

# Chapter 5

## Lists and Arrays

*He's making a list, checking it twice. 'Gonna find out who's naughty or nice ...*

SANTA CLAUS IS COMING TO TOWN

*For seven men she gave her life. For one good man she was his wife. Beneath the ice by Snow White Falls, there lies the fairest of them all.*

VIRGINIA (KIMBERLY WILLIAMS), IN *THE 10TH KINGDOM*

*The generation of random numbers is too important to be left to chance.*

ROBERT R. COVEYOU

*When a cat is dropped, it always lands on its feet, and when toast is dropped, it always lands with the buttered side down. I propose to strap buttered toast to the back of a cat; the 2 will hover, inches above the ground. With a giant buttered-cat array, a high-speed monorail could easily link New York with Chicago.*

JOHN FRAZEE

### Objectives

When you complete this chapter, you will be able to:

- ☐ Use a list to store multiple items
- ☐ Use Alice's **forAllInOrder** and **forAllTogether** statements
- ☐ Use random numbers to vary the behavior of a program
- ☐ Use an array to store multiple items

In the preceding chapters, we have often used variables to store values for later use. Each variable we have seen so far has stored a *single* value, which might be a **Number**, a **Boolean**, a **String**, an **Object**, a **Sound**, a **Color**, or any of the other types that Alice supports. For example, if we have three variable definitions like this in our program:

```
Number result = 0.0;
Boolean done = false;
String name = "Jo";
```

then we might (simplistically) visualize these three variables as shown in Figure 5-1.

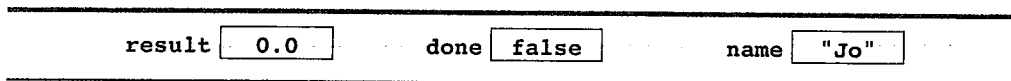


FIGURE 5-1 Storing three values in three variables

Each variable stores a single value (of a given type) that can be changed by the program.

It is sometimes convenient to be able to define a variable that can store *multiple* values. For example, suppose you have 12 songs (call them  $s_0 \dots s_{11}$ ) in your music player that you want to represent in a program. You could define 12 single-value variables (for example, **song<sub>0</sub>**, **song<sub>1</sub>**, **song<sub>2</sub>**, ..., **song<sub>10</sub>**, **song<sub>11</sub>**), but it would be more convenient if you could define one variable capable of storing all 12 songs, as shown in Figure 5-2.

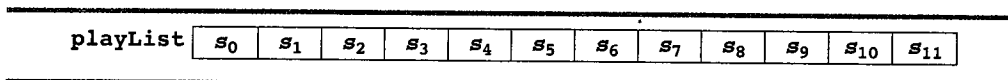


FIGURE 5-2 Storing 12 values in one variable

One advantage of this approach is that if I need to pass my song collection to a method, I only have to pass one argument (**playList**) instead of 12. Also, my method needs only one parameter.

A variable like this is called a **data structure** — a structure for storing a group of data values. In this chapter, we will examine two data structures that are available in Alice:

- The **list**, which stores a group of values where the group's size changes frequently
- The **array**, which stores a group of values where the group's size does not change

Each structure is used for storing *sequences* of values, but the two have very different properties.

## 5.1 The List Structure

### 5.1.1 List Example 1: Flight of the Bumble Bees

Suppose Scene 2 of a story requires a dozen bees to take off, one at a time, to defend the honor of their queen bee. We might begin by using the Alice Gallery to build the scene as shown in Figure 5-3.

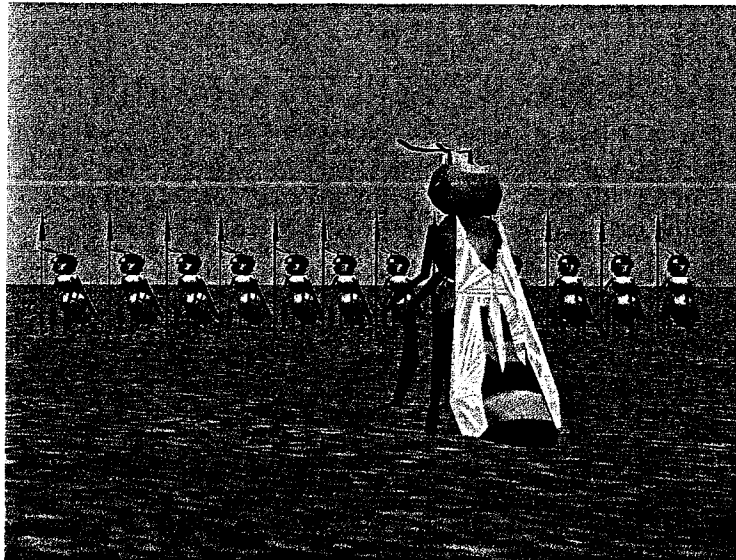


FIGURE 5-3 The queen and her 12 bees

To make the bees take off one at a time, we could use 12 separate statements:

```
bee.move(UP, verticalDistance);
bee2.move(UP, verticalDistance);
...
bee12.move(UP, verticalDistance);
```

Note, however, that although the bee to which we are sending the `move()` message changes, each statement is otherwise the same. Remember: *any time you find yourself programming the same thing over and over, there is usually a better way.* In this case, the better way is to create a data structure variable named `bees` that stores references to the 12 bees, and which we might visualize as shown in Figure 5-4.

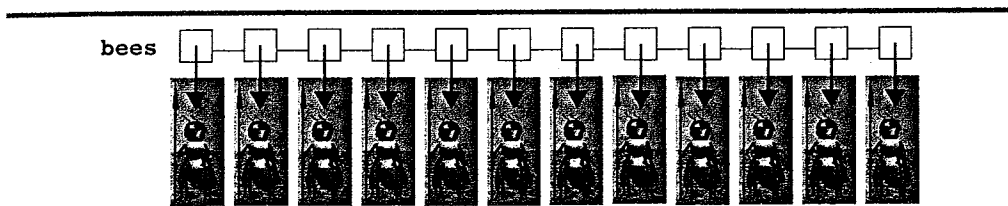


FIGURE 5-4 A list of 12 bees

As indicated in the caption of Figure 5-4, this kind of data structure is called a *list*, as in *shopping list*, *guest list*, or *play list*. Alice's list data structure can store a sequence of items, which can be any Alice type (for example, **Number**, **Boolean**, **Object**, **String**, **Color**, etc.).

Given a list variable, we can use Alice's **forAllInOrder** statement to send each item in the list the message **move(UP, verticalDistance)**:

```
for all bees, one item_from_beas at a time {
    item_from_beas.move(UP, verticalDistance);
}
```

We will look at each of these steps separately.

### Defining a List Variable

We can begin by defining a **playScene2()** method, and then defining a list variable within it. To create a list variable, we click the **create new variable** button as usual. Because the things we want to store in the list (bees) are objects, we select **Object** as the type in the dialog box that appears. We then click the checkbox labeled **make a List**, which expands the dialog box with a **values** pane, as shown in Figure 5-5.

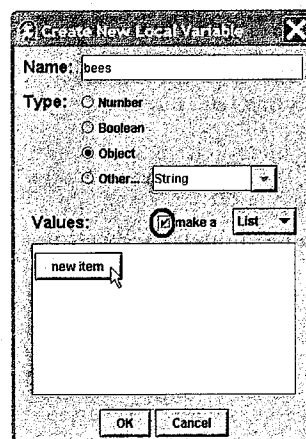


FIGURE 5-5 Creating a list variable

To store the bees in the list, we click the **new item** button visible in Figure 5-5. Alice then adds an item to the list whose value is **<None>**, as shown in Figure 5-6 (left side).

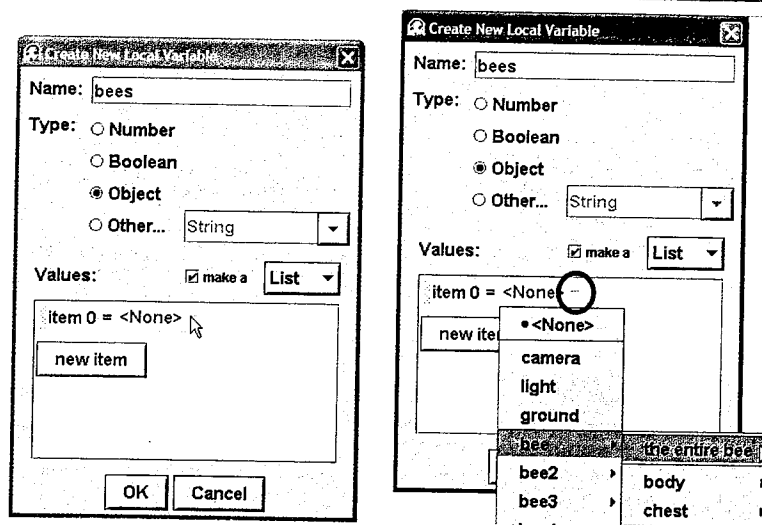


FIGURE 5-6 Defining initial values in a list variable

To make the value of this new item the first bee, we click the list arrow next to **<None>**, choose the **bee** from the menu that appears, and select **the entire bee**, as shown in Figure 5-6 (right side). We then repeat these steps to create new items for each additional bee in the story, and finally click the dialog's **OK** button. The result is the list variable shown in Figure 5-7.

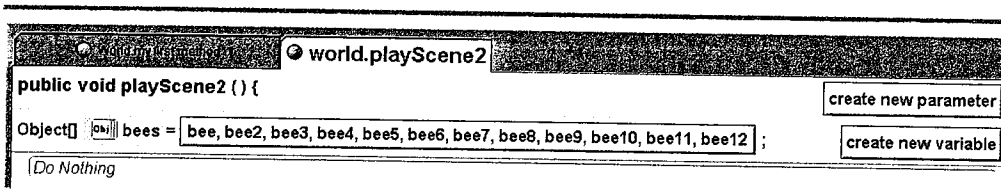


FIGURE 5-7 A list variable definition

While Alice's choice of font makes it a bit difficult to see, the form of this definition is:

```
Object [] bees = bee, bee2, ... bee12;
```

Alice uses square brackets ([ and ]) to distinguish data structures from "normal" variables.

## Processing List Entries

Now that we have defined a list variable, the next step is to use a new Alice statement — the **forAllInOrder** statement — to send the **move()** message to each of its items. To do so, we click the **forAllInOrder** control at the bottom of the *editing area*, and then drag it into the **playScene2()** method. When we drop it, Alice generates a **list** menu from which we can choose the **bees** variable, as shown in Figure 5-8.

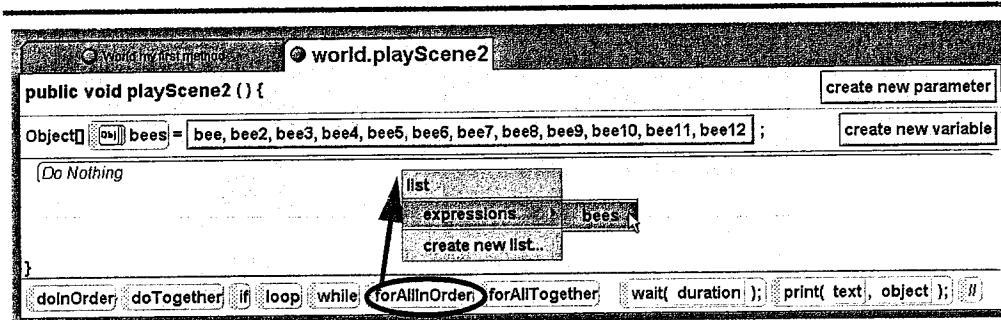


FIGURE 5-8 Dragging the **forAllInOrder** statement

When we choose **bees** from this menu, Alice generates the **forAllInOrder** statement shown in Figure 5-9.

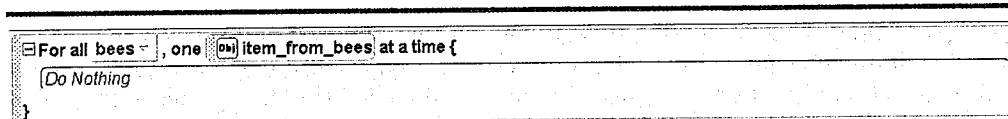


FIGURE 5-9 The **forAllInOrder** statement

With this in place, we construct the necessary **move()** message, using one of the bees as a placeholder, as shown in Figure 5-10.

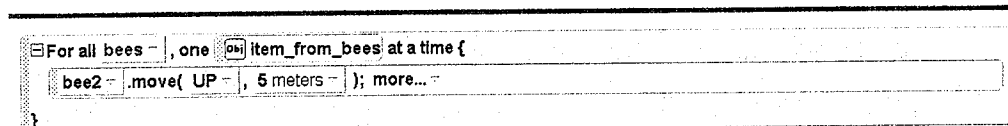
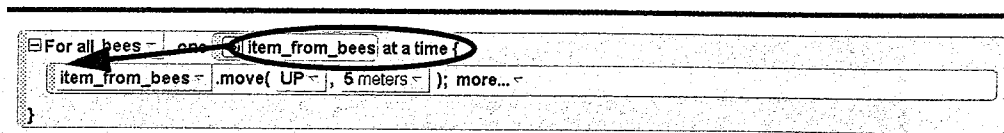
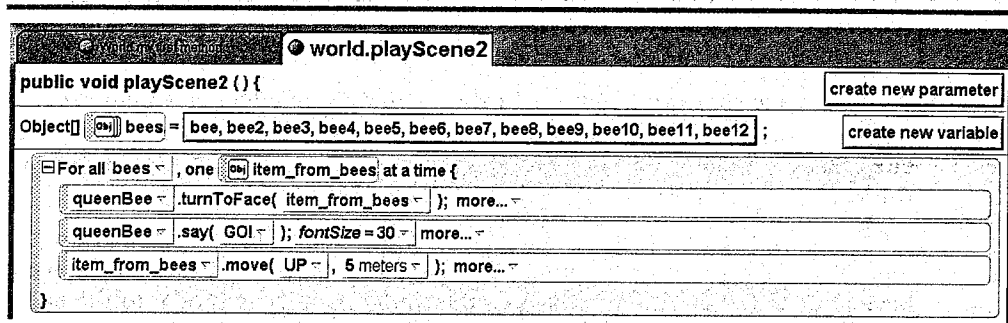


FIGURE 5-10 The **forAllInOrder** statement with a bee placeholder

We then replace the placeholder with an item from the **bees** list. To do so, we drag **item\_from\_bees** onto the placeholder and drop it, as shown in Figure 5-11.

FIGURE 5-11 The `forAllInOrder` statement with an item from `bees`

The resulting loop will send the `move()` message to each bee in the list, one at a time, causing them to “take off.” We can similarly add statements to make the queen bee turn to face each bee and order it to take off. Figure 5-12 shows the completed scene method.

FIGURE 5-12 The `playScene2()` method (final version)

With Figure 5-4 as the starting scene, we can trace the flow through this loop as follows:

- In the first repetition of this loop, the `queenBee` faces `bee` and says `Go!`; then `bee` moves up five meters, because `item_from_bees` refers to `bee`.
- In the second repetition of the loop, the `queenBee` faces `bee2` and says `Go!`; then `bee2` moves up five meters, because `item_from_bees` refers to `bee2`.
- In the third repetition of the loop, the `queenBee` faces `bee3` and says `Go!`; then `bee3` moves up five meters, because `item_from_bees` refers to `bee3`.
- This process repeats for each bee, up to the eleventh bee.
- In the eleventh repetition of the loop, the `queenBee` faces `bee11` and says `Go!`; then `bee11` moves up five meters, because `item_from_bees` refers to `bee11`.
- In the twelfth (and final) repetition of the loop, the `queenBee` faces `bee12` and says `Go!`; then `bee12` moves up five meters, because `item_from_bees` refers to `bee12`.

Figure 5-13 shows the scene during the loop's first, third, and last repetitions.

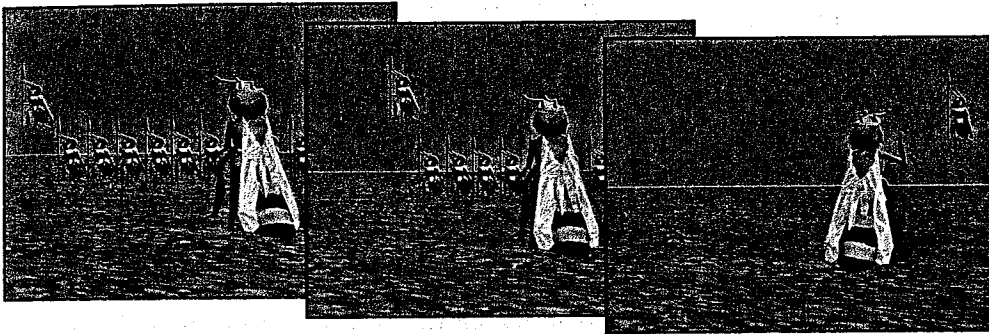


FIGURE 5-13 Repetitions 1, 3, and 12 of the loop

The code shown in Figure 5-12 thus achieves the effect of 12 **turnToFace()** messages, 12 **say()** messages, and 12 **move()** messages. However it does so using only one **turnToFace()** statement, one **say()** message, one **move()** message, a **forAllInOrder** statement, and a list!

Moreover, suppose later on we decide that, to be more convincing, the scene needs more bees taking off (for example, positioned behind those already in the scene). All we have to do is (1) add the new bees to the world, and (2) add them to the **bees** list.<sup>1</sup> We need not add any new **turnToFace()**, **say()**, or **move()** statements to **playScene2()**.

In any situation for which you need to do the same thing to multiple items, a data structure can save you a lot of work!

## 5.1.2 List Operations

The preceding example illustrates how the **forAllInOrder** statement can be used to process each of the items in a list in turn. It provides a very simple way to iterate (or loop) through the entries in the list, doing the same thing to each item in the list.

You may have noticed that there is also a **forAllTogether** control at the bottom of the editing pane. This can be used to create **forAllTogether** statements. Like the **forAllInOrder** statement, the **forAllTogether** statement operates on a list. However, where the **forAllInOrder** statement performs the statements within it once for each item in the list *sequentially*, the **forAllTogether** statement performs the statements within it once for each item in the list *simultaneously*, or in parallel.

To illustrate, if we wanted all of the bees in Figure 5-3 to take off at the same time instead of one at a time, we could rewrite the **playScene2()** method using the **forAllTogether** statement, as shown in Figure 5-14.

1. To add new values to a list variable, just click the box of values (for example, **bee**, **bee2**, ... **bee12**) in its definition.



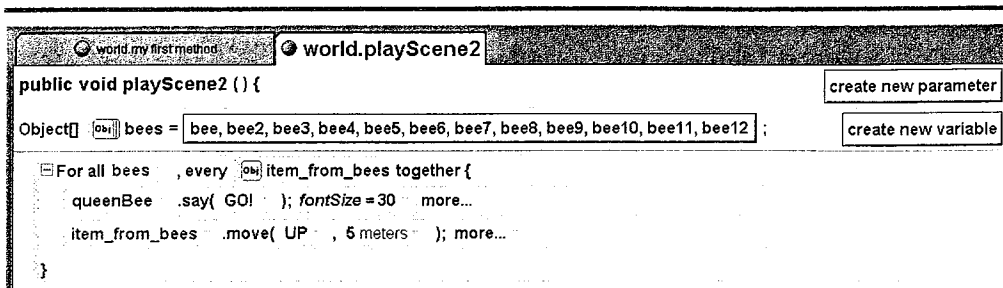


FIGURE 5-14 Making the bees take off together

Using this version of `playScene2()`, clicking Alice's **Play** button produces the screen shown in Figure 5-15.

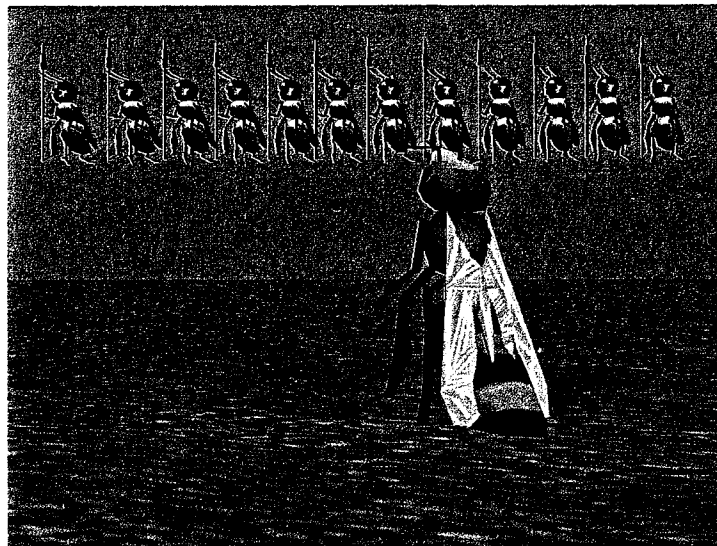


FIGURE 5-15 The bees take off together

Alice provides the **forAllInOrder** and **forAllTogether** statements to simplify the task of processing all of the values in the list data structure. In addition to these *statements*, Alice provides *messages* that you can send to a list variable to modify it or its items. More precisely, if you drag a list variable into the *editing area* and drop it anywhere a *statement* can appear, Alice generates the menu shown in Figure 5-16.

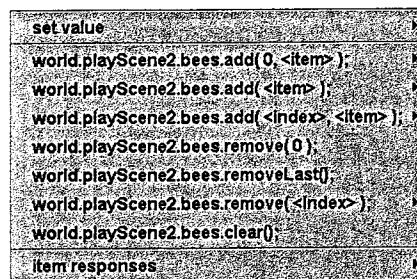


FIGURE 5-16 The list methods menu

The three sections in this menu let you:

- Set the value of the list to another list (the **set value** choice)
- Send a message to the list (the middle portion of the menu)
- Send a message to any of the items in the list (the **item responses** choice)

Because the middle portion of the menu is unique to lists, we will examine it next.

**List Methods.** The messages you can send to a list include those shown in Figure 5-17.

Alice List Method	Behavior
<code>aList.add(0, val);</code>	Create a new item containing <i>val</i> at <i>aList</i> 's beginning
<code>aList.add(i, val);</code>	Insert a new item containing <i>val</i> at position <i>i</i> in <i>aList</i> (the item at position <i>i</i> shifts to position <i>i+1</i> , and so on)
<code>aList.add(val);</code>	Create a new item containing <i>val</i> at <i>aList</i> 's end
<code>aList.remove(0);</code>	Remove the first item from <i>aList</i>
<code>aList.remove(i);</code>	Remove the item at position <i>i</i> from <i>aList</i> (the item at position <i>i+1</i> moves to position <i>i</i> , and so on)
<code>aList.removeLast();</code>	Remove the last item from <i>aList</i>
<code>aList.clear();</code>	Remove all items from <i>aList</i>

FIGURE 5-17 List methods

Figure 5-6 showed how to initially define a list with a group of values. However, there are situations in which a program needs to modify the contents of a list *as it is running*. For example, once the bees are in the air, we might want to have the queen take off too, and add her to the **bees** list. The messages in Figure 5-17 allow a program to modify a list by adding and/or removing items.

Each item in a list has a **position**, or **index**, by which it can be accessed. The index of the first item is always zero, the index of the second item is always one, and so on. To illustrate, Figure 5-18 shows **bees** again, but this time showing the index of each list item.

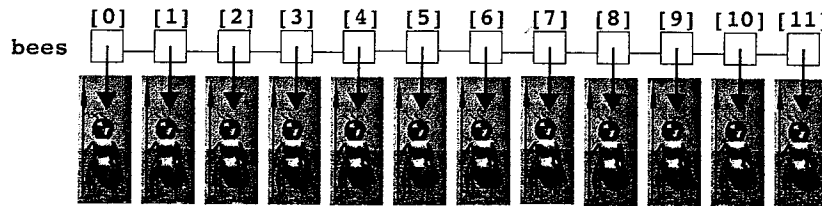


FIGURE 5-18 The list of 12 bees with index values

In the list messages **add(i, val)** and **remove(i)**, the value of **i** is the position or index at which the value will be added or removed. To illustrate, suppose we have the following list:

```
List aList = 2, 8, 4;
```

Suppose we then perform the following statements:

```
aList.remove(1);    // remove the item at index 1 (the 8)
aList.add(0, 1);    // insert 1 at the beginning
aList.add(2, 3);    // insert 3 at index 2
aList.add(5);       // append 5
```

As a result, the contents of **aList** will be 1, 2, 3, 4, 5.

### List Functions

Alice also provides function messages that we can send to a list to get information from it, as shown in Figure 5-19.

List Function	Return Value
<b>aList.size()</b>	The number of items in <b>aList</b>
<b>aList.firstIndexOf(val)</b>	The position of the first item containing <b>val</b> in <b>aList</b> (or -1 if <b>val</b> is not present in <b>aList</b> )
<b>aList.lastIndexOf(val)</b>	The position of the last item containing <b>val</b> in <b>aList</b> (or -1 if <b>val</b> is not present in <b>aList</b> )
<b>aList[0]</b>	The value of the first item in <b>aList</b>

FIGURE 5-19 List functions

*continued*

List Function	Return Value
<code>aList[i]</code>	The value of the item at position <i>i</i> in <i>aList</i>
<code>aList.getLastItem()</code>	The value of the last item in <i>aList</i>
<code>aList.getRandomItem()</code>	The value of an item at a random position in <i>aList</i>

FIGURE 5-19 List functions (*continued*)

To use these functions, you must drag a list definition into the *editing area* and drop it onto a *placeholder* whose type is the function's return type. For example, the top three functions — `size()`, `indexOf()`, and `lastIndexOf()` — each return a **Number**, so if you drop a list onto a **Number** placeholder, Alice will display a menu whose choices are these messages, as shown in the *left-hand* menu in Figure 5-20.

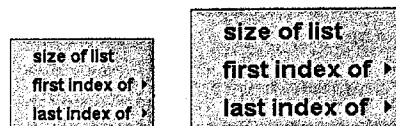


FIGURE 5-20 The list functions menus

However, if you drop a list onto a placeholder whose type is the type of item in the list (for example, **Object**), Alice will display the *right-hand* menu in Figure 5-20, from which you can choose one of the bottom four functions of Figure 5-19.

### 5.1.3 List Example 2: Buying Tickets

Suppose Scene 3 of a story has a line of people waiting for something (for example, to buy tickets to a film). After the first person has been served, she turns and walk away. The remaining people in the line then move forward, so that the person who was second is the new first person in line.

We might begin by building the scene shown in Figure 5-21.

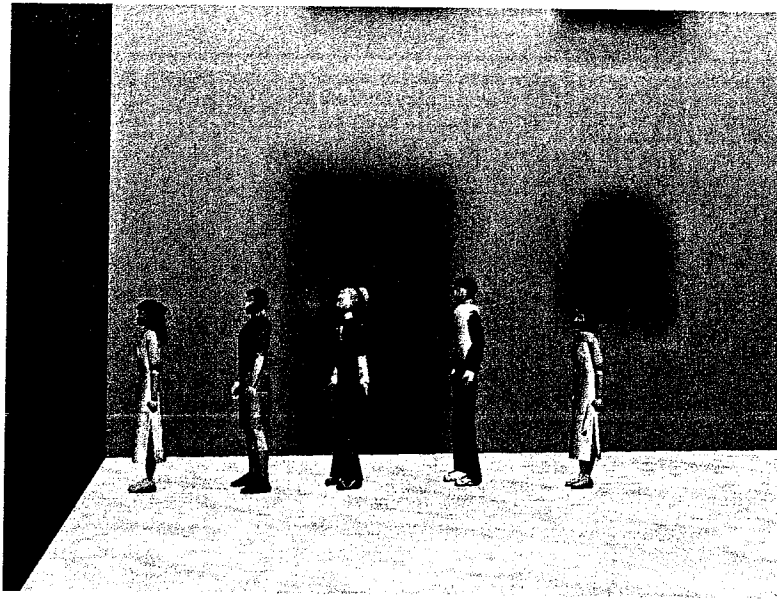


FIGURE 5-21 People waiting in a line

Alice's list data structure makes it fairly easy to animate such a scene. The basic idea is to represent the line of people with a list containing each of the people in the scene. Then we can use the list methods and functions to move them around, using an algorithm like this:

```

1 personList = isis, randomGuy2, skaterGirl, skaterGuy, cleo;
2 while personList is not empty {
3   Set firstPerson to the first item in personList
4   Have firstPerson say "Two tickets please", and then "Thank you"
5   Have firstPerson turn left
6   Have firstPerson move off-screen
7   Remove the first item from personList
8   Advance the line, moving each person in personList forward
9 }
```

To determine whether a list is empty, we can compare its `size()` to zero. To get the first item in the list, we can use the `[0]` function. To "advance the line" we can either use a `forAllTogether` statement or a `forAllInOrder` statement. To remove the first item from the list, we can use the `remove(0)` method.

