it comes before the Cheshire Cat. At the next node you would do the same
thing—if the node does not contain Alice, then you go to that node’s left branch or right
branch, and so on until you find Alice. The efficiency of the binary tree comes from the fact
that each time you look at a node, you either find the item you want, or move down one
level in the tree, which cuts the remaining data to be searched in half, since you now only
need to look to the left or to the right of the item you want. With a set of 1,000 objects, you
could always find an object in the tree in 10 tries or less.

If it takes 1/100th of a second to look at an item, then finding an item in a queue of 1,000 items
could take any where from 1/100th of a second if it is the first item, to 10 seconds if it is the
last item. In a binary tree with a thousand items, the longest it would take to find any item is
1/10th of a second. So, you can see that a binary tree is a much better data structure to use in a
situation where objects need to be retrieved quickly.

So, if a programmer needs to organize data so that the first item placed in a data structure
will be the first item to leave the data structure, then a queue is a quick simple way to do
this, but if a programmer wants to organize data so that any item can be retrieved quickly,
then a more complex binary tree would be better. A programmer chooses a data structure
based on how the data will be used. Someone who wants to become an effective profes-
sional programmer must become familiar with many different data structures and their
characteristics.

It’s impossible to learn all about data structures in an introductory course in programming,
although you can start to learn about the simple and commonly used data structures known
as lists and arrays, which are the focus of the remainder of this chapter.
LISTS IN ALICE

One of the simplest of all data structures is a list, which is an ordered set of data. It is often used to store objects that are to be processed sequentially, meaning one at a time in order. In fact, lists are often used to set up queues like the print queue described above.

Alice has a number of instructions that can be used to manipulate lists, some of which are shown on the menu in Figure 8-4. You can see that there are commands to insert or remove items from the beginning of the list, the end of the list, or at any position in the list, which in this example is named world.bunnies.

Alice also has two instructions among those at the bottom of the Editor area to manipulate lists—For all in order and For all together, as shown in Figure 8-5. For all in order will perform an operation on each item in a list one at a time, beginning with the first item in the list and going through the list in order. To iterate a list means to go through the list in this manner.

For all together will perform an operation on all of the items in a list at the same time. Notice how For all in order and For all together are different from the Do in order and Do together instructions that you saw in earlier chapters. Do in order and Do together operate on a set of instructions. For all in order and For all together operate on a set of objects.
course, the *Do* and *For all* instructions can be combined to perform sets of instructions on sets of objects.

Figure 8-6 shows a series of screen shots from an Alice world window capturing the operation of *For all in order* on a list of cars. You can see that the cars each turn and begin to move away one at a time in order. The same instructions are being executed one at a time by each object in the list; the first car turns and moves, then the second car turns and moves, and so on.

**FIGURE 8-6:** The operation of *For all in order*

```plaintext
For all CARS in order
{
  pull out
  turn right
  move forward
}
```

Figure 8-7 shows a similar series of screen shots capturing the operation of *For all together* on the same list of cars. You can see that the cars each turn and begin to drive away all at the same time. In Tutorial 8A you will create a list containing a group of toy soldiers, and use *For all in order* and *For all together* to make the soldiers perform a drill routine.
**ARRAYS IN ALICE**

An array is a set of indexed variables, each containing objects of the same data type. For example, if we wished to store a list of five phone numbers for use by a computer program, we might call them phone\[0\], phone\[1\], phone\[2\], phone\[3\], and phone\[4\]. The value in brackets after each occurrence of phone is the index value for that item. There is not one phone variable, but several phone variables, each identified by its unique index value. Notice that the index values start with 0 rather than 1. This is true in almost all programming languages.

In mathematics, especially matrix algebra, index values for elements in an array are usually subscripted—which means they are placed half a line below the other characters in the name of the variable, much like the 2 in H\(_2\)O, the chemical formula for water. In many programming languages, however, they are simply placed in brackets. An array of the names of four cities might be referred to as city\[0\], city\[1\], city\[2\], and city\[3\], or as city[0], city[1], city[2], and city[3]. Sometimes both brackets and subscripting are used. In any case, such variables are often referred to as subscripted variables, even when brackets are used without true subscripting.

At first glance, it might seem that there is little difference between an array and a list, but actually, a list is a set of objects, while an array is a set of variables that hold objects. It is as if a list were a collection of things, and an array were a collection of boxes that hold things. If we remove an item from the middle of a list, then the list simply gets smaller by one object; whereas, if we remove an item from a “box” in an array, then that “box” is simply empty. Figure 8-8 shows this. If we remove the second item in a list, then the old third item becomes the new second item, and so on for all of the rest of the items in the list. If we remove the second object from an array, then the second “box” in the array is simply empty. The third item will remain in the third “box,” and so on throughout the array.
There is one other crucial difference between an array and a list: a list is linear, while an array may be multi-dimensional. To say that a list is linear means that we can think of it as a straight line. It has one item, then a second, then a third, and so on, as if in one continuous straight line. To say that an array has more than one dimension means that each object in an array could have more then one subscript, referring to its location in the array in different dimensions. You might recall from chapter two that a dimension is simply a way of measuring something, such as the length, width, and height of three-dimensional objects. A two-dimensional array, for example, could be thought of as having rows and columns, with one subscript for the row and one for the column. Such a two dimensional array is sometimes called a table or a two-dimensional matrix. A simple one-dimensional array is sometimes called a vector.

