After finishing this chapter, you should be able to:

- Provide a brief definition of the following terms: binary branching, binary bypass, binary choice, branch, concurrency, control variable, count-controlled loop, flowchart, linear sequence, loop, multiple branching, parallel algorithm, post-test loop, pre-test loop, pseudo-code, repetition sequence, selection sequence, sentinel loop, thread.
- List and describe the three major elements of logical structure found in algorithms and describe how they relate to one another.
- List several criteria that should be met by each linear sequence.
- Describe how binary bypass and binary choice branching routines work, create simple flowchart segments for each, pseudo-code for each, and implement each in at least one Alice method.
- Describe how count-controlled and sentinel loops work, create simple flowchart segments for each, pseudo-code for each, and implement each in at least one Alice method.
- Describe what is meant by concurrent execution of instructions in an algorithm, and how to implement concurrent execution in Alice.
This chapter includes readings about the logical structure of algorithms—including linear sequences, selection sequences, repetition sequences, and concurrent execution of instructions in an algorithm—followed by four tutorials that will provide you with experience implementing these in Alice.

ELEMENTS OF LOGICAL STRUCTURE

Algorithms contain the steps necessary to complete a particular task or solve a particular problem. A recipe for baking a cake will have a list of all the ingredients needed, as well as step-by-step instructions on what to do with those ingredients. The recipe provides an algorithm for baking a cake.

When young children learn to perform long division, they are learning an algorithm. Professionals, such as engineers, architects, and doctors, apply many different algorithms in the course of their daily work. Some algorithms are simple; some can be quite long and complex. The Holtrop and Mennen Algorithm, which is used by naval architects to design the optimum propellers for an ocean going ship, involves several thousand steps and must be run on a computer.

Algorithms are sequential in nature. There are examples where several instructions in an algorithm are executed at the same time, but generally, we can think of the instructions in an algorithm as being executed one at time. They form a kind of sequential logic. Modern approaches to developing software recognize that this is only part of the story, but programmers still need to be able to design, manipulate, and implement sequential algorithms. They need to understand sequential logic.

There are certain patterns that exist in the design of sequential logic. These patterns fall into categories that can be understood as elements of logical structure, which can be combined in a myriad of ways to form the logical structure of algorithms in modern computer software. A programmer who is familiar with the design patterns of logical structure can more easily create and edit software.

Think about how this compares to a plumber or an electrician. A person who wishes to design a plumbing system for a building, such as a residential home, has a selection of existing parts from which to choose. We can see these parts in a hardware store or building supply warehouse—elbow joints, T-joints, certain kinds of valves, and so on. Despite the differences from one home to another, the plumbing systems will mostly be composed of the same parts, which we might think of as the elements of structure for a plumbing system. The architects who will design the system need to know how the parts work and how they fit together. The plumbers who will build or repair the system need to know how to work with each of the parts.
The same thing is true for an electrical system. The electrical engineers and electricians who design and build such systems need to be familiar with the parts that are available, how they work, and how they fit together. Switches, wires, outlets, junction boxes, circuit breakers, and so on, can be thought of as the building blocks of the system.

So it is with the elements of logical structure in an algorithm. They form the building blocks of the algorithm’s sequential logic. Each element of logical structure is a set of instructions that forms part of an algorithm. However, there are only a handful of basic elements of logical structure that programmers need to learn about, not hundreds or even thousands of different parts, as in plumbing and electrical systems. In the 1960’s two Italian mathematicians, Corrado Böhm and Giuseppe Jacopini, showed that algorithms are composed of three major structures: linear sequences, branching routines, and loops. Modern computer programming focuses on these three elements of logical structure.

FLOWCHARTS

Böhm and Jacopini used a system they called flow diagrams to describe their work. In Figure 4-1 you can see part of their manuscript showing some of their flow diagrams, which soon became known as flowcharts. A flowchart is a diagram showing us the structure of an algorithm. They weren’t the first to use such diagrams, but they formalized them and used them in their work on algorithms.

BOEHM AND JACOPINI'S ORIGINAL MANUSCRIPT

Böhm and Jacopini used a simple system of flowcharting with two symbols: rectangles to show each step in an algorithm, and diamond-shaped boxes to show what they called a “logical predicative.” More commonly, the diamond symbol for a logical predicative is called a “decision diamond,” a “decision box,” or a “conditional.”
To say that one thing is “predicated” on another means that one thing is determined by another. In other words, there is some condition that will determine what happens next. In an algorithm, these conditions will be either true or false. If the condition is true, one thing happens; if the condition is false, then something else happens. The path through an algorithm each time it is executed is determined by the state of the true or false conditions in that algorithm at that time. Flowcharts are designed to show the possible paths through an algorithm.

**FLOWCHARTING TEMPLATE**

Böhm and Jacopini’s notion of flow diagrams was relatively simple, but, in practice, flowcharts quickly became complicated as people continued to add more shapes. Figure 4-2 shows a flowcharting template first introduced by IBM in 1969. It was accompanied by a 40-page manual showing the proper way to use all of the symbols.