Figure 5-22 presents an Alice version of this algorithm, using a **forAllInOrder** statement to “advance the line.”

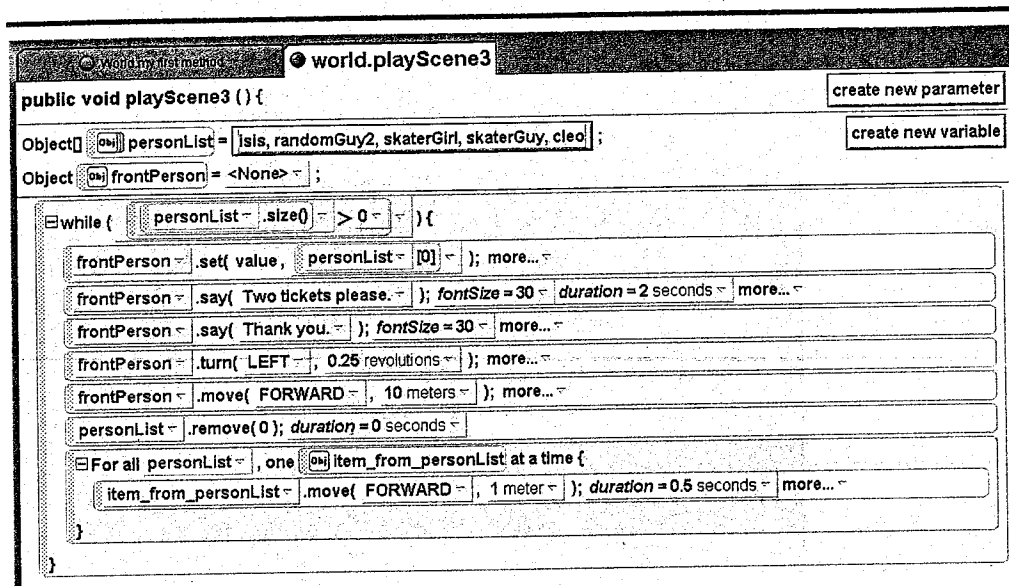


FIGURE 5-22 Animating a line of people

Figure 5-23 shows three screen shots of this scene, all taken during the first pass through the **while** statement in **playScene3()**.

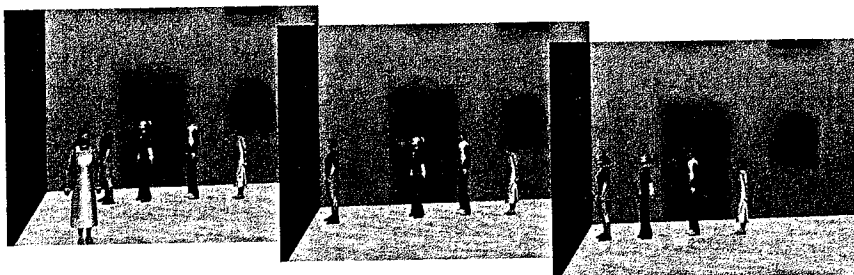


FIGURE 5-23 Screen captures from **playScene3()**

To see how the people in line are moving, compare their positions against the background in each screen capture. For example, **cleo** is in front of the rounded window in the leftmost capture; in the rightmost capture, she and the others have moved forward.

The list is one of the two data structures available in Alice. The other is called the array, and we examine it next.

## 5.2 The Array Structure

Alice's second data structure is called the **array**. Like an Alice list, an array can store a group of values, each of which can be accessed through its position or index. However, unlike the list, the array is a *fixed-sized data structure*, meaning it cannot grow or shrink as your program runs. You can still change the values of the items in an array, but once your program begins running, its *capacity* (the maximum number of values it can store) cannot change. An array is thus a somewhat less flexible data structure than a list.

Why would anyone want to use an array instead of a list? There are two answers:

1. In Alice and most other programming languages, it takes less of a computer's memory to store a group of items in an array than it does to store the same group of items in a list. Put differently, if you have a group of items to store and the size of the group never changes, it is more *memory-efficient* to store the group in an array instead of a list.
2. In most other programming languages, items in a list cannot be accessed via index values. Instead, only the first and last item in the list can be accessed. The exact reason is beyond the scope of our discussion, but accessing an arbitrary item from a list is *much* more time-consuming than accessing an arbitrary item from an array, so most languages don't let you do it. So, if the solution to a problem requires a program to access arbitrary values from a group, then it's better to store the group in an array instead of a list. Put differently, to access an arbitrary group item, an array is more *time-efficient* than a list.

To see the Alice array in action, let's see an example.

### 5.2.1 Array Example 1: The Ants Go Marching

Suppose a user story has an ant marching along, singing the song "The Ants Go Marching." The lyrics to the song are as follows:

<p>The ants go marching one-by-one Hurrah! Hurrah! The ants go marching one-by-one Hurrah! Hurrah! The ants go marching one-by-one, the little one stopped to suck his thumb, and they all went marching down to the ground to get out of the rain BOOM! BOOM! BOOM!</p>	<p>The ants go marching two-by-two Hurrah! Hurrah! The ants go marching two-by-two Hurrah! Hurrah! The ants go marching two-by-two, the little one stopped to tie his shoe, and they all went marching down to the ground to get out of the rain BOOM! BOOM! BOOM!</p>
<p>...</p> <p>The ants go marching three-by-three, the little one stopped to climb a tree, ...</p>	<p>...</p> <p>The ants go marching four-by-four, the little one stopped to shut the door, ...</p>

... The ants go marching five-by-five The little one stopped to take a dive, ...	... The ants go marching six-by-six The little one stopped to pick up sticks, ...
... The ants go marching seven-by-seven The little one stopped to pray to heaven, ...	... The ants go marching eight-by-eight The little one stopped to shut the gate, ...
The ants go marching nine-by-nine Hurrah! Hurrah! The ants go marching nine-by-nine Hurrah! Hurrah! The ants go marching nine-by-nine, the little one stopped to check the time, and they all went marching down to the ground to get out of the rain BOOM! BOOM! BOOM!	The ants go marching ten-by-ten Hurrah! Hurrah! The ants go marching ten-by-ten Hurrah! Hurrah! The ants go marching ten-by-ten, the little one stopped to say, 'THE END', and they all went marching down to the ground to get out of the rain BOOM! BOOM! BOOM!

One way to build this story would be to send the ant 10 **say()** messages per verse times 10 verses = 100 **say()** messages. But many of the song's lines are exactly the same from verse to verse, so this approach would result in lots of wasteful, replicated effort.

Another way would be to recognize that this is basically a *counting problem*: the song is counting from 1 to 10. So perhaps we could use a **for** statement to count through the verses, and put statements within the **for** statement to make the ant sing a verse? This is good thinking; the difficulty is that each verse differs from the others in *two* ways:

- the number being sung (*one, two, ..., nine, ten*); and
- what the little ant does (*suck his thumb, tie his shoe, ..., check the time, say "THE END"*).

One solution is to make two indexed groups, one for each way the verses differ, as shown in Figure 5-24.

Group 1	
Index	Numbers
0	one
1	two
2	three
3	four

Group 2	
Index	What the little ant does
0	suck his thumb
1	tie his shoe
2	climb a tree
3	shut the door

FIGURE 5-24 Groups of strings

*continued*



Group 1		Group 2	
Index	Numbers	Index	What the little ant does
4	five	4	take a dive
5	six	5	pick up sticks
6	seven	6	pray to heaven
7	eight	7	shut the gate
8	nine	8	check the time
9	ten	9	say 'THE END'

FIGURE 5-24 Groups of strings (continued)

If we defined two data structures (one for each group), then the **for** statement could count from 0 to 9, and on repetition *i*, retrieve the value associated with index *i* from each data structure.

Because the number of verses in the song is fixed, it makes sense to use array data structures to store the two groups. Defining an array variable is similar to defining a list variable, which we saw in Figures 5-5 and 5-6. The only difference is that we must specify that we want an **Array** variable instead of a **List** variable, as shown in Figure 5-25.

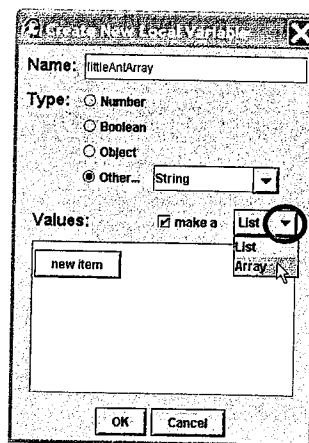


FIGURE 5-25 Creating an Array variable

Once we have created the array variable, we can fill it with values in exactly the same way as we would a list (see Figure 5-6).

Given the ability to define arrays, we can build this algorithm to solve the problem:

```

1 Define numberArray = one, two, three, four, five, six, seven, eight,
  nine, ten;
2 Define littleAntArray = suck his thumb, tie his shoe, climb a tree,
  shut the door, take a dive, pick up sticks, pray to heaven, shut the
  gate, check the time, say 'THE END',
3 for each index 0 through 9 {
4   repeatedLine = "The ants go marching " + numberArray[index] + "-by-"
5   + numberArray[index];
6   ant.say(repeatedLine);
7   ant.say("Hurrah! Hurrah!");
8   ant.say(repeatedLine);
9   ant.say("Hurrah! Hurrah!");
10  ant.say(repeatedLine);
11  ant.say("The little one stopped to " + littleAntArray[index]);
12  ant.say("and they all went marching");
13  ant.say("down to the ground");
14  ant.say("to get out of the rain.");
15  ant.say("BOOM! BOOM! BOOM!");
16 }

```

Using this algorithm, we can build the program, as shown in Figure 5-26.

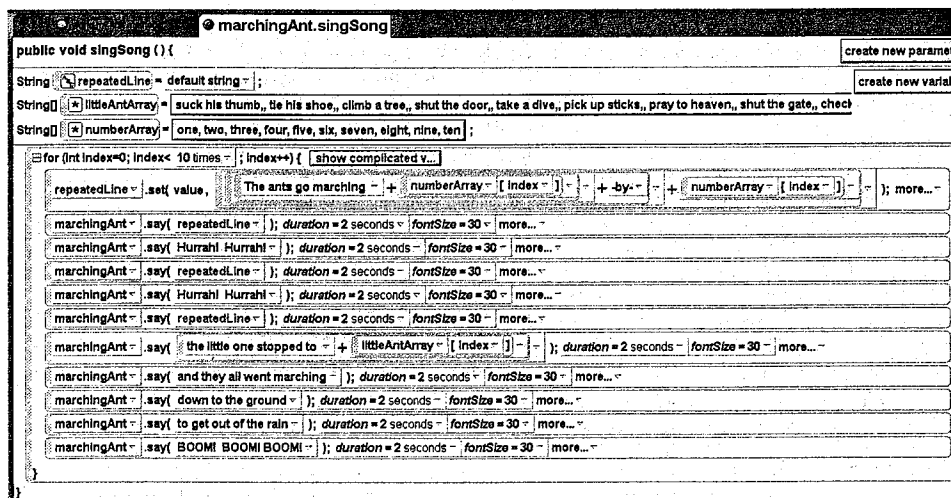


FIGURE 5-26 Singing "The Ants Go Marching"

When performed, this method (using just 12 statements and three variables) causes the `marchingAnt` to “sing” the entire 10-verse, 100-line song!

If we add a method that makes the ant march, and then have the ant sing the song as it marches, the result will appear something like what is shown in Figure 5-27.

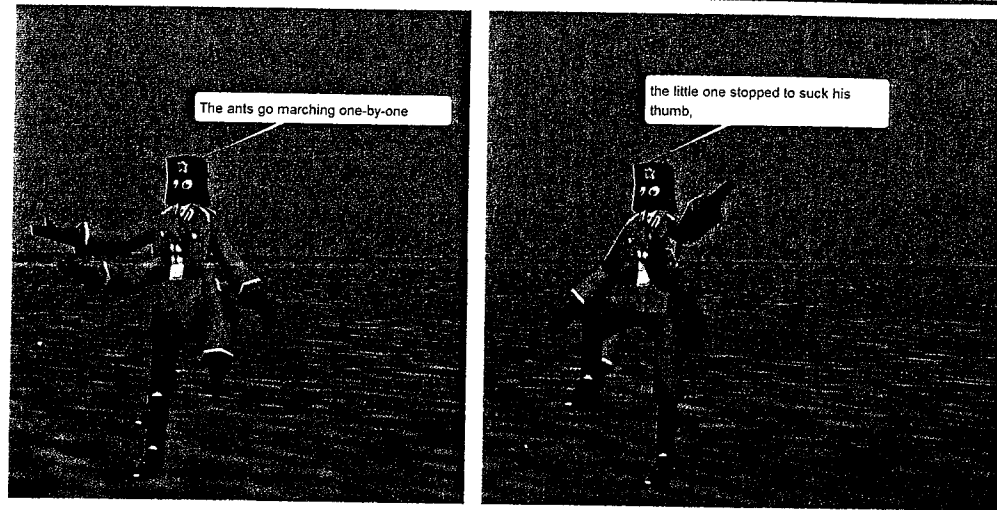


FIGURE 5-27 The singing ant in action

## 5.2.2 Array Operations

Like lists, arrays are **indexed variables**, meaning their items can be accessed using an index value. For example, we could have written the “bees” program in Figure 5-12 using an array instead of a list. If we had done so, we could have drawn the `bees` group as shown in Figure 5-28:

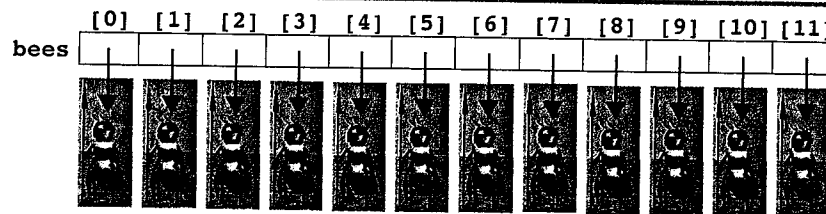


FIGURE 5-28 An array of 12 bees

We saw in Section 5.1.2 that Alice provides a variety of predefined operations that can be used with list variables. By contrast, there are only a few operations for array variables. These are listed in Figure 5-29.

Alice Array Operation	Behavior
<code>anArray[i] = val;</code>	Change the value at position <i>i</i> in <i>anArray</i> to <i>val</i>
<code>val.set(value, anArray[i]);</code>	Retrieve the value at position <i>i</i> in <i>anArray</i>
<code>anArray.length</code>	Retrieve the number of values in <i>anArray</i>

FIGURE 5-29 Array operations

The notation `anArray[i]` is called the **subscript operation**. As shown in Figure 5-29, there are two versions of the subscript operation. The first one is sometimes called the *write version*, because it changes (that is, writes) the value of item *i* of the array. The second one is sometimes called the *read version*, because it retrieves (that is, reads) the value of item *i* of the array.

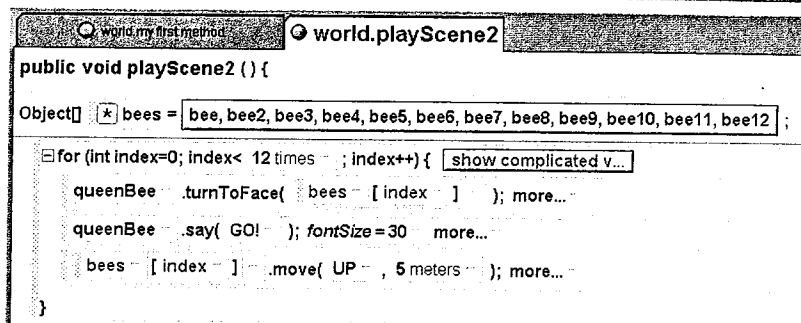
If an array variable is dropped where a *statement* can appear, Alice displays a menu from which you can select the write version of the subscript operation. If an array variable is dropped onto a *placeholder* variable or value, Alice displays a menu from which you can select either the array's **length** attribute or the read version of the subscript operation.<sup>2</sup>

At the time of this writing, the Alice **forAllInOrder** and **forAllTogether** statements can only be used on a list, not on an array. Until this changes, if you want to process each of the values in an array, you must use a **for** statement like this:

```
for (int index = 0; index < anArray.length; index++) {
    // do something with anArray[index]
}
```

To illustrate, Figure 5-30 presents an alternative version of Figure 5-12 using an array.

2. When this was written, Alice was very inconsistent in displaying these menus, for both arrays and lists. Hopefully, these problems will be fixed by the time you read this!



```

world.myfirstmethod  world.playScene2
public void playScene2 () {
    Object[] bees = { bee, bee2, bee3, bee4, bee5, bee6, bee7, bee8, bee9, bee10, bee11, bee12 };
    for (int index=0; index< 12 times ; index++) { show complicated v...
        queenBee .turnToFace( bees [ index ] ); more...
        queenBee .say( GO! ); fontSize=30 more...
        bees [ index ] .move( UP , 5 meters ); more...
    }
}

```

FIGURE 5-30 The bees take off using an array

This method produces exactly the same behavior as that of Figure 5-12. However, note that because we cannot use a **forAllTogether** statement on an array, we cannot use an array to produce the simultaneous behavior shown in Figure 5-14 and Figure 5-15.

### 5.2.3 Array Example 2: Random Access

Suppose we want to build the following simple story:

---

*Scene:* A castle has two magical doors. The left door tells the right door a random knock-knock joke.

---

If the left door told the right door the same joke every time, then this story would quickly become boring and the user would not want to play the story more than twice. However, if each time the scene is played, the left door tells a *random* (that is, potentially different) knock-knock joke, we make the scene more interesting and worth revisiting.

We can begin by positioning the **camera** and **castle** as shown in Figure 5-31.

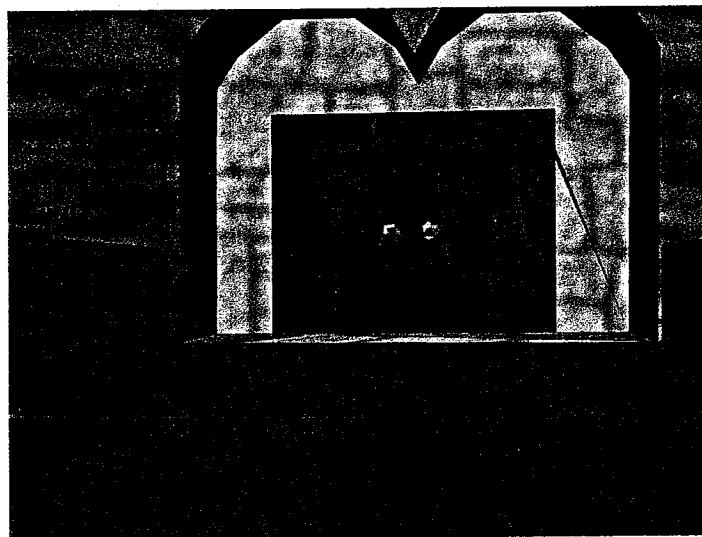


FIGURE 5-31 The castle doors

Our next problem is to figure out how to make the doors tell a knock-knock joke. Let's look at several jokes, to see what is the same and what is different about each one:

Joke 1	Joke 2	Joke 3
L: Knock-knock	L: Knock-knock	L: Knock-knock
R: Who's there?	R: Who's there?	R: Who's there?
L: Boo.	L: Who.	L: Little old lady.
R: Boo who?	R: Who who?	R: Little old lady who?
L: Don't cry, it's just a joke.	L: Is there an owl in here?	L: I didn't know you could yodel!

Comparing these (lame) jokes, we see that knock-knock jokes have the following structure:

L: Knock-knock  
 R: Who's there?  
 L: *name*  
 R: *name* who?  
 L: *punchline*

where *name* and *punchline* are the parts that vary from joke to joke.

If we make *name* and *punchline* array variables, then we can store multiple jokes in them. For example, to store the three jokes above, we would define *name* and *punchline* as follows:

```
String [] name = {"Boo", "Who", "Little old lady" };
String [] punchline = {"Don't cry, it's just a joke.",
                      "Is there an owl in here?",
                      "I didn't know you could yodel!"};
```

Such definitions create parallel data structures in which *punchline*[0] corresponds to *name*[0], *punchline*[1] corresponds to *name*[1], and so on. We can visualize them as shown in Figure 5-32.

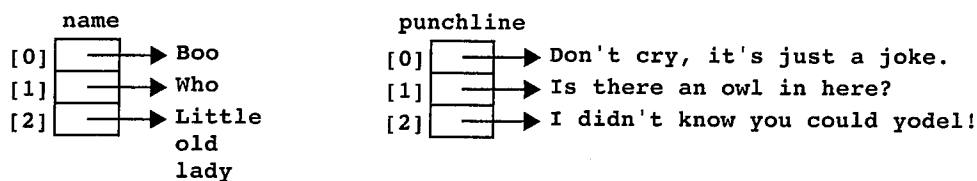


FIGURE 5-32 The *name* and *punchline* arrays

Once we have the parts of the jokes stored in arrays, we can tell the joke at index *i* as follows:

L: Knock-knock

R: Who's there?

L: *name*[*i*]

R: *name*[*i*] who?

L: *punchline*[*i*]

That is, if *i* has the value 0, then this will tell the "Boo who" joke; if *i* has the value 1, then it will tell the "Who who" joke, and so on.

### Generating Random Numbers

To tell a random joke from the array, we need to generate a **random number** for the index *i*. That is we need to set *i* to a value that is randomly selected from the range of possible index values for the array. Fortunately, Alice makes this fairly

easy by providing a **world** function named `Random.nextDouble()`, as shown in Figure 5-33.

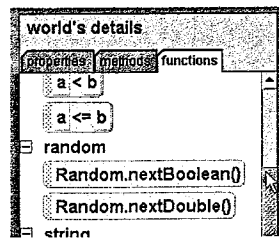


FIGURE 5-33 The world function `Random.nextDouble()`

Using this function, we can set a **Number** variable `i` to a random number by (1) setting the value of `i` to a placeholder value, (2) dragging the function onto the placeholder, and (3) setting its **minimum**, **maximum**, and **integerOnly** attributes to appropriate values (for example, `0`, `name.length`, and `true`, respectively). Figure 5-34 shows the completed `tellRandomKnockKnockJoke()` method, which includes the jokes above, plus two others.

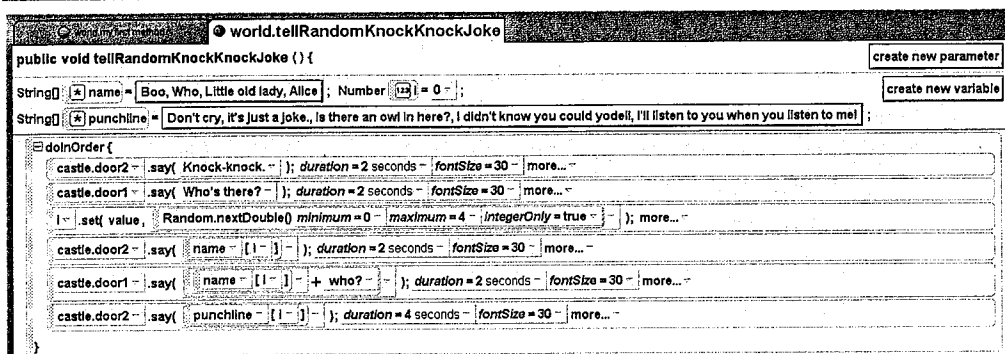


FIGURE 5-34 The `tellRandomKnockKnockJoke()` method

Each time this method is performed, it tells a knock-knock joke selected at random from the **name** and **punchline** arrays. By randomly generating the value of `i`, and then using that same value as the index for both `name[i]` and `punchline[i]`, we ensure that the name and punchline for a given joke match one another.



## Random Details

The `Random.nextDouble()` function has two quirks to keep in mind:

- If you wish to generate an *integer* (that is, a whole number without decimal places like -1, 0, 1, or 1234), be sure to set the `integerOnly` attribute to `true`, or else the function will produce a *real* number (that is, a number with decimal places like -1.25, 0.05, 98.7654, etc.).
- In Figure 5-34, the arrays contain 4 items, indexed 0, 1, 2, and 3. To generate a random index value from the group {0, 1, 2, 3}, we specified a minimum value of 0, but a maximum value of 4. In general, if we want to generate a random number from the range *a* through *b*, then we should specify *a* as the minimum value and *b*+1 as the maximum value. Put differently, whatever minimum value we specify is *included* in the range of randomly generated values, but whatever maximum value we specify is *excluded* from the range of randomly generated values.

Recall that in Figure 5-19, we saw that one of the messages we can send to an Alice list is the `getRandomItem()` function. In situations where we just need to retrieve one random item from a data structure, a list and this function provide an easy way to solve the problem.

However, we cannot use a list and the `getRandomItem()` function to solve the random knock-knock joke problem (at least not as easily). Do you see why not? The issue is that the problem has *two* data structures: one containing the names and one containing the corresponding punchlines. If we were to store the names and punchlines in two lists and then send each list the `getRandomItem()` function, the randomly selected punchline would be unlikely to correspond to the randomly selected name.<sup>3</sup>

## 5.3 Alice Tip: Using the `partNamed()` Function

Suppose that Scene 1 of a story begins as follows:

---

*Scene: The court of the fairy queen is crowded with fairy-courtiers talking amongst themselves. One of the fairies announces, "Her majesty, the Queen!" The fairy queen enters her court, and all the courtiers turn toward her. As she moves along the promenade leading to her throne, each courtier in succession turns to her and bows. Upon reaching her throne, the queen turns and says "Please rise." As one, the courtiers turn toward her and rise from their bows.*

---

Looking over the nouns in the story, we might begin building this scene by creating a "fairy court" in a woodland setting, with a promenade leading to a throne, and a crowd of fairies flanking each side of the promenade. Figure 5-35 shows one possible realization of this scene using various fairy, forest, and other classes from the Alice Web Gallery.

3. We could replace the two arrays in Figure 5-34 with two lists. Because Alice lists support the subscript operation, we could randomly generate an index *i* and then access the item at position *i* in each list. However, because the data structures' sizes remain fixed as the program runs, using an array is preferable.



FIGURE 5-35 The court of the fairy queen

We also chose class `OliveWaterblossom` as the fairy queen, added her to the world, and positioned her behind the camera (13 meters from the throne) to set up her entry to the court.

With the scene set, we are ready to think about generating the behavior required by the user story. We might break the actions down into the following sequence of steps:

1. One of the fairies announces, "Her majesty, the Queen!"
2. As the queen enters the court, each courtier simultaneously turns to face the camera.
3. Do together:
  - a. Move the queen 13 meters forward (down the promenade, toward her throne).
  - b. Make the queen's wings flap.
  - c. As she passes each courtier, have him or her turn toward the queen and bow.
4. The queen turns 1/2 revolution (so that she is facing her courtiers).
5. The queen says "Please rise."
6. Together each courtier turns toward the queen and rises from his or her bow.

Together, these steps make up an algorithm we can use for the `playScene1()` method.

### Defining The Method

What is the best way to implement this algorithm? Steps 1 and 5 require all courtiers to take a simultaneous action, and Step 2c requires each courtier to take an action one at a time. One way to elicit these simultaneous and one-at-a-time actions is to place the courtiers into a list data structure. Given such a list, we can use the **forAllTogether** statement to make all courtiers do the same thing simultaneously in Steps 1 and 5, and we can use the **forAllInOrder** statement to make them all do the same thing one at a time in Step 2c.

