8 Repetition: Recursion

This chapter continues our discussion of repetition with an introduction to a powerful form of repetition known as *recursion*. The Loop construct presented in chapter 7 is somewhat limited. To use a Loop, Alice needs to know a *count* -- how many times the loop will be executed. Using a counted loop is reasonable for many problems (such as those presented in the last chapter), but there are many problems for which such information is unavailable. When we cannot know (at the time the program is being written) a count for the number of times a repetition is to occur, we may use recursion. **Recursion means that a method (or a question) calls itself.** This is an extremely powerful technique that greatly enhances the types of problems that can be solved in Alice.

Under what situations will we not know the *count* of repetitions at the time the program is written? In Alice, there are two major situations. The first is when *random motion* is involved. Random motion means that an object is moving in some way that is unpredictable. Section 8-1 will explore the topic of random motion. Random motion is used to introduce a form of recursion called *generative* recursion.

The second situation where we do not have a count of repetitions is when some complex computation is to be done that depends on an ability to break a problem down into smaller sub-problems. Once the smaller sub-problems are solved, the solutions to the smaller sub-problems are used to cooperatively solve the larger problem. In these kinds of situations, we can use a form of recursion called *structural* recursion. In Section 8-2, we will look at a famous puzzle and present a solution using structural recursion.

Note to the instructor: Most textbooks cover while loops (the topic of Chapter 9) before recursion. We intentionally do the opposite. We look at the general situation first and then consider the conditions in which a while loop is used. Topic sequence and coverage, of course, can be varied to suit your preferences and teaching style.

8-1 Repetition: Random motion and recursion

Introduction

A *Loop* construct, as introduced in Chapter 7, requires that the programmer specify the number of times the loop is to be repeated. This could be a numeric constant, such as 10, or a question such as *boy.distanceTo(girl)*. But, exactly how many times a loop should repeat might not be known ahead of time. This is often the case in games and simulations. In this section, a form of looping will be introduced to handle situations where the programmer does not know how many times the loop should be repeated. The technical type of looping that will be presented in this section is generative recursion. Also, we will see that our code is *tail-recursive*.

Chase scene

Let's look at an example of a situation where we do not know (ahead of time) how many times a segment of code should be repeated. This animation will simulate a "chase scene." Chase scenes are common in video games and animation films where one character is trying to catch another character as part of the game or story. In this world, a big fish is hungry for dinner. The big fish is going to chase after and catch a goldfish that, unaware that he is about to become a meal, is swimming in a random fashion. Figure 8-1-1 shows a very simple initial world. Our task is to animate the big fish chasing the goldfish, trying to get close enough (within 1 meter) to gobble the goldfish down for dinner. Naturally, as the big fish chases the goldfish, the goldfish is not standing still. Instead, the goldfish is moving away in a random direction.

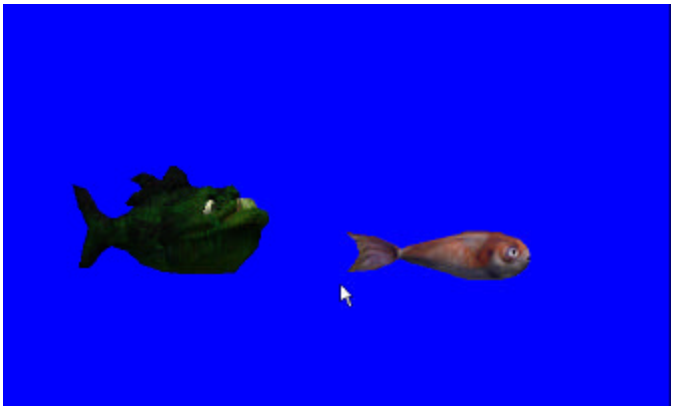


Figure 8-1-1. Initial Scene

Unknown number of repetitions

Before the program for this world can be developed, we should think about the problems involved. The goldfish is moving at the same time the big fish is moving. Let's assume that the big fish moves toward the goldfish, the same distance each time it moves. One problem is that the goldfish is swimming to a random position nearby. This means we cannot know how many times to repeat the swimming actions of the two fish. If the goldfish moves a long distance away in a move, it will take more repetitions for the big fish to catch it. But, if the goldfish moves a short distance away in a move, the big fish may catch up quickly. Clearly, the number of

repetitions is unknown. The second problem is how a goldfish can be made to move to a random position.
Now that we know what problems have to be solved, a storyboard can be created to list the steps to animate the big fish chasing the goldfish. A possible storyboard is:

Do in order
    point the big fish at the goldfish
    Do together
        move the big fish towards the goldfish
        randomly move the goldfish to a new position.

The above storyboard points the big fish at the goldfish and then the two fish move at the same time. The difference between the two fish movements is that the big fish moves towards the goldfish a given distance, but the goldfish moves to a new random position.

Of course, this is a chase – which means that the above actions need be repeated until the big fish gets close enough to the goldfish to eat it, thus ending the chase. But, before we worry about repeating the instructions, let's implement this one set of instructions as a method. A world-level method named *World.chase* will be written. (A world-level method is appropriate, since more than one object is involved in the action.). We will use a *Do together* block nested within a *Do in order* block, as shown in Figure 8-1-2. The *bigfish.swim* method is illustrated in Figure 8-1-3.