**FIGURE 4-2:** The IBM flowcharting template introduced in 1969

**FLOWCHART SYMBOLS**

In the rest of this chapter, we will use a simple version of flowcharting to help describe the elements of logical structure found in algorithms. We will use only three symbols: rectangles and diamonds as Böhm and Jacopini did, along with an oval-shaped box to mark the beginning and end of an algorithm, as shown in Figure 4-3.

The oval shape is called a terminator. There should be only one terminator at the beginning of an algorithm and one terminator at the end of an algorithm, since each algorithm should have one beginning, called an entry point, and one end, called an exit point. Usually they are labeled with the words “start” and “stop,” or sometimes “begin” and “end.”
The simplest element of logical structure in an algorithm is a linear sequence, in which one instruction follows another as if in a straight line. The most notable characteristic of a linear sequence is that it has no branching or looping routines—there is only one path of logic through the sequence, which doesn’t divide into separate paths, and nothing is repeated.

On a flowchart this would appear as a single path of logic, which would always be executed one step after another, as shown in Figure 4-4.

**FIGURE 4-3:** A flowchart drawn using only three simple symbols

![Flowchart](image-url)
Linear sequences are deceptively simple. It doesn’t seem very complicated to do one thing, then another, and then another, but it can be. Programmers need to make sure that linear sequences meet the following criteria:

- They should have a clear starting and ending point.
- Entry and exit conditions need to be clearly stated. What conditions need to exist before the sequence starts? What can we expect the situation to be when the sequence is finished?
- The sequence of instructions needs to be complete. Programmers need to be sure not to leave out any necessary steps. (This is harder than it sounds. See Exercise 2 at the end of this Chapter.)
- The sequence of instructions needs to be in the proper order.
- Each instruction in the sequence needs to be correct. If one step in an algorithm is wrong, then the whole algorithm is wrong.

In short, linear sequences must have clearly stated entry and exit conditions, and they need to be complete, correct, and in the proper order.
SELECTION SEQUENCES—BRANCHING ROUTINES

Sometimes an algorithm reaches a point where a decision can go one way or another. That is, the code is executing a selection sequence. As an example of a selection sequence, consider this example of a student who has chemistry lab at 2:00 p.m. on Fridays only:

Start
IF (Today is Friday)
THEN (Get to chemistry lab by 2:00 p.m.)
Stop

Diagrammed as part of flowchart, it would look like Figure 4-5.

This is an example of a branching routine. A branching routine occurs whenever the path or flow of sequential logic in an algorithm splits into two or more paths. Each path is called a branch. Branching routines are also known as selection sequences or selection structures.

POSSIBLE PATHS

If there are two possible paths, then the routine is known as binary branching. If there are more than two paths, then it is called multiple branching. “Would you like vanilla ice cream?” is a binary question—it has two possible answers, yes and no. “What flavor ice cream would you like?” is a question with multiple answers, not just yes or no. Binary branching is similar to the first question above; multiple branching is similar to the second.

It is possible to rewrite each multiple branching routine as a collection of binary branching routines. Consider an ice cream parlor with 28 flavors of ice cream. Instead of asking the multiple question, “What flavor ice cream would you like?”, a series of binary questions
could be asked—“Would you like vanilla ice cream?”, “Would you like chocolate ice cream?”, “Would you like strawberry ice cream?”, and so on. In a similar manner, every multiple branching routine in an algorithm can be rewritten as a series of binary branching routines.

The exercises in Alice later in this chapter only look at binary branching, not multiple branching. In fact, Alice does not have an instruction for multiple branching.

**BINARY BYPASS AND BINARY CHOICE ROUTINES**

There are two kinds of binary branching. One is called a *binary bypass*, and one is called a *binary choice*. In a binary bypass, an instruction is either executed or bypassed, as shown in Figure 4-5. In a binary choice, one of two instructions is chosen, as shown in Figure 4-6. The difference between a bypass and a choice is subtle but significant. In a binary bypass, it is possible that nothing happens, whereas in a binary choice, one of the two instructions will occur, but not both.

---

**FIGURE 4-6:** A binary choice

---

**PSEUDO-CODE**

Sometimes computer programmers use a more formal language, called *structured language* or *pseudo-code*, to describe algorithms. The term pseudo-code comes from the fact that it looks something like the code in a computer programming language, but not quite. It’s like code, but not really code, only a tool to help describe and understand algorithms, just as flowcharts do.
In pseudo-code a bypass is equivalent to an IF (condition) THEN (instruction) instruction. If the condition is true, then the instruction is executed; if the instruction is not true, then the instruction is ignored, and nothing happens. The chemistry lab example prior to Figure 4-5 shows a binary bypass.

A binary choice is equivalent to an IF (condition) THEN (instruction A) ELSE (instruction B). If the condition is true, then instruction A is executed; if the instruction is not true, then instruction B is executed. Either instruction A or instruction B will be executed, but not both. One of the two always happens, as seen in the example in Figure 4-6, in which a student has Math class on Monday, Wednesday, and Friday, and History class on Tuesday and Thursday. We will assume the student only needs to consider weekdays and not weekends. The pseudo-code showing an algorithm for the student’s day might include this:

```
IF (today is Monday, or today is Wednesday, or today is Friday)
    THEN (go to math class)
    ELSE (go to history class)
```

A set of instructions, called a block of instructions or block of code, could take the place of a single instruction anywhere in an algorithm, including in binary branching routines. In the preceding example above, `go to math class` could be a whole series of instructions.