With this approach, we can revise the algorithm as follows:

1. Let **courtierList** be a list of all the courtier fairies.
2. The courtier nearest the throne announces the queen.
3. For all items in **courtierList** together:
  - Each item in **courtierList** turns to face the camera.
4. Do together:
  - a. The queen moves 13 meters forward (down the promenade, toward her throne).
  - b. The queen's wings flap.
  - c. For each item in **courtierList**, one at a time:
    - The item in **courtierList** turns toward the queen and bows.
5. The queen turns 1/2 revolution.
6. The queen says "Please rise."
7. For all items in **courtierList** together:
  - Each item in **courtierList** turns toward the queen and rises from his/her bow.

Most of these steps are straightforward to program in Alice. However, there are two subtle points to keep in mind as we do so.

### Defining The List

One subtle part is that when we define the **courtierList** variable as a list of **Object** and then add fairies to it, the order of the fairies in the list is significant. That is, because we are using the **forAllInOrder** statement in Step 3c and this statement goes through the items in the list from first to last, we must be careful to add the fairies to the list so that those closest to the camera are earlier in the list and those who are farthest from the camera are later in the list. Otherwise, the fairies will not turn toward the queen and bow to her as she moves past them. Figure 5-36 presents a fragment of this list from the **playScene1()** method.

---

```
Object[] courtierList = {shadeAniseed, petalBeamweb, sprightlyReedsmoke, mabHazelut, meadSeafeather, lichenZenspider, leafFlameglimmer, r
```

---

FIGURE 5-36 Defining the courtier list

---

### Making A Fairy Bow

The second subtle part is generating the bowing and rising behaviors for the courtiers. It is easy to make an individual fairy-courtier bow and rise, by "opening" the individual in

the *object tree*, selecting their **upperBody** component, and then sending this component the **turn()** message, as shown in Figure 5-37.

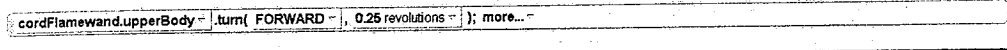


FIGURE 5-37 Making an individual courtier bow

The difficulty arises when we seek to use this approach with an item from the list within a **forAllInOrder** or **forAllTogether** statement. Although each item in the list is a fairy that has an **upperBody** component, we defined the **courtierList** variable as a list of **Objects**. Because not all Alice **Objects** have **upperBody** components (for example, buildings, fish, trees, etc.), Alice will not let us access the **upperBody** component of an item from the list. So we *can* make each courtier turn and face the queen by programming:

```
for all courtierList, one item_from_courtierList at a time:
    item_from_courtierList.turnToFace(oliveWaterblossom);
```

But we *cannot* make each courtier bow to the queen by programming:

```
for all courtierList, one item_from_courtierList at a time:
    item_from_courtierList.upperBody.turn(FORWARD, 0.25); // NO!
```

Because the lists are lists of **Objects**, we can only send a list item a message (or select a component) that is common to all **Objects**.

For this situation, every Alice object provides a function message called **partNamed(component)** that can be sent to that object to retrieve its part named **component**. In our situation, we know that every fairy in the list contains a component named **upperBody**, so we can send each fairy the **partNamed(upperBody)** message to retrieve its **upperBody** part, and then send that part the **turn()** message to make the fairy bow.

To use the **partNamed()** function, we begin with the statement shown in Figure 5-37, using the courtier's **upperBody** component as a placeholder. We then drag the courtier's **partNamed()** function onto the placeholder, yielding the statement shown in Figure 5-38.

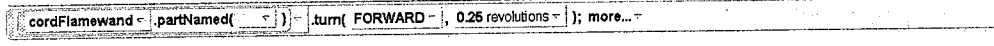


FIGURE 5-38 Using the **partNamed()** function

If we click the list arrow for **partNamed()**'s argument, and choose **other...** from the menu that appears, Alice displays a dialog box where we can type the name of the part we wish to access (**upperBody** in this case). When we do this, we get the statement shown in Figure 5-39.

```
cordFlamewand .partNamed( upperBody ) .turn( FORWARD , 0.25 revolutions ); more...
```

FIGURE 5-39 The `partNamed()` function

In the statement in Figure 5-39, `cordFlamewand` is a placeholder that we need to replace with an item from the list. To do so, we can drag and drop this statement into a `forAllInOrder` (or `forAllTogether`) statement, specify the `courtierList` as the `forAllInOrder` statement's list variable, and then drag the loop's `item_from_courtierList` variable onto the placeholder to replace it. The resulting statement is shown in Figure 5-40.

```
cordFlamewand - .partNamed( upperBody ) - .turn( FORWARD , 0.25 revolutions ); more...
```

FIGURE 5-40 Replacing the placeholder with a list item

Using this same approach, we can make the courtiers rise at the end of the scene. Figure 5-41 shows the completed `playScene1()` method.

```
public void playScene1() {
    // Object[] courtierList = [shadeAniseed, petalBeamweb, sprightlyReedsSmoke, mabHazelNut, meadSeafather, lichenZenspider, leafFlameglimmer];
    cordFlamewand .say( Her majesty, the Queen! ); duration = 2 seconds; fontSize = 40; more...
    For all courtierList, every item_from_courtierList together {
        item_from_courtierList .turnToFace( camera ); more...
    }
    doTogether {
        For all courtierList, one item_from_courtierList at a time {
            item_from_courtierList .pointAt( oliveWaterblossom ); duration = 0.25 seconds; onlyAffectYaw = true; more...
            item_from_courtierList .partNamed( upperBody ) .turn( FORWARD , 0.25 revolutions ); more...
        }
        oliveWaterblossom .move( FORWARD , 13 meters ); duration = 16 seconds; more...
        oliveWaterblossom .flapWings( duration = 16 , amount = 1 );
    }
    oliveWaterblossom .turn( LEFT , 0.5 revolutions ); more...
    oliveWaterblossom .say( Please rise. ); duration = 2 seconds; fontSize = 30; more...
    For all courtierList, every item_from_courtierList together {
        item_from_courtierList .turnToFace( oliveWaterblossom ); more...
        item_from_courtierList .partNamed( upperBody ) .turn( BACKWARD , 0.25 revolutions ); more...
    }
}
```

FIGURE 5-41 The `playScene1()` method (final version)

When we run the program, we get the desired behavior. Figure 5-42 presents three screen captures: one partway through the **forAllInOrder** statement, one after all have bowed and the queen says "Please rise.", and one at the end of the scene.



FIGURE 5-42 Screen captures from `playScene1()`

### Components Are Objects

The `partNamed()` function thus provides a means of retrieving a component of an object. The components of an object are themselves objects, so in `playScene1()`, we could have defined an **Object** variable named `torso`, and then used it in the **forAllInOrder** statement as follows:

```
for all courtierList, one item_from_courtierList at a time {
    item_from_courtierList.turnToFace(oliveWaterblossom);
    torso.set(value, item_from_courtierList.partNamed(upperBody));
    torso.turn(FORWARD, 0.25);
}
```

Either approach is okay. The point is that the components of an Alice **Object** are **objects**, and can be referred to by **Object** variables.

### Sending Messages to `null`

What happens if the object to which we send the `partNamed(component)` function does not have a part named `component`? For example, suppose we defined two **Object** variables, one named `part1`, the other named `part2`, and then set their values as follows:

```
part1.set(value, OliveWaterblossom.partNamed(upperBody));
part2.set(value, OliveWaterblossom.partNamed(xyz));
```

Because `OliveWaterblossom` does not have a component named `xyz`, the `partNamed()` function returns a special "zero" value named `null`, that denotes the absence of an `Object`. This value `null` is stored in variable `part2`, instead of a reference to a component. We can envision these two variables as shown in Figure 5-43.

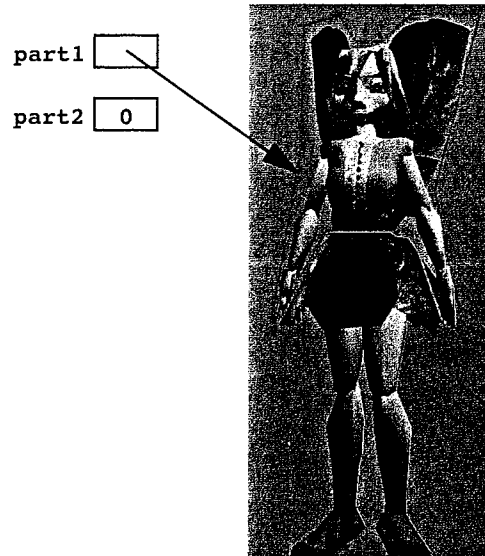


FIGURE 5-43 Variables with non-null and null values

Where `part1` refers to the queen from the waist up, `part2` refers to *nothing*. If the program then erroneously tries to send a message to `part2`:

```
part2.turn(FORWARD, 0.25);
```

Alice will generate an error message:

```
Alice has detected a problem with your world:
subject must not be null.
```

Alice will display this error any time a message is sent via a variable whose value is `null`, which Alice usually displays as `<None>`. This error may appear for a variety of reasons, including the following:

- You deleted an object from your world to which your program was sending a message.
- You deleted a variable from a method or world through which a message was being sent.
- You misspelled the name of the component in the `partNamed()` function.
- You sent the `partNamed(component)` function to an object that does not have a part named `component`.

To correct the first two kinds of errors (which are by far the most common), look through your program's methods for statements in which a message is sent to **<None>**. When you find such a statement, either replace **<None>** with a valid object or delete/disable the statement.

To correct the second kind of error, check the spelling of the component in each statement where you send a **partNamed()** message. If you find one that is incorrect, correct its spelling.

To correct the third kind of error, you must ensure that the **partNamed(component)** function is only sent to objects that have a part named **component**. Check the parts of each object to which you are sending the **partNamed()** message. If you find one that does not have a part named **component**, then either rename the component in that object, or replace that object with a different object that does have a component named **component**.

## 5.4 Chapter Summary

- ☐ An array is a data structure that uses a minimal amount of your computer's memory to store a sequence of items, but cannot grow or shrink as your program runs.
- ☐ A list is a data structure that can grow and shrink as your program runs, at the cost of using some additional computer memory (compared to the array).
- ☐ The **forAllInOrder** statement allows for sequential processing of the items in a list.
- ☐ The **forAllTogether** statement allows for parallel processing of the items in a list.
- ☐ The **Random.nextDouble()** function provides a way to generate random numbers.
- ☐ The **partNamed(component)** function lets us retrieve a part of an object (usually so that we can send it a message).
- ☐ The **null** value is a special "zero" value used to indicate the *absence* of an object. In the *editing area*, Alice usually displays **<None>** to represent the **null** value.

### 5.4.1 Key Terms

array	iterate
data structure	list
<b>forAllInOrder</b> statement	<b>null</b>
<b>forAllTogether</b> statement	<b>partNamed()</b> function
index	position
item	random number

## Programming Projects

- 5.1 Using the **Cheerleader** class from the Alice Gallery, build a world containing 5–6 cheerleaders who lead a cheer at a sporting event. Your cheer can be either funny or serious, and it can either be a cheer unique to your school or a standard cheer (for example, "The Wave").



- 5.2 *This Old Man* is a silly song with the lyrics below. Create an Alice program containing a character who sings this song, using as few statements as possible.

<i>This old man, he played one.  He played knick-knack on my drum,  with a knick-knack paddy-wack give a dog a bone.  This old man came rolling home.</i>	<i>This old man, he played two.  He played knick-knack on my shoe,  with a knick-knack paddy-wack give a dog a bone.  This old man came rolling home.</i>
<i>This old man, he played three.  He played knick-knack on my knee,  ...</i>	<i>This old man, he played four.  He played knick-knack on my door,  ...</i>
<i>This old man, he played five.  He played knick-knack on my hive,  ...</i>	<i>This old man, he played six.  He played knick-knack on my sticks,  ...</i>
<i>This old man, he played seven.  He played knick-knack up in heaven,  ...</i>	<i>This old man, he played eight.  He played knick-knack on my gate,  ...</i>
<i>This old man, he played nine.  He played knick-knack on my spine,  with a knick-knack paddy-wack give a dog a bone.  This old man came rolling home.</i>	<i>This old man, he played ten.  He played knick-knack once again,  with a knick-knack paddy-wack give a dog a bone.  This old man came rolling home.</i>

- 5.3 Create a city scene featuring a parade. Store the paraders (that is, vehicles, people, etc.) in a data structure and use it to coordinate their movements. Make your parade as festive as possible.
- 5.4 Build a world containing a person who can calculate the average, minimum, and maximum of a group of numbers in his or her head. Use a **NumberDialog** to get the numbers from the user. Have your person and each **NumberDialog** tell the user to enter a special value (for example, -999) after the last value in the sequence has been entered. Store the group of numbers in a data structure, and write three new world functions — **average()**, **minimum()**, and **maximum()** — that take a data structure as their argument and return the average, minimum, and maximum value in the structure, respectively. When all the numbers have been entered, have your person “say” the group’s average, minimum, and maximum values.
- 5.5 Create a scene in which a group of Rockettes do a dance number (for example, the Can-Can). Store the Rockettes in a data structure, and use **forAllInOrder** and/or **forAllTogether** statements to coordinate the movements of their dance routine.
- 5.6 Create a “springtime” scene that runs for a minute or so, starting with an empty field but ending with the field covered with flowers. The flowers should “grow” out of the ground as your scene plays. Make your program as short as possible by storing the flowers in a data structure.
- 5.7 Proceed as in Problem 5.6, but use random-number generation to make the flowers appear in a different order or pattern every time your program is run.

- 5.8 Create a scene in which two people are talking near a not-very-busy intersection, which uses four stop signs to control the traffic. Build the intersection using buildings and roads. Define two data structures: one containing a group of vehicles, and one containing the four directions a vehicle can move through the intersection (for example, north, south, east, and west). As your characters talk, use random numbers to select a vehicle and its direction.
- 5.9 Choose an old pop song that has several unique arm or body motions and whose lyrics are available on the Internet (for example, YMCA by the Village People, *Walk Like An Egyptian* by the Bangles, etc.). Using Alice, create a “music video” for the song, in which several people sing the song and use their arms or bodies to make the motions. Make your video as creative as possible, but try to avoid writing the same statements more than once. If you have access to a *legal* digital copy of the song, use the `playSound()` message to play it during your video.
- 5.10 Create a scene containing a group of similar creatures from the Alice gallery (for example, a herd of horses, a school of fish, a pack of wolves, etc.). Store your group in a data structure, and write a method that makes the group exhibit *flocking behavior*, in which the behavior of one member of the group causes the rest of the group to behave in a similar fashion. (Hint: designate one member of the group as the leader, and make the leader the first item in the data structure.)

*It's**To  
of th  
the  
of a  
man  
upon**In t**Obj**Upo*☐☐☐

# Chapter 6

## Events

*It's not the events of our lives that shape us, but our beliefs as to what those events mean.*

ANTHONY ROBBINS

*Often do the spirits  
Of great events stride on before the events,  
And in to-day already walks to-morrow*

SAMUEL TAYLOR COLERIDGE

*To understand reality is not the same as to know about outward events. It is to perceive the essential nature of things. The best-informed man is not necessarily the wisest. Indeed there is a danger that precisely in the multiplicity of his knowledge he will lose sight of what is essential. But on the other hand, knowledge of an apparently trivial detail quite often makes it possible to see into the depth of things. And so the wise man will seek to acquire the best possible knowledge about events, but always without becoming dependent upon this knowledge. To recognize the significant in the factual is wisdom.*

DIETRICH BONHOEFFER

*In the event of a water landing, I have been designed to act as a flotation device.*

DATA (BRENT SPINER), IN *STAR TREK: INSURRECTION*

### Objectives

Upon completion of this chapter, you will be able to:

- ☐ Create new events in Alice
- ☐ Create handler methods for Alice events
- ☐ Use events to build interactive stories

Most of the programs we have written so far have been scenes from stories that, once the user clicks Alice's **Play** button, simply proceed from beginning to end. For some of our **interactive programs**, the user must enter a number or a string, but entering such values via the keyboard has been all that we have required of the user in terms of interaction with the program.

When a user clicks Alice's **Play** button for a program, it *triggers* a change in the program — usually creating a flow that begins at the first statement in `world.my_first_method()`. An action by the user (or the program) that causes a change in the program is called an **event**. For example, clicking Alice's **Play** button triggers a **When the world starts** event.

Alice supports a variety of events, including those listed in Figure 6-1.<sup>1</sup>

Alice Event	Triggered By	Triggered When
When the world starts	the user	the user clicks Alice's <b>Play</b> button
*While the world is running		the world is running
When a key is typed	the user	the user releases a keyboard key
*While a key is pressed		the user holds down a keyboard key
When the mouse is clicked on something	the user	the user clicks the left mouse button while pointing at an object
*While the mouse is pressed on something		the user holds down the left mouse button while pointing at an object
While something is true	the program	a condition remains true
*When something becomes true		a condition becomes true
When a variable changes	the program	a variable changes its value
Let the mouse move <objects>	the user	the user moves the mouse
Let the arrow keys move <subject>	the user	the user presses one of the arrow keys
Let the mouse move the camera	the user	the user moves the mouse
Let the mouse orient the camera	the user	the user moves the mouse

FIGURE 6-1 Alice events

1. An event marked with an asterisk (\*) is accessible by (1) creating the event above it in Figure 6-1, (2) right-clicking on that event, and then (3) choosing **change to ...** from the menu that appears.

There are two steps to making a program respond when an event occurs:

1. Choose (or define) a method providing the behavior to occur in response to the event.
2. Tell Alice to invoke that method whenever the event occurs.

Invoking a method in response to an event is called **handling the event**, and a method that is invoked in response to an event is often called an **event handler**. A program that solves a problem or tells a story mainly through events and handlers is called an **event-driven program**.

In the rest of this chapter, we will see how to build event-driven programs. While we will not cover all Alice events, we will provide a representative introduction to what they can do.

## 6.1 Handling Mouse Clicks: The Magical Doors

To let us see how an event-driven world differs from those we have built before, let us revise the scene-story from Section 5.2.3 as follows:

---

*Scene: A castle has two magical doors. When the user clicks on the right door, it opens; but when the user clicks on the left door, it tells the right door a random knock-knock joke.*

---

Because some of the behavior is the same as in the world we built in Section 5.2.3, we will begin with that world. As before, the initial shot is as shown in Figure 6-2.

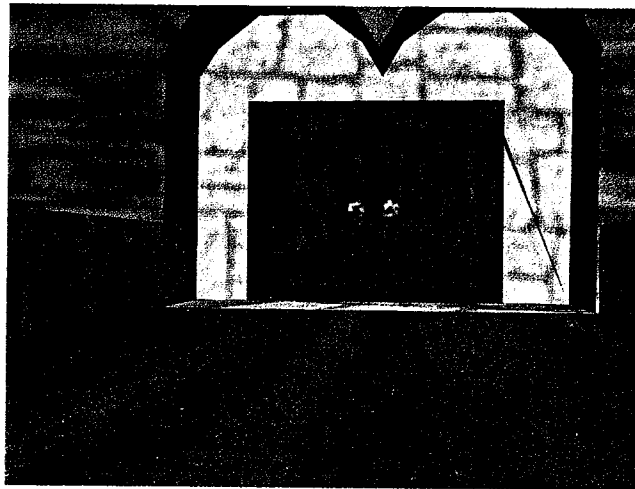


FIGURE 6-2 The castle doors

We will deal with each door separately. However, before we begin, we should prevent the left door from telling jokes when we run the world. To do so, we delete the `tellRandomKnockKnockJoke()` message from `my_first_method()`.<sup>2</sup>

### 6.1.1 The Right Door

You may recall from the Alice tutorials that handling mouse clicks is easy in Alice. To do so, we follow two steps:

1. If the event should trigger behavior that requires more than one message, we define a method that produces that behavior. This method will be the handler.
2. We create a new event in the *events area* that invokes the handler — either the method we defined in Step 1, or the single message that produces the required behavior.

To illustrate, the behavior to make the right door open can be elicited with a single message, `turn()`, so we need not create a new method. Instead, we proceed to Step 2 by clicking the **create new event** button and choosing **When the mouse is clicked on something** from the menu that appears, as shown in Figure 6-3.

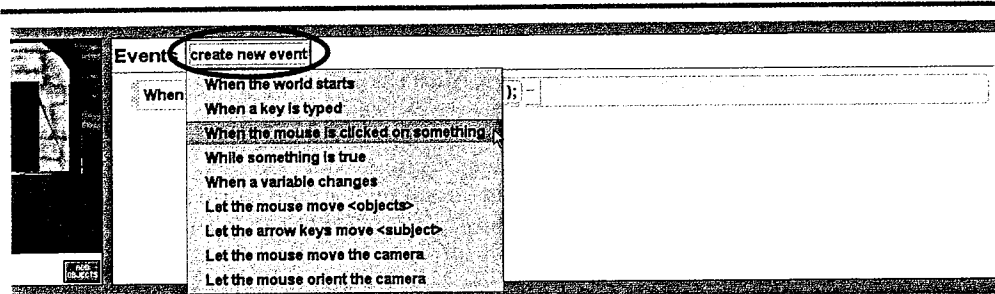


FIGURE 6-3 Creating a new mouse event

When we select this choice, Alice creates a new event in the *events area*, as shown below.

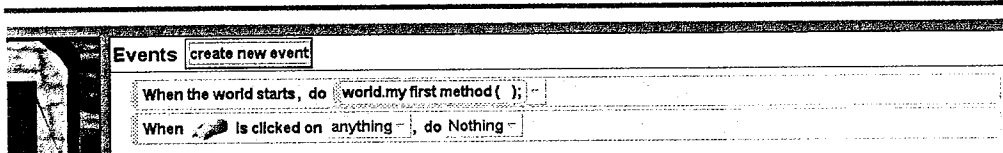


FIGURE 6-4 A new mouse event

2. Alternatively, we could achieve the same effect by deleting **When the world starts do `world.my_first_method()`** from the *events area*.

To satisfy the user story, we need this event to be triggered by clicking on the right door (`castle.door1`), rather than *anything*. To make this happen, we click the list arrow next to *anything* and select `castle->door1`, modifying the event as shown in Figure 6-5.



FIGURE 6-5 A mouse event for the right door

The object from which an event originates — `castle.door1` in this case — is called the **event source**.

To handle this event, we can replace *Nothing* with the message `castle.door1.turn(LEFT, 0.25)`; by opening up the `castle` in the *object tree*, selecting `door1`, and then from the *methods* pane of the *details area*, dragging the `turn()` message to the *events area* and dropping it on *Nothing*. Figure 6-6 shows the resulting event.

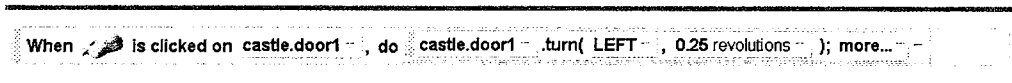


FIGURE 6-6 Handling the mouse event for the right door

Congratulations — you have just handled your first event! When we click Alice's **Play** button, nothing happens until the user clicks the right door, at which point it swings open.

### 6.1.2 The Left Door

Dealing with the left door is nearly as easy as the right door, but only because we already have a method that makes the left door tell a random knock-knock joke (see Figure 5-33). That is, if we had not already written `world.tellRandomKnockKnockJoke()`, we would have to write a handler method for this event, as described in Step 1 above.

Since we already have a method to serve as a handler, we can proceed to Step 2 of the event steps. To do so, we use the same approach we saw in Figure 6-3 through Figure 6-6, but specifying `castle.door2` as the source of this event, and dragging-and-dropping `world.tellRandomKnockKnockJoke()` as its handler. This is shown in Figure 6-7.

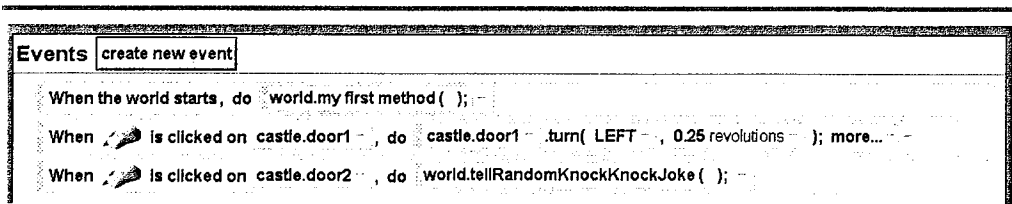


FIGURE 6-7 Handling the mouse event for the left door

That's it! Now, when we click Alice's **Play** button, clicking on the left door produces a random knock-knock joke, while clicking on the right door causes that door to open.

Note that, unlike past worlds we have built, `world.my_first_method()` does nothing in this new world. Instead, all of the interesting behavior lies in the handler methods, which are triggered by the event of the user clicking the mouse.

### 6.1.3 The Right Door Revisited

If we test the world thoroughly, we find that the right door opens correctly *the first time* we click on it. However if we subsequently click on the right door again, it turns left *again* (precisely what we told it to do). The mistake lies in the *logic* we used in defining how that door should behave. Such mistakes are called **logic errors**. A better response to a mouse click on the right door would be to open the door if it is closed and to close the door if it is open.

It is important to see that it is okay to revise the user story when testing reveals a weakness. Just as a filmmaker may rewrite a scene the night before it is shot, a programmer may have to rewrite a part of the user story to improve the overall program.

Generating the new behavior requires more than one message, so we will write a handler method and invoke it in place of the `turn()` message shown in Figure 6-6.

#### Design

To design this method, we can revise the right door part of the user story as follows:

---

*When the user clicks on the right door, if it is closed, it opens; otherwise, it closes.*

---

Notice that the revised story contains the magic word *if*. This strongly suggests that the method will need an **if** statement.

#### Programming: Storing the Door's State

One way to produce this new behavior is to add a new property to the castle, to indicate whether or not its right door is closed.<sup>3</sup> To do so, we select **castle** in the *object tree*, click the *properties* tab in the *details area*, and then click the **create new variable** button, as we saw back in Section 3.3.

The right door is in one of two states: either it is *closed* or it is *open*. This means that we can represent whether or not the right door is closed with a **Boolean** variable named **rightDoorClosed**, using **true** to represent the *closed* state and **false** to represent the *open* state. Since the door is closed at the outset, its initial value should be **true**, as shown in Figure 6-8.

---

3. Since this is a characteristic of the right door, such a property really should be defined in `castle.door1`. Unfortunately, Alice only lets you add properties, methods, and questions to an object at the "top" level of the *object tree*, so the best available place to store this property is **castle**.



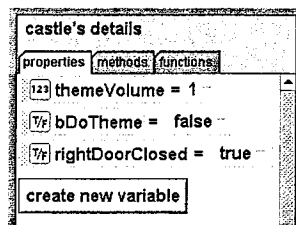


FIGURE 6-8 Storing the state of the castle's right door

When the user clicks on the right door, the handler method for that event can use an **if** statement to determine which way to **turn()** it (**LEFT** or **RIGHT**), and then update the value of **rightDoorClosed** from **true** to **false** (or vice versa) to reflect the door's changed state.

### Programming: Defining the Handler

We create a handler method the same way as any other method: by choosing the *methods* tab in the *details area*, clicking the **create new method** button, naming the method, and then defining its behavior. As usual, if a handler method affects the behavior of a single object, it should be defined within that object; otherwise it should be stored in the *world*. Figure 6-9 shows the **castle.openOrCloseRightDoor()** method.