Figure 8-1-2. *Do together* nested within *Do in order*

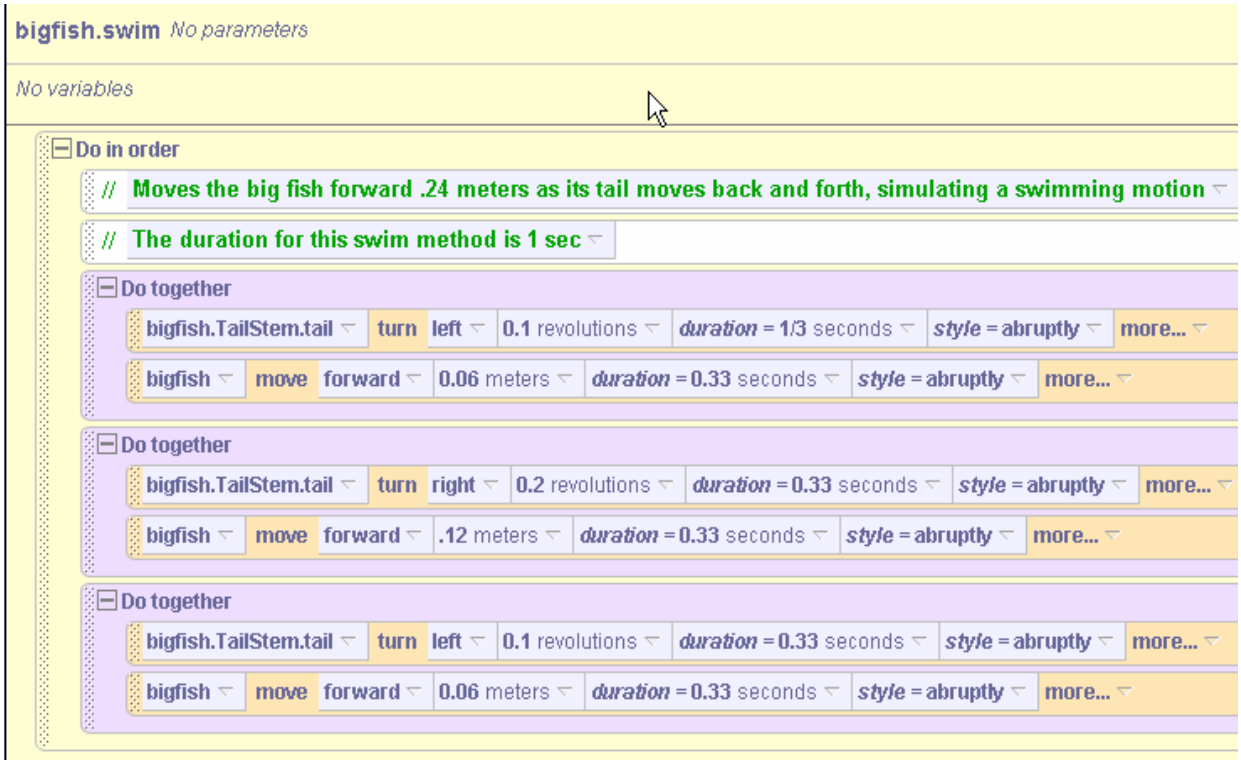


Figure 8-1-3. The *bigfish.swim* method

Random motion

The real challenge is to get the goldfish to move to a random location. The reason this is a challenge is the *move* instruction we have been using in our animations is for moving an object forward, back, up, down, left, or right a specific amount relative to where it is right now. It does not provide a way to say "move this object to this specific location in the world."

But, every object in Alice also has a *move to* instruction that can be used to move to a specific location. In the Tips & Techniques 4 section, we looked at an example of the *move to* instruction where we moved a basketball to the location of the rim in a basketball hoop. However, in this example, the goldfish should move to a random position (but fairly close to its current position), rather than to another object's position.

Using *move to* for random motion

We can use the *move to* instruction with random motion by following a few simple steps. First, issue a *move to* instruction for the goldfish, accepting the default Vector3 position.

Then, drag the world-level question "*right, up, forward*" dragged onto the "*Vector(0,0,0)*", as in Figure 8-1-4. This question will give us a location template where we can specify the right, up, and forward movements to move the goldfish to a new location.

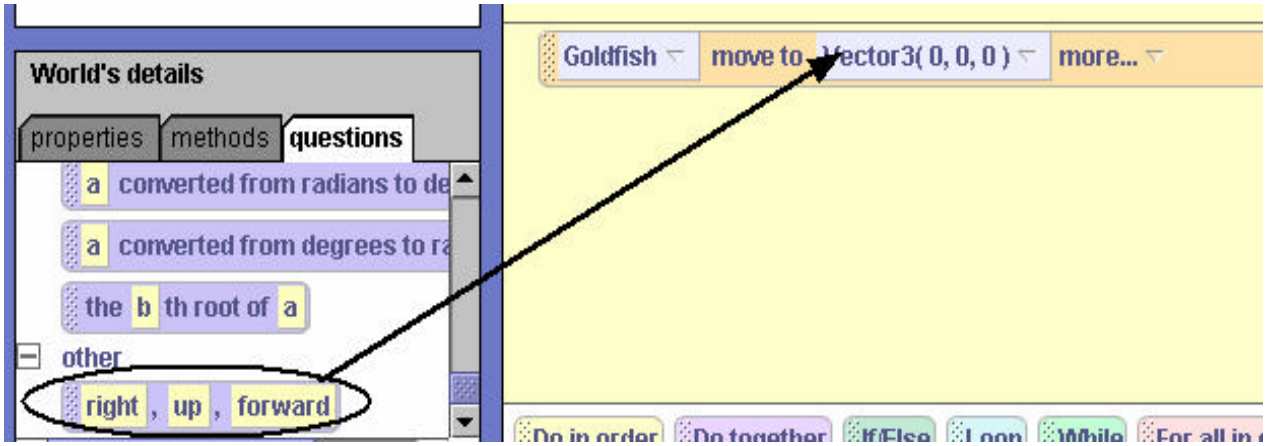


Figure 8-1-4. Drag in template for *right, up, forward* position

The result is shown below, having **dummy values** of 1 for the right, up, and forward positions:

Of course, we do not really want the goldfish to move to the location (1,1,1), we want it to move to a random location. To get random values for the (right, up, forward) coordinates, the world *random number* question is used, see Figure 8-1-5. By default, the *random number* question generates a random number between 0 and 1. But, optional parameters can be selected that allow numbers to be generated over a different range of values. We can drag the random number tile into the right, up, and forward tiles of the position template to get a random location.

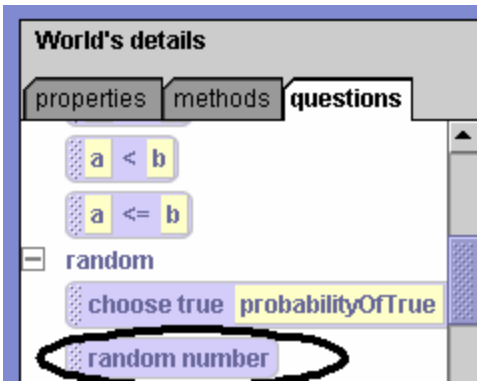


Figure 8-1-5. World random number question

But, what we really want to do is limit the random movement to a nearby location (so our animation looks more realistic). In other words, the goldfish's new location will be random – but will be adjusted to stay near to its current position. To keep the goldfish near its current position, we must find out what the goldfish's current position is. The built-in character-level question *goldfish's position* (see Figure 8-1-6) returns whichever coordinate of the goldfish's location (right/left, up/down, forward/back) is desired.