You won’t be working with any two-dimensional arrays in Alice, but in Tutorial 8D, you will look at a world that sorts the objects in a simple linear array. The world uses a special object in Alice called an Array Visualization Object, which is used to show us an array in an Alice world, rather than just creating it in the memory of the computer.

**TUTORIAL 8A—EIGHT BALLERINAS**

In this tutorial you will create a list with eight ballerinas in the list, and then write a method to make the ballerinas perform movements individually and all at once. The purpose of the exercise is to learn how to create lists in Alice and how to use the *For all in order* and *For all together* instructions tiles.
EXPLORING THE BALLERINA MOVEMENT METHODS

You will begin by opening a world that already has eight ballerinas in it, with methods to enable the ballerinas to perform some individual movements. You will explore the methods this world contains before creating and working with a list of the ballerinas.

1. Start the Alice software and open the eight ballerinas world from the CD that comes with this book.

2. Notice that the world contains eight ballerinas, whose names can be seen in the object tree: Bronwyn, Ava, Addie, Mardi, Evelyn, Daphne, Kristen, and Meagan. The opening scene from the world is shown in Figure 8-9. Click the world tile in the Object tree and the methods tab in the Details area, and you will see that in addition to my first method, there are generic instructions to make any ballerina jump, put its arms up or down, spin, jump and move at the same time, and bow. Before creating a list of the ballerinas, you will try each of these methods with different ballerinas.

3. Click the edit button next to the my first method tile, and you will see world.my first method appear in the Editor area.

4. First, drag a copy of the jump who tile from the methods tab into world.my first method in the Editor area. When the menus appear, select Bronwyn, the entire Bronwyn.
5. Next, drag a copy of the arms up who tile into world.my first method and drop it below the world.jump who = Bronwyn instruction tile. When the menus appear select Ava, the entire Ava.

6. Now, drag a copy of the arms down who tile and drop it below the world.arms up who = Ava instruction tile in world.my first method. When the menus appear again, select Ava, the entire Ava.

7. Drag a copy of the spin who tile into world.my first method and drop it below the world.arms down who = Addie instruction tile. When the menus appear, select Addie, the entire Addie.

8. Drag a copy of the jumpMove who direction tile into world.my first method and drop it below the world.spin who = Daphne instruction tile. When the menus appear, select Daphne, the entire Daphne, and left as the direction.

9. Drag another copy of the jumpMove who direction tile into world.my first method and drop it below the first world.jumpMove instruction tile. When the menus appear again, select Daphne, the entire Daphne, but this time choose right as the direction.

10. Finally, drag a copy of the bow who tile into world.my first method and drop it below all of the other instruction tiles. When the menus appear, select Mardi, the entire Mardi.

11. Now world.my first method contains an example of each of the instructions for the individual movements for a ballerina. Save the world with the name ballerina movements, and then play it to see what each of the movements looks like. You may want to try it a few times or to experiment with the various instructions for the ballerinas before continuing.

**CREATING A LIST OF THE BALLERINAS**

Your next task is to create a list containing the eight ballerinas so that you can try the For All in order and For all together instructions. You won’t need the sample instructions you just tried, so they can be discarded. It’s actually easier to restart the eight ballerinas world than to delete each of the instructions in world.my first method. You will begin by reopening the eight ballerinas world and creating a list of the ballerinas.

1. Close and reopen the Alice software. This will ensure that memory is clear before creating the new world. If the system warns you that the world has been modified and asks you if you want to save it, select no.

2. Reopen the original eight ballerinas world from the CD provided with this book.
3. Save the world with the name ballerina company so that your changes will not affect the original eight ballerinas world.

4. Select the world tile in the Object tree and the properties tab in the Details area.

5. Click the create new variable button on the properties tab and the create new variable dialog window should appear. Type company as the name, select object as the type, and make sure that the make a option is checked and list is selected in the values section of the window. Do not click the OK button at this time.

6. Next, you will add the ballerinas to the new list. Click the new item button in the create new variable dialog window, and a line for item 0 should appear just above the button, as shown in Figure 8-10.

7. Click the word None next to item 0, and select Bronwyn, then the entire Bronwyn from the menus that appear.

8. Click the new item button again, and a line for item 1 will appear. This time, click the word None next to item1, and select Ava, then the entire Ava from the menus that appear.

9. In a similar manner, add Addie as item 2, Mardi as item 3, Evelyn as item 4, Daphne as item 5, Kristen as item 6, and Meagan as item 7.
10. When you are finished, click the OK button in the create new variable dialog window, and a new tile for the company list should appear in the properties tab. You now have a list containing the eight ballerinas.

11. Save the world with the name **ballerina company** before continuing.

**CREATING A DANCE ROUTINE FOR THE BALLERINAS**

Now that you have a list with the eight ballerinas in the list, you can place some instructions in `world.my first method` to create a routine for the ballerina company. The routine will be a fairly simple one, just enough for you to learn how to use the *For all in order* and *For all together* instructions.