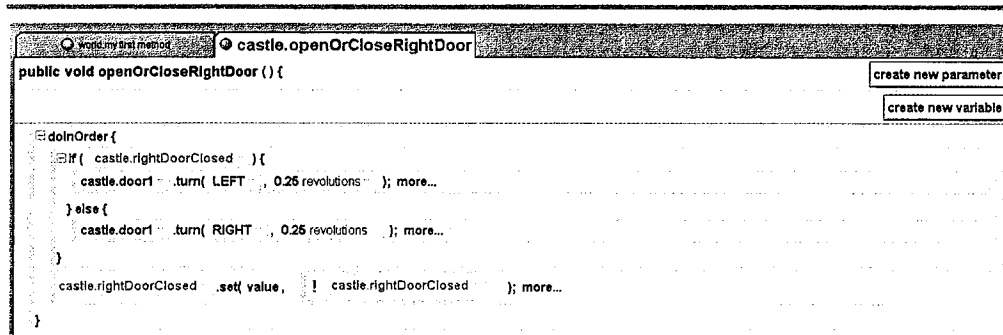


FIGURE 6-9 Handling clicks on the castle's right door

The last statement in Figure 6-9 uses the logical *not* operator (!) we saw in Section 4.1.4 to invert the value of **rightDoorClosed** from **true** to **false** when opening the door, and from **false** to **true** when closing the door.

The approach shown in Figure 6-9 can be generalized into a standard pattern for situations in which an object can be in one of two states (for example, *open-closed*, *in-out*, *on-off*, etc.) to switch the object from one state to the other. We can generalize the pattern for such **two-state** behavior as follows:

```

if (booleanStateVariable) {
    // do what is needed to change the object
    // from the first state to the second state
} else {
    // do what is needed to change the object
    // from the second state to the first state
}
booleanVariable = !booleanVariable. // update the state variable

```

### Programming: Handling the Event

Given the `openOrCloseRightDoor()` handler method shown in Figure 6-9, we can finish the program by specifying that it be the handler for the **When the mouse is clicked on castle.door1** event, by dragging the new method and dropping it on top of the previous handler (the `castle.door1.turn()` message). Figure 6-10 shows the resulting *events area*.

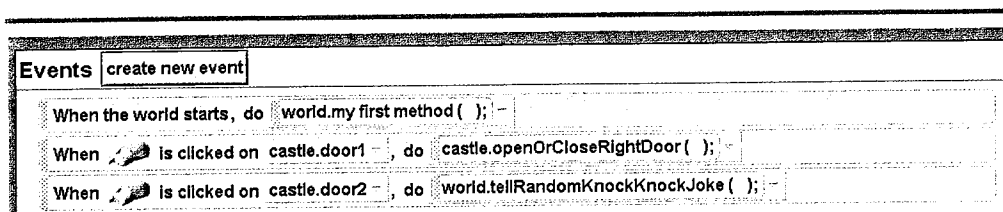


FIGURE 6-10 The (revised) *events area*

Now, a click on the closed right door opens it, and a click on the open right door closes it.

### 6.1.4 Event Handling Is Simultaneous

One tricky thing about events is that two events can occur almost simultaneously, requiring their handlers to run at the same time. For example, in the “castle doors” program we wrote in this section, a user could click on the left door and then click on the right door, before the left door finishes the joke. Figure 6-11 shows two screen captures under these circumstances.

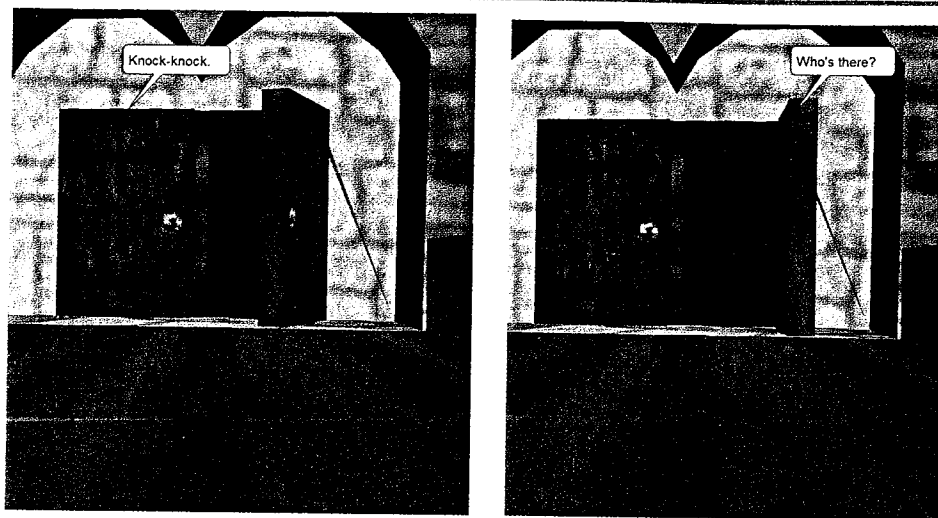


FIGURE 6-11 Handling simultaneous events

As shown in Figure 6-11, Alice handles such simultaneous events quite well. The first (left) screen capture shows the left door beginning its knock-knock joke while the right door is opening. The handlers for the left and right doors thus run simultaneously. When the right door's handler finishes, the left door's handler keeps running, as seen in the second screen capture.

Handlers running simultaneously usually work well, but if two running handlers both modify the same property of an object, a conflict may arise. For example, to tell a knock-knock joke, the left door's handler sends both doors `say()` messages. If the right door's handler also sent either door a `say()` message, then the simultaneous performance of both handlers would interfere with the joke and create a conflict. To avoid such conflicts, avoid designing programs in which two different events simultaneously modify the same property of the same object.

### 6.1.5 Categorizing Events

In this section, we have seen how to handle a **mouse event** — an event that is triggered when the user moves the mouse or clicks a mouse button. A **keyboard event** is triggered when the user presses a keyboard key. Because they are initiated by a user action, mouse and keyboard events are both known as **user events**. By contrast, a **program event** is triggered when the world starts running, or the program changes the value of a variable or condition.

## 6.2 Handling Key Presses: A Helicopter Flight Simulator

Now that we have seen how mouse click events can be handled, let's look at keyboard events.

## 6.2.1 The Problem

---

*Scene: Catastrophel! The mayor's cat is lost, bringing your city's government to a halt. As the city's only helicopter pilot, you have been asked to help find the mayor's cat. You find yourself in the cockpit of a running helicopter at an airport outside the city. Fly the helicopter and find the mayor's cat.*

---

## 6.2.2 Design

We can build the scene using the classes from the Alice Gallery (see below), but let's first spend a few minutes thinking what the user must do: how he or she will fly the helicopter.

Flying a helicopter is complicated: the user needs to be able to make the helicopter move *up*, *down*, *forward*, *backward*, and turn *left* or *right*. It would be difficult to elicit all six of these behaviors using a mouse, so we will instead use keyboard keys for each of them. After a bit of thought, we might decide to operate the helicopter as follows:

---

To make the helicopter *ascend* (take off), use the 'a' key. To *descend* (land), use the 'd' key. When the helicopter is in the air, use the up and down arrow keys to move it forward and backward. Similarly, when the helicopter is in the air, use the left and right arrow keys to turn it left and right.

---

These keys are chosen for their:

- **Mnemonic values:** 'a' is the first letter in *ascend*, and 'd' is the first letter in *descend*, making these keys easy to remember. Likewise, the up, down, left, and right arrow keys point in the directions we want the helicopter to move, making their meanings easy to remember.
- **Convenient positions:** 'a' and 'd' are near one another on most keyboards, allowing the user to easily control the helicopter's elevation with two fingers of one hand. Likewise, the four arrow keys are usually grouped together, allowing the user to easily control the helicopter's forward, backward, left, and right motion with the fingers on the other hand.

It is important to consider *human factors* when building interactive stories. If the story requires complex behaviors, make the controls for your user as convenient and easy to use as possible. Making programs easy to use is an important aspect of programming known as **usability**.

## 6.2.3 Programming in Alice

To construct the scene, we can build an Alice world containing an airport, a city terrain, an assortment of buildings, a helicopter, and a (Cheshire) cat. After arranging the buildings to resemble a small city, we place the cat somewhere within the city (exactly where is for you to find out), position the helicopter at the airport, and then position the camera to be peering out the front of the helicopter. We then set the **camera's vehicle** property

to be the **helicopter**, so that any message that moves the **helicopter** will also move the **camera**.

### Making the Helicopter's Propellor Spin

The helicopter's engine is running when the story begins, so its propellor should be spinning when the world starts. The **Helicopter** class has a method named **heli blade()** that continuously spins its **propellor** and **rotor**. By using this method as the handler for Alice's default **When the world starts** event (see Figure 6-12), the helicopter's propellor begins spinning as soon as we press Alice's **Play** button.

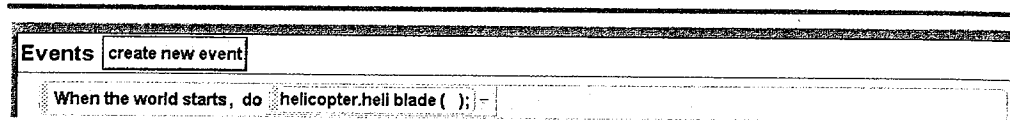


FIGURE 6-12 Making the helicopter's propellor spin

### Making the Helicopter Ascend

A helicopter must be in the air before it can go forward or backward, turn left or right, or descend. As a result, it makes sense to define the **helicopter.ascend()** method first, while keeping those other operations in mind.

Our other five operations must be able to determine if the helicopter is in the air. (If it is not, those operations should do nothing.) To store this information, we can create a **Boolean** property for the **helicopter** named **inTheAir**, initially **false**, as shown in Figure 6-13.

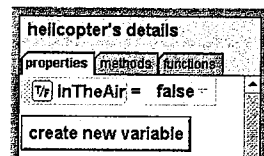


FIGURE 6-13 The **helicopter.inTheAir** property

With this property in place, we can define the **ascend()** method as shown in Figure 6-14.

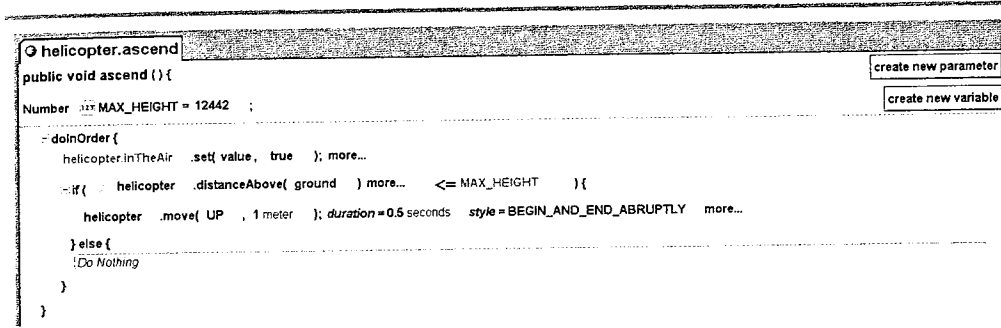


FIGURE 6-14 The ascend() method

Since helicopters cannot fly infinitely high, we first define a constant named **MAX\_HEIGHT** and set its value to the maximum altitude a helicopter has attained (according to the Internet, 12,442 meters). The body of the method then sets **helicopter.inTheAir** to **true**, and moves the helicopter up 1 meter if it has not already attained the maximum altitude. To smooth the animation, we set the **move()** message's **style** attribute to **BEGIN\_AND\_END\_ABRUPTLY**. To make it move upwards at a reasonable rate, we set its **duration** attribute to **0.5** seconds, which effectively makes the helicopter ascend at 2 meters per second.

With the **ascend()** method defined, our next task is to associate it with the 'a' keyboard event. To do so, we click the **create new event** button in the *events area*, and select **When a key is typed**, as shown in Figure 6-15.

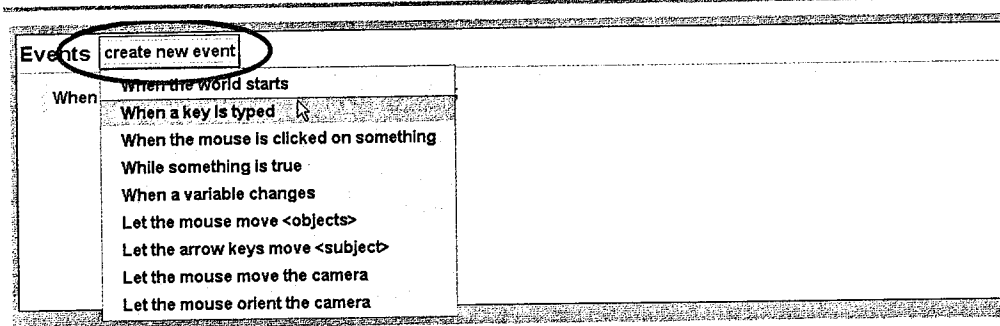


FIGURE 6-15 Creating a new keyboard event

Alice then generates the **When a key is typed** event shown in Figure 6-16.

---




---

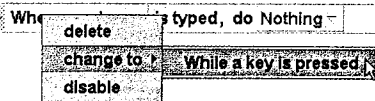
FIGURE 6-16 The when a key is typed event

---

If we use this event to make the 'a' key trigger the `ascend()` method, then each press of the 'a' key will move the helicopter up 1 meter. Put differently, to climb just 100 meters, the user would have to press the 'a' key 100 times, which is no fun for the user!

A better approach is to use the **While a key is pressed** event from Figure 6-1. As indicated there, we can convert a **When a key is typed** event to a **While a key is pressed** event by right-clicking on the **When a key is typed** event, and then selecting **change to -> While a key is pressed** from the menu, as shown in Figure 6-17.

---



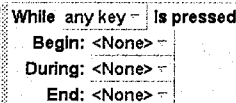

---

FIGURE 6-17 Changing when a key is typed into while a key is pressed

---

Selecting this choice causes Alice to replace the **When a key is typed** event with a **While a key is pressed** event, as shown in Figure 6-18.

---




---

FIGURE 6-18 The while a key is pressed event

---

As seen in Figure 6-18, this event allows *three* different handlers to respond to one key event:

- **Begin:** a handler here is performed *once*, when the key is first pressed.
- **During:** a handler here is performed *continuously*, as long as the key remains down.
- **End:** a handler here is performed *once*, when the key is released.

For the problem at hand, we will only need one of these parts: the **During** part.

To specify that we want the 'a' key to trigger this event, we click the list arrow next to **any key** and choose **letters->A** from the menu that appears, as shown in Figure 6-19.

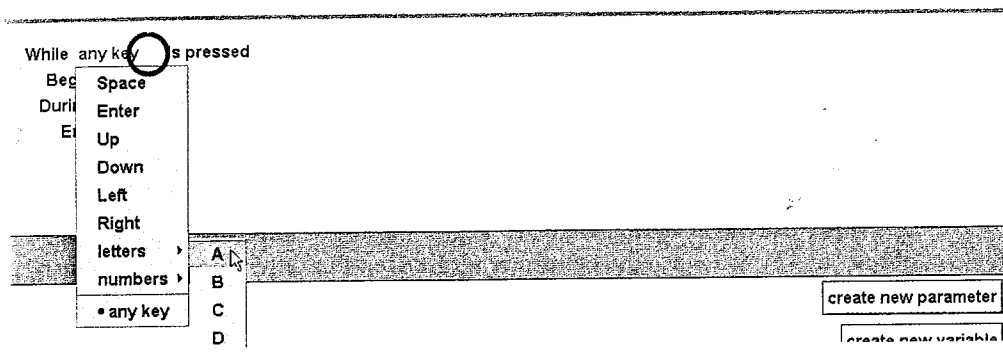
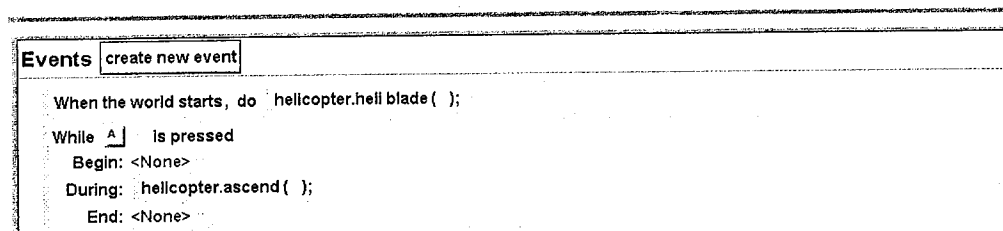


FIGURE 6-19 Making 'a' trigger an event

We then make the `ascend()` method the handler for the **During** part of this event, as shown in Figure 6-20.

FIGURE 6-20 Associating 'a' with `helicopter.ascend()`

With this event defined, holding down the 'a' key will make the `helicopter` ascend smoothly into the air, and set `helicopter.inTheAir` to `true`.

### Making the Helicopter Descend

To make the helicopter descend, we write a `descend()` method to serve as a handler for the 'd' keyboard event. If we think through its behavior, it has two things to accomplish:

1. If the `helicopter` is above the ground, the `helicopter` should move down 1 meter.
2. Otherwise, the `helicopter.inTheAir` property should be set to `false`.



Figure 6-21 presents a definition for `descend()` that achieves both of these goals.

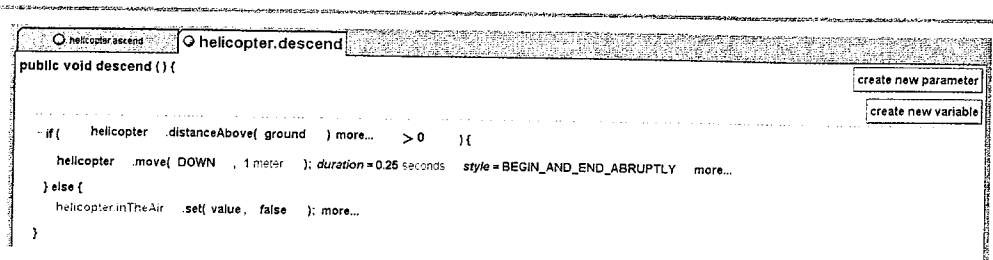


FIGURE 6-21 The `descend()` method

Our method first checks to see if the helicopter is above the ground. If not, it sets the `inTheAir` property to `false`. Since the `ascend()` and `descend()` methods control the helicopter's vertical movement, they are responsible for changing the value of `inTheAir` when necessary. No other methods should modify this property (though they will read its value).

If the helicopter is above the ground, our method moves it down 1 meter. As in `ascend()`, we set the `style` of the `move()` method to `BEGIN_AND_END_ABRUPTLY` to smooth the animation. Unlike `ascend()`, we set its `duration` to 0.25 seconds, so that the helicopter descends at a rate of 4 meters per second (to simulate the effect of gravity on its descent).

We can associate the `descend()` method with the 'd' event using the same approach we used for 'a' in Figure 6-15 through Figure 6-19. The result is the event shown in Figure 6-22.

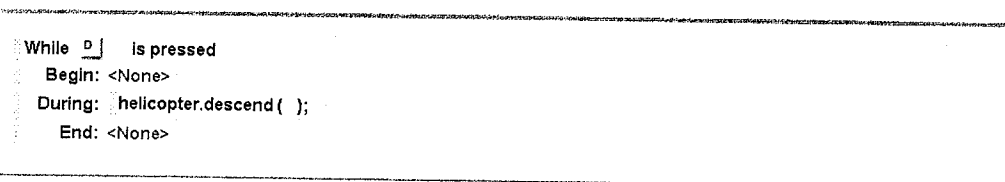


FIGURE 6-22 Associating 'd' with `helicopter.descend()`

With this event in place, we can now use 'd' key to land the helicopter!

### The Arrow Keys

As we saw in Figure 6-1, Alice provides a `Let arrow keys move <subject>` event. You might be tempted to use this event to control the `helicopter` in the program, by creating an event like that shown in Figure 6-23.

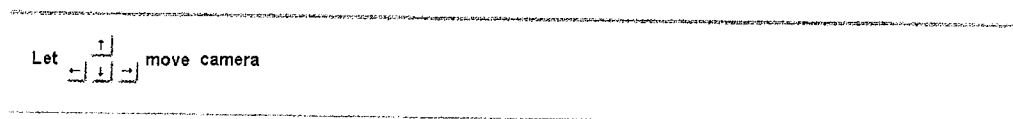


FIGURE 6-23 Control using the arrow keys?

This kind of an event works great in many situations, and it would be nice if it worked in ours. The problem for us is that a helicopter should *not* move unless it is in the air. If we were to use this event, then the arrow keys *would* cause the **helicopter** to move, even when it is “parked” on the ground! Moreover, Alice provides no easy way to modify the behavior triggered by this event. As a result, we will not use this event in the program. Instead, we will define four separate events: one for each of the four arrow keys.

The good news is that we will not need four separate event handlers. As we shall see, two methods are all we need to handle all four arrow events.

### Making the Helicopter Turn

To make the helicopter turn left or right, we could define two separate methods like we did for ascending and descending. However, these methods would be nearly identical, differing only in the direction we want the helicopter to turn. Instead of defining separate methods, we will use a single method, and pass an argument (**LEFT** or **RIGHT**) to specify the direction we want the helicopter to turn. The basic logic is as shown in Figure 6-24.

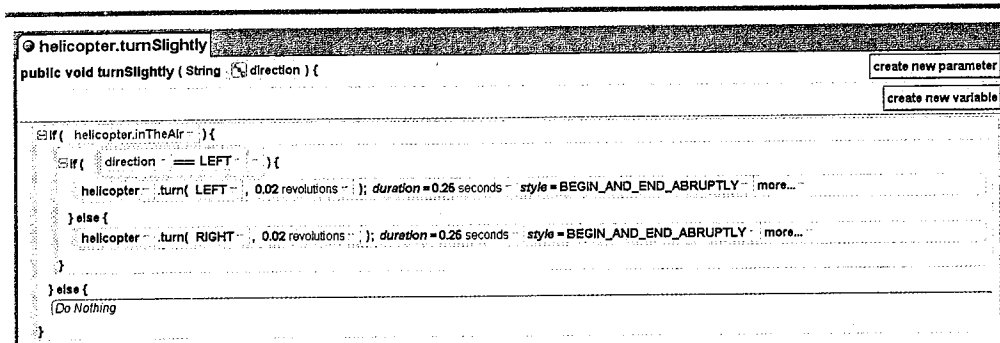


FIGURE 6-24 The `turnSlightly()` method

We named the method `turnSlightly()`, to keep it distinct from the existing `turn()` method, and because it only turns the **helicopter** a small, fixed amount (0.02 revolutions).

With this definition to serve as a handler, we can associate it with the left and right arrow keys using the approach shown in Figure 6-15 through Figure 6-19, but passing **Left** as an argument to the handler for the left arrow key event, and **Right** as an argument to the handler for the right arrow key event. Doing so produces the two events shown in Figure 6-25.

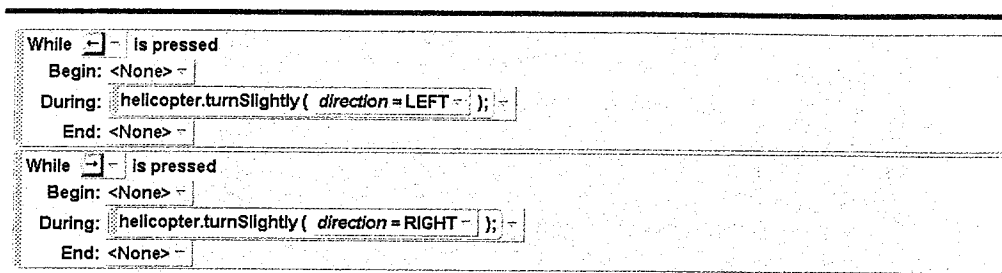
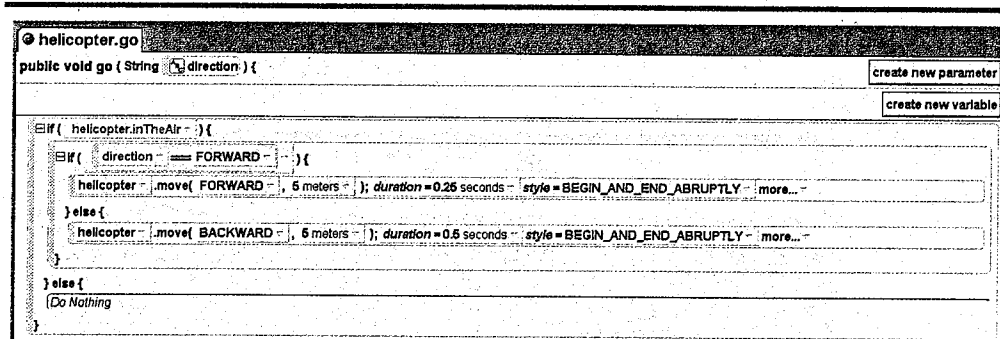


FIGURE 6-25 The left and right arrow events

Using these events, we can turn the helicopter left or right, but only when it is in the air.

### Moving the Helicopter Forward or Backward

Our final operations are to move the helicopter forward or backward in response to the up and down arrow keys. As with turning the helicopter, we can do both of these in a single method, by passing **FORWARD** or **BACKWARD** as an argument to specify which direction to go. As in `turnSlightly()`, this method must only let the helicopter move if it is in the air. Figure 6-26 presents a definition for this method, which we have named `go()`.

FIGURE 6-26 The `go()` method

To make the helicopter move forward twice as fast as it goes backward, we use a *distance* of 5 meters for each, but a *duration* of 0.25 seconds for forward and 0.5 seconds for backward. In each case, we use the **BEGIN\_AND\_END\_ABRUPTLY** style to smooth the animation.

With a handler in place, all we have to do is associate it with the appropriate up and down arrow key events, using the same approach we have seen before, as shown in Figure 6-27.

```

While ↑ is pressed
  Begin: <None>
  During: helicopter.go ( direction = FORWARD );
  End: <None>

While ↓ is pressed
  Begin: <None>
  During: helicopter.go ( direction = BACKWARD );
  End: <None>

```

FIGURE 6-27 The up and down arrow events

At this point, our program is operational, provided there is someone there to explain to users what they are supposed to do. In the next section, we will see how to add instructions.

### 6.3 Alice Tip: Using 3D Text

In the last section, we built a working helicopter flight simulator. However, for a program with such a complex user interface (the user must use both hands and operate six keys), it is a good idea to present some operating instructions when the program begins.

To do so, we return to Alice's **Add Objects** screen, and click the **Create 3D Text** button (at the far end of Alice's **Local Gallery**), as shown in Figure 6-28.

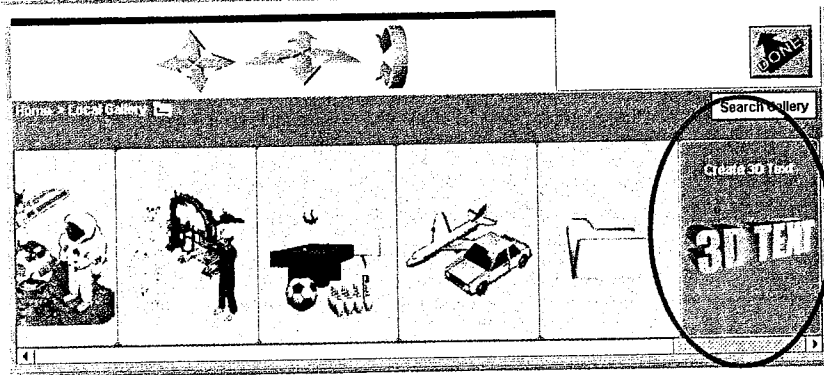


FIGURE 6-28 The create 3D text button

When this button is clicked, Alice displays the **Add 3D Text** dialog box shown in Figure 6-29.

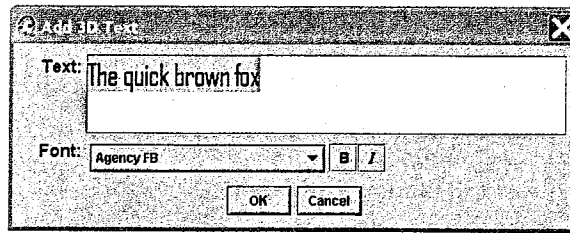


FIGURE 6-29 The Add 3D Text dialog box

Using this dialog box, we can replace **The quick brown fox** with any textual information we want to appear in our world, such as:

- *instructions* for the user
- an *opening title* for the story
- *closing credits* for the story

and so on. We can specify that the text be displayed in a particular font using the **Font** drop-down box, and make the text bold or italic using the **B** and **I** buttons.