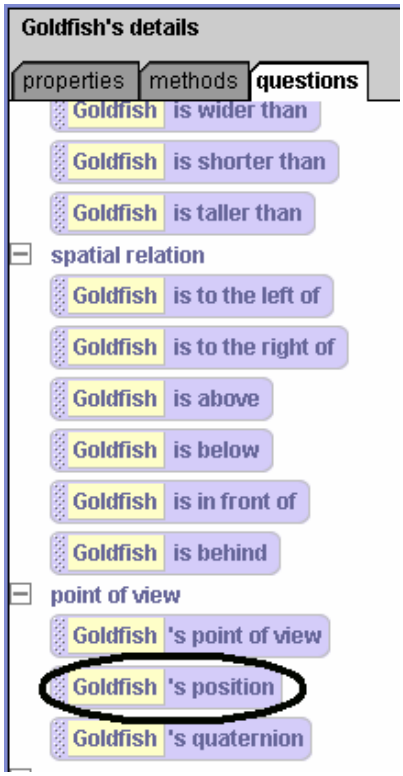


Figure 8-1-6. Finding the goldfish's current position

Now, we can put it all together to move the goldfish a small random amount from its current position. For convenience, we created a question, *adjust*, that will take a given coordinate as a parameter, and return that value adjusted by a small, random amount, arbitrarily between –1/5 and 1/5. See the adjust question in Figure 8-1-7.

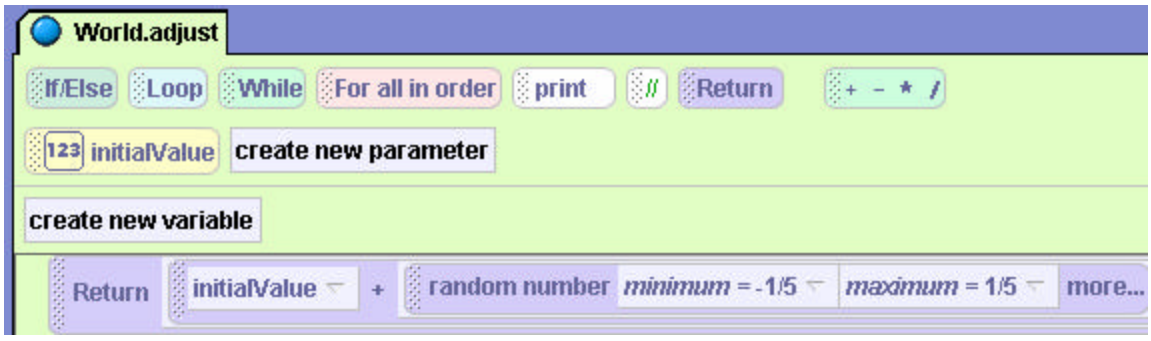


Figure 8-1-7. The world-level adjust question

Finally, the *move to* instruction is completed by invoking the adjust method, passing in the goldfish's coordinates as parameters. Figure 8-1-8 shows the complete *move to* instruction.

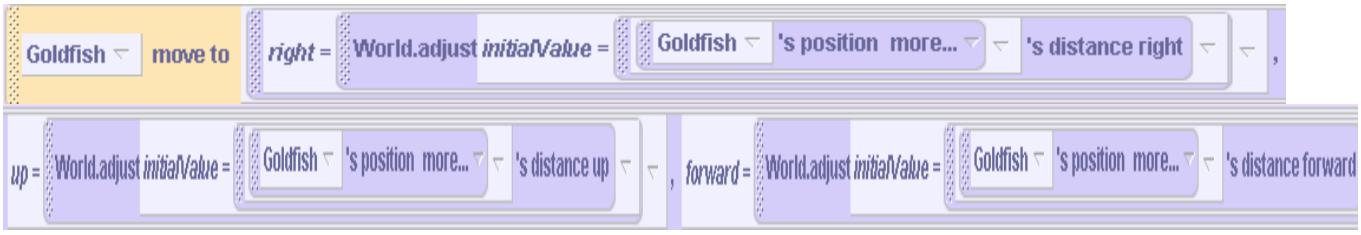


Figure 8-1-8. The complete "move to" instruction

So, let us put the pieces of the code together. The result should look like the chase method shown in Figure 8-1-9.

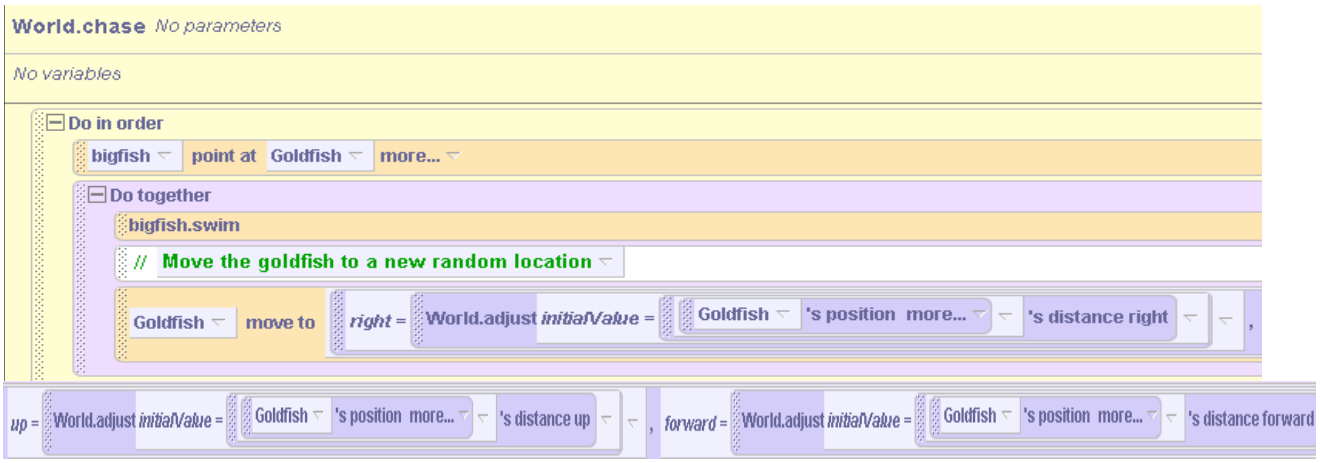


Figure 8-1-9. *World.chase* method

Recursion

What happens when the above code is run? The *chase* method executes once and then stops. Now is the time to think about repetition. What is needed is a way to allow for repetition of the method until the big fish gets close enough to catch the goldfish. The following storyboard expands the previous version:

If the big fish is too far away from the goldfish,
    Do in order
        point the big fish at the goldfish
        Do together
            move the big fish towards the goldfish
            randomly move the goldfish to a new position.
    Do everything again!
Else
    big fish eats goldfish

Three statements have been added to the original storyboard. First, a decision (represented as an *If* statement) must be made: is the big fish too far away from the goldfish? The second addition is the statement to "Do everything again." This means that the method is to be repeated by calling itself. **This is where we will use *recursion***. The third addition is the Else part of the IF…ELSE construct – the big fish eats the goldfish.

Clearly, an *If* statement is important for this example because it not only decides whether the instructions to move the two fish will be executed but also decides whether the method will be repeated. The conditional test that will be used is whether the big fish is "more than 1 meter from" the goldfish. The *If* statement is shown in Figure 8-1-10.

Figure 8-1-10. The if statement

If the condition is true, both the big fish and the goldfish are moved. After both objects have moved, the method will call itself. When the method is invoked again, the question is asked again: Is the big fish more than 1 meter from the goldfish? The statement to invoke the method again is shown in Figure 8-1-11.

Figure 8-1-11. Invoking the *chase* method

When the statements in Figure 8-1-10 and 8-1-11 are added to the chase method, the revised chase method code is shown in Figure 8-1-12.

An important concept to understand is that the recursive call to the method is repeated over and over as long as the condition for the *If* statement is *true*. But, when the condition for the *If* statement becomes *false* the *Else* kicks in and the recursive calls stop. In the *else-part* of the *if-else* statement in *World.chase*, a call to *bigFishEatGoldfish* spells the end of the chase. The *bigFishEatGoldfish* method is shown in Figure 8-1-13.

The form of recursion illustrated in this example is said to be *generative*. In *generative* recursion, a decision is made. Depending on the results of that decision, the method is either finished or will be executed again. If the method is executed again, the same decision is made once more. This repetition may go on and on until, eventually, the tested condition changes and the solution is finally complete. This kind of recursion is *generative*, because more executions are "generated" each time the result of the previous decision (in this case, "Is the big fish too far from the goldfish?") is true.