One at a time, each of the ballerinas will spin while saying her name, then the ballerinas will perform a few movements together. When they are finished, each will bow, and then the entire company will bow together.

1. Click the **methods** tab in the Details area, and the blank `world.my first method` should appear in the Editor area. If it does not appear, then click the edit button next to **My first method** tile in the Details area. If it is not blank, then delete any instructions it contains.

2. First, each of the ballerinas will do something one at time, so drag a copy of the *For all in order* tile from the bottom of the Editor area and drop it into `world.my first method` in place of *Do Nothing*. When the menus appear select **expressions** and then `world.company`. An instruction tile will appear in `world.my first method` that says *For all world.company, one item_from_company at a time*.

3. Each ballerina will do two things together – say her name and spin—so drag a copy of the *Do together* tile from the bottom of the Editor area and drop it in the *For all* instruction tile in place of *Do Nothing*.

4. Now an instruction needs to be added to make the ballerina say her name. Figure 8-11 show what this instruction will look like when it is complete. Drag the **one item_from_company** object tile from the *For all world.company, [obj] one item_from_company at a time* tile, and drop it into the *Do together* tile in place of *Do Nothing*.

**FIGURE 8-11:** Instructing a ballerina say her name

![Instruction tile](image-url)
5. A menu will appear allowing you to select which of the primitive methods you wish to have the item_from_company perform. Select item_from_company say, and then hello.

6. However, item_from_company should say its own name, not hello. You must use a function to do this. Select the functions tab in the Details area, find what as a string, then drag and drop a copy of it in the item_from_company say hello instruction in place of the word hello.

7. When the menu appears, select expressions, then item_from_company. Now the instruction looks as it should.

8. Save the world again and test it before continuing. Each ballerina should say her own name in turn.

If everything works okay so far, then you can continue. If not, then you need to find and fix the error before continuing. Once everything is okay you can proceed with creating the dance routine. To do this, you need to add the spin instruction to the Do together tile; then you can start adding the dance routine.

1. Click the methods tab in the Details area so that you can again see the list of generic methods in the world.

2. Drag a copy of the spin who tile from the methods tab and drop it below the item_from_company say instruction in the Do together tile in world. My first method. When the menu appears, select expressions, then item_from_company.

3. A spin takes two seconds to complete, so, to synchronize the spin and say instructions, click word more the item_from_company say instruction tile and set the duration to 2 seconds.

4. Test the world again, and this time each ballerina should say her name while spinning. If it looks correct, then save the world again.

Next you will add instructions to world.my first method to create the dance routine. All of the ballerinas will perform the routine together, so the instructions will be contained in a For all together tile. The ballerinas will jump, spin, jump left, jump right, and then spin again. After they are finished, they will each bow while saying their names, and then the company will bow together.

1. Drag a copy of the For all together tile from the bottom of the Editor area and drop it into world.my first method below the For all world.company, one item_from_company at a time tile. When the menus appear, select expressions, and then world.company. An instruction tile will appear in
world.my first method that says For all world.company, every item_from_company together.

2. Drag a copy of the jump who tile from the methods tab and drop it in the For all world.company, every item_from_company together tile in place of Do nothing. When the menu appears, select expressions, then item_from_company.

3. Drag a copy of the spin who tile from the methods tab and drop it in the For all world.company, every item_from_company together tile below the world.jump who = item_from_company instruction. Again select expressions, then item_from_company when the menu appears.

4. In a similar manner, add instructions to jump move left, jump move right, and then spin again.

5. Save the world before continuing.

Now you can add the instructions to make the ballerinas each say their names and bow at the end of the routine. They will drop their arms after bowing, and then all bow together.

1. Drag a For all in order tile from the bottom of the Editor area and drop it into the bottom of world.my first method after all of the other instruction tiles in the method. When the menu appears, select expressions and item_from_company.

2. Drag the second object tile that says one item_from_company from the top of the new For all world.company, one item_from_company at a time tile, and drop it into the tile in place of Do Nothing.

3. Select item_from_company say, and then hello from the menus that appear.

4. Select the functions tab in the Details area, and then scroll through the functions until you find the what as a string function. Drag a copy of it from the functions tab and drop it in the item_from_company say hello instruction in place of the word hello. When the menu appears, select expressions, then item_from_company. The say instruction is now complete.

5. Select the methods tab in the Details area and drag a copy of the bow who instruction from the methods tab and drop it just below the item_from_company say instruction tile. When the menu appears, select expressions, then item_from_company.

6. Drag a copy of the arms down who instruction from the methods tab and drop it below the For all world.company, one item_from_company at a time instruction tile. When the menu appears, select expressions, then item_from_company.
Finally, the last movement that needs to be added to the routine is the company bowing together.

1. Drag a copy of the For all together tile from the bottom of the Editor area and drop it into world.my first method below all of the other instructions. When the menus appear, select expressions, and then world.company.
2. Drag a copy of the bow who instruction from the methods tab and drop it in the new For all world.company tile in place of Do Nothing. When the menu appears, select expressions, then item_from_company.
3. That’s it! Save the method, and then play the world to see what happens. If everything is correct, the ballerinas should each say their name while spinning, complete several moves together, bow individually while saying their names, and then bow together. If they do not, then find and fix the error before continuing.