To make the instructions for the helicopter, we can type the text shown in Figure 6-30.

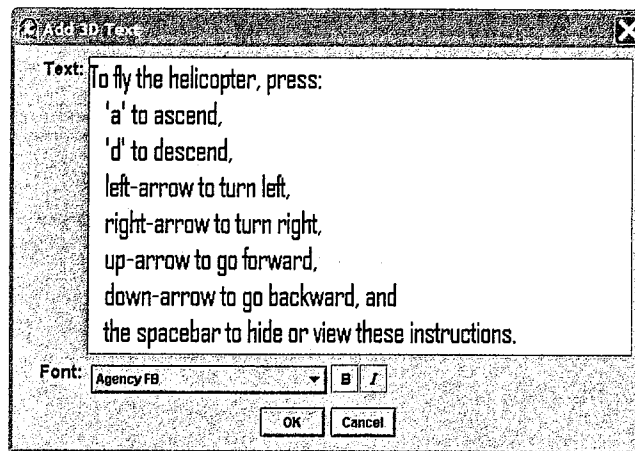


FIGURE 6-30 Instructions for the flight simulator

When we click the **OK** button, Alice inserts a three-dimensional version of these instructions into the world, and names it **To fly the helicopter**. To discuss it more conveniently, we will right-click on it and rename it **instructions**.

### 6.3.1 Repositioning Text that Is Off-Camera

If we have moved the camera from its original position, Alice will add new 3D text at a position that is off-camera. To position the text in front of the camera, we can use these steps:

1. Right-click on **instructions** in the *object tree*, and then choose **methods->setPointOfView(<asSeenBy>->camera**. This moves the text to be in the same position and orientation as the camera.
2. Right-click on **instructions** in the *object tree*, and then choose **methods->move(<direction>,<amount>)->FORWARD->10 meters**. This moves the text forward so that we can see it. However since its point of view is like that of the camera, its front is facing away from us, making it backwards to our view.
3. Right-click on **instructions** in the *object tree*, and then choose **methods->turn(<direction>,<amount>)->LEFT->1/2 revolution**. This spins the text 180 degrees, making it readable to us.

From here, we can use the controls at the upper right of the **Add Objects** window to resize and reposition the text as necessary to make the **instructions** fit the screen.

We will make the instructions appear or disappear whenever the user presses the spacebar. One way to make this occur is to make the **instructions** *move with the camera*, so that the spacebar event handler just has to make them visible to make them appear, or invisible to make them disappear.<sup>4</sup> To make them move with the camera, we set the **instructions.vehicle** property to **camera** in the *details area*. At the same time, we set the **instructions.color** property to **yellow**, to (in theory) make them show up well.

### 6.3.2 Adding a Background

Unfortunately, when we view the results of our work, the **instructions** are nearly impossible to read, as can be seen in Figure 6-31.

4. This approach is simple because the camera moves with the helicopter which flies throughout the world. For text that should only appear once (like titles or credits), it makes more sense to build a dedicated scene for displaying the text, and then moving the camera to and from that scene when necessary.

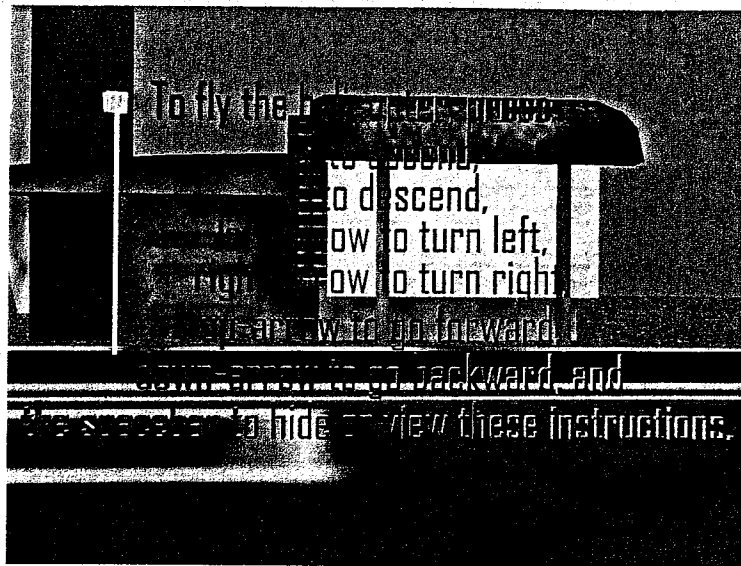


FIGURE 6-31 Instructions that are hard to read

To improve their visibility we can add a *background* behind the **instructions**. To make such a background, we can add an object from the **Shapes** folder in Alice's **Local Gallery**, as shown in Figure 6-32.

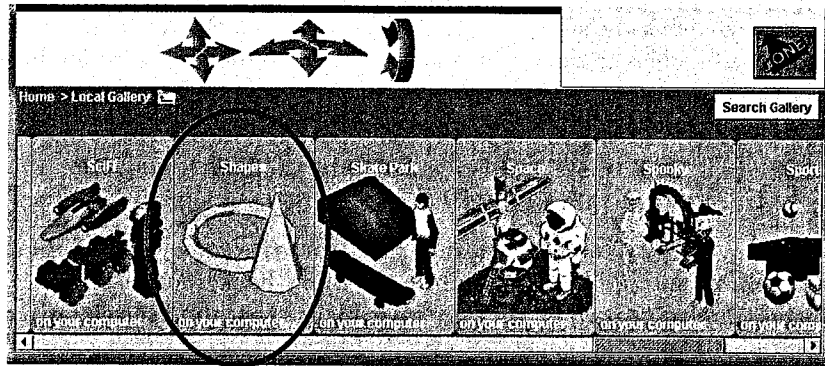


FIGURE 6-32 Alice's shapes folder

There, we find a **Square** class that (since our view is rectangular) we can use as a background for the **instructions**. If we drag and drop it into the world, we can use the controls at the upper right of the **Add Objects** window to resize the **square** to fill the screen, and reposition the **square** so that it is behind the **instructions**. (Or reposition

the **instructions** to be in front of the **square**.) When we are finished, we can increase the contrast by setting the **square.color** property to **black**, and using right-click->**methods** to make the **light** turn to face these instructions.<sup>5</sup> The result is the easier-to-read screen shown in Figure 6-33.

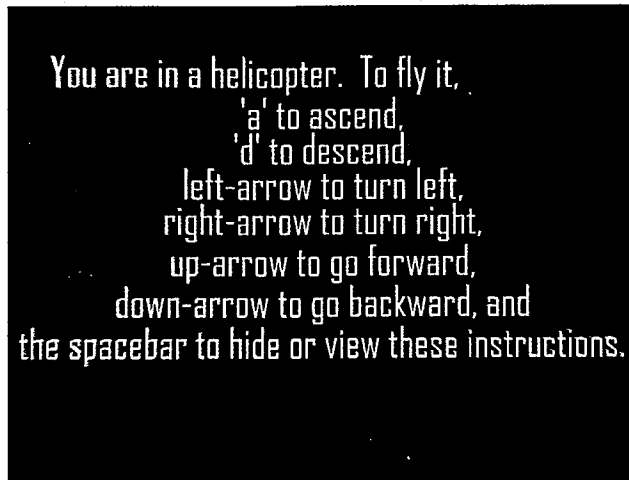


FIGURE 6-33 Instructions that are easy to read

Since we want the background to move with the **instructions**, we set the **square.vehicle** property to be **instructions**.

This approach can be used to create any kind of on-screen text we want to appear in the story, such as titles or credits. Once we have one 3D text object in the right position, we can add others to the world, and move them to the same position by using right-click->**methods**->**objectName.setPointOfView(<asSeenBy>)** to move new text to the position and orientation of the existing 3D text.

### 6.3.3 Making Text Appear or Disappear

We are almost done! Our next task is to write an event handler that makes the **instructions** and **square** (the instructions' background) disappear when they are visible, and appear when they are invisible, so that the user can make them appear or disappear using the spacebar.

This is another example of the two-state behavior we saw back in Figure 6-9. However, **instructions** and **square** already have an **isShowing** property that indicates whether or not they are visible, so we need not define any new properties. Instead, we can just toggle **instructions.isShowing** and **square.isShowing** to elicit the desired behavior, as shown in the **toggleInstructionVisibility()** method in Figure 6-34.

5. If we leave the light as it is, some portions of the 3D text may be darker than others, if the text is in the shadows. By making the light face the instructions, we illuminate the text uniformly, eliminating such shadows.



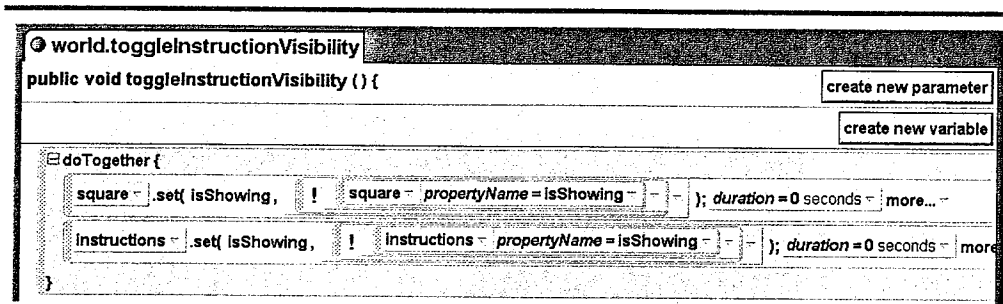


FIGURE 6-34 Toggling the visibility of instructions and square

When this method is performed, it uses the ! (NOT) operator to invert the **square** and **instruction** objects' **isShowing** properties. That is, if **isShowing** were **true** for each object *before* the method was performed, **isShowing** is **false** for each of them *after* the method finishes. Conversely, if **isShowing** was **false** for each of them *before* the method runs, **isShowing** is **true** for each *after* the method finishes.

To finish the program, we must make this method the handler for spacebar events. To do so, we can use the **When a key is typed** event shown in Figure 6-16, replace **any** with **Space**, and then drag the **world.toggleInstructionVisibility()** method onto **Nothing** to make it the event's handler. The result is the event shown in Figure 6-35.



FIGURE 6-35 Associating space with world.toggleInstructionVisibility()

Now, when the program begins running, the **instructions** appear as shown in Figure 6-33. When the user is ready and presses the spacebar, the instructions and background disappear, revealing the scene behind them. The scene that appears depends on the position and orientation of the **camera**. Figure 6-36 shows the scene.

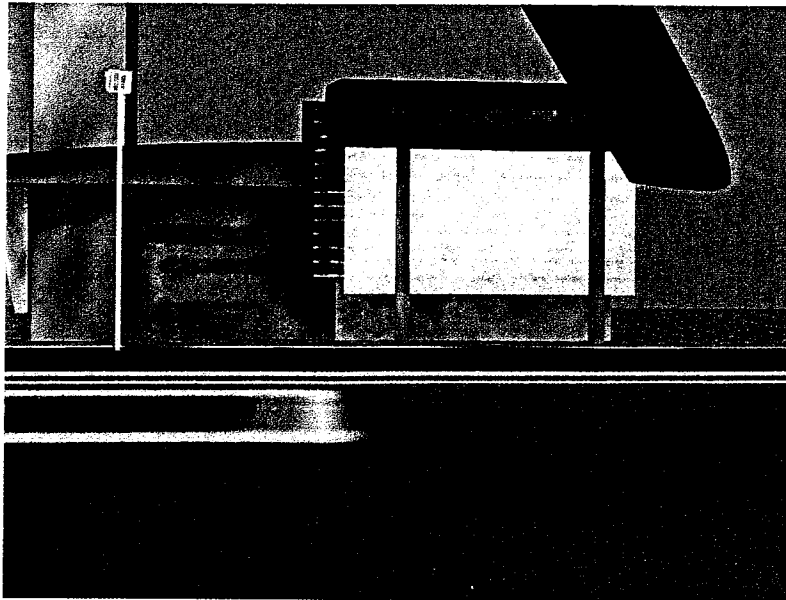


FIGURE 6-36 After the user presses the spacebar

The black object visible in the upper-right corner of Figure 6-36 is the helicopter's rotor blade.

At this point, we have a version of the program that is sufficient for testing with users. For additional enhancements (for example, adding a title, closing, and so on), see the Programming Problems at the end of the chapter.

---

*If 3D text objects or backgrounds are to be fixed in place in front of the camera, they should be the last objects you add to a world or scene. The reason is that they will usually lie between the camera and any objects you subsequently place in the world; if you try to click on these latter objects, the 3D text or background will intercept your click.*

---

## 6.4 Alice Tip: Scene Transitional Effects for the Camera

In Section 2.4, we saw how to use **Dummies** to mark camera positions, and how to use the `setPointOfView()` message to change the position of the **camera** to that of a dummy. This approach provides a convenient way to shift the camera from its position at the end of a given scene to a new position at the beginning of the next scene.

Instead of instantaneously jumping from the end of one scene to the beginning of the next scene (a transition called a *cut*), filmmakers often use special camera effects like *fades* or *wipes* to smooth the transition between scenes. Such **transition effects** can

make the transition between scenes seem less abrupt and jarring to the viewer, or be used to convey a sense of time elapsing between the scenes.

Alice does not provide any built-in transitional effects. However, it does provide us with raw building blocks that we can use to create our own. With a little time and effort, we can build credible transitional effects. In this section, we will see how to do so.

### 6.4.1 Setup for Special Effects

Before we see how to create the effects themselves, we need to do a bit of setup work. The basic idea is to add four black shutters or “flaps” outside of the camera’s viewport, that we can manipulate to create the effects. These shutters should be positioned at the top-left-right-bottom positions outside of the camera’s viewing area, as shown in Figure 6-37.

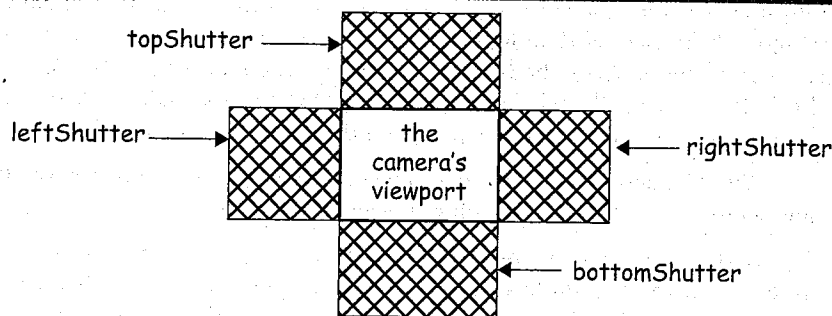


FIGURE 6-37 Surrounding the camera with four shutters

To create such shutters in Alice, we add four **square** objects to a world; change the **color** property of each **square** to **black**; change the **vehicle** property of each **square** to the **camera**; rename them **topShutter**, **leftShutter**, **rightShutter**, and **bottomShutter**; and position them as shown in Figure 6-38.

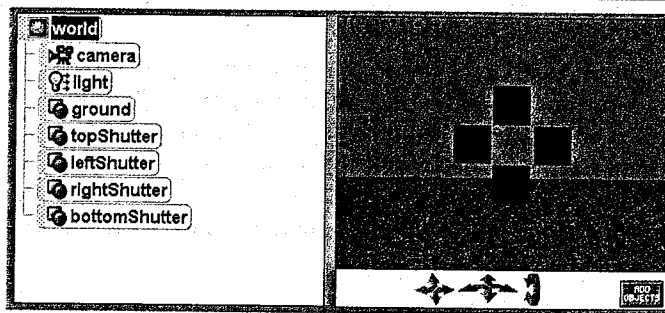


FIGURE 6-38 Using four squares for shutters

Moving each square to a position outside of the camera's viewing area is tricky, because we cannot easily drag them to the right position. Instead, we can position a square at the center of the screen, and then drag it towards the camera until it completely fills the viewing area. We can then use right-click->methods->**objectName.move(<direction>,<amount>)** with the appropriate arguments to move the square just outside the viewing area, using trial-and-error to find the right distance. (For us, this distance was about 0.08 meters.<sup>6</sup>)

With shutters in place, we can perform a variety of special effects by writing methods that move the shutters. We perform such effects using *complementary pairs* of methods, in which one method “undoes” the actions of the other.

## 6.4.2 The Fade Effect

Our first effect is a *fade* effect, which causes the entire screen to gradually darken until it is completely black, and then lightens, exposing a new scene. To achieve this effect, we write two complementary methods: **fadeToBlack()** and **fadeFromBlack()**. We can perform the fade-to-black effect by setting the **topShutter**'s opacity to zero percent, moving it down to cover the camera's viewport, and then setting its opacity back to 100 percent.

With the screen dark, we can move the camera to its position at the beginning of a new scene without the user seeing the scenery flash by.

With the camera in place for the new scene, we can perform a fade-from-black effect by setting **topShutter**'s opacity to zero percent, moving it up to its original position, and setting its opacity back to 100 percent, so that the **topShutter** is exactly as it was at the beginning of the fade-to-black effect.

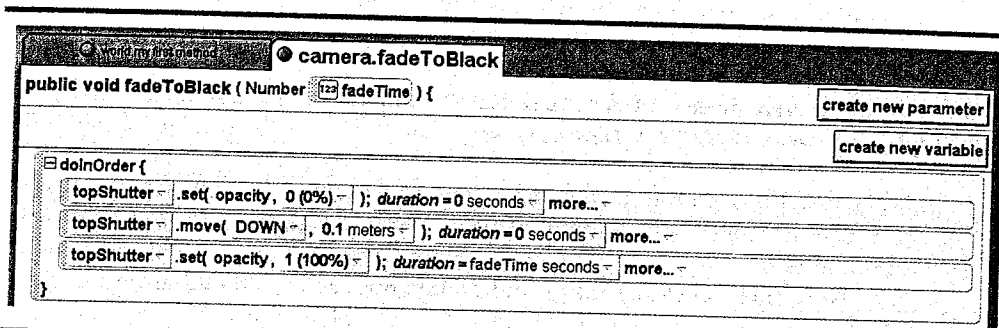
### Fade to Black

Using the **topShutter**, we can achieve the fade-to-black effect as follows:

1. Set the **opacity** property of **topShutter** to 0 percent, so that it is invisible.
2. Move **topShutter** down so that it is in front of the camera's viewing area.
3. Set the **opacity** property of **topShutter** back to 100 percent, making it visible.

For Steps 1 and 2, the **duration** should be 0 so that the steps happen instantaneously. For Step 3, the **duration** will determine how long the fade takes. While we could make this duration last a fixed length of time, a better approach is to let the sender of this message specify how long the fade should take. To let the sender pass this fade time as an argument, we must define a parameter to store it, and set the **duration** attribute of Step 3 to that parameter. Figure 6-39 shows the resulting definition, which we define within **camera**.

6. To move the **leftShutter** left and the **rightShutter** right, we had to turn each 180 degrees, as the way they were facing caused the LEFT and RIGHT directions to move them the opposite way.

FIGURE 6-39 The `fadeToBlack()` method

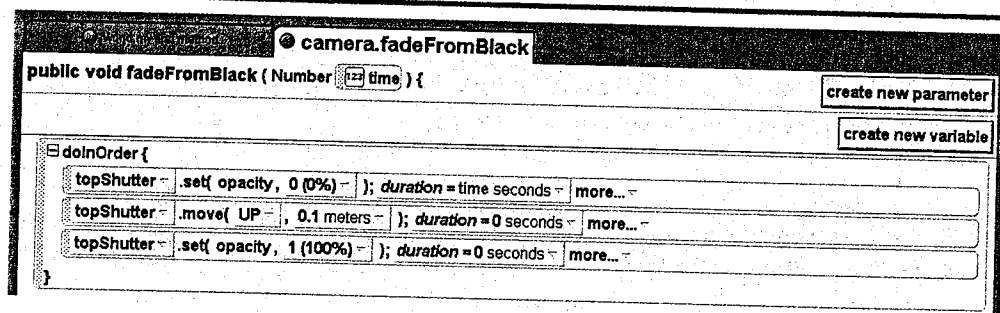
With this method, the message `camera.fadeToBlack(5)`; will cause the screen to darken over the course of five seconds. You may have to adjust the distance you move `topShutter`, depending on its placement and size with respect to the camera.

### Fade From Black

The fade-from-black method has to “undo” everything the fade-to-black method did, in the reverse order, so as to leave the `topShutter` in its original position:

1. Set the **opacity** property of `topShutter` to 0 percent.
2. Move `topShutter` up so that it is out of the camera's viewing area.
3. Set the **opacity** property of `topShutter` to 100 percent.

As before, we should allow the sender to specify the effect's *time*. In this method, it controls the **duration** of Step 1, while Steps 2 and 3 occur instantaneously, as shown in Figure 6-40.

FIGURE 6-40 The `fadeFromBlack()` method

With these two methods, the messages:

```
camera.fadeToBlack(4);
camera.setPointOfView(dummyForNextScene); duration = 0
camera.fadeFromBlack(3);
```

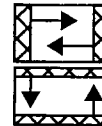
will cause the screen to change from light to dark over the course of four seconds at the end of one scene, and then change from dark to light over three seconds, with a new scene in view.

Note that `fadeToBlack()` and `fadeFromBlack()` should always be used in pairs, because each manipulates `topShutter` in a complementary way.

### 6.4.3 The Barndoor Edge Wipe Effect

*Edge wipe* effects are transitions in which one or more edges move across the screen to hide the end of one scene and expose the beginning of the next scene. One kind of edge wipe transition is the *barndoor wipe*, in which the shutters move like the doors of a barn, sliding closed at the end of a scene and then opening on a new scene. Two common barndoor edge wipes are:

- *vertical*, in which the “doors” close and open from the sides of the screen.
- *horizontal*, in which the “doors” close and open from the top and bottom of the screen.



In this section, we will show how to use the shutters to achieve the vertical effect. The horizontal effect is similar and is left for the exercises.

#### Vertical Barndoor Effects

We can perform a *vertical barndoor close* effect by simultaneously moving the left and right shutters towards one another. As before, the best approach is to let the sender of the message pass the effect's *time* as an argument, and then use that argument's parameter as the `duration` value for each shutter's movement. We define this method within `camera`, using the definition shown in Figure 6-41.

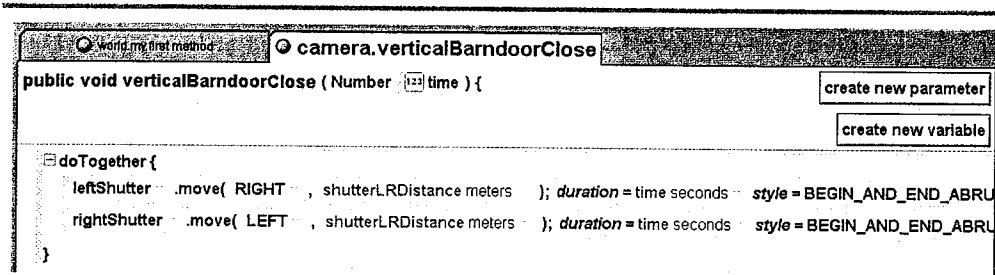


FIGURE 6-41 The `verticalBarndoorClose()` method

To simplify changing the distance the left and right shutters must move, we defined a **camera** property named **shutterLRDistance**, which we then used to control the shutter movements.

The complementary effect — the *vertical barndoor open* effect — can be achieved by simultaneously moving the left and right shutters apart, as shown in Figure 6-42.

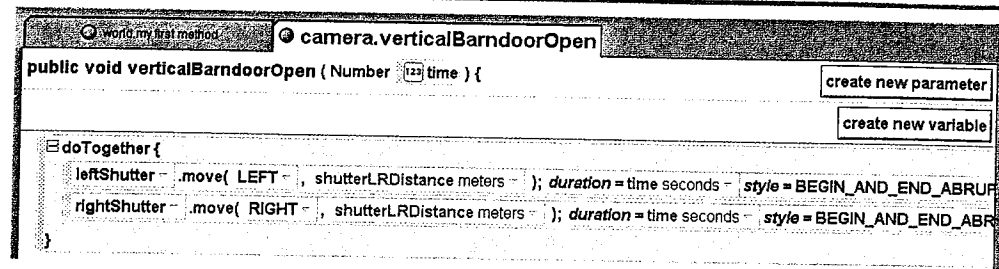


FIGURE 6-42 The `verticalBarndoorOpen()` method

Using these two methods, a programmer can send messages like this:

```
camera.verticalBarndoorClose(4);
camera.setPointOfView(dummyForNextScene); duration = 0
camera.verticalBarndoorOpen(3);
```

to “close the door” over the course of four seconds, shift the camera to the next scene, and then “open the door” over three seconds. Because they act as complementary operations, these methods should be used in pairs, or the shutters will not be in place for subsequent transitions.

The *horizontal barndoor edge wipe* is similar, but involves moving the top and bottom shutters instead of the left and right shutters. Building this effect is left as an exercise.

#### 6.4.4 The Box Iris Wipe Effect

Our last effect is a different wipe effect called an *iris wipe*, in which the screen is darkened except for an area called the *iris* that shrinks (the iris is closing) at the end of a scene, and expands (the iris is opening) to expose a new scene. This effect is usually used to center the viewer’s attention on something in the scene, which is encircled by the iris as it closes.

We can define a “box iris close” effect using the shutters, by simultaneously moving all four shutters towards the center of the camera’s viewport. As in the preceding effect methods, we let the sender of the message specify the *time* the effect should take. For added flexibility, we also let the sender specify what *percentage* the iris should close, as shown in Figure 6-43.



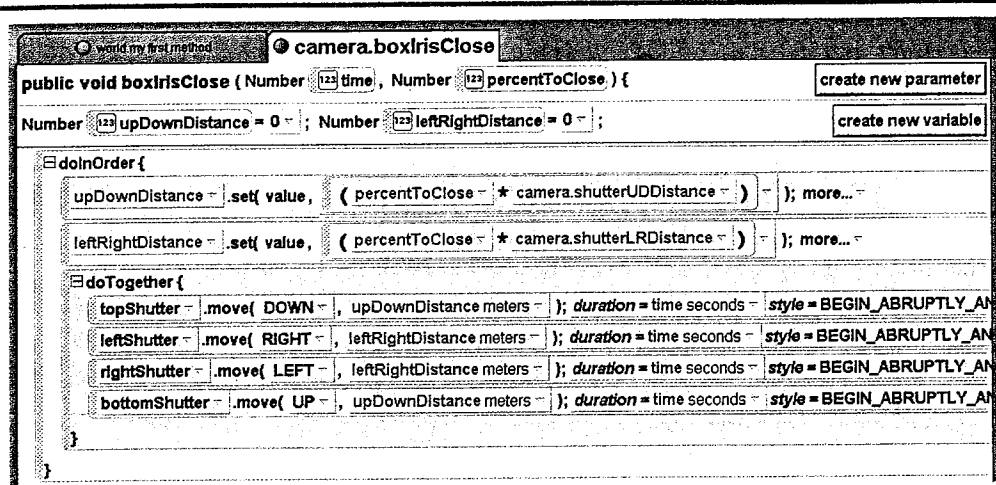


FIGURE 6-43 The boxIrisClose() method

We first compute how far to close each shutter, by multiplying the parameter **percentToClose** by each shutter's close distance. We then simultaneously move each shutter that distance using the value of parameter **time** as the **duration**. As shown in Figure 6-44, when this method runs, the shutters outline a shrinking box that contains roughly **percentToClose** of the screen area.