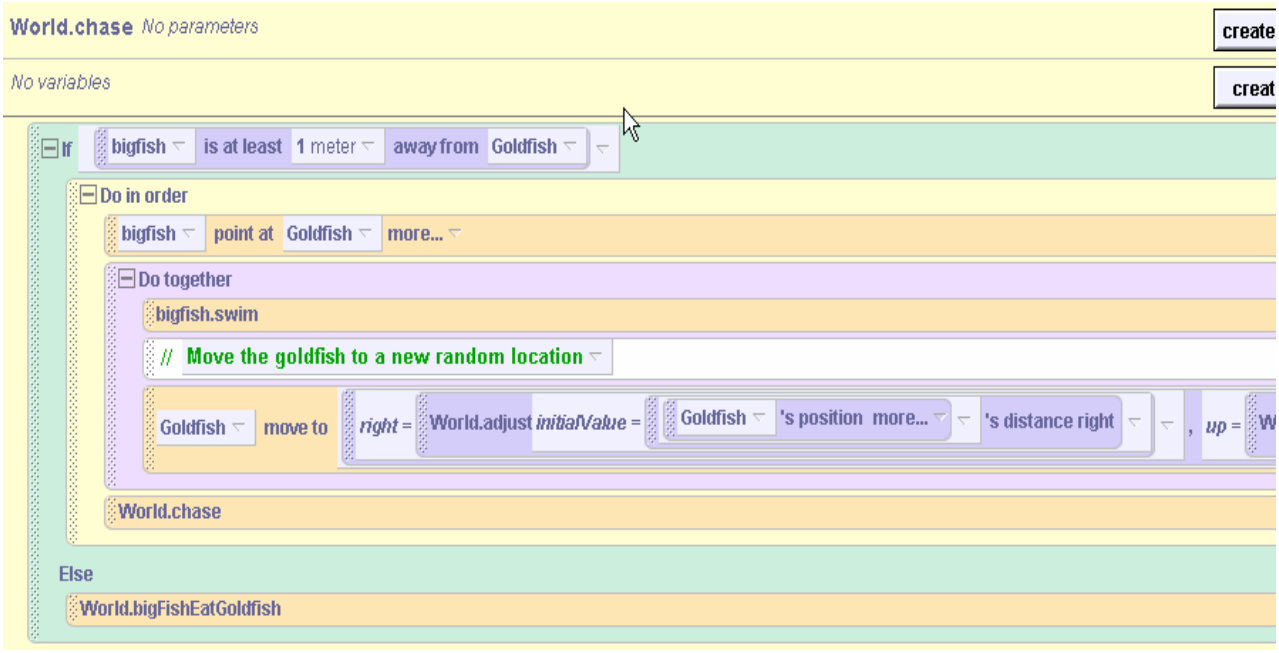


Figure 8-1-12. The complete chase method

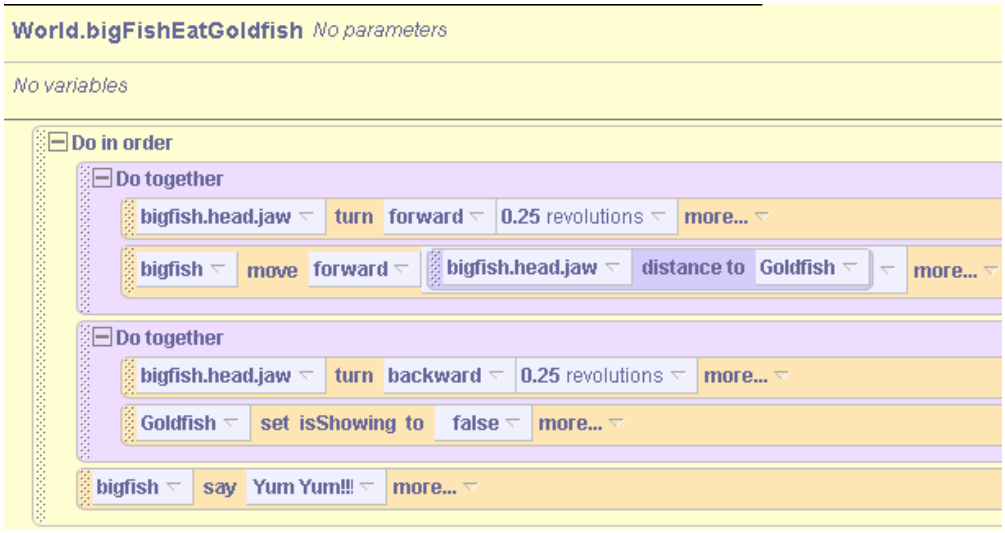


Figure 8-1-13. World.bigFishEatGoldfish

Not only is this an example of generative recursion, but it is also *tail recursive.* A method is said to be tail recursive when two conditions are met:

(1) **only one** recursive call appears in the method, and

(2) the recursive call is the **last** statement in the *if-part* or the *else-part* of an *if-else* statement.

In the code illustrated in Figure 8-1-12, there is only one call to *World.chase* and it is the last statement in the *if-part* of the *if-else* statement. So, the *chase* method is tail recursive.

8-1 Exercises

1. ButterflyChase

Consider the following initial scene and scenario:

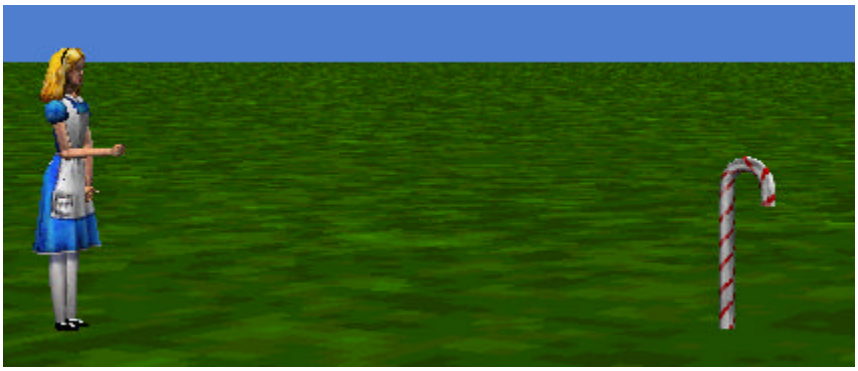
A White Rabbit is chasing after a butterfly. The world to build is similar to the chase scene from the text except in two important respects:

- The White Rabbit must always remain on the ground (rabbits do not fly). Use the "*only affect yaw=true*" option of the *point at* instruction.
- In order to prevent the butterfly from flying away (too high or too low), the butterfly's up-down coordinate should be set to a random value between 0 and 1 rather than a slight change from its current up-down value.

When the White Rabbit gets close enough to the butterfly, have him catch the butterfly in his net.

2. MidasTouch1

A magician has cast a spell on Alice. The effect of the magician's spell is to give Alice the Midas touch. Everything she touches turns to gold.



Alice is facing the candycane. (Use PointAt to make this alignment.) Write a recursive method, named *CheckCandy*, that checks whether Alice's right hand is very close to the candycane (within 1 meter). If it is, Alice bends over to touch the candycane. After Alice touches the candycane, the candycane turns to gold (color changes to yellow). If Alice is not yet close enough to the candycane to be able to touch it, Alice moves a small distance forward (one-tenth meter) and the CheckCandy method is recursively called.

3. **MidasTouch2**. Create a second version of the MidasTouch1 world. This new version will be interactive. The idea is allow the user to guide the movement of Alice towards a candy cane. To make this a bit more challenging, Alice should **NOT** be pointing towards the candycane in the initial scene. This will require the the *CheckCandy* method be modified so that whenever Alice is close enough to the candycane to touch it, she first turns to point towards the candycane before bending over to touch it.

Use the left and right arrow keys to turn Alice left or right. Create 2 methods: *TurnRight* and *TurnLeft*. The methods should turn Alice 0.05 meters to the right or left when the user presses the right-arrow key or the left-arrow key. These methods will allow the user to guide Alice towards a candycane.

Hint: It is possible that Alice will wander out of the range of the camera. There are two possible solutions to this problem. One is to make the camera point at Alice each time she moves. The other is to make the camera's vehicle be Alice!

8-2 Structural Recursion

In the previous section of this chapter, *generative* recursion was introduced. In this section, we examine a second kind of recursion, *structural recursion*. Our goal in this section is to illustrate structural recursion and some of the differences between generative and structural recursion.

A closer look at generative recursion

In *generative* recursion, a decision is made. Depending on the results of that decision, the method is either finished or will be executed again. If the method is executed again, the same decision needs to be considered once more to determine whether another repetition will occur. This repetition may go on and on until, eventually, the tested condition changes and the solution is finally complete. In the example presented in section 8-1, the big fish is chasing the goldfish and if the big fish is more than 1 meter from the goldfish, both of the big fish and the goldfish are moved. After both objects have moved, the question is asked again: Is the big fish more than 1 meter from the goldfish? Such recursion is called generative because another execution may be generated each time the result of the decision is true. We do not know whether executing the method one more time will produce an acceptable solution to the problem (getting the big fish within 1 meter of the goldfish) until the method has run and the results can be tested again.