This concludes the exercise. You should now know how to create a list and how to use the For all in order and For all together instructions to perform operations on lists.

**TUTORIAL 8B—MARCHING TOY SOLDIERS**

In this tutorial you will work with a list of toy soldiers to make them complete a marching drill routine. You will use the For all in order and For all together instructions to make the soldiers complete marching maneuvers sequentially and concurrently in the routine.

**THE TOY SOLDIERS WORLD**

On the CD that comes with this book, there is a toy soldier world containing a list of four toy soldiers with methods to make each of them march. You probably could create such a world on your own, but it is a tedious process, so you will start with a world in which this has already been done. You will start by exploring the existing world to become familiar with its features.

1. Start the Alice software and open the toy soldiers world from the CD that accompanies this book. When the world loads you should see a squad of four toy soldiers on the screen, as in Figure 8-12. The first toy soldier has black pants, and the rest all have blue pants. Perhaps the first toy soldier is an officer. You should also notice that there are tiles for four toy soldiers in the Object tree, named toySoldier1 through toySoldier4.
2. Before doing anything else, save the world with the name **toy soldiers marching** so that your changes do not affect the original toy soldiers world.

3. Click the **properties** tab in the Details area, and you will see that there are two variables: a list of objects named `squad`, and a Boolean variable named `marching`, which is initialized to **true**, as in Figure 8-13.

4. Click the button after the equals sign following the `squad` tile on the properties tab, and you will see the Collection Editor window open, as shown in Figure 8-14. The Collection Editor window shows that the list contains the four toy soldiers, with `toy soldier1` as the first element, and so on up to `toy soldier4` as the last element in the list. Click the **OK** button to close the Collection Editor window.
5. Look in the Events area and you will see that there is an event to run the method `world.squadMarch` while the world is running.

6. Click the `methods` tab in the Editor area and then the `edit` button next to `squadMarch` to see what this method does. If you look at the method in the Editor area, you will see that if the Boolean variable `world.marching` is true, then each of the four toy soldiers will march, and if it is not true, then nothing happens—in other words they will not march. `world.marching` is a state control variable, which controls the status of the toy soldiers. When it is true, they will be marching; when it is false they will not be marching.

7. Since the Boolean variable `marching` is initialized to `true`, the squad should march if the world is played. Play the world and you should see the soldiers march off the screen. Once they have marched away, you can stop the world.

8. From now on, the soldiers should not march away when the world starts, so click the `true` button next to the `marching` tile on the world’s properties tab in the Details area, and change it to `false`.

**CREATING A MARCHING ROUTINE**

You are now ready to create a method to make all of the soldiers complete a marching drill routine. They will start marching, then after two seconds they will turn right one at a time, then after another two seconds, they will turn right all at once. This process will be repeated four times, which should create an interesting marching routine using only a few simple instructions. Figure 8-15 shows what this method will look like when it is finished.
1. Click the `world` tile in the Object tree, and then the `methods` tab in the Details area.

2. Click the `create new method` button on the methods tab, type the name `routine` in the new method dialog window that appears, and then click the `OK` button.

   Click the `properties` tab and drag a copy of the `marching` Boolean variable tile from the properties tab and drop it into the `world.routine` method in place of `Do Nothing`. Select `true` from the menu that appears. Setting the `marching` variable to `true` will have the affect of a “Forward, march!” command and will start the soldiers marching when the routine is run.

3. The routine will contain a process to be repeated four times, so drag a copy of the `loop` instruction from the bottom of the Editor area and drop it in the `world.routine` method below the `world.marching set value to true` instruction. Select `other` from the menu that appears and enter the value `4` using the calculator style keypad that appears, and then click the `Okay` button.

4. Drag a copy of the `Wait` instruction from the bottom of the Editor area and drop it into the `loop` instruction in place of `Do Nothing`. Set the duration for the `wait` to `2 seconds.`
5. Drag a copy of the **For all in order** tile from the bottom of the Editor area and drop it into the *loop* instruction below the *wait* instruction. When the menus appear, select *expressions*, and then *world.squad*.

6. The new instruction now says *For all world.squad one [obj] item_from_squad at a time*. Drag a copy of the *item_from-squad* parameter in this instruction and drop it in the same instruction in place of *Do Nothing*.

7. When the menus appear, select *item_from_squad turn*, then *right*, then *¼ revolution*.

8. Next right-click the *wait* tile in the *world.routine* method and select **make copy**. Move the copy of the *wait* instruction to the end of the *loop* instruction, inside the *Loop* tile but below the *For all* tile.

9. Drag a copy of the **For all together** tile from the bottom of the Editor area and drop it into the *loop* instruction below the second *wait* instruction. When the menus appear, select *expressions*, and then *world.squad*.

10. The new instruction now says *For all world.squad every [obj] item_from_squad together*. Drag a copy of the *item_from-squad* parameter in this instruction and drop it in the same instruction in place of *Do Nothing*.