One thing is common to all binary branching routines, and to all repetition sequences as well—there must be a condition to determine what to do. These conditions will be either true or false when the algorithm is executed. They are a form of conditional logic known as Boolean logic, which will be discussed in the next chapter.

**REPETITION SEQUENCES—LOOPING**

In the branching routines that you saw earlier in the chapter, the algorithms split into different paths that all moved forward; nothing was repeated. Whenever we branch backward to a previous instruction, and then repeat part of an algorithm, we have what is known as a repetition sequence. A repetition sequence forms a loop in an algorithm, which can be seen on a flowchart, such as in the following example for printing the numbers from 1 to 10. Figure 4-7 shows both the pseudo-code and a flowchart for the algorithm.

In this algorithm, the word “WHILE” is used for looping instead of the word “IF” that was used for branching. In pseudo-code, as in many programming languages, this tells the computer to loop back to the conditional expression when the block of code following the WHILE instruction is finished. Each time the condition is true, the computer will execute the block of code, and then come back to the condition again. When the condition is no longer true, the block of code will be ignored, much like a binary bypass, and the computer will move on to whatever comes next in the algorithm.
Like all loops, this loop has a control variable in its condition. A variable holds a value that can change, much like a variable from algebra, which stands for a number that could change. A control variable is a variable whose value controls whether or not a selection sequence will be executed. In this loop, the variable X stands for a number that is used to keep track of how many times to go through the loop. X starts at 1, the WHILE instruction tests to see if X is still less than or equal to 10, and 1 is added to X each time the loop is executed. The loop is executed while the control variable X is less than or equal to 10. 10 is the last value that is printed. When the value of control variable reaches 11, the loop is no longer executed.

**PRE-TEST AND POST-TEST LOOPS**

The loop in Figure 4-7 is a pre-test loop, meaning that the test to determine whether or not to go through the loop comes before the block of code to be executed. Traditionally, there are four parts to every pre-test loop:

- Initialization: an instruction that sets the first value of the control variable
- Test: the instruction that looks at the control variable to see if the loop should be executed
- Processing: instructions defining the process to be repeated
- Update: an instruction that changes the value of the control variable
In a pre-test loop, the test to determine whether or not to continue executing the loop comes before any other instructions that are to be repeated. It is also possible to set up a **post-test loop**, with the test to determine whether or not to repeat a loop coming after the instructions that are to be repeated. Figure 4-1, near the beginning of this chapter, shows diagrams of four different logical structures from Böhm and Jacopini’s original manuscript. Look closely at both the upper-right diagram and the lower-right diagram. In both cases, the condition in the diamond-shaped box is labeled with the Greek letter “α” (alpha), and the rectangular box representing an instruction to be repeated is labeled with the letter “a”. Notice that the upper structure is a pre-test loop with the decision diamond before the instruction to be repeated, and the lower structure is a post-test loop, with the decision diamond after the instruction to be repeated.

Some computer programming languages contain a `REPEAT (instruction) UNTIL (condition)` structure to set up a post-test loop, yet many computer scientists suggest that only pre-test
loops should be used in programming. To see why they suggest this, consider the following two programs:

1. Program 1—When the computer is turned on, the program erases the first hard drive and then asks “Should I do that again for the next hard drive?” (Of course, this assumes the program will still run after the first hard drive has been erased.)

2. Program 2—When the computer is turned on, the program asks “Do you want me to erase the first hard drive?” If the answer is yes, it erases the first hard drive, and then it asks if you want to repeat the process for the next hard drive.

The second program is slightly more complicated than the first, but which do you think is a safer program to run?

Unfortunately, people often think very differently from the way a computer works. We tend to do something first, then ask if it should be repeated, like a post-test loop instead of a pre-test loop—just the opposite of what computer scientists suggest. Alice has a WHILE instruction for pre-test loops, but it does not contain any commands to set up a post-test loop.

**COUNT-CONTROLLED AND SENTINEL LOOPS**

In addition to being a pre-test loop, the example in Figure 4-8 is also a count-controlled loop. Every loop in a computer program is either a count-controlled loop, or a sentinel loop. A count-controlled loop causes a process to be repeated a specific number of times. A sentinel loop causes a process to be repeated until a condition or marker, called a sentinel, is encountered.

In a count-controlled loop, the control variable is a called a **counter**. We need to know the initial value, the final value, and the increment for the counter. The loop starts with the counter at the initial value. The increment is the amount added to the counter each time through the loop. If the increment is positive, then the index increases by the increment each time through the loop. If the increment is negative, then the index decreases by the increment each time through the loop. The final value is the last value processed by the loop. In Figure 4-8, the initial value is 1, the increment is 1, and the final value is 10.

It’s important to make sure that the initial value, the final value, and the increment all match each other. If a computer were programmed to start the counter at 100, and then increase it by 1 each time through the loop until it reached 0, we would probably get some unexpected results. If the increment is positive, then the final values should be higher than the initial value. If the increment is negative, then the final values should be lower than the initial value.