While generative recursion is fairly useful and easy to implement, it has one serious flaw: **It is often very difficult to determine that the program will ever end.** How can we be sure that the big fish will ever catch the goldfish? What if, by some random chance, the goldfish were to constantly move far enough away from the big fish that the big fish will never catch up. (Actually, we expect in random behavior that sometimes the goldfish will move towards the big fish, rather than away.) To show that the big fish will catch the goldfish eventually, it would be necessary to argue that the total amount of distance the big fish moves (in each step) is greater than the total distance traveled by the goldfish in each step. Do not worry if the last statement does not make too much sense. Running the program several times, and seeing that each time the program runs, the big fish does eventually catch the goldfish, is enough to provide a good sense that the program does actually work.

Structural recursion

A second form of recursion is known as *structural* recursion. Structural recursion depends on an ability to break a problem down into smaller and smaller sub-problems. Once the smaller sub-problems are solved, the solutions to the smaller sub-problems are used to cooperatively solve the larger problem. Many mathematicians (and computer scientists interested in logic) often prefer this form of recursion because it is easier to show that the program does end – and that it ends with the correct solution. Rather than discussing structural recursion in theory, it is perhaps best to look at an example.

Towers of Hanoi puzzle

The problem to be considered is the Towers of Hanoi, as illustrated in Figure 8-2-1. The Towers of Hanoi is a legendary puzzle. In this puzzle, disks of varying widths have been placed on a tower. To solve the puzzle, all the disks must be moved from one tower to one of the other towers, following certain strict rules.

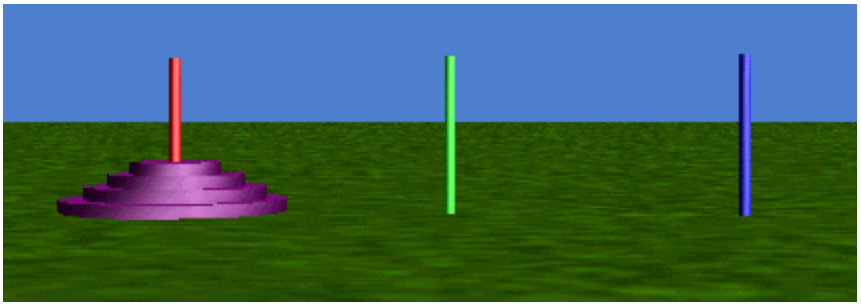


Figure 8-2-1. The Towers of Hanoi

In the world shown in Figure 8-2-1, four disks on the leftmost tower. Each tower is exactly 5 meters from its nearest neighbor. And, each disk is exactly 0.2 meters in height. To make it easier to describe this puzzle and our solution to the puzzle, let's number the disks 1, 2, 3, 4 …in order of increasing size. The smallest disk is disk 1 at the top and the largest disk is disk 4 at the bottom of the stack. Also, let's name the towers "one, two, three" left to right from the camera point of view. Our goal is to move all of the disks from the leftmost tower, tower one, to another tower. In this example, we will move the disks to the rightmost tower (tower 3).

In this puzzle, the rules that must be strictly followed are:

1) Only one disk may be moved at a time.
2) A larger disk may never be placed on top of a smaller disk.

So, for example, the following attempts to solve the problem are illegal:

1) Simultaneously move disks 1 and 2 from tower one to tower two. (This move violates rule #1.)
2) Move disk 1 (the smallest disk) from tower one to tower two. Then move disk 2 from tower one to tower two. (This would result in disk 2 being on top of disk 1, thus violating rule #2.)

In the ancient story about this puzzle, there were 64 disks on the Towers in Hanoi. Solving the puzzle with 64 disks would be a huge task and would take much too long to run! (In fact, assuming that it takes 1 second to move a disk from one tower to another, it would take many centuries to run!) However, we can use just 4 disks to illustrate a solution to the puzzle that uses structural recursion. Most people can solve the 4-disk puzzle in just a few minutes, so this will allow a quick check for a correct solution.

Two requirements

To solve this puzzle for 4 disks using structural recursion, two requirements must be met. **The first requirement is that we must assume we know how to solve the problem for a smaller sub-problem.** Well, let's assume that we do know a solution for solving the problem for 3 disks. If we know how to solve the problem of moving 3 disks, it would be quite easy to write a program to solve it for 1 more disk (4 disks). The following steps would work:

1) Move the 3 disks (imagining the solution for the puzzle with only 3 disks is already known) from tower one to tower two. See Figure 8-2-2(a).
2) Move the last disk, disk 4, from tower one to tower three. See Figure 8-2-2(b). (Remember this move is now safe, as all of the 3 smaller disks are now located on tower two.)
3) Move the 3 towers (again, imagining the solution is already known) from tower two to tower three.

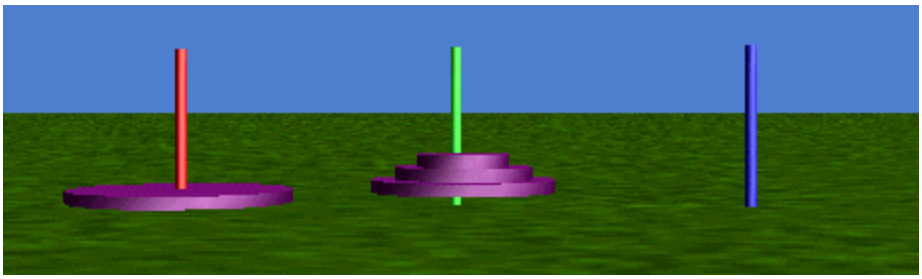


Figure 8-2-2(a). After step 1

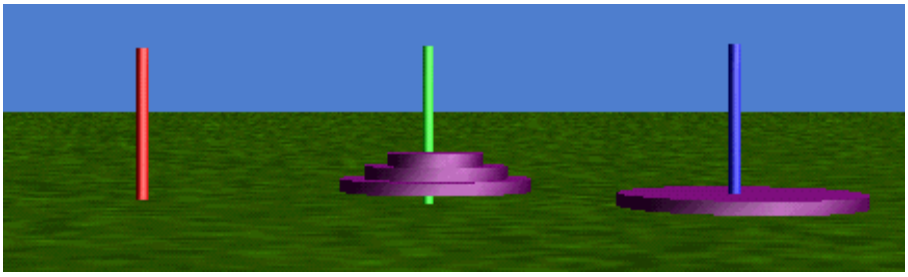


Figure 8-2-2(b). After step 2

The second requirement for a structural recursive solution is that we must have a "base case." A "base case" is the simplest possible situation where the solution is unquestionably obvious. Since the solution to the base case is obvious, the base case does not require a solution to a smaller sub-problem. The obvious "base case" in the Towers of Hanoi puzzle is the situation where we have only 1 disk. In order to move 1 disk, we can just move it, as there are no smaller disks than disk 1, it can always be moved from whichever tower to whichever tower without question!