11. When the menus appear, select *item_from_squad turn*, then *right*, then *¼ revolution*.

12. Finally, drag a copy of the **marching** Boolean variable tile from the properties tab and drop it into the *world.routine* method at the very bottom, below the *loop* instruction tile. Select *false* from the menu that appears. This will have the affect of a “Halt” command and will stop the soldiers from marching.

13. Your new method is now complete. Save the world again before continuing.

Finally, to make the soldiers carry out the routine when the world starts, you need to put a *routine* instruction in *world.my first method*. It would also be better if the soldiers turned right before starting their routine.

1. Click the *world.my first method* tab in the Editor area and the method should become visible. It should be blank except for *Do Nothing*.
2. Drag a copy of the *routine* tile from the methods tab in the Details area and drop it into *world.my first method* in place of *Do Nothing*.
3. Save the world again before continuing.
4. Play the world and watch what happens. The soldiers should complete two circuits of their routine before stopping. Each iteration through the loop in the *routine* method is only ⅔ of the routine’s circuit, which is why they will complete two circuits when the loop is set to repeat four times. If the method does not perform as expected, then find and fix any errors before continuing.
TUTORIAL 8C—SALUTING TOY SOLDIERS

In this tutorial you will create a method to make the toy soldiers from the previous tutorial salute, which will provide you with practice in manipulating the parts of objects that are contained in a list. Manipulating the parts of an object can be the most tedious part of programming objects in three-dimensional virtual worlds like Alice, but the result is that the objects function more like similar items in the real world.

You will actually create two methods. First, you will create a generic salute method to make any soldier salute; then you will use the For all together instruction to create a method to make the entire squad salute. It is necessary to use several move and turn instructions to manipulate the right arms of the toy soldiers to make them salute.

CREATING A GENERIC SALUTE METHOD

To salute, a soldier needs to raise its arm into a saluting position, then drop the arm back to its original position. You will first create a generic method with a who parameter to enable any soldier to salute.

1. If necessary, start the Alice software and open the toy soldiers marching world that you saved in the previous tutorial, or open the world from the CD that accompanies this book. When the world loads, you should see a squad of four toy soldiers on the screen, as shown back in Figure 8-12.
2. Save the world with the name toy soldiers salute before continuing so that you do not change the existing toy soldiers marching world.
3. Click the create new method button on the methods tab in the world’s Details area, type the name salute in the New Method dialog window that opens, and then click the OK button. A new method named world.salute will appear in the Editor area.
4. Click the create new parameter button at the top of the new method in the Editor area, and the Create New Parameter window will open. Type the name who, make sure that Object is selected as the parameter type, and then click the OK button.

Figure 8-16 shows what the first part of the salute method will look like. This is the part that will cause the soldier to raise his arm into a saluting position.
You can see that three things need to happen to make the soldier salute:

- The right forearm needs to roll right .2 revolutions.
- The right forearm needs to turn backward .3 revolutions.
- The entire right arm needs to turn backward .2 revolutions.

The numeric values for the amounts of these movements were determined through calculation combined with trial and error. All three of these instructions need to be carried out at the same time, so they will be placed inside a Do together tile. Since the method will be a generic method to make any soldier salute, you will also need to use an object parameter in the method to indicate which soldier should salute.

There is one complication that you will encounter in creating the salute method. To manipulate parts of an object in a generic method, such as a soldier’s arm or forearm, it is often necessary to use a function that points to the object’s part by name. The two parts you need to manipulate are the rightArm, and the rightArm.forearm. When the time comes below to use these, you will need to type in their names, so be careful: The capitalization and spelling are important.

1. Drag a Do together tile from the bottom of the Editor area and drop it into the world.salute method in place of Do nothing.

2. Drag a copy of the who parameter tile from the top line of the world.salute method and drop it in the Do together tile in place of Do Nothing.

3. When the menus appear, select world.Salute who roll, then right, then other. Use the calculator style keypad to enter .2, and then click the Okay button.

4. The last instruction in the method now reads who roll right .2 revolutions. This needs to be modified, because you do not actually want who to roll, but who’s forearm to roll. You will now need to use the function that points to an object’s part. To get to this function, you will need to look at the functions tab for one of the toy soldiers.
1. Click the `toySoldier1` tile in the Object tree and then the `functions tab` in the Editor area.

2. Scroll through the functions until you find the `toySoldier1's part named key function` tile. It is near the bottom of the functions tab. Drag the tile into the Editor area and drop it in the `who roll right .2 revolutions` instruction in place of `who`.

3. Now click the `toy soldier1` parameter just before the words `part named`, and select `expressions`, then `who` from the menu that appears.

4. Finally, click the empty white box just after `part named` in the same instruction, select `other` from the menu that appears, and the Enter a string dialog window will appear. Carefully type `rightArm.forearm` in the box, and then click the `OK` button. The instruction should now look like the roll instruction in Figure 8-17.

Now you will add the instruction to make the forearm turn backward .3 revolutions.

1. Drag a copy of the `who parameter` tile from the top of the `world.salute` method and drop it in the `Do Together` tile just below the `roll` instruction.

2. When the menus appear, select `world.Salute who turn`, then `backward`, then `other`. Use the calculator style keypad to enter .3, and then click the `Okay` button.