Alice handles count-controlled loops with a special Loop instruction, so, most of the time counters and increments will be handled for you automatically in Alice. However, Alice’s loop instruction does not let us use a negative increment. That is, you could not use the
loop instruction to add a negative number to the counter each time through the loop, or, effectively, subtract a number from the counter. If you wanted to start at 100 and count backwards until you reached zero, such as the countdown for launching the space shuttle, then you would need to set up your own count-controlled loop using the While instruction instead of the special Loop instruction.

A count-controlled loop is a special case of a sentinel loop, in which the sentinel involves a counter, but the term “sentinel loop” is generally used to refer only to loops that are not count-controlled.

As an example of such loops, imagine a machine that tests a car door. The machine, which is controlled by a computer program, opens the door, and then closes the door. The machine could be programmed to repeat this a certain number of times with a count-controlled loop, but it could also be programmed to repeat the process until the door falls off, as shown in the following pseudo-code:

```
BEGIN
LET counter = 0
WHILE (door is still on the car)
  
  open the door
  close the door
  increment counter by 1

PRINT "The door fell off after opening and closing this many times:"
PRINT counter
END
```

This loop has a counter, but the counter does not control when the loop stops running, so this would not be a count-controlled loop, but a sentinel loop. It is the sentinel condition, the door falling off, that controls when the loop will stop.

In summary, when code in a computer program is repeated, the algorithm contains a repetition structure, which is also called a loop. Algorithms can contain count-controlled loops or sentinel loops that are not count-controlled. Each loop is also a pre-test loop or a post-test loop. Alice has a WHILE instruction for pre-test loops and does not allow post-test loops. Alice also has a special LOOP instruction for count-controlled loops.

There are two methods of programming that are often more appropriate than loops in many situations—event-driven programming and recursion. You already know enough about events in Alice to ask yourself if it might be more appropriate to prepare an event to handle the situation whenever you are considering the use of a loop. The seaplane world in Chapter 3 is an example where this occurs. Recursion, a powerful programming tool in which a method calls itself, will be covered in Chapter 7. Events and recursion are sometimes a little harder to use than loops, but in the long run, they often work better than loops.
CONCURRENCY IN ALGORITHMS

It is possible for one computer, or several computers working together, to work on several parts of an algorithm at the same time. Each path of logic that is being executed is called a thread of sequential logic, and algorithms that run multiple threads at the same time are called parallel algorithms. The process of running multiple threads is called concurrent execution, or concurrency.

Parallel algorithms can be quite powerful, but they can be difficult to design and use. Many problems arise, such as the different threads interfering with each other. It might be easier to run a restaurant kitchen with four chefs instead of one, but if things aren’t carefully coordinated, then chaos could ensue.

Concurrency is mentioned here for two reasons: first, it is becoming more common, even in simple programs, and second, concurrency is important in Alice, such as when an object should move and turn at the same time, or when two objects should move at the same time.

You have already seen a simple version of concurrency in Alice. In Chapter 3 you used the Do together logical structure, which causes concurrent execution of separate instructions. Alice also has a For all together tile that can be used with lists; this tile is covered in Chapter 8.

TUTORIAL 4A—BRANCHING IN ALICE METHODS

In this exercise you will modify the generic triple jump world from Chapter 2 to include user input and branching. The world contains three objects, each a character from Alice in Wonderland. The existing version of the world contains a method to make all three characters jump, one at a time. The algorithm in world.my first method is simply a linear sequence. You will modify it to include user input and If...Then instructions. The new program will ask the user questions about which character should jump, then have one of the three characters jump, depending on the answers to those questions.

Alice has a world-level function to ask the user a yes or no question. You are going to add two questions to world.my first method. First, the method will ask if the user wants Alice to jump. If the answer is yes, then Alice will jump. If the answer is no, then the method will ask if the user wants the White Rabbit to jump. If the second answer is yes, then the White Rabbit will jump, if the answer is no, then the Cheshire Cat will jump. The pseudo-code and flowchart in Figure 4-9 below describes this algorithm.
Before you start, you need more information about the user input functions in Alice. There are three world-level functions in Alice to ask the user a question: *ask user for a number*, *ask user for yes or no*, and *ask user for a string*. Figure 4-10 shows the tiles for these three functions on the function tab in the Details area for the world.
In the following steps, you will use the function to ask user for yes or no, which returns a value of true if the user answers “yes” to the question and false if the user answers “no”. This function may be used any place in an Alice method where true or false can be used, such as in a condition in an If/Else instruction.

1. Start the Alice software and open the generic triple jump world created in Chapter 2. If you cannot find the world, then either load the world from the CD that comes with this book, or complete tutorials 2B and 2C to create and save the world before continuing.

2. Look at the code for world.my first method, as shown in Figure 4-11. You can see that there are several instructions that form a linear sequence in the program. You need to add an If/Else instruction to the method. Drag a copy of the If/Else tile from the bottom of the Editor area and drop it into the method just below the three jump instructions.