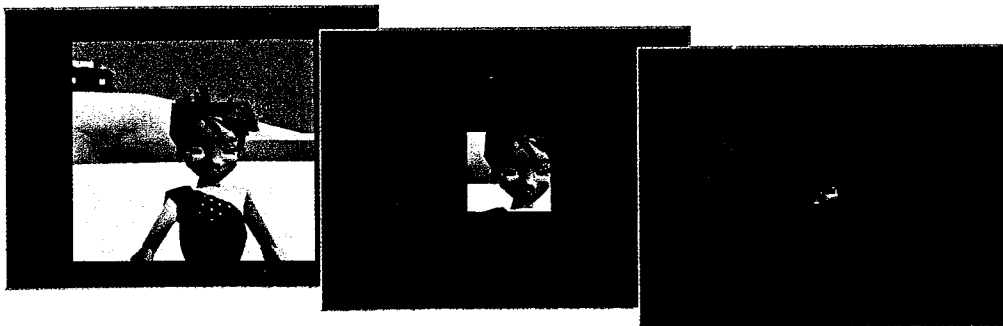


FIGURE 6-44 The box iris effect (closing)

The complementary **boxIrisOpen()** method is similar, as shown in Figure 6-45.



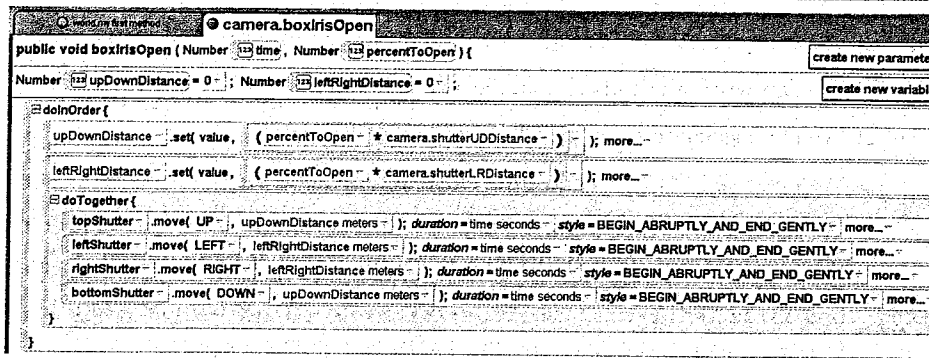


FIGURE 6-45 The boxIrisOpen() method

With these methods, we can now create interesting transitions:

```
camera.boxIrisClose(3, 0.75);
// do something interesting inside the iris
camera.boxIrisClose(2, 0.25);
// move camera to the next scene
camera.boxIrisOpen(5, 100);
```

## 6.4.5 Reusing Transition Effects

If you search on the Internet for terms like *transition*, *effect*, *fade*, and *wipe*, you can find many other transition effects that can be defined using techniques like those we presented in this section. (To define them, you may need to add more shapes to the camera.) We hope that this section has provided you with an introduction into how such effects can be created. However, once we have defined a nice group of transition effects, how do we reuse them in different programs?

Unfortunately, the **save object as...** technique presented in Section 2.3 will not save properties that are objects, so with the **camera**'s shutters being **Squares**, we cannot rename, save, and import the modified **camera** into a different world and have its shutters come with it, even if we were to make the four shutters properties of the **camera**.<sup>7</sup>

Instead, we define all of these transitions in a "template world" we call **TransitionEffects** that contains nothing but the **camera**, the **light**, the **ground**, and the **squares** we use for the transitions. For any story in which we want transitions, we open this **TransitionEffects** world as the starting world for the story, and then use **File->Save world as...** to save it using a name appropriate for that story. Of course, this means that we must plan ahead and know in advance that we will be using transitions in the story. This is one more reason to spend time carefully designing your program before you start programming.

7. This was true when this book was written. It may not be true by the time you read this. Check and see!

## 6.5 Chapter Summary

- ❑ We can create new events in Alice, including both mouse and keyboard events.
- ❑ We can write methods that act as event handlers.
- ❑ We can associate event handlers with specific events.
- ❑ We can use 3D text to add titles, instructions, and credits to a world.
- ❑ We can use the Alice **square** shape as a background for 3D text, and to create “special effects” for transitions between scenes. Alice shapes can be used as “building blocks” to build other structures in Alice.

### 6.5.1 Key Terms

event  
event-driven program  
event handler  
event source  
handling an event  
interactive program  
keyboard event

logic error  
mouse event  
program event  
transition effect  
two-state behavior  
usability  
user event

## Programming Projects

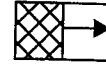
- 6.1 Choose one of the robots from the Alice Gallery, and provide events and handlers so that the user can control the robot using the keyboard. For example, use the arrow keys to make the robot go forward, backward, left, or right; use other keys to control the robot’s arms (or other appendages). Build a world in which the user must navigate the robot through obstacles.
- 6.2 Using the **dragon.flapWings()** method we wrote in Section 2.2.1, build a short story in which a dragon flies from place to place in search of adventure, landing periodically to eat, talk, and anything else required by your story. Make **dragon.flapWings()** the handler of a **While something is true** event, so that the dragon automatically flaps its wings whenever it is above the ground.
- 6.3 Build a world containing a puzzle the user must solve. Place characters in the world who can provide hints to the puzzle’s solution when the user clicks on them. Let the user navigate through the world using the arrow keys.
- 6.4 Add the following enhancements to the helicopter flight simulator program we built in Sections 6.2 and 6.3 (in increasing order of difficulty):
  - a. Add a “title” screen that names the program, and describes the problem to be solved.
  - b. Add a “congratulations” screen that appears when the user finds and clicks on the cat.

- c. Modify the program so that when the helicopter descends over the roof of any of the buildings, it lands on the roof instead of passing right through it.
- d. Modify the program so that if the helicopter collides with anything in the world as it moves forward, backward, left, or right, the helicopter "crashes" and the world displays a "better luck next time" screen.

6.5 Build methods to perform the following transition effects:

- a. Build a method `fadeTo(someColor, fadeTime)`; that lets the sender specify the color to which the screen should fade, and `fadeFrom(fadeTime)` that complements `fadeTo()`. Then revise the `fadeToBlack()` and `fadeFromBlack()` methods so that they use `fadeTo()` and `fadeFrom()`.

- b. Build a method `barWipeCover(direction, time)`; that, at the end of a scene, moves a single shutter from one of the edges to cover the screen; and a method `barWipeUncover(direction, time)`; that complements `barWipeCover()`. The direction argument should be either **LEFT**, **RIGHT**, **UP**, or **DOWN**.



- c. In a *diagonal wipe*, a shutter crosses the screen from one corner to the opposite corner. Write complementary methods that perform the two parts of a diagonal wipe.



- d. A *bowtie wipe* is like a barndoor wipe, but the shutters coming from the sides are wedges that form a bow-tie when they first touch one another. Write complementary methods that perform the two parts of a bowtie wipe.



- e. A *rotating octagonal iris wipe* is an iris wipe in which the iris is a rotating octagon rather than a rectangle. Write complementary methods that perform the two parts of a rotating octagonal iris wipe.



- 6.6 Choose a popular game like chess, checkers, mancala, master mind, etc. Create a board and pieces for the game. Add event handlers that allow the user to move the pieces interactively.
- 6.7 Using the **Carrier** and **FighterPlane** classes from Alice's Web Gallery, create a carrier-jet simulation, in which the user must fly the **fighterPlane**, taking off from and landing on the **carrier**.
- 6.8 Using the **WhackAMoleBooth** class from the Alice Gallery **Amusement Park** folder, program a whack-a-mole game in which the **mole** pops its head out of a random hole in the booth for a short, random length of time before ducking down again, and the user tries to bop the mole with the **bopper**. Play a sound each time the user successfully bops the mole.
- 6.9 Proceed as in Problem 6.8, but make your program a continuously running series of games. Limit each game to some fixed length of time (for example, 60 seconds). Have the user enter his or her name at the beginning of a game. Your program should keep track of how many times the user bops the mole during the game, and when the time expires, display (a) that number as the user's score for this game, and (b) the top five scores since the program began running. Play a special sound if the user beats the highest score (becoming the new top score).
- 6.10 Design and build your own original, interactive computer game.

# Appendix A

## Alice Standard Methods and Functions

### A.1 Alice Standard Methods

Alice *methods* are messages that we can send to an object commanding it to do something. The object then responds with a behavior (hopefully the one we intended). The following table provides a complete list of Alice's standard methods, which are the commands to which all Alice objects will respond.

Method	Behavior Produced
<code>obj.move(dir, dist);</code>	<i>obj</i> moves <i>dist</i> meters in direction <i>dir</i> = UP, DOWN, LEFT, RIGHT, FORWARD, or BACKWARD
<code>obj.turn(dir, revs);</code>	<i>obj</i> turns <i>revs</i> revolutions in direction LEFT, RIGHT, FORWARD, or BACKWARD (that is, about its UD- or LR-axis)
<code>obj.roll(dir, revs);</code>	<i>obj</i> rotates <i>revs</i> revolutions in direction LEFT or RIGHT (that is, about its FB-axis)
<code>obj.resize(howMuch);</code>	<i>obj</i> 's size changes by a factor of <i>howMuch</i>
<code>obj.say(message);</code>	<i>obj</i> says <i>message</i> (via a cartoon balloon)
<code>obj.think(thought);</code>	<i>obj</i> thinks <i>thought</i> (via a cartoon balloon)
<code>obj.playSound(soundFile);</code>	<i>obj</i> plays the audio file <i>soundFile</i>
<code>obj.moveTo(obj2);</code>	<i>obj</i> 's position becomes that of <i>obj2</i> ( <i>obj</i> 's orientation remains unchanged)
<code>obj.moveToward(obj2, dist);</code>	<i>obj</i> moves <i>dist</i> meters toward the position of <i>obj2</i>

*continued*

Method	Behavior Produced
<code>obj.moveAwayFrom(obj2,dist);</code>	<i>obj</i> moves away from <i>obj2</i> , <i>dist</i> meters from its current position
<code>obj.orientTo(obj2);</code>	<i>obj</i> 's orientation becomes that of <i>obj2</i> ( <i>obj</i> 's position remains unchanged)
<code>obj.turnToFace(obj2);</code>	<i>obj</i> rotates about its UD-axis until it is facing <i>obj2</i>
<code>obj.pointAt(obj2);</code>	<i>obj</i> rotates so that its FB-axis points at <i>obj2</i> 's center
<code>obj.setPointOfView(obj2);</code>	<i>obj</i> 's position and orientation change to that of <i>obj2</i>
<code>obj.setPose(pose);</code>	<i>obj</i> assumes the pose specified by <i>pose</i>
<code>obj.standUp();</code>	<i>obj</i> rotates so that its UD-axis is vertical
<code>obj.moveAtSpeed(dir,mps);</code>	<i>obj</i> moves direction UP, DOWN, LEFT, RIGHT, FORWARD, or BACKWARD at <i>mps</i> meters/sec (for <i>duration</i> secs) <sup>1</sup>
<code>obj.turnAtSpeed(dir,rps);</code>	<i>obj</i> turns direction LEFT, RIGHT, FORWARD, or BACKWARD at <i>rps</i> revolutions/sec (for <i>duration</i> secs)
<code>obj.rollAtSpeed(dir,rps);</code>	<i>obj</i> rolls direction LEFT or RIGHT at <i>rps</i> revolutions/sec (for <i>duration</i> secs)
<code>obj.constrainToPointAt(obj2);</code>	<i>obj</i> points at <i>obj2</i> for the duration of this message

1. To make *obj* accelerate: use `obj.moveAtSpeed(dir,speed)`, make *speed* a variable, and use a `doTogether` block to simultaneously perform the `moveAtSpeed()` method while changing the value of *speed*.

## A.2 Alice Standard Object Functions

Alice *functions* are messages we can send to an object to ask it a question. The object responds by producing a *result* — the answer to our question. The following table provides a complete list of Alice's standard functions — the questions that all Alice objects will answer:

Function	Result Produced
<code>obj.isCloseTo(dist,obj2)</code>	true, if <i>obj</i> is within <i>dist</i> meters of <i>obj2</i> ; false, otherwise
<code>obj.isFarFrom(dist,obj2)</code>	true, if <i>obj</i> is at least <i>dist</i> meters away from <i>obj2</i> ; false, otherwise
<code>obj.distanceTo(obj2)</code>	the distance between <i>obj</i> and <i>obj2</i> 's centers
<code>obj.distanceToTheLeftOf(obj2)</code>	the distance from the left side of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not left of <i>obj2</i> )
<code>obj.distanceToTheRightOf(obj2)</code>	the distance from the right side of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not right of <i>obj2</i> )
<code>obj.distanceAbove(obj2)</code>	the distance from the top of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not above <i>obj2</i> )
<code>obj.distanceBelow(obj2)</code>	the distance from the bottom of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not below <i>obj2</i> )
<code>obj.distanceInFrontOf(obj2)</code>	the distance from the front of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not in front of <i>obj2</i> )
<code>obj.distanceBehind(obj2)</code>	the distance from the back of <i>obj2</i> 's bounding box to <i>obj</i> 's bounding box (negative if <i>obj</i> is not in back of <i>obj2</i> )
<code>obj.getWidth()</code>	the width (LR-axis length) of <i>obj</i> 's bounding box
<code>obj.getHeight()</code>	the height (UD-axis length) of <i>obj</i> 's bounding box
<code>obj.getDepth()</code>	the depth (FB-axis length) of <i>obj</i> 's bounding box
<code>obj.isSmallerThan(obj2)</code>	true, if <i>obj2</i> 's volume exceeds that of <i>obj</i> ; false, otherwise

*continued*

Function	Result Produced
<i>obj.isLargerThan(obj2)</i>	true, if <i>obj</i> 's volume exceeds that of <i>obj2</i> ; false, otherwise
<i>obj.isNarrowerThan(obj2)</i>	true, if <i>obj2</i> 's width exceeds that of <i>obj</i> ; false, otherwise
<i>obj.isWiderThan(obj2)</i>	true, if <i>obj</i> 's width exceeds that of <i>obj2</i> ; false, otherwise
<i>obj.isShorterThan(obj2)</i>	true, if <i>obj2</i> 's height exceeds that of <i>obj</i> ; false, otherwise
<i>obj.isTallerThan(obj2)</i>	true, if <i>obj</i> 's height exceeds that of <i>obj2</i> ; false, otherwise
<i>obj.isToTheLeftOf(obj2)</i>	true, if <i>obj</i> 's position is left of <i>obj2</i> 's left edge; false, otherwise
<i>obj.isToTheRightOf(obj2)</i>	true, if <i>obj</i> 's position is right of <i>obj2</i> 's right edge; false, otherwise
<i>obj.isAbove(obj2)</i>	true, if <i>obj</i> 's position is above <i>obj2</i> 's top edge; false, otherwise
<i>obj.isBelow(obj2)</i>	true, if <i>obj</i> 's position is below <i>obj2</i> 's bottom edge; false, otherwise
<i>obj.isInFrontOf(obj2)</i>	true, if <i>obj</i> 's position is before <i>obj2</i> 's front edge; false, otherwise
<i>obj.isBehind(obj2)</i>	true, if <i>obj</i> 's position is in back of <i>obj2</i> 's rear edge; false, otherwise
<i>obj.getPointOfView()</i>	the point of view ( <i>position</i> + <i>orientation</i> ) of <i>obj</i>
<i>obj.getPosition()</i>	the <i>position</i> (with respect to the world's axes) of <i>obj</i>
<i>obj.getQuaternion()</i>	the <i>orientation</i> (with respect to the world's axes) of <i>obj</i>
<i>obj.getCurrentPose()</i>	the current <i>Pose</i> ( <i>position</i> + <i>orientation</i> of subparts) of <i>obj</i>
<i>obj.partNamed(piece)</i>	the subpart of <i>obj</i> named <i>piece</i>

### A.3 Alice World Functions

Alice *world functions* are implementations of commonly needed computations. The following table provides a complete list of Alice's world functions.

Function	Result Produced
<code>!a</code>	true, if <code>a</code> is false; false, otherwise
<code>(a &amp;&amp; b)</code>	true, if <code>a</code> and <code>b</code> are both true; false, if <code>a</code> or <code>b</code> is false
<code>(a    b)</code>	true, if either <code>a</code> or <code>b</code> are true; false, if neither <code>a</code> nor <code>b</code> is true
<code>a == b</code>	true, if <code>a</code> and <code>b</code> have the same value; false, otherwise
<code>a != b</code>	true, if <code>a</code> and <code>b</code> have different values; false, otherwise
<code>a &lt; b</code>	true, if <code>a</code> 's value is less than <code>b</code> 's value; false, otherwise
<code>a &gt; b</code>	true, if <code>a</code> 's value is greater than <code>b</code> 's value; false, otherwise
<code>a &lt;= b</code>	true, if <code>a</code> 's value is less than or equal to <code>b</code> 's value; false, otherwise
<code>a &gt;= b</code>	true, if <code>a</code> 's value is greater than or equal to <code>b</code> 's value; false, otherwise
<code>Random.nextBoolean()</code>	a pseudo-randomly chosen true or false value
<code>Random.nextDouble()</code>	a pseudo-randomly chosen number
<code>a + b</code>	the string consisting of <code>a</code> followed by <code>b</code> (concatenation)
<code>what.toString()</code>	the string representation of <code>what</code> (string conversion)
<code>NumberDialog(question)</code>	a number entered by the user in response to <code>question</code>

*continued*



Function	Result Produced
<code>BooleanDialog(question)</code>	<code>true</code> if the user responds to <i>question</i> by clicking the dialog box's <b>Yes</b> button; <code>false</code> otherwise.
<code>StringDialog(question)</code>	a string entered by the user in response to <i>question</i>
<code>mouse.getDistanceFromLeftEdge()</code>	the number of pixels the mouse is from the left edge of the window (corresponds to x of an [x,y] coordinate)
<code>mouse.getDistanceFromTopEdge()</code>	the number of pixels the mouse is from the top edge of the window (corresponds to y of an [x,y] coordinate)
<code>getTimeElapsedSinceWorldStart()</code>	the number of "ticks" since the world began running
<code>getYear()</code>	a number representing the current year
<code>getMonthOfYear()</code>	a number representing the current month (Jan-0, Feb-1, etc.)
<code>getDayOfYear()</code>	a number representing the current day of the year
<code>getDayOfMonth()</code>	a number representing the current day of the month
<code>getDayOfWeek()</code>	a number representing the current day of the week (Sun-1, etc.)
<code>getDayOfWeekInMonth()</code>	a number for how many times the current day of the week has occurred in the current month
<code>isAM()</code>	<code>true</code> if the current time is between midnight and noon; <code>false</code> , otherwise
<code>isPM()</code>	<code>true</code> if the current time is between noon and midnight; <code>false</code> , otherwise
<code>getHourOfAMOrPM</code>	the hour value of the current time, 12-hour format
<code>getHourOfDay</code>	the hour value of the current time, 24-hour format
<code>getMinuteOfHour</code>	the minute value of the current time

*continued*

Function	Result Produced
<code>getSecondOfMinute</code>	the second value of the current time
<code>Math.min(a, b)</code>	the minimum of <i>a</i> and <i>b</i>
<code>Math.max(a, b)</code>	the maximum of <i>a</i> and <i>b</i>
<code>Math.abs(a)</code>	the absolute value of <i>a</i>
<code>Math.sqrt(a)</code>	the square root of <i>a</i>
<code>Math.floor(a)</code>	the largest integer smaller than <i>a</i>
<code>Math.ceiling(a)</code>	the smallest integer larger than <i>a</i>
<code>Math.sin(a)</code>	the sine of <i>a</i>
<code>Math.cos(a)</code>	the cosine of <i>a</i>
<code>Math.tan(a)</code>	the tangent of <i>a</i>
<code>Math.asin(a)</code>	the angle whose sine is <i>a</i>
<code>Math.acos(a)</code>	the angle whose cosine is <i>a</i>
<code>Math.atan(a)</code>	the angle whose tangent is <i>a</i>
<code>Math.atan2(x, y)</code>	the polar coordinate angle associated with Cartesian coordinate ( <i>x</i> , <i>y</i> )
<code>Math.pow(a, b)</code>	<i>a</i> raised to the power <i>b</i> ( $a^b$ )
<code>Math.natural log of(a)</code>	the number <i>x</i> such that $e^x == a$ ; <i>e</i> being Euler's number
<code>Math.exp(a)</code>	Euler's number <i>e</i> raised to the power <i>a</i> ( $e^a$ )
<code>Math.IEEEremainder(a, b)</code>	the remainder of <i>a/b</i> using integer division
<code>Math.round(a)</code>	the integer whose value is closest to <i>a</i>
<code>Math.toDegrees(r)</code>	the angle in degrees corresponding to radians <i>r</i>
<code>Math.toRadians(d)</code>	the angle in radians corresponding to degrees <i>d</i>
<code>superSquareRoot(a, b)</code>	the $b^{\text{th}}$ root of <i>a</i>
<code>getVector(right, up, forward)</code>	an x-y-z vector [ <i>x</i> ==right, <i>y</i> == up, <i>z</i> ==forward]

# Appendix B

## Recursion

Hundreds of years before there were computers, programming languages, or loop statements, mathematicians were defining functions, many of which required repetitive behavior. One way to provide such behavior without using a loop is to have a function or method *invoke itself*, causing its statements to repeat. Such a method (or function) is called **recursive**. To illustrate, suppose we were to define a method for Alice's **camera** named **repeatRoll()** as follows:

```
void camera.repeatRoll() {  
    camera.roll(LEFT, 1);  
    camera.repeatRoll();  
}
```

When invoked, this method will make the **camera** roll left one revolution, and then it will invoke itself. That second invocation will make the **camera** roll left one revolution, and then it will call itself. That third invocation will make the **camera** roll left one revolution, and then it will call itself, and so on. Thus, the result is an "infinite" repetition, or **infinite recursion**.<sup>1</sup>

To avoid infinite repetition, recursive methods and functions typically have (1) a **Number** parameter, (2) an **if** statement that only performs the recursion if the parameter's value exceeds some lower bound, and (3) a recursive call within the **if** statement that passes a value smaller than the parameter as an argument. The net effect is that the function or method counts downward toward the lower bound, typically 0 or 1. To illustrate, we might revise the preceding **repeatRoll()** method as follows:

```
void camera.repeatRoll(Number count) { // the parameter count  
    if (count > 0) { // if statement guards  
        camera.roll(LEFT, 1); // the recursive call  
        camera.repeatRoll(count - 1);  
    }  
}
```

---

1. Since each recursive call consumes additional memory, the looping behavior will eventually end — when the program runs out of memory. However, we will become tired of the **camera** rolling long before that occurs!

When invoked with a numeric argument *n*, this version of the function will roll the **camera** *n* times and then stop. For example, if we send the message **camera.repeatRoll(3)**;

1. This starts **repeatRoll(3)**, in which parameter **count == 3**.
2. The method checks the condition **count > 0**.
3. Since the condition is true, the method (a) rolls the **camera** left 1 revolution, and (b) sends the message **camera.repeatRoll(2)**;
4. This starts **repeatRoll(2)**, a new version in which parameter **count == 2**.
5. The method checks the condition **count > 0**.
6. Since the condition is true, the method (a) rolls the **camera** left 1 revolution, and (b) sends the message **camera.repeatRoll(1)**;
7. This starts **repeatRoll(1)**, a new version in which parameter **count == 1**.
8. The method checks the condition **count > 0**.
9. Since the condition is true, the method (a) rolls the **camera** left 1 revolution, and (b) sends the message **camera.repeatRoll(0)**;
10. This starts **repeatRoll(0)**, a new version in which parameter **count == 0**.
11. The method checks the condition **count > 0**.
12. Since the condition is false, the method terminates; flow returns to the sender of **repeatRoll(0)** — **repeatRoll(1)** — the version where **count == 1**.
13. The version in which **count == 1** terminates; flow returns to the sender of **repeatRoll(1)** — **repeatRoll(2)** — the version in which **count == 2**.
14. The version in which **count == 2** terminates; flow returns to the sender of **repeatRoll(2)** — **repeatRoll(3)** — the version in which **count == 3**.
15. The version in which **count == 3** terminates; flow returns to the sender of **repeatRoll(3)**.

Steps 1 through 11, in which the repeated messages are counting downward toward the lower bound, are sometimes called the **winding phase** of the recursion. Steps 12 through 15, in which the chain of recursive messages terminate, are sometimes called the **unwinding phase** of the recursion.

Recursion thus provides an alternative way to achieve repetitive behavior. When the recursive message is the last behavior-producing statement in the method, as follows:

```
void camera.repeatRoll(Number count) {
    if (count > 0) {
        camera.roll(LEFT, 1);
        camera.repeatRoll(count - 1); // the last statement
    }
}
```

it is called **tail recursion**, because the recursive message occurs at the end or “tail” of the method. Any function defined using tail recursion can be defined using a loop, and vice

versa. But in Section B.2, we will see that one recursion method can produce behavior that would require multiple loops.

## B.1 Tail Recursion

Suppose that at the end of Scene 1 of a story, the main character goes to sleep at 11 p.m., and the **camera** zooms in to a closeup of the clock in his or her bedroom. Shot 1 of Scene 2 begins with that same clock, showing the time to be 11 p.m. Suppose that our story calls for the clock's hands to spin, indicating that time is "flying ahead." When the hands reach 3 a.m., a fairy appears and works some sort of mischief on the sleeping main character. (Exactly what mischief the fairy works is left up to you.)

To build the scene, we can go to the Alice Gallery, add a **bedroom** from the **Environments** folder, add a **Dresser** from the **Furniture** folder, add a **mantleClock** from the **Objects** folder, and add **OliveWaterblossom** from the **People** folder as our fairy. To set the scene, we can manually advance the clock's hands to 11 p.m. (using right-click->**methods**->**mantleClock.roll()** messages), make **OliveWaterblossom** smaller, position her next to the clock, set her **opacity** to zero, and then position the **camera** appropriately. Our scene thus starts as shown in Figure B-1.

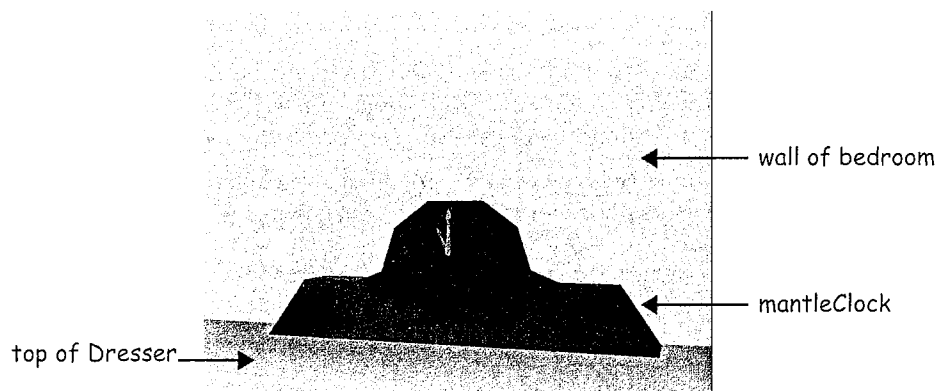


FIGURE B-1 Beginning of Scene 2

To follow the story, we need a way to make the clock's hands spin forward to 3 a.m. Since this is a counting problem, we could use a **for** loop; but for variety, let's use tail recursion. The basic algorithm is as follows:

**Algorithm:** *advance-the-clock's-hands hours hours*  
**Given:** *hours*, the number of hours to spin the clock's hands forward

```

1 If hours > 0:
  a Spin the hour and minute hands forward one hour
  b advance-the-clock's-hands hours-1

```

Building a `mantleClock.advanceHands()` method this way is straightforward. However, when we perform the recursion by dragging and dropping the `mantleClock.advanceHands()` method into the same method, Alice warns us that we're sending a recursive message, as shown in Figure B-2.

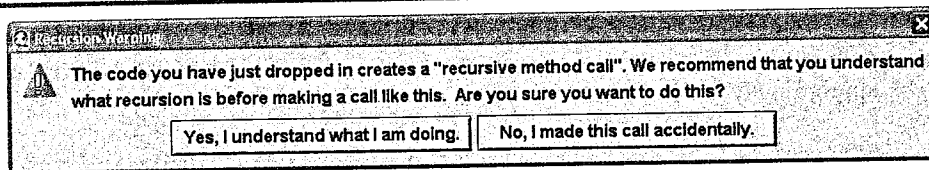


FIGURE B-2 Alice's recursion warning dialog box

Since we think we know what we are doing, we click the **Yes, I understand what I am doing.** button. The resulting method is shown in Figure B-3.