3. You need to again use the part name function, but this time you can copy it from the previous instruction. Drag a copy of the purple box that says `who's part named rightArm.forearm from the roll` instruction and drop it on the clipboard; then drag it from the clipboard and drop it into the `who turn` instruction at the bottom of the `Do together` tile in place of `who`. Your turn instruction should now look like the turn instruction in Figure 8-18.
Next, you will add the instruction to make the entire arm turn backward 0.2 revolutions. You will do this by copying and modifying the instruction that you just completed.

1. Drag a copy of the `who's part named rightArm.forearm turn backward .3 revolutions` instruction to the clipboard; then drag it from the clipboard and drop it into the bottom of the `Do together` tile as the last instruction in the tile.

2. Click the `rightArm.forearm` box in the last instruction and select `other` from the menu that appears. Carefully type `rightArm` in the Enter a string dialog window that appears, and then click `OK`.

3. Click the `.3 revolutions` box in the last instruction and select `.2 revolutions` from the menu that appears. Your method should now look like Figure 8-19.

![Figure 8-19: The first part of the salute method](image)

The `salute` method is not finished yet, but you can unit test it to see what it does so far. It should cause a soldier to raise its arm into a saluting position. To test the method, you will need to modify `world.my first method`, and then play the world. `World.my first method` currently contains the `routine` instruction, which the soldiers carry out in their marching routine. You will temporarily disable the `routine` instruction to unit test the `salute` method.

1. Click the `world` tile in the object tree and then the `methods` tab in the details area.

2. Click the `world.my first method` tab in the Editor area, and the method should become visible.

3. Right-click the `world.routine` instruction tile and select `disable` from the menu that appears.

4. Drag a copy of the `salute who` method tile from the methods tab and drop it into `world.my first method` in the Editor area above the `routine` instruction so that it becomes the first instruction in the method.

5. Select `toySoldier1`, the entire `toySoldier1`, from the menus that appear.
6. Now play the world. The first soldier should raise his arm into a saluting position. If it does not, then check your work and fix the problem in the `salute` method. If the method runs but the salute looks funny, then you should check the direction and amount of the parameters in the `turn` and `roll` instructions in the `salute` method. If you get a “subject must not be null” error message, then the name of one of the parts you typed as the `part named` parameters in the `turn` and `roll` instructions may be wrong.

7. Save the world again before continuing.

The method makes the toy soldier raise its arm into a saluting position, but it also needs to make it drop its arm. To add this to the method, you will simply make a copy of the `Do Together` tile in the `salute` method, and then reverse the direction of each of the arm movements.

1. Click the `world.salute` tab in the Editor area.
2. Drag a copy of the `Do Together tile from the world.salute` method in and drop it on the clipboard, then drag it from the clipboard and drop it into the `world.salute` method below the original `Do Together` tile.
3. Now, one at a time, click each of the direction parameters in the three instruction tiles inside the bottom `Do Together` tile and reverse the direction. Change `right` to `left` in the first tile, and `backward` to `forward` in each of the other two tiles. When you are finished, the second `Do together` instruction should look like Figure 8-20.

4. Now the soldier should raise its salute, and then drop its salute. Play the world again to see if it works, fix any errors you encounter, and then save the world again before proceeding.

**MAKING ALL OF THE SOLDIERS SALUTE**

Now that you have a generic method to make any soldier salute, you need to create a method to make the entire squad of soldiers salute together. To do this, you will create a
squadSalute method that contains the generic salute method inside a For all together instruction for the squad of soldiers. Your finished method should look like Figure 8-21.

![Figure 8-21: A method to make all of the toy soldiers salute](image)

1. Click the create new method button on methods tab in the world’s Details area.
2. Type the name squadSalute in the New Method dialog window that opens, then click the OK button. A new method named world.squadSalute will open in the Editor area.
3. You will use a For all together instruction to have the soldiers all salute at the same time. Drag a copy of the For all together tile from the bottom of the Editor area and drop it in the squadSalute method in place of Do Nothing.
4. When the menus appear, select expressions, and then world.squad.
5. Drag a copy of the salute who tile from the methods tab and drop it into the For all tile in place of Do Nothing.
6. When the menus appear, select expressions, then item_from_squad.
7. The squadSalute method is finished and should resemble Figure 8-21. Save your world before continuing.

You are now ready to test the world.squadSalute method.

1. Click the world. my first method tab in the Editor area to see the method. The method still contains the instruction world.salute who = toySoldier1 and the disabled routine instruction.
2. Drag a copy of the squadSalute method from the Editor area and drop it into world. My first method between the two instructions.
3. Now play the world to see if it works as expected. If it does, then the first soldier should salute, followed by all of the soldiers saluting together.
The world would look nicer if the soldiers turned toward the camera before saluting, then turned back again before starting their marching routine. First, you will add the instruction to make the soldiers turn left before saluting.

1. Drag a copy of the For all together tile from the bottom of the Editor area and drop it into world.my first method above the existing instructions so it becomes the first instruction in the method.
2. When the menus appear, select expressions, and then world.squad.
3. Now drag a copy of the item_from_squad object parameter after the word every from the new For all together instruction and drop it into the same For all together instruction tile in place of Do Nothing.
4. Select item_from_squad turn, then left, and then $\frac{1}{4}$ revolution from the menus that appear.