3. A short menu will appear asking you if you want to use a true or false condition in the If/Else instruction. Select true, and a light greenish-blue If/Else tile will appear in your method, as in Figure 4-12.
4. Next you need to replace `true` as the condition for the `If/Else` instruction with the function to `ask user yes or no`. Select the `world` tile in the Object tree, and then click the `functions` tab in the Details area. Scroll through the list of functions and find the function titled `ask user for yes or no`. Drag and drop a copy of this function into the `If/Else` tile in place of `true` following the word `If`.
5. A short menu will appear with the options Yes or No? and other .... This menu is asking you how you want to word the question that the user will see. Click other ..., and the Enter a string dialog box will appear. The character string entered here will form the text of the question the user will see. Type Do you want Alice to jump? as the string, and then click the OK button. Your question will now appear in the If/Else tile in place of true as the condition for the If/Else instruction, as shown in Figure 4-13.

FIGURE 4-13: The If/Else instruction with the ask user for yes or no function in place

6. Drag the Alice jump tile and drop it into the If/Else instruction in place of Do Nothing immediately below the If clause and above the word Else. Now if the person answers “yes” to the question, Alice will jump.

7. If the person answers “no” to the first question, he or she should see a second question. Thus, another If/Else instruction is needed following the word Else. Drag and drop another IF/Else tile from the bottom of the Editor area and drop it in place of Do Nothing following the word Else, then, click True when the short menu appears. Now you have nested If/Else instructions—one IF/Else tile inside another one.

8. You need to put another question in place of true in the second If/Else instruction. As before, find the function titled ask user for yes or no on the world’s functions tab. Drag and drop a copy of this function into the If/Else tile in place of true following the word If in the second If/Else instruction.

9. A short menu will appear. Click other ..., and the Enter a string dialog box will appear. Type Do you want the White Rabbit to jump? as the string and click the OK button. Your second question will now appear in the If/Else tile in place of true as the condition for the If/Else instruction.

10. If the person answers “yes,” the whiteRabbit should jump. Drag the whiteRabbit jump tile and drop it in the instruction in place of Do Nothing below the If clause and above the word Else.

11. Drag the cheshireCat jump tile, and drop it in the instruction in place of Do Nothing below the word Else.
12. Your method should now match the specifications as shown in Figure 4-9 and should look like the code shown in Figure 4-14. You are now ready to test the new program, but first you should save your work. Save the world with the name *jump user choice*.

It’s now time to test the world. It’s a good idea to test the world under all possible circumstances, which in this case means trying the world with all possible combinations of user input. This calls for a testing plan.

The specifications back in Figure 4-9 show that there are three possible paths for the logic in the program. The answer to the first question could be yes or no. If it’s yes, then Alice should jump and the program is done. If it’s no, then the second question appears. If the answer to the second question is yes, then the White Rabbit jumps, and the program ends. If the answer to the second question is no, then the Cheshire Cat jumps, and the program ends. The testing plan must include three trials, one for each possibility, as follows:

- **Trial 1**—first answer “yes”  
  expected outcome—Alice jumps

- **Trial 2**—first answer “no”, second answer “yes”  
  expected outcome—White Rabbit jumps

- **Trial 3**—first answer “no”, second answer “no”  
  expected outcome—Cheshire Cat jumps
Test your program according to the testing plan, and see if it works as expected. If it does, you’re done, if not, then it’s time to debug, etc. Remember to save your world again if you make any significant changes.

**TUTORIAL 4B—A SIMPLE COUNT-CONTROLLED LOOP**

In this exercise you will experiment with count-controlled loops in Alice.

Alice has a special *Loop* instruction to make it easier to set up a count-controlled loop. The *Loop* instruction has two different versions: a simple version and a complicated version. Both versions of the same loop are shown in Figure 4-15.

In the simple version, the programmer simply tells Alice how many times to repeat the loop, and Alice will deal with the counter, increment, and final value automatically to stop the loop when it has been executed the specified number of times. In the complicated version, the programmer has access to the initial value, final value, and increment.

In the next several steps you will modify the generic triple jump world created in Chapter 2 using the simple versions of Alice’s loop instruction to make the characters jump a specified number of times.

1. Open the *generic triple jump* world created in Chapter 2, or create it again as described in Exercises 2B and 2C in Chapter 2. A copy of the finished world is on the CD accompanying this book.
2. In this program, all three characters will jump at the same time. Drag a *Do together* tile from the bottom of the Editor area and place it in your method after the three jump instructions.
3. Drag each of the *jump* instructions into the middle of the *Do together* tile, as shown in Figure 4-16. This is an example of concurrency in an algorithm.
4. Save the world first with the name *triple jump loop*, and then play the world. If all three characters jump at the same time, then move on to the next step. If not, then find and fix the error.
5. Next you will add a simple count-controlled loop to the program to make the three characters jump a certain number of times. Drag a **Loop** tile from the bottom of the Editor area and drop a copy of it into the method just below the **Do together** tile. When you do this, an end dialog box will appear asking you how many times you wish to repeat the loop. Select 5 as the number of times.

6. Drag the **Do together** tile into the **Loop** tile. Your method is complete and should now look like Figure 4-17.

7. You can now test the world again, to make sure that the characters jump together five times. Save the world first, and then play it to see if it works. This demonstrates the use of the simple version of a loop instruction, and all of the characters should jump together five times. If the program doesn’t work properly, review your work to find the error in your program, and then fix it. Once it works, you are finished with this exercise.
TUTORIAL 4C—A MORE COMPLICATED COUNT-CONTROLLED LOOP

In this exercise you are going to modify the triple jump loop world created in Tutorial 4B. You will work with the more complicated version of the Loop instruction, using the loop's control variable to determine how high the Cheshire Cat jumps. First, you will modify the jump method `world.jump [who]` to include a height parameter. Next, you will use the new method to make the characters each jump a different height—Alice will continue to jump one meter, the White Rabbit will jump two meters, and the Cheshire Cat’s height will depend on the value of the counter in the loop. Let’s start by adding a parameter to the jump method.