In summary, two requirements must be met when using structural recursion for solving a problem:

1) Use the solution to a smaller sub-problem to help solve the larger problem.
2) Have a base case, which does not require a solution to a smaller sub-problem.

Towers Method

Putting these two ideas together, we can now write a method, named *Towers*, to animate a solution to the Towers of Hanoi puzzle. The method will have instructions to move some number

(**n**) of disks from a **source tower** (the tower where the disks are currently located) to a **target tower** (the tower where the disks will be located when done). In the process of moving the disks from source tower to target tower, a **spare tower** (the tower that is neither source nor target) will be used as a temporary holder for towers on their journey from source to target.

What information has to sent to parameters in the *Towers* method so it can do its work? Well, the *number of disks to be moved* must be known along with *which tower is the source*, *which tower is the target*, and *which tower is the spare*. To provide this information to the *Towers* method, four parameters will be needed: **howmany, source, target**, and **spare**. Here is the storyboard:

```
Towers (howmany, source, target, spare)
        if howmany equals 1
              move the disk (the smallest one) from the source to the target.
        otherwise
              Recursively call Towers() to move howmany-1 disks from source to
                  spare (using target as spare).
              Move disk # howmany from the source to the target.
                      (This is ok because all smaller disks are now located on
                      the spare peg, after step 1).)
              Recursively call Towers() to move howmany-1 disks from the spare to
                      the target (using the source as the spare)
```

We know this looks kind of complicated; but hang in there -- it really is not too difficult. All we are saying is: (1) Move all but one of the disks from the source to the spare tower. (2) Now that only one disk is left on the source, move it to the target tower. (3) Now, move all the disks that have been temporarily stored on the spare tower to the target tower. Based on this storyboard, the code for the Towers method is shown in figure 8-2-3.

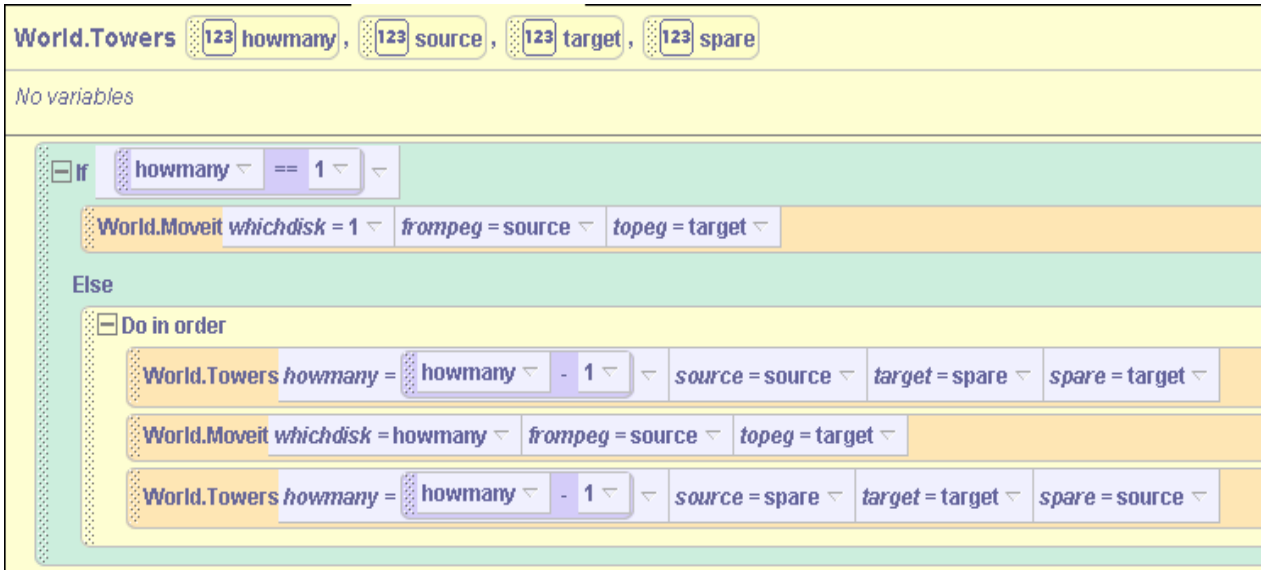


Figure 8-2-3. The Towers method

Special *Moveit* Method

No doubt as you looked at the code in Figure 8-2-3, you realized that we called a method named *Moveit*. (In fact, we called the *Moveit* method a couple of times.) So, exactly what does the *Moveit* method do? Well, *Moveit* is a special method we wrote to move a specific disk from one tower to another tower. The *Moveit* method needs 3 parameters because it needs to know: which disk is to be moved, the source tower, and the target tower. A storyboard for *Moveit* could be:

```
Do in order
       Lift the disk up above the top of the source peg
       Move it (forward or back) to a location above the target peg.
       Lower the disk down onto the target peg.
```

This sounds easy enough. But, how high should the disk be lifted? Each disk is at a different initial height on the tower. So, it will be necessary to raise each disk a different amount. (Note that disk1 has been placed on top, disk2 immediately below it, disk3 immediately below disk2, and disk4 immediately below disk3.) Also, in our example world, we made each disk be 0.2 meters in height. Then, we determined the height to lift each disk by trial and error. In the sample world, it was necessary to lift the first disk approximately 2.0 meters, disk2 approximately 2.2 meters, disk3 approximately 2.4 meters, and disk4 approximately 2.6 meters (to "clear" the tower).