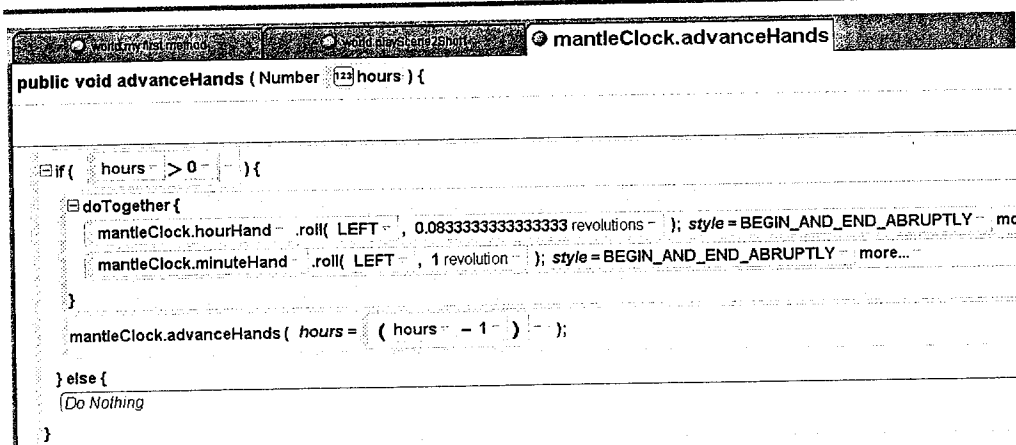


FIGURE B-3 An `advanceHands()` method

When invoked with a positive **hours** value, this method spins the clock's hands forward one hour, and then invokes itself recursively with **hours-1** as an argument. The method thus "counts down" recursively from whatever **hours** value it receives initially, until it is invoked with an **hours** value of 0, at which point the recursion terminates.

We can use this method to build the **playScene2Shot1()** method, as shown in Figure B-4.

```

world.my first method  world.playScene2Shot1  mantleClock.advanceHands
public void playScene2Shot1 () {
    doInOrder {
        mantleClock.advanceHands( hours = 4 );
        OliveWaterblossom = set( opacity, 1 (100%) ); more...
        OliveWaterblossom = say( I'm feeling mischievous... ); duration = 2 seconds - fontSize = 30 - more...
        // The fairy does something magical...
    }
}

```

FIGURE B-4 The **playScene2Shot1()** method

When performed, the scene begins with the setup shown in Figure B-1. The **advanceHands(4)** message then spins the clock's hands forward four hours, after which **OliveWaterblossom** appears and says she's feeling mischievous, as shown in Figure B-5.



FIGURE B-5 The end of Shot 1 of Scene 2

If you compare the definition of the `advanceHands()` method with the `repeatRoll()` method we described earlier, you'll see that both follow the same basic pattern:

---

Simplified Pattern for Tail Recursion:

```
void tailRecursiveMethod ( Number count ) {
    if ( count > LOWER_BOUND ) {
        produceBehaviorOnce();
        tailRecursiveMethod( count-1 );
    }
}
```

where:

*produceBehaviorOnce() produces the behavior to be repeated.*

---

A method that follows this pattern will produce results equivalent to those produced by the following nonrecursive pattern:

```
void nonRecursiveMethod( Number count ) {
    for ( int i = count; i > 0; count-- ) {
        produceBehaviorOnce();
    }
}
```

Tail recursion provides an alternative way to solve counting problems and other problems in which solutions require repetition. In the next section, we will see that useful work can be done *following* the recursive call.

## B.2 General Recursion

Suppose that Scene 3 of our story begins the same way as Scene 2: with a closeup of the clock in the main character's bedroom showing 11 p.m., the next night. In this scene, our story calls for time to fly ahead eight hours to 7 a.m., once again indicated by the clock's spinning hands. Then **OliveWaterblossom** appears, once again intent on mischief. In this scene, her mischief is to reverse time everywhere except for the main character, so that upon waking up after eight hours of sleep — fully rested — it will be 11 p.m. again! To indicate that time is flowing in reverse, we must spin the clock's hands backward eight hours.

We could accomplish this by using our `advanceHands()` method to spin the clock's hands forward eight hours, and then writing a `reverseHands()` method to make the hands spin backward eight hours, using either tail recursion or a `for` loop. Instead, let's see how recursion lets us perform both of these steps in one method.

The key idea is to use recursion's winding phase to spin the hands forward (as before), and then to use the unwinding phase to make the hands spin backward. In between the two phases — when we have reached our lower bound — **OliveWaterblossom** can work her magic.



```
Algorithm: wind-and-unwind-the-clock's-hands hours hours
Given: hours, the number of hours to spin the clock's hands forward

1 If hours > 0:
  a Spin the hour and minute hands forward one hour
  b wind-and-unwind-the-clock's-hands hours-1
  c Spin the hour and minute hands backward one hour
2 Else:
  a OliveWaterblossom appears
  b OliveWaterblossom works her magic
```

Understanding how this works can be difficult the first time you see it. One way to understand it is to see that Step 1c does the exact opposite of Step 1a. That is, during the winding phase, Step 1a spins the clock's hands forward one hour; then Step 1b sends the recursive message, preventing flow from reaching Step 1c (for the time being). When the lower bound is reached, the **if** statement's condition is false, so **OliveWaterblossom** works her magic. And since no recursive message is sent, the repetition halts. The recursion then starts to unwind, with flow returning to Step 1c in each message, which "undoes" the effects of Step 1a. Figure B-6 gives a numbered visualization of what happens when **hours** has the value 8. The steps that are performed within each message at a given point are highlighted.

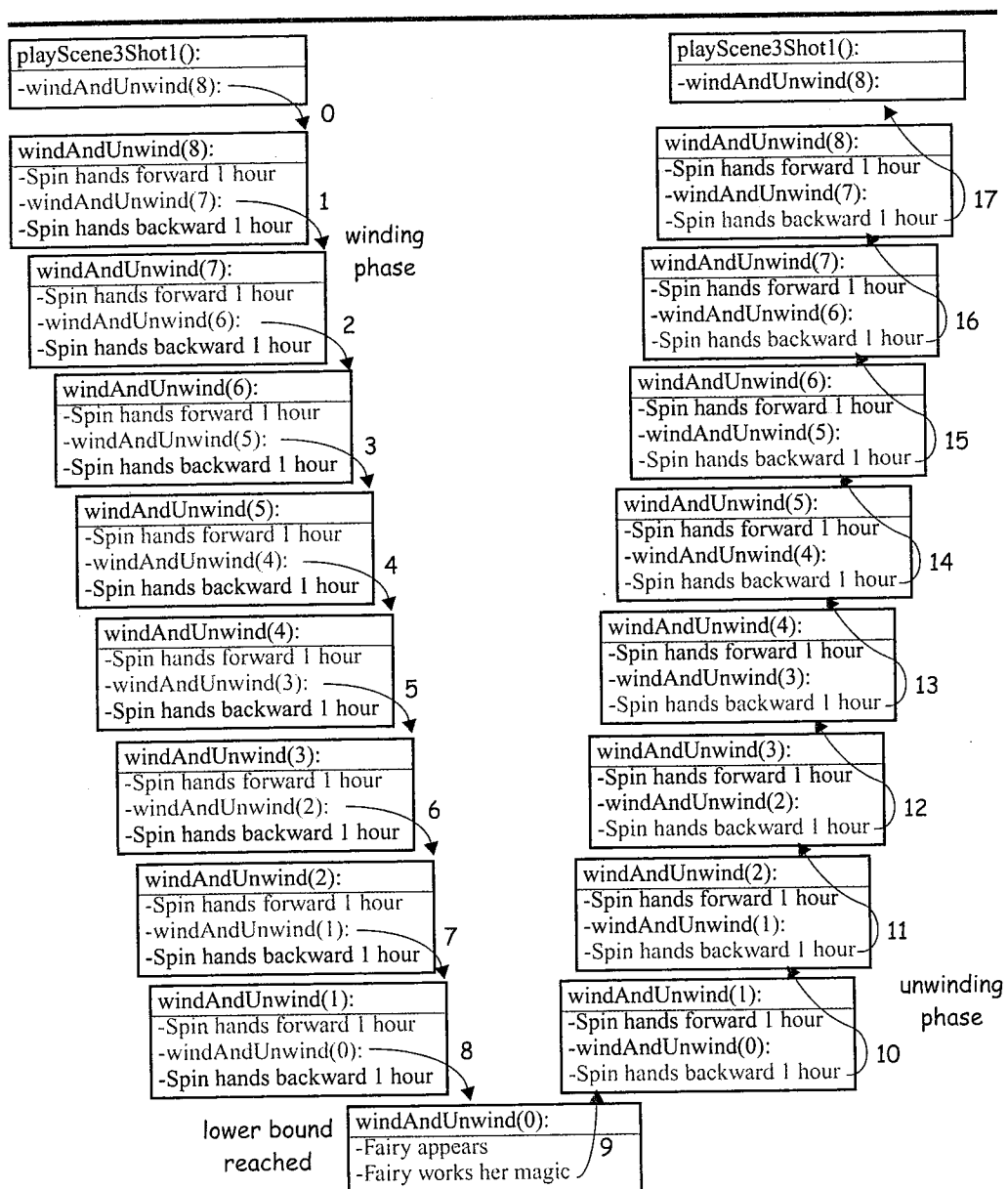
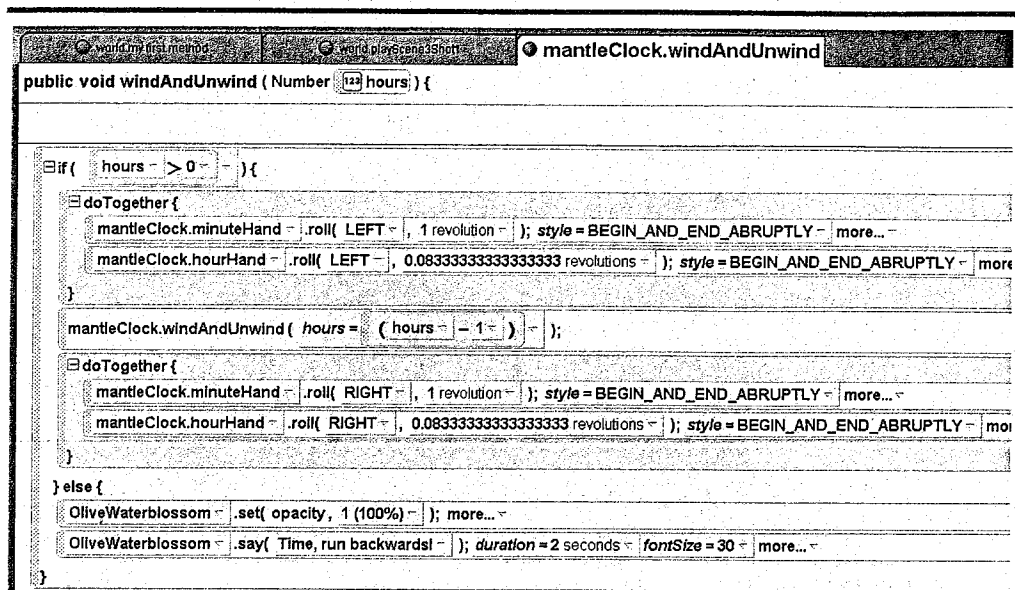


FIGURE B-6 Recursive winding and unwinding

We can define this method in Alice, as shown in Figure B-7.



```

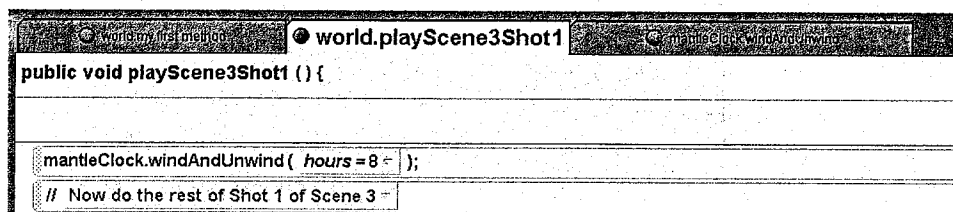
public void windAndUnwind ( Number hours ) {

    if ( hours > 0 ) {
        doTogether {
            mantleClock.minuteHand .roll( LEFT , 1 revolution ); style = BEGIN_AND_END_ABRUPTLY ; more...
            mantleClock.hourHand .roll( LEFT , 0.08333333333333333 revolutions ); style = BEGIN_AND_END_ABRUPTLY ; more...
        }
        mantleClock.windAndUnwind ( hours = ( hours - 1 ) );
        doTogether {
            mantleClock.minuteHand .roll( RIGHT , 1 revolution ); style = BEGIN_AND_END_ABRUPTLY ; more...
            mantleClock.hourHand .roll( RIGHT , 0.08333333333333333 revolutions ); style = BEGIN_AND_END_ABRUPTLY ; more...
        }
    } else {
        OliveWaterblossom .set( opacity, 1 (100%) ); more...
        OliveWaterblossom .say( Time, run backwards! ); duration = 2 seconds ; fontSize = 30 ; more...
    }
}

```

FIGURE B-7 The windAndUnwind() method

Given such a method, `playScene3Shot1()` is quite simple, as shown in Figure B-8.



```

public void playScene3Shot1 () {

    mantleClock.windAndUnwind ( hours = 8 );
    // Now do the rest of Shot 1 of Scene 3
}

```

FIGURE B-8 The windAndUnwind() method

When performed, the method starts out with the scene shown in Figure B-1. Once again, we see “time fly” as the hands wind forward, but this time they advance eight hours. Our fairy then appears and says her line, as shown in Figure B-9.

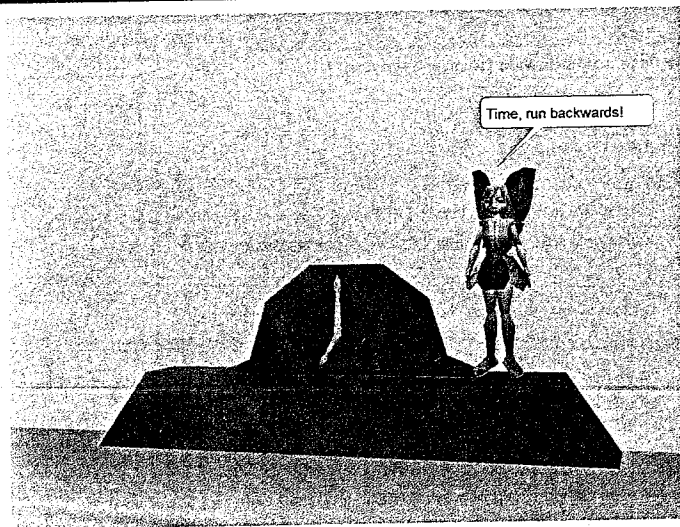


FIGURE B-9 Time has flown forward eight hours

The hands then spin backward, returning to their original positions, as shown in Figure B-10.

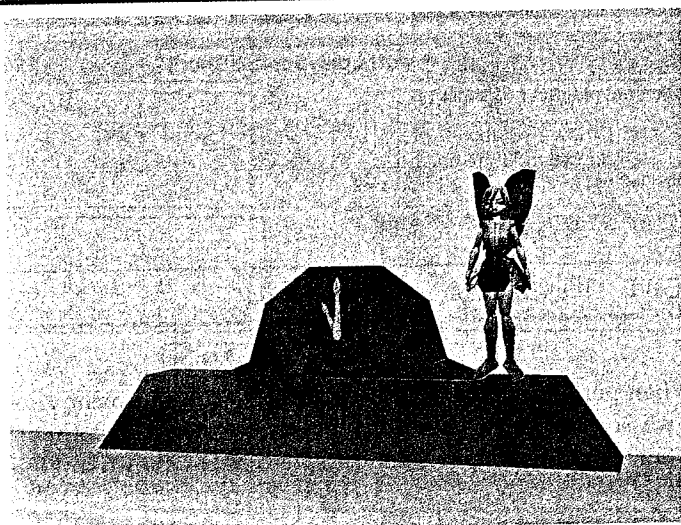


FIGURE B-10 Time has flown backward eight hours

It is thus possible to do work during both the winding and the unwinding phases of a chain of recursive messages. Any statements that we want to be performed during the winding phase must be positioned before the recursive call, and any statements that we want to be performed during the unwinding phase must be positioned after the recursive call.

The following pattern can be used to design many recursive methods:

---

Simplified Pattern for Recursion:

---

```
void recursiveMethod ( Number count ) {
    if ( count > LOWER_BOUND ) {
        windingPhaseBehavior();
        recursiveMethod( count-1 );
        unwindingPhaseBehavior();
    } else {
        betweenPhasesBehavior();
    }
}
```

---

### B.3 Recursion and Design

Now that we have seen some examples of recursive methods, how does one go about designing such methods?

Recall that recursive methods usually have a **Number** parameter. Designing a recursive method generally involves two steps: (1) identifying the **trivial case** — how to solve the problem when the value of this parameter makes the problem trivial to solve; and (2) identifying the **nontrivial case** — how to use recursion to solve the problem for all of the other cases. Once we have done so, we can plug these cases into this template:

```
someType recursiveMethod(Number count) {
    if (count indicates that this is a nontrivial case) {
        solve the problem recursively, reducing count
    } else { // it's the trivial case
        solve the trivial version of the problem
    }
}
```

To illustrate, let's apply this approach to one of the functions mathematicians defined recursively long before there were computers.

Pretend for a moment that you are an elementary school student, and your teacher just caught you misbehaving during math class. As a "punishment," your teacher makes you stay in at each recess until you have calculated 10! (10 factorial), 20! (20 factorial), and 30! (30 factorial). Even with a calculator, this will take a long time because the factorial function  $n!$  is defined as shown in Figure B-11.

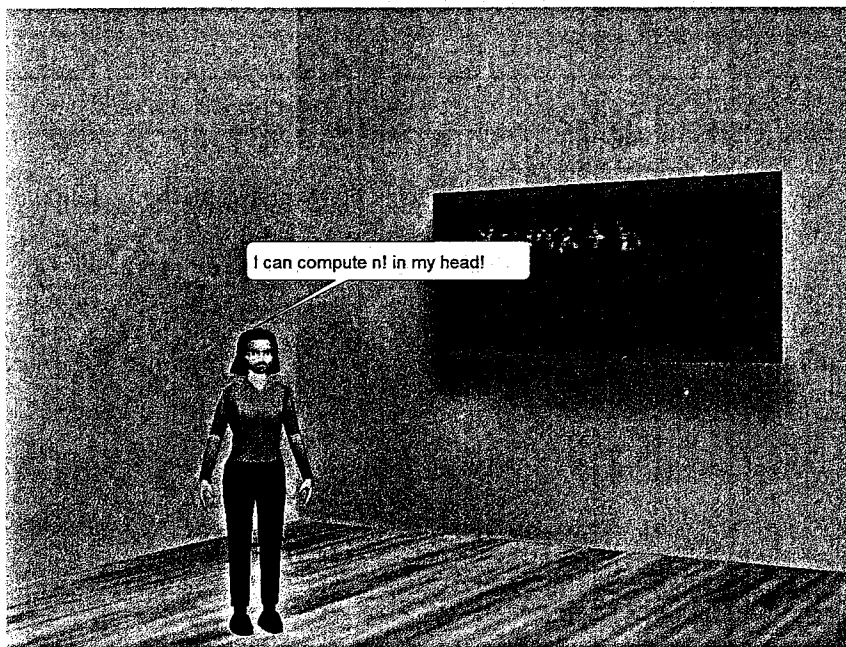
$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

FIGURE B-11  $n!$ , in open-form notation

That is,  $1! == 1$ ;  $2! == 2$ ;  $3! == 6$ ,  $4! == 24$ ,  $5! == 120$ , and so on.  $0!$  is also defined to equal 1, and the function is not defined for negative values of  $n$ .

While we could solve this problem by hand, doing so would be long and tedious, and we would lose lots of recess time. Instead, let's write an Alice program to solve it!

To do so, we can begin as we did in Section 3.5.2, and build a scene containing a character (**Roommate**, in this case) who can do factorials "in her head," positioned within an Alice **School** environment, as shown in Figure B-12.

FIGURE B-12 Setting the scene to compute  $n!$ 

With such a scene in place, we just have to (1) write a **factorial()** function, (2) get  $n$  from the user, (3) invoke and save the answer of **factorial( $n$ )**, and (4) display the answer.

Let's begin by writing the **factorial()** function. If we examine the description given in Figure B-11, it should be evident that this is a counting problem, and so we could solve it using a **for** loop. However, let's instead see how the mathematicians would have solved it back in the days before there were **for** loops.

### B.3.1 The Trivial Case

We start by identifying the trivial case. What is a version of the problem that is trivial to solve? Since  $0! == 1$  and  $1! == 1$ , we actually have two trivial cases: when  $n == 0$ , and when  $n == 1$ . In either case, our function needs to return the value 1.

### B.3.2 The Nontrivial Cases

To solve the nontrivial cases, we look for a way to solve the general  $n!$  problem, assuming that we can solve a smaller but similar problem (for example,  $(n-1)!$ ). If we compare the two:

$$\begin{aligned} n! &= 1 \times 2 \times \dots \times (n-1) \times n \\ (n-1)! &= 1 \times 2 \times \dots \times (n-1) \end{aligned}$$

it should be evident that we can rewrite the equation in Figure B-11 by performing a substitution, as shown in Figure B-13.

---


$$n! = (n-1)! \times n$$


---

FIGURE B-13  $n!$ , in recursive, closed-form notation

### B.3.3 Solving the Problem

The trivial and nontrivial cases can be combined into a complete solution to the problem, as shown in Figure B-14.

---


$$n! = \begin{cases} (n-1)! \times n, & \text{if } n > 1 \\ 1, & \text{if } n == 0 \text{ OR } n == 1 \\ \text{undefined,} & \text{otherwise} \end{cases}$$


---

FIGURE B-14 Recursive algorithm for  $n!$

The equation given in Figure B-14 can serve as an algorithm for us to define our **factorial()** function in Alice, as shown in Figure B-15.

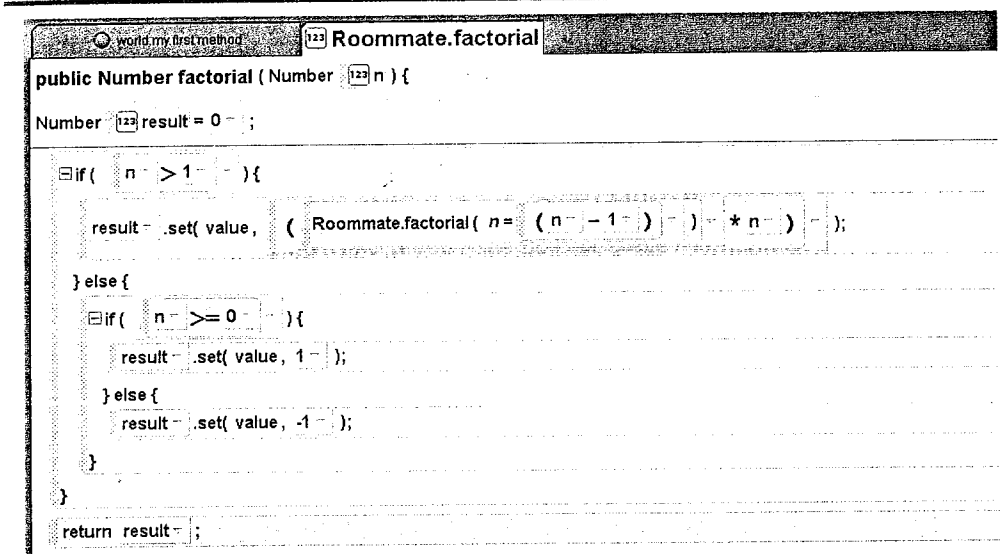


FIGURE B-15 The factorial() function in Alice

Note that because  $n!$  is undefined when  $n$  is negative, and  $n!$  never returns  $-1$  under normal circumstances, we have our function return  $-1$  when  $n$  is negative.

With this function defined, we can now finish our program, as shown in Figure B-16.

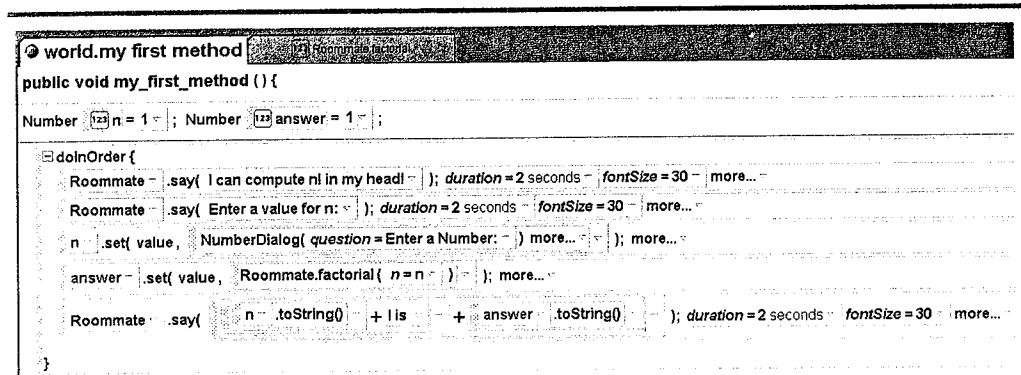


FIGURE B-16 The factorial() program in Alice

When run, the program has us enter a value for  $n$ , and then displays  $n!$ . After testing our function on easily verified values (such as 0, 1, 2, 3, 4, and 5), we can solve the problems our teacher assigned. Figure B-17 shows the result when we use the program to compute  $10!$ .



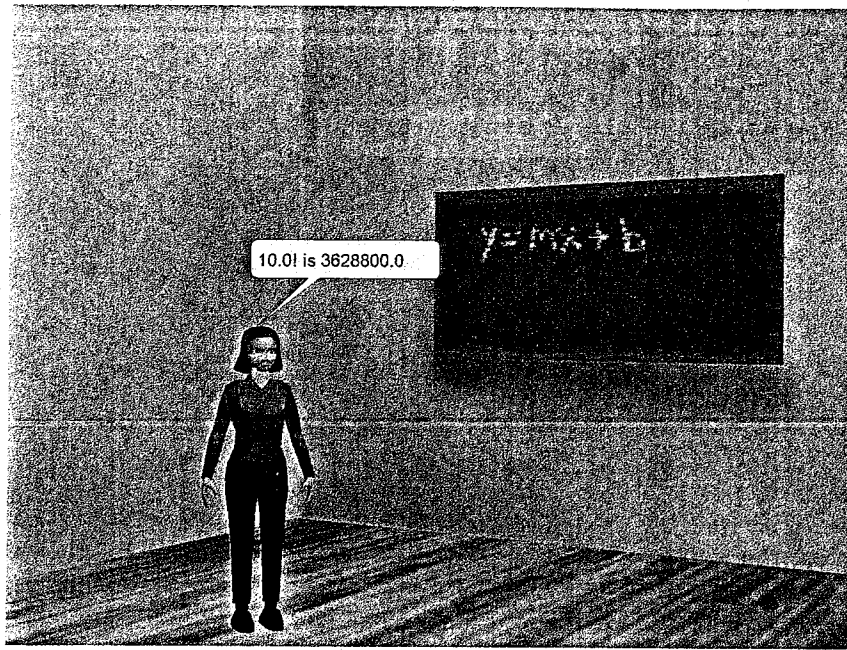


FIGURE B-17 The `factorial()` program in Alice

It's recess time!

## B.4 A Final Recursive Method

As a final example, consider the following user story.

*Scene 1, Shot 1: zeus, socrates, aliceLiddell, plato, euripides, and the whiteRabbit are all waiting to practice basketball. The coach says, "Okay, everyone line up by height!" The players line up, tallest to shortest.*

*Scene 1, Shot 2: The coach says, "No, line up the other way — shortest to tallest!" The players reverse their order.*

Scene 1, Shot 1 is mainly to get things set up, so we will leave it as an exercise. What we want to do is to build Scene 1, Shot 2, especially the part in which the players reverse their order.