1. Open the triple jump loop world from Tutorial 4B.
2. Click File on the menu bar, and then click Save World As to save a copy of the world with the name triple jump loop 2 so that the changes you make will not alter the original triple jump loop world.
3. Select the world tile in the Object tree and the methods tab in the Details area. Click the edit button next to the `jump [who]` tile on the methods tab, and the method `world.jump` should open in the Editor area as shown in Figure 4-18.

![Figure 4-18: The method world.jump opened in the Editor area](image)

4. You will now add a height parameter to the jump method. Click the create new parameter button on the right side of the top of the method, as identified in Figure 4-18. A dialog box will appear asking you for the name and type of the new parameter. Type height as the name and select Number as the type, and then click the OK button. Now the method has two parameters—who, which is an object parameter, and height, the number parameter that you just added.
5. Next you will modify the move up and move down instructions in the method to use the height parameter as the amount to jump instead of 1 meter. Drag the height parameter tile from the top of the method and
drop a copy of it into the *move up* instruction tile in place of the value *1 meter*. Do the same thing for the *move down* instruction. Now, instead of jumping up and down one meter each time the *generic jump* method is used, the object will jump up and down the amount specified by the *height* parameter. Figure 4-19 shows the two move instructions.

Now that a *height* parameter has been added to the *jump* method, you should be able to modify how high each character jumps in the program that calls the *jump* method. In this world, the method *world.my first method* calls the *jump* method for each of our three characters. You need to change the amount passed to the *jump* method from *world.my first method*.

1. Click the *edit* button next to the *my first method* tile in the Details area, and you will now see *world.my first method* in the Editor area. Notice that a *height* parameter has been added to each jump instruction with the default value 1.

2. The rabbit should be able to jump higher than Alice, so click the *height* parameter in the *world.jump who= whiteRabbit* tile in the middle of the method, and change the value to 2.

3. Save and play your world. The White Rabbit should be jumping twice as high as Alice and the Cheshire Cat. If not, find and fix your error.

Each time through a loop is called an iteration of the loop. The loop control variable, named *index*, starts at zero, and increases by one with each iteration, as follows: 0, 1, 2, 3, and 4. The loop starts counting at zero and stops before reaching five. So, even though the loop executes five times, the first value of the index is 0, and the last value is 4. You will now make the amount the Cheshire Cat jumps each time equal to the value of the index—0 the first time through the loop, 1 the second time, and so on.

Next you will make the Cheshire Cat jump a different amount each time the loop repeats. To do this, first you need to be able to see the complicated version of the loop instruction. The *Loop* tile contains a *show complicated version* button. If you clicked this button now, you would see the complicated version of the loop, as shown in Figure 4-20.
1. Drag a copy of the index tile from the Loop instruction and drop it into the `world.jump who= cheshireCat` tile in place of the value 1 as the height parameter. Your code should now look like Figure 4-20.

2. Save the world and test it. The amount the Cheshire Cat jumps should be equal to the index for the loop, which starts at 0 and increments by 1 each time, stopping before 5 is reached. The Cheshire Cat should first jump 0 meters, then 1 meter, 2 meters, 3 meters, and 4 meters. He might be jumping off the screen the last few times.

   **NOTE**  The loop instruction in Alice is really intended to be used only in situations where the programmer wants to make something happen a certain number of times, such as jumping five times. Remember, that a count-controlled loop is just a special case of a sentinel loop. Whenever a more sophisticated loop is called for, such as one that counts backwards, it is best to create your own version of a count-controlled loop with a While instruction.

### TUTORIAL 4D—USING THE WHILE INSTRUCTION

In this tutorial you are going to use the While instruction to duplicate the effect of the Loop instruction used in Tutorial 4C. Remember that a sentinel loop has a value or condition that tells a loop when to stop executing. A count-controlled loop is just a special case of a sentinel loop.

You will use the triple jump loop world from Tutorial 4B as your base world, modifying it to use the While instruction instead of the Loop instruction, but the new world should function
in a way very similar to the old world. Figure 4-21 shows the algorithm for a simple count-controlled loop alongside the new algorithm for the \texttt{While} loop you will create. Notice that the \texttt{Loop} instruction handles the initialization, test and update automatically, whereas the programmer must include instructions to deal with these steps in the \texttt{While} loop.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure421.png}
\caption{A simple count-controlled loop on the left, and a \texttt{While} loop on the right that functions as a count-controlled loop}
\end{figure}

You will need to add a variable to \texttt{world.my first method} to function as the control variable, add a \texttt{While} loop to the method, and delete the \texttt{Loop} instruction. As you go through the following steps, use Figure 4-22 as a reference.