Next, you will add the instruction to make the soldiers turn right after saluting.

1. Drag a copy of the For all together tile from the bottom of the Editor area and drop it into world.my first method between the squad salute instruction and the disabled routine instruction.
2. When the menus appear, select expressions, and then world.squad.
3. Now drag a copy of the item_from_squad object parameter after the word every from the new For all together instruction and drop it into the same For all together instruction tile in place of Do Nothing.
4. Select item_from_squad turn, then right, and then $\frac{1}{4}$ revolution from the menus that appear.

Finally, you need to enable the disabled routine instruction in world.my first method.

1. Right-click the disabled world.routine instruction and select enable from the menu that appears. world.my first method is now complete and should resemble Figure 8-22.
2. Save the world again, and then play the world to see if it performs as expected. The soldiers should turn; the first soldier salutes, all soldiers salute, and then they turn back and complete their routine. If the world does not perform as expected, then find and fix any errors before continuing.

If you would like, you could copy and paste from the existing instructions in `world.my first method` to make the soldiers turn and salute again when they are finished their marching routine.

**TUTORIAL 8D—SORTING AN ARRAY OF SIXTEEN BALLERINAS**

This tutorial is a little different from the others because you will not create any new worlds but simply examine a world from the CD provided with this book. The world, named sixteen ballerinas, has an array that contains sixteen ballerinas. The array exists as an Array Visualization Object in Alice, which will let you see the array as a set of adjacent boxes in the Alice world much like the way an array would be stored as a set of consecutive locations in a computer’s memory.

The array has room for sixteen elements, with one ballerina stored in each spot in the array. The ballerinas are different sizes, and the world contains a method to sort the ballerinas in the array according to their height. The sorting method uses the bubble sort technique, which is not the most efficient sort, but it is fairly easy to understand.
1. Start the Alice software and open the world named **sixteen ballerinas** that is on the CD supplied with this book.

2. You will see that the world has a set of sixteen ballerinas, all lined up in separate “boxes” in an array. The ballerinas are all different heights, as seen in Figure 8-23.

3. Play the world, and you will see that the ballerinas are sorted into order according to their heights.

4. Click the **array** tile in the Object tree and the **methods** tab in the Details area. You will see that there are two different user-created methods for the array—**swap** and **bubbleSort**.

5. Click the **edit** button next to the **swap** method; the method becomes visible in the Editor area. There are two spots in the method where the **duration** is set to .1 seconds. Changing these to a larger number will slow down the sort.

6. Click the **edit** button next to the **bubbleSort** method; the method becomes visible in the Editor area.

Notice that there are two loops in the **bubbleSort** method, a **while** loop and a **count-controlled** loop. The count controlled loop goes through the array once, comparing each element in the array to the next element. This is done by using the index variable from the loop to refer to the element in the array. **array[index]** and **array[index +1]** are adjacent elements in the array. If **index** is 2 for example, then **index +1** will be 3, and the two elements being compared will be **array[2]** and **array[3]**. If the two elements are out of order, then they are swapped. Each time through the loop, comparing each item to its adjacent item, is called a pass through the loop.
The method continues to make passes though the loop until no more swaps occur. The Boolean variable called *changed* is used to keep track of when a swap occurs. It is set to *false* at the beginning of the While loop, and is only reset to *true* if a swap occurs when going through the inner count-controlled loop. This will cause the program to keep repeating passes through the list until no swap occurs; then the program stops.

In other words, here is how the bubble sort works: The computer goes through the array, comparing each value to the one that follows it. If they are out of order, then they are swapped; otherwise, they are left in place. The computer continues to make passes through the array until it makes one pass where no swaps occur. Once it can do this, the list is all in the correct order, and the sorting stops. As stated above, this is not the most efficient way to sort an array, but it is a reasonably easy to understand technique that correctly sorts an array.

If you look through the various parts of this world, you will see that it contains both an array and a list. The world is really just included for demonstration purposes so that you can see a program that operates on an array.

Close the Alice software when you are finished examining the *sixteen ballerinas* world.
CHAPTER SUMMARY

This chapter consisted of a discussion of data structures, followed by four tutorials. The discussion of data structures included the following:

- **A data structure is a scheme for organizing data in the memory of a computer. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.**

- **Alice has instructions that will allow you to manipulate two of the most basic data structures — lists and arrays.**

- **Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data because the way in which the data is organized affects the performance of a program for different tasks.**

- **A queue is a set of data items with a beginning and end, called the front and back of the queue. Data enters the queue at one end and leaves at the other. Because of this, data exits the queue in the same order in which it entered the queue, like people in a checkout line at a supermarket.**

- **A binary tree is a data structure that looks like an upside down tree. Each spot on the tree, called a node, holds an item of data along with a left pointer and a right pointer. The pointers are lined up so that the structure forms the upside down tree, with a single node at the top, called the root node, and branches increasing on the left and right as you go down the tree.**

- **By comparing the queue with the binary tree, you can see how the structure of the data affects what can be done efficiently with the data.**