It is fairly easy to get our scene to the point shown in Figure B-18.

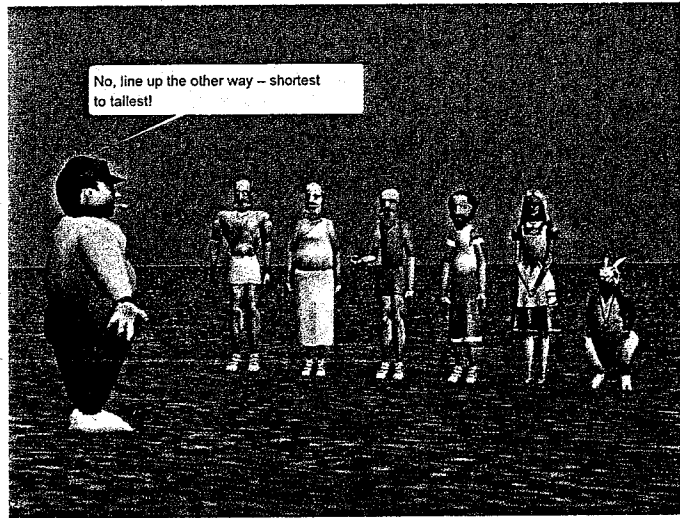


FIGURE B-18 Scene 1, Shot 2 (beginning)

But how can we make our players reverse their order?

Since we have a group of players, and their number is fixed, one idea is to store them in an array, tallest to smallest, as shown in Figure B-19.

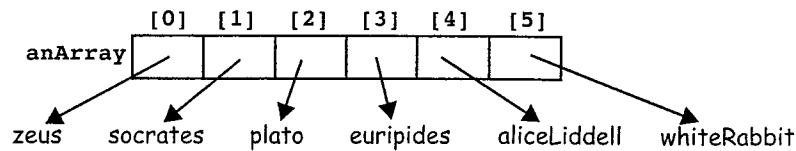
```

world.my first method
public void my_first_method () {
  Object[] anArray = { zeus, socrates, plato, euripides, aliceLiddell, whiteRabbit };
  world.playScene1Shot2 ( );
}

```

FIGURE B-19 Scene 1, Shot 2 (beginning)

The first array element is the tallest player, the second array element is the second tallest player, and so on. We can visualize **anArray** as shown in Figure B-20.

FIGURE B-20 Visualizing `anArray`

With the players in order within the data structure, we can transform our problem into this one:

---

*Reverse the positions of the players in `anArray`.*

---

One way to accomplish this is to (1) make the first and last players in the array swap positions within our world, as shown in Figure B-21, and then (2) reverse the remaining players in the array (that is, ignoring the `whiteRabbit` and `zeus`) the same way — a recursive solution!

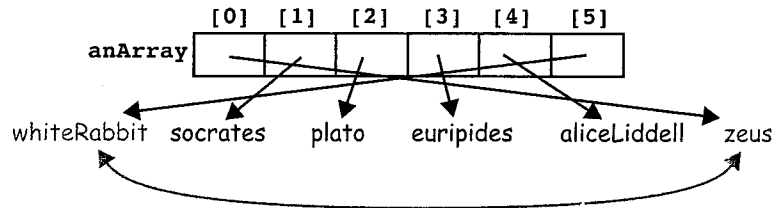


FIGURE B-21 The tallest and smallest players swap positions

To do so, we would need a method named `reverse()`, to which we can pass the array containing our players, plus the indices of the players that are to swap positions:

```
reverse(anArray, 0, 5);
```

Our method requires three parameters: an **Object** array, a **Number** to store the first index, and a **Number** to store the second index:

```
void reverse(Object [] arr, Number index1, Number index2) {
}
```

To get two objects to swap positions, we can write a method named `swapPositions()`, and then pass it the two objects whose positions we want to swap. Figure B-22 shows one

way to do this, which is by adding two dummies to our world and then using them within our method to mark the original positions of the two objects we wish to move.

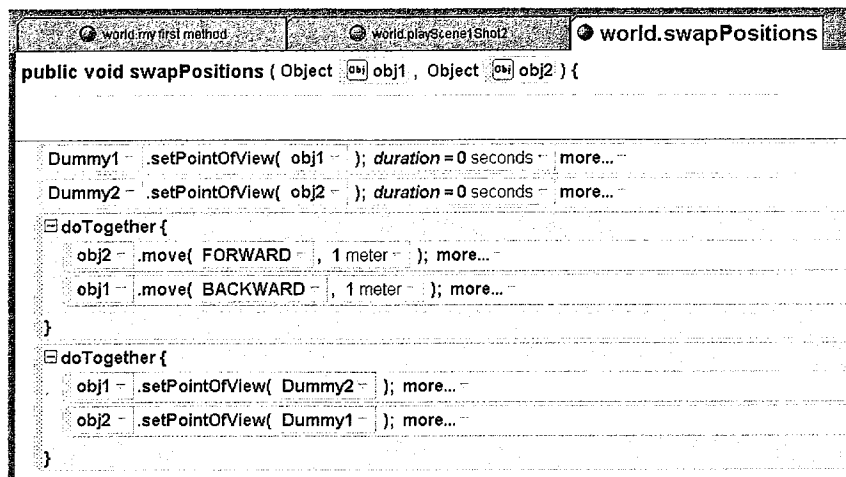


FIGURE B-22 Exchanging two objects' positions

In this definition, the two objects move simultaneously, one moving in front of the line of players, and the other moving behind the line of players, to avoid colliding with one another.

With method `swapPositions()` in hand, we are ready to define the recursive `reverse()` method.

### B.4.1 The Trivial Case

As we have seen, the first step in defining a recursive method is to find a case where the problem is trivial to solve. Since our `reverse()` method has this form:

```

void reverse(Object [] arr, Number index1, Number index2) {
}

```

any trivial cases must be identified using the `Number` parameters, `index1` and `index2`.

At this point, it is helpful to generalize from the specific problem at hand to the more general problem of reversing the positions of objects stored in an arbitrary array `arr`, where `index1` contains the index of the array's first element, and `index2` contains

the index of the array's last element. Thinking this way, there are two cases in which the problem of reversing the positions of the items in **arr** is trivial to solve:

1. If there is just one object in **arr**, then the object is already in its final position, so we should do nothing. There is one object in the array when **index1 == index2**.
2. If there are zero objects in **arr**, then there are no objects to move, so we should do nothing. There are zero items in the array when **index1 > index2**.

Since we do the same thing (nothing) in each of our trivial cases, the condition **index1 >= index2** will identify both of our trivial cases. Conversely, the condition **index1 < index2** can be used to identify our nontrivial cases.

## B.4.2 The Nontrivial Cases

We have hinted at how the nontrivial cases can be solved. Since **index1** is the index of the first (tallest) object in the array, and **index2** is the index of the smallest object in the array, we:

1. Swap the positions of the objects in **arr[index1]** and **arr[index2]**.
2. Reverse the rest of the objects (ignoring the ones we just swapped) recursively.

The trick is to figure out how to do Step 2. Drawing a diagram is often helpful, as shown in Figure B-23:

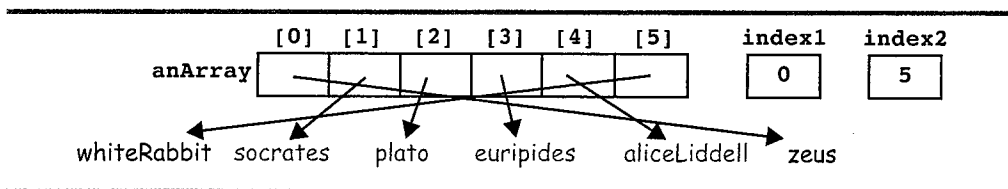
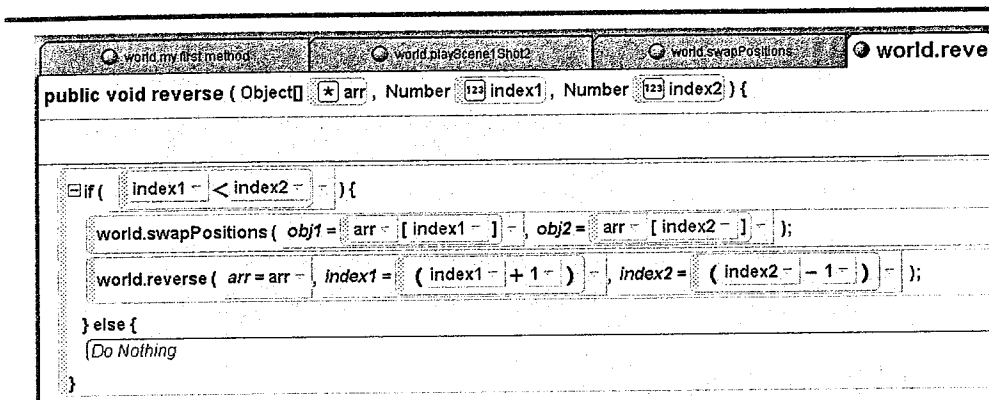


FIGURE B-23 Visualizing the recursive step

This allows us to clearly see the sub-array of objects that Step 2 must reverse; it begins at index 1 and ends at 4. However, to correctly solve the problem, we must express the arguments in Step 2 in terms of changes to our method's parameters, **index1** and **index2**. Expressed this way, the sub-array to be processed by Step 2 begins at index **index1+1**, and ends at **index2-1**. That is, we can solve the nontrivial cases of the problem as follows:

1. Swap the positions of the objects in **arr[index1]** and **arr[index2]**.
2. Recursively invoke **reverse(arr, index1+1, index2-1)**.

That's it! Figure B-24 presents a definition of **reverse()** that uses this approach.

FIGURE B-24 The recursive `reverse()` method

Note that our `reverse()` method does not change the order of the objects within the array. It merely uses the array as a table from which it can identify the tallest and shortest players, the next tallest and next shortest players, and so on.

Given this definition, we can finish `playScene2Shot2()`, as shown in Figure B-25.

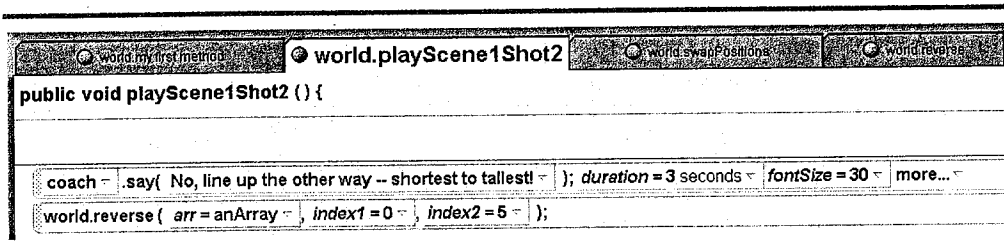
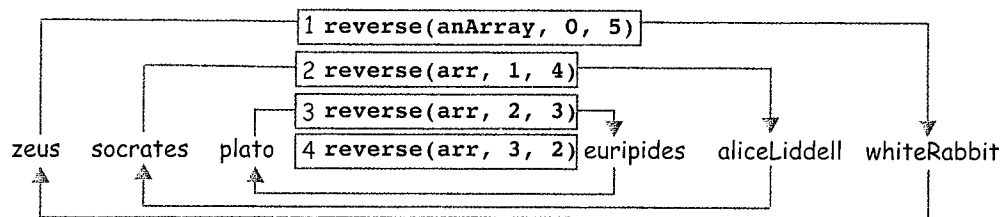
FIGURE B-25 The recursive `reverse()` method

Figure B-26 presents some screen captures taken as `playScene1Shot2()` runs. Compare them to the initial setting shown in Figure B-18 to see the progression of changes.



FIGURE B-26 Screen captures of Scene 1, Shot 2

Figure B-27 provides a conceptual view of what happens as `reverse()` runs.

FIGURE B-27 Conceptualizing `reverse()`

The fourth message, `reverse(arr, 3, 2)`, invokes the trivial case, halting the recursion.

Recursion is a powerful programming technique that can be used to solve any problem that can be decomposed into one or more “smaller” problems that are solved in the same way.

# Index

## Symbols

++ (increment) operator 124  
== (equality) operator 115

## Numerics

3D objects  
  orientation 59–62  
  position 57–59  
3D text 193–199

## A

ADD OBJECTS button 12  
Alice  
  downloading 3  
  installation 3  
  Statements 18  
Alice Gallery 12  
animal parameter 83  
arguments  
  recursion 220  
arrays 144, 157  
  indexed variables 161  
  integers, generating 167  
  marching ants example 157–161  
  random access example 163–167  
  read version 162  
  subscript operations 162  
  write version 162  
arrow keys, keyboard events and 191  
asSeenBy attribute 127, 139  
attributes 91  
  asSeenBy 127, 139  
  duration 127  
  methods 18  
  objects, retrieving 95–98  
axis 58

## B

background 196–198  
bees example 145–150  
Boolean functions 109–110  
Boolean operators 111–112, 139  
Boolean type 108  
Boolean variables 110  
borders  
  green 15  
  red 15  
bounding boxes 14  
  functions 26  
buttons  
  ADD OBJECTS 12  
  capture pose 89  
  create new event 180  
  create new function 99  
  create new method 32  
  create new variable 110, 146  
  drop dummy at camera 51  
  Play 10  
  Redo 10  
  Undo 10, 22  
buying tickets example 154–156

## C

camera object 10  
  dummies and 50  
  editing area and 16  
  setPointOfView( ) message and 54–57  
capture pose button 89  
classes, objects and 13  
clipboard 45–46  
  editing area and 45  
  statements and 46  
code, reusing 45–49  
comments, methods 40  
computeHypotenuse( ) method 75  
concatenating strings 79



- conditions in if statements 115–117
- control structures 108
- controls
  - doInOrder 15
  - doTogether 19
  - editing area 11
- counting loop, for statement 123–125
- create new event button 180
- create new function button 99
- create new method button 32
- create new variable button 110, 146

## D

- data structures 144
- debugging 16–17
- declaring variables 78
- defining variables 67
- depth, objects 24
- design
  - recursion and 229–233
- details area 25–27
  - functions tab 96
  - methods pane 15
  - panes 11
  - subparts 14
- distanceInFrontOf( ) function 71
- distanceTo( ) function 71
- doInOrder control 15
- doTogether control 19
- downloading Alice 3
- dragging methods 15
- dragon flapping wings example 38–42
- drop dummy at camera button 51
- dummies 50–54
- dummy objects 52
- duration attribute 127

## E

- Edit menu 10
- editing area 11
  - camera and 16
  - clipboard and 45
  - controls 11
- equality (==) operator 115
- errors, logic errors 182
- event handling 184, 184–185
- events 178
  - creating 180
  - handlers, defining 183
  - keyboard 185, 186–193
  - logic errors 182
  - mouse clicks 179–185
  - program events 185

- events area 11
- examples
  - bees 145–150
  - buying tickets 154–156
  - dragon flapping wings 38–42
  - jumping fish 85–88
  - Old MacDonald's farm 81–85
  - storing computed values 67–75
  - storing user-entered values 75–80
  - toy soldier marching 42–44
- expressions menu 72

## F

- Fibonacci series 135
- fibonacci( ) function 136–139
- File menu 10
- fixed-sized data structure (see arrays)
- flapWings( ) message 38
- flow 5
- flow control
  - for statement 121–125
  - functions and 134–139
  - pausing 118
  - selective 112–114
  - statements 108
  - while statement 127–134
- flow diagram 108
- for statement 108, 121–123, 139
  - counting loop 123–125
  - nested 126
  - while statements and 131
- fullName variable 97
- functions 26
  - attributes, retrieving 95–98
  - Boolean 109–110
  - distanceInFrontOf( ) 71
  - distanceTo( ) 71
  - fibonacci( ) 136–139
  - flow control and 134–139
  - lastIndexOf( ) 154
  - lists 153–154
  - Math.sqrt( ) 78
  - NumberDialog( ) 77
  - parameters and 99–102
  - partNamed( ) 167–174
  - size( ) 154
  - standard 213–215
  - wait( ) 139
  - world 215–218
- functions pane 11, 25–27
- functions tab 14
  - details area 96

**G**

general recursion 224–229  
 green borders 15  
 ground object 10

**H**

heBuilder 25  
 height, objects 24  
 Help menu 10

**I**

if statement 108, 114–115  
   conditions 115–117  
 if statements 139  
 increment (++) operator 124  
 indexed variables, arrays 161  
 indexOf() function 154  
 infinite recursion 219  
 installing Alice 3  
 integers, generating 167  
 isShowing property 22  
 iteration 150

**J**

jumping fish example 85–88

**K**

keyboard events 185, 186–193

**L**

lastIndexOf() function 154  
 light object 10  
 list menu 148  
 lists 144  
   bees example 145–150  
   buying tickets example 154–156  
   defining 169  
   entries 148  
   functions 153–154  
   iterations 150  
   methods 152  
   operations 150–154  
   variables, defining 146  
 local variables 67  
 logic errors 182  
 loops  
   for statement 123–125

iterations 150  
 nested 125–127

**M**

make a List checkbox 146  
 marchRight() method 43  
 Math.sqrt() function 78  
 menus  
   Edit 10  
   expressions 72  
   File 10  
   Help 10  
   Tools 10  
 messages 14, 21  
   attributes 18  
   flapWings() 38  
   move() 43  
   resize() 24  
   roll() 40  
   say() 17  
   sending 15–16  
   set() 23  
   statements and 32  
   turn() 43  
 method variables 67, 67–80  
 methods 15, 211–213  
   comments 40  
   computeHypotenuse() 75  
   defining 169  
   dragging 15  
   lists 152  
   marchRight() 43  
   move() 59  
   names 33  
   object methods 38–44  
   pane 25  
   pointAt() 15  
   scenes 32–36  
   shots 36–38  
   singVerse() 82  
 methods pane 11, 15, 24  
 methods tab 14  
 mnemonic values, keyboard events and 186  
 more... 18  
 mouse clicks 179–185  
 move() message 43  
 move() method 59

**N**

naming  
   methods 33  
   objects 13  
 nested for statements 126  
 nested loops 125–127

noise parameter 83  
 nontrivial case, recursion 229, 231, 237  
 nouns 5, 21  
 NumberDialog() function 77

## O

object methods 38–44  
 object tree 10  
   properties 21  
 object variables 67, 89  
 objects 21  
   adding 12–14  
   attributes 91  
     retrieving 95–98  
   bounding box, functions 26  
   bounding boxes 14  
   classes and 13  
   color 91  
   depth 24  
   dummy 52  
   height 24  
   naming 13  
   opacity 91  
   orientation 59–62  
   position  
     3D 57–59  
     quad view 27–28  
   renaming 13  
   subparts 14–15  
   vehicle 91  
   width 24  
 Old MacDonald's farm example 81–85  
 operators  
   Boolean 111–112, 139  
   equality (==) 115  
   increment (++) 124  
   relational 110–111  
 orientation of objects, 3D 59–62  
   pitch 60  
   point of view 62  
   roll 61  
   yaw 60

## P

panes  
   details area 11  
   functions 25–27  
   methods 24, 25  
   properties 21

parameters 67, 80–88  
   animal 83  
   functions 99–102  
   noise 83  
   values  
     validation 118–120  
 partNamed() function 167–174  
 pausing program flow 118  
 pitch, orientation 60  
 Play button 10  
 point of view, orientation 62  
 pointAt() method 15  
 positioning objects  
   3D 57–59  
   axis 58  
 print() statement 34  
 pristine 9  
 program design 4–5  
 program events 185  
 program style 11–12  
 properties  
   modifying from within program 22  
   vehicle 92–95  
 properties pane 11, 21  
 properties tab 14  
 property variables 67, 89–92

## Q

quad view 27–28

## R

read version, arrays 162  
 recursion  
   arguments  
     numeric 220  
   design and 229–233  
   general 224–229  
   infinite recursion 219  
   introduction 219  
   nontrivial 237  
   nontrivial case 229, 231  
   tail recursion 220, 221–224  
   trivial case 229, 231, 236  
   unwinding phase 220  
   winding phase 220  
 red borders 15  
 Redo button 10  
 relational operators 110–111  
 renaming objects 13

rep  
 res  
 retu  
 reu  
 (

## S

say  
 sce  
 1  
 1

scr  
 sel  
 ser  
 set  
 set

shu  
 shu  
 sir  
 siz  
 so  
 so  
 so  
 sp  
 sta  
 sta  
 sta

st  
 st  
 st  
 st  
 st  
 su

su

repetition, for statement and 121–123  
 resize() message 24  
 return statements 97  
 reusing code 45–49  
   objects in different worlds 46  
 roll() message 40  
 roll, orientation 61

## S

say() message 17  
 scenes  
   methods 32–36  
   transitions 200–207  
     barndoor wipe effect 203  
     box iris effect 205  
     edge wipe effect 203  
     fade effect 201  
     reusing transitions 206  
     special effects 200  
 scrolling through statements 36  
 selective flow control 112–114  
 sending messages 15–16  
 set() message 23  
 setPointOfView() message  
   camera control and 54–57  
   dummies and 50  
 sheBuilder 25  
 shots, methods for 36–38  
 singVerse() method 82  
 size() function 154  
 software design 21  
 software engineering 21  
 software implementation and testing 21  
 spirals, Fibonacci series and 135  
 standard functions 213–215  
 state, storing 182  
 statements 18  
   clipboard and 46  
   flow control 108  
   for 108, 121–123, 139  
     nested 126  
   if 108, 114–115, 139  
     conditions 115–117  
   messages and 32  
   print() 34  
   return 97  
   scrolling through 36  
   wait() 117  
   while 108, 127–134, 139  
 storing computed values 67–75  
 storing user-entered values 75–80  
 storyboard sketches 6–8, 21  
 string operations 78  
 strings, concatenating 79  
 subparts, objects 14–15  
   details area 14  
 subscript operations, arrays 162

## T

tail recursion 220, 221–224  
 targets, pointAt() method 16  
 templates 8  
 testing 16–17, 21  
 text  
   3D 193–199  
   hiding/unhiding 198–199  
   off-camera, repositioning 195  
 Tools menu 10  
 toy soldier marching example 42–44  
 transition diagrams 8  
 transitions between scenes 200–207  
   barndoor wipe effect 203  
   box iris effect 205  
   edge wipe effect 203  
   fade effect 201  
   reusing transitions 206  
   special effects 200  
 trivial case, recursion 229, 231, 236  
 turn() message 43  
 tutorials 3–4  
 types, Boolean 108

## U

Undo button 10, 22  
 unwinding phase, recursion 220  
 user stories 5–6, 21

## V

validation 118–120  
 values  
   computed, storing 67–75  
   sequences 144  
   user-entered, storing 75–80  
 values, validation 118–120  
 variables  
   Boolean 110  
   data structures 144  
   declaring 78  
   defining 67  
   fullName 97  
   initial value 69  
   lists, defining 146  
   local variables 67  
   method variables 67, 67–80  
   object variables 67, 89  
   parameters 67  
   property variables 67, 89–92  
 vehicle property 92–95  
 verbs 5

**W**

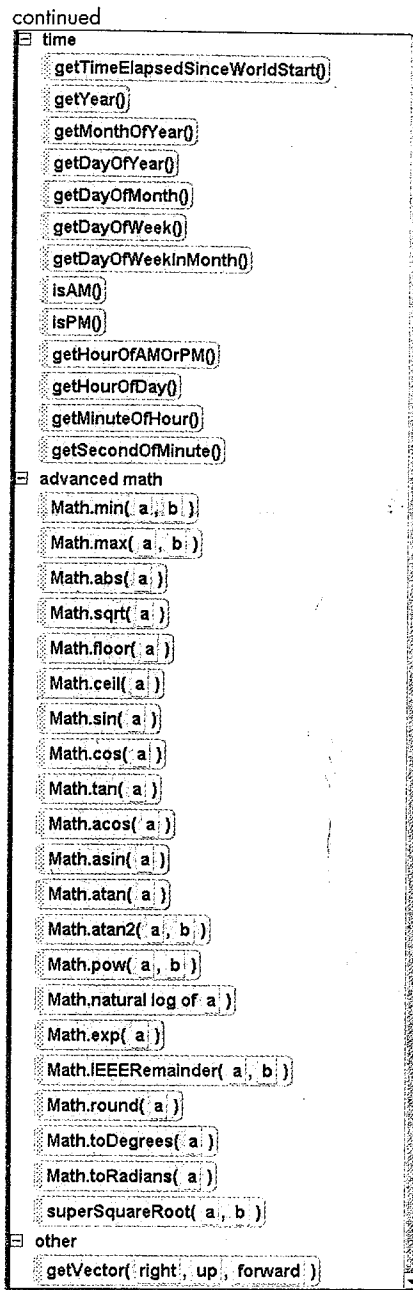
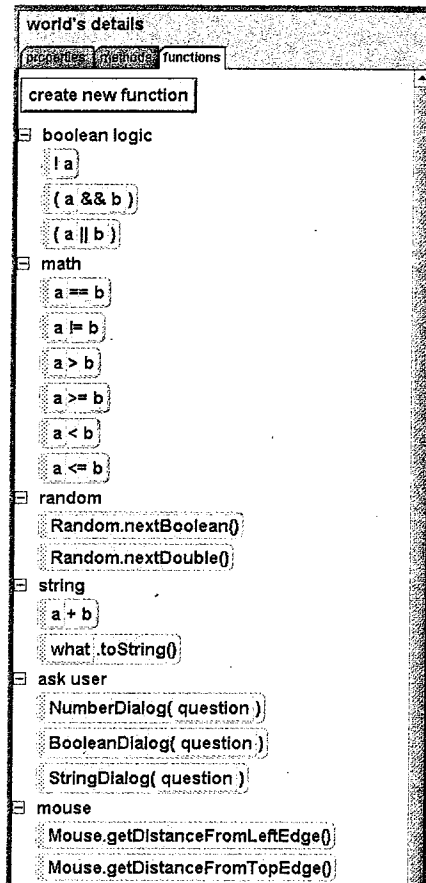
wait( ) function 139  
wait( ) statement 117  
welcome window 8  
while statement 108, 127–134, 139  
    for statements and 131  
width, objects 24  
winding phase, recursion 220

wizards 89  
world functions 215–218  
write version, arrays 162

**Y**

yaw, orientation 60

# World Functions



# in Action

## Computing Through Animation

Human-computer interaction (HCI) is a multidisciplinary field that combines principles from psychology, computer science, and design to create effective and user-friendly systems. This book explores the fundamental concepts and techniques of HCI, providing a comprehensive overview of the field. It covers the theoretical foundations of HCI, including the principles of human factors and ergonomics, and the practical applications of HCI in various domains, such as user interface design, usability testing, and user-centered design. The book also discusses the latest research and trends in HCI, such as the use of animation in HCI and the development of intelligent user interfaces.

### Features of the Text

• Provides a comprehensive overview of the field of HCI, covering both theoretical and practical aspects.

• Discusses the latest research and trends in HCI, such as the use of animation in HCI and the development of intelligent user interfaces.

• Includes numerous examples and case studies to illustrate the concepts and techniques discussed in the text.

• Provides a comprehensive overview of the field of HCI, covering both theoretical and practical aspects.

• Discusses the latest research and trends in HCI, such as the use of animation in HCI and the development of intelligent user interfaces.

### About the Author

Dr. [Name] is a Professor of Computer Science at [University Name] and has been working in the field of HCI for over 20 years. He has published numerous papers in the field of HCI and has been involved in several major research projects. He is also the author of several books on HCI, including this book. He is currently working on research in the area of intelligent user interfaces and the use of animation in HCI.



**COURSE TECHNOLOGY**  
CENGAGE Learning

For your lifelong learning solutions, visit [course.cengage.com](http://course.cengage.com)  
Purchase any of our products at your local college store or at our  
preferred online store [www.ichapters.com](http://www.ichapters.com)

ISBN-13: 978-1-4188-3771-6  
ISBN-10: 1-4188-3771-7



9 781418 837716