1. Open the \texttt{triple jump loop} world that you saved in Tutorial 4B. If you cannot find the world, then either load the world from the CD that comes with this book, or redo Tutorial 4B to create and save the world before continuing.
2. Select the \texttt{world} tile in the Object tree and the \texttt{methods} tab in the Details area. Click the \texttt{edit} button next to the \texttt{my first method} tile on the methods tab; the method \texttt{world.my first method} should open in the Editor area.
3. You need to create a new control variable for the \texttt{While} loop that you will add to the program. Click the \texttt{create new variable} button, and a dialog box will appear asking you for the name and type of the new variable. Type \texttt{count} for the name and select \texttt{Number} as the type, and then click the \texttt{OK} button.
4. The variable tile at the top of the method shows that the count is initialized to 1. Click the 1 and change the value to 0.
5. Drag a **While** instruction tile from the bottom of the Editor area and drop it in the method below the **Loop** tile. Select **true** from the short menu that appears.

6. Next, drag each of the three **jump** instructions from the **Loop** tile and drop them in the **While** tile.

7. Now the **Loop** tile is no longer needed. Right-click the **Loop** tile and click **delete**.

Remember from the reading at the beginning of this chapter that there are four parts to every pre-test loop, which is what you are creating: initialization, test, processing, and update. Each of these parts of the loop needs to be properly in place for the loop to function as desired. Count is initialized to 1 in the **count** variable tile at the top of the method, so this will suffice as the initialization step for the loop. The three **jump** instructions are the processing in the center of the loop. You only need to modify the test and add the update step.

The condition in the **While** instruction will be the test to see if the loop needs to be repeated. The algorithm in Figure 4-21 shows that the loop should continue while the count is less than 5. It also shows that 1 should be added to **count** at the end of the loop. You need to modify the code in **world.my first method** to match this.
1. Drag a copy of the count variable tile and drop it into the While tile in place of the value true. When the menu appears with different choices for the conditional expression, choose \texttt{count <}, then \texttt{other}, and set the value to \texttt{5}. Now the loop will repeat while \texttt{count} is less than \texttt{5}.

2. Drag the count variable tile from the top of the method and drop a copy in the While tile after the three jump instructions. When you do this, a short menu will appear asking you how you wish to set the value of count. Choose \texttt{expression}, and then click \texttt{count}.

3. Now the tile says \texttt{count set value to count}. You need to build a math expression so the tile will say \texttt{count set value to count +1}. Click the second word \texttt{count}, and select \texttt{math} from the menu that appears. Then select \texttt{count +} and then \texttt{1}. Now \texttt{world.my first method} should match the specifications as shown in Figure 4-21 and should look like Figure 4-22.

4. You need to save and test the world. Save the world with the name \texttt{triple jump while loop}, then play the world to make sure that the characters jump together five times, just as they did with the simple count-controlled loop in Tutorial 4B. If the program doesn’t work properly, review your work to find the error in your program, and then fix it. Once it works, you are finished with this tutorial.
CHAPTER SUMMARY

This chapter consisted of several readings about the logical structure of algorithms—including linear sequences, selection sequences, repetition sequences, and concurrent execution of instructions in an algorithm—followed by four hands-on tutorials.

The readings discussed the following:

- **Algorithms are sequential in nature;** we can think of the instructions in an algorithm as being executed one at a time.

- **Each element of logical structure is a set of instructions that forms part of an algorithm.** Corrado Böhm and Giuseppe Jacopini showed that algorithms are composed of three major structures: linear sequences, selection sequences (branching routines), and repetition sequences (loops).

- **A flowchart is a diagram showing us the structure of an algorithm.** Flowcharts are designed to show the possible paths through an algorithm.

- **The simplest element of logical structure in an algorithm is a linear sequence,** in which one instruction follows another as if in a straight line. Linear sequences must have clearly stated entry and exit conditions, and they need to be complete, correct, and in the proper order.

- **A selection sequence (branching) occurs whenever the path or flow of sequential logic in an algorithm splits into two or more paths.**

- **There are two kinds of binary branching in algorithms:** a binary bypass and a binary choice. In a binary bypass an instruction is either executed or bypassed. In a binary choice one of two instructions is chosen.

- **Whenever we branch backward to a previous instruction,** and then repeat part of an algorithm, we have what is known as a repetition sequence (loop).

- **A control variable is a variable whose value controls whether or not a selection sequence will be executed.**

- **In a pre-test loop,** the test to determine whether or not to continue executing the loop comes before any other instructions that are to be repeated. In a post-test loop, it comes afterward. Many computer scientists recommend that only pre-test loops be used.

- **There are four parts to every pre-test loop:** initialization, test, processing, and update.

- **Every loop in a computer program is either a count-controlled loop or a sentinel loop.** A count-controlled loop causes a process to be repeated a specific number of times. A sentinel loop causes a process to be repeated until a condition or marker, called a sentinel, is encountered. Actually, a count-controlled loop is a special case of a sentinel loop.

- **It is possible for a computer to execute several instructions from the same algorithm at the same time.** This is called concurrency, and algorithms that include concurrency are called parallel algorithms.

In Tutorial 4A, you learned how to use the `If/Else` instruction to include binary branching in an Alice method, and to use the `ask user a yes or no question` function.
In Tutorial 4B, you learned how to include a simple count-controlled loop in Alice with the Loop instruction, and used the Do together instruction to perform concurrent execution of several methods.