- **One of the simplest of all data structures is a list, which is an ordered set of data. It is often used to store objects that are to be processed sequentially.**

- **Alice has a number of instructions that can be used to manipulate lists, including For all in order and For all together.**

- **For all in order will perform an instruction on each item in a list one at a time, beginning with the first item in the list and going through the list in order. This is known as iterating the list.**

- **For all together will perform an operation on all of the items in a list at the same time.**

- **An array is a set of indexed variables, each containing objects of the same data type.**

- **Each variable in the array is identified by its unique index value.**

- **A list is a set of objects, while an array is a set of variables that hold objects. It is as if a list were a collection of things, and an array were a collection of boxes that hold things.**
A list is linear, while an array may be multi-dimensional. A two-dimensional array, for example could be thought of as having rows and columns, with one subscript for the row and one for the column. A two dimensional array is sometimes called a table, or a two-dimensional matrix. A simple one-dimensional array is sometimes called a vector.

Alice has a special object called an Array Visualization Object, which is used to show us an array in an Alice world, rather than just creating it in the memory of the computer.

In Tutorial 8A you created a list of ballerinas and used the For all in order and For all together instructions to perform operation on the list.

In Tutorial 8B you again used the For all in order and For all together instructions to perform operations on a list of toy soldiers so they would complete a marching routine.

In Tutorial 8C you created a generic method that performs operations on the parts of objects contained in a list to make the toy soldiers salute.

Tutorial 8D demonstrated the use of an Array Visualization Object in a method that uses the bubble sort technique to sort a set of objects stored as an array.

REVIEW QUESTIONS

1. Define the following terms:
   - array
   - Array Visualization Object
   - binary tree
   - data structure
   - index value
   - iterate a list
   - list
   - matrix
   - node
   - queue
   - root node
   - vector

2. How do programmers decide what data structures to use for different programs?

3. Why is a queue a good data structure to use for keeping track of documents to be printed on a network printer?

4. Why would a binary tree be a good data structure to use for storing a dictionary on a computer?

5. What is the difference between a list and an array?

6. What is the role of index values in an array?

7. What does it mean to say that an array can be multi-dimensional?

8. What is the difference between using the Do together and the For all together instructions in an Alice world?

9. The swap method in Tutorial 8C contains an object variable called temp. Why is it needed in this method?
10. An encapsulated data structure is one which has functions available to programmers who use it, such as adding an element to the data structure, but the details of those methods are hidden from the user. In other words, they can use the methods, but they cannot see inside them. List several encapsulated methods that would be useful for all of the following data structures: a queue, a binary tree, a list, and an array. List several methods that would each be useful for one of the data structures but not necessarily the others.

EXERCISES

1. Create an Alice world with four helicopters and a list containing the helicopters. Program the world to make the helicopters each lift off from the ground one at a time, then all turn and fly away together.

2. The Animals folder in the Alice Local Gallery contains several different kinds of fish, including a goldfish and a shark. The Ocean folder contains an ocean floor background. Using these items, create an Alice world with a school of goldfish that swim around together, then scatter when a shark appears. You may be tempted to use the Lilfish in the Ocean folder instead of the goldfish, but be careful—each instance of the Lilfish requires more than 2 megabytes of memory, while each Goldfish requires only 30 kilobytes.

3. Create a drill routine of your own for the toy soldiers world from Tutorial 8B that demonstrates the use of the For all in order and For all together instructions along with good modular programming techniques.

4. Create an Alice world with several objects from the People gallery. Place some of the objects in one list and some of the objects in another list. Create a method to make each object in the first list say hello to each object in the second list, and vice-versa.

5. Create several new generic movement methods for the eight ballerinas world, then demonstrate their use in methods that use the For all in order and For all together instructions.

6. Create a series of keyboard and mouse events for either the eight ballerinas world that will allow the user to control a ballet routine, or for the toy soldiers world that will allow the user to control a marching drill routine while the world is running.

7. Open the generic triple jump world from the CD that accompanies this book and add a set of five different heights to the world as a list of numbers. Create a method to iterate through the list and have the three characters jump each of the heights contained in the list.

8. Create a search method for the sixteen ballerinas world that will do all of the following:
   a. allow the user to input a ballerina’s name,
   b. then go through the list to find that ballerina in the list that has that name
   c. have the ballerina respond by performing a dance maneuver.

   If the list does not contain a ballerina with the name which the user entered, then one of the ballerinas should tell the user that this is the case.
9. The Bugs folder inside the Animals folder in the Alice Local Gallery contains a Butterfly class of objects. Complete the following:
   a. Create an Alice world with six butterflies in the world.
   b. Create a generic method to make any butterfly flap its wings.
   c. Create a generic random movement method that will use random numbers to pick a direction and an amount, and then cause a butterfly to move accordingly.
   d. Place the six butterflies in a list, and create methods and events to randomly make all of the butterflies fly around gently in an Alice world, flapping their wings and moving about randomly.

10. The National Institute for Standards and Technology has a Dictionary of Algorithms and Data Structures on the Web at http://www.nist.gov/dads/. Look up the definitions for data structure, stack, and queue, and then describe the differences between a stack and queue.