In Tutorial 4C, you learned how to use the complicated version of the Loop instruction, and to use the loop index variable within an instruction in the loop.

In Tutorial 4D, you learned how to create a properly structured While loop.

**REVIEW QUESTIONS**

1. Define the following terms:
   - binary branching
   - binary bypass
   - binary choice
   - branch
   - concurrency
   - control variable
   - count-controlled loop
   - flowchart
   - linear sequence
   - loop
   - multiple branching
   - parallel algorithm
   - post-test loop
   - pre-test loop
   - pseudo-code
   - repetition sequence
   - selection sequence
   - sentinel loop
   - thread

2. Create a set of instructions for a simple everyday process that contains a number of steps, such as making a cup of coffee or getting from your school to where you live. Exchange directions with another student, and critique each other's work. In particular, are the linear sequences in your algorithm complete, correct, and in the proper order?

3. Compare the structures created by using an If/Else instruction and a While instruction in pseudo-code. How are they the same? How are they different? What would each look like on a flowchart?

4. To add two fractions, such as 1/2 and 1/3, the fractions must have a common denominator. Using both pseudo-code and a flowchart, describe a general algorithm for adding two fractions.

5. What is meant by the term “nested” If/Else instructions? Give at least one example of nested If/Else instructions using pseudo-code and flowcharts to describe your answer.

6. List and describe the four parts of every pre-test sentinel loop.

7. The following algorithm was intended to result in the numbers from 10 to 1 being printed. What will it actually do? What is wrong with it, and how can it be corrected?
   ```
   BEGIN
   count = 10
   While count > 0
   {
   Print count
   Count = count +1
   }
   Print "The countdown is finished."
   END
   ```
8. Tutorial 4B shows how to set up a count-controlled loop using the Loop instruction. Tutorial 4D shows how to do the same thing using a While instruction. Figure 4-20 shows the two algorithms side-by-side. What are the advantages to using the Loop instruction to set up a count controlled loop? What are the advantages in using the While instruction to set up a count controlled loop?

9. Describe the difference between a pre-test loop and a post-test loop. Which is generally safer to use in a computer program and why?

10. A data file for a payroll program consists of a set of records. There is one record for each employee, containing the employee’s ID number, first name, last name, and hours worked. The last record contains “0000” as the employee number. Using pseudo-code and a flowchart, describe an algorithm that will read in and print each record no matter how many records are in the file when the algorithm is executed.

EXERCISES

1. Alice contains a world level function that will ask the user for a number. Create an Alice world with a character of your choice that will ask the user for a number, cause the character to jump up, use a loop to spin around the number of times specified, and then come back down. What is the difference between using the turn and roll methods to make the character spin one revolution each time through the loop?

2. Modify the triple jump loop world so that Alice jumps if the index is equal to 1, the White Rabbit jumps if the index is equal to 2, the Cheshire Cat jumps if the index is equal to 3, and all three characters jump if the index is equal to 4.

3. Add a tree to the triple jump while loop world. Make the sentinel to be the tree’s height, rather than the number 5, so that the loop will execute when the distance is less than or equal to the tree’s height, but stop when the distance passes its height. To do this, you can use the character level function that returns an object’s height.

4. Modify the triple jump loop2 world from Tutorial 4C to make the height the Cheshire Cat jumps increase by .5 meters each time through the loop.

5. Make the height in the triple jump loop2 start at a higher number and decrease each time until it is less than or equal to a smaller value. For example, the initial value could be 5, the increment could be –1, and the final value could be 1.

6. Modify the triple jump loop2 world to make the height the White Rabbit jumps increase each time through the loop, while the height the Cheshire Cat jumps decreases at the same time. Consider the following: what happens to \(5 - x\) as \(x\) increases from 1 to 5?

7. Modify the finished triple jump While loop world from Tutorial 4D to use the increment instruction instead of the instruction that says count set value to count +1. To do this, delete the old set value instruction, and drag the count variable tile, and drop it into the code for the method, and select increment count by 1 instead of set value.
8. Alice contains Hebuilder and Shebuilder class tiles in the People folder of the Local Gallery. You may create your own character objects in Alice using these. The new character will have a method to walk. Create an Alice world with a character of your own creation as an object in the world. Using the walk, move, and turn methods, create an Alice method to make the character walk around in a complete circle. (You may also approximate a circle with a polygon.)

9. The Animals object gallery contains a pterodactyl. The disk that comes with this book contains a world named “flapping pterodactyl” with a character-level method named `flap` that will cause the pterodactyl to flap its wings. Do each of the following:
   a. Create a method called `pterodactyl fly` that will make the pterodactyl move forward while flapping its wings, and then create a loop in `world.my first method` to make the pterodactyl fly away.
   b. Modify the world to use an event instead of a loop to make the pterodactyl fly while the world is running. You may need to refer back to Chapter 3, which covers events.
   c. Add controls to your pterodactyl world so that the user can steer the pterodactyl.
   d. Add a user control to point the camera at the pterodactyl when the spacebar is pressed.

10. In Exercise 9, why is it better to use an event instead of a loop to make the Pterodactyl continue flying while the world is running?