Once the lift-height is determined for each disk, then a plan can be created for carrying out the lift to the appropriate height for the particular disk. One possibility is to pass a parameter (to the *Moveit* method) that contains the id number of the disk to be moved. If the disk id is 1, move Disk1, if the disk id is 2, move Disk2, and so on. Then, a cascading *If* statement can be used to check on the id of the disk passed in and lift the appropriate disk the appropriate amount. The storyboard (for the *If* statement) would look something like this:

```
If whichdisk is 1 then
   Move Disk1 Up 2.0 meters
Else
   If whichdisk is 2 then
      Move Disk2 Up 2.2 meters
   Else
      If whichdisk is 3 then
            Move Disk3 Up 2.4 meters
      Else
            Move Disk4 Up 2.6 meters
```

We thought about this for a while, realizing that our code is a bit awkward. Then, we came up with a more clever approach using a nifty mathematical expression. Note that the amount any disk should move Up is **1.8 + 0.2 * whichdisk**. So, it would seem that the above storyboard could be condensed into just one step:

```
Move the appropriate disk Up 1.8 + 0.2 * whichdisk
```

Conversion Question

One question still needs to be answered: "How is Alice to be told the name of the disk object to be moved when only the disk id number is known?" The problem is there is no easy way to convert the disk *id* number to the *disk-object* name. What is needed is a conversion method that takes the id as a parameter and then returns the appropriate name of the disk object to move. Such a conversion method can be written in Alice using a question. The *which* question is illustrated in Figure 8-2-4. Note that this question returns an object – namely which disk should be moved.

In this example, each *If* statement includes an instruction containing the keyword `return`. The return statement means that the function will be sending information back to the method that called it. For instance, suppose "*which(i = 2)*" is called, the information that will be returned is "*disk2*". So, the *which* question provides a way to convert from an id number (the *whichdisk* parameter) to an object name so the *Moveit* method will know which object is to be moved!

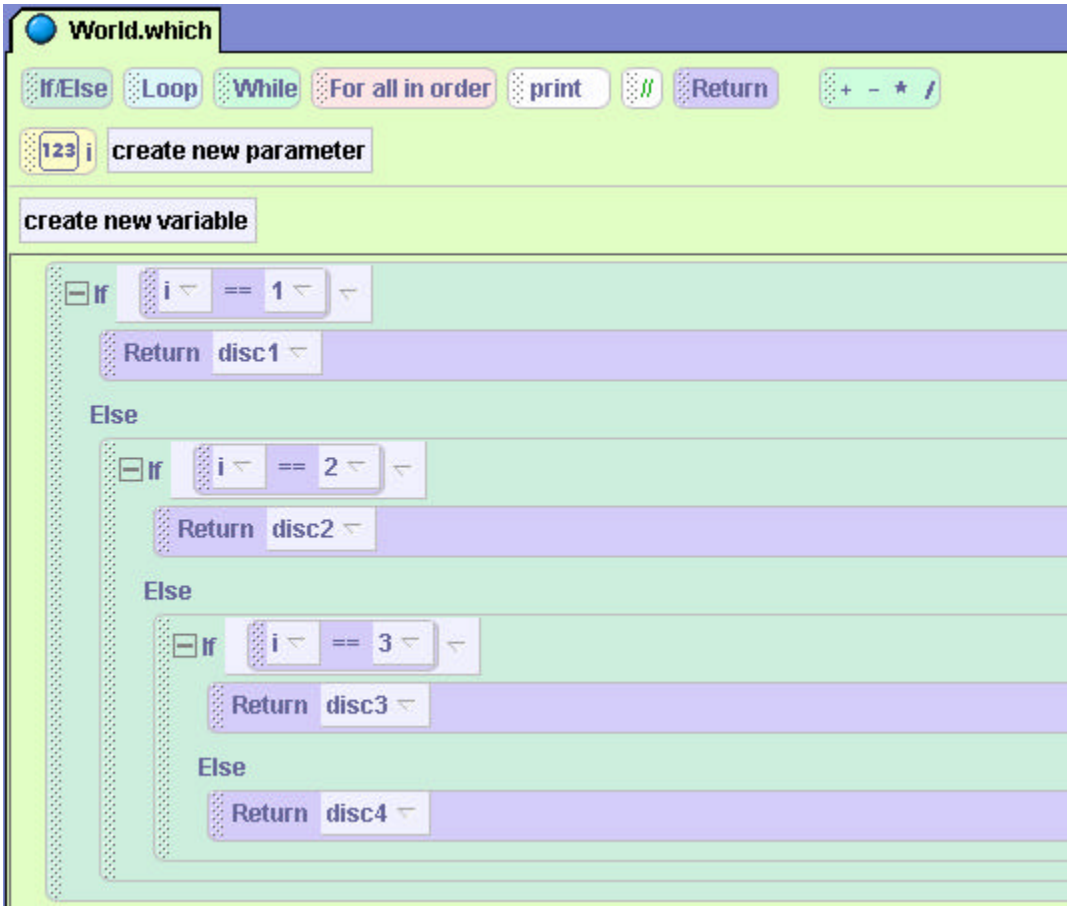


Figure 8-2-4. A World-level Conversion Question

An instruction is now written in the *Moveit* method that calls the *which* question to determine which disk to move, as illustrated in Figure 8-2-5.

Figure 8-2-5. Invoking the question to determine which disk to move

In this instruction, *which(i = id)* is a **call** to the *which* function. When executed, *which(i = id)* will select the appropriate disk and return the name of the object that will move it the appropriate amount, as explained in the earlier expression.

The next piece of information we need to know is the distance (forward or back) to move the disk (in order to move it over to the destination tower). The towers are purposely positioned exactly 5.0 meters from one another. So, one possibility is to have several if statements, covering all possible cases of source and destination towers – perhaps something like:

```
if the Source peg is 1
   if the target peg is 2
       move the appropriate disk 5 meters forward
   else
       move the appropriate disk 10 meters forward
else
   if the Source peg is 2
       if the target peg is 3
          move the appropriate disk 5 meters forward
       else
          move the appropriate disk 5 meters back
   else (the source peg is 3)
       if the target peg is 1
          move the appropriate disk 10 meters back
       else
          move the appropriate disk 5 meters back
```

While this storyboard is correct, the nested if – else structures make it quite confusing. Once again, we gave some thought to what we might do to come up with an easier way to figure out the distance to move forward or back. With a bit of thought, you can see that the forward amount to move is: (*totower – fromtower*) * 5.0. (5 is the distance between two adjacent towers) Also, notice that moving forward –5.0 meters is the same as moving back 5.0 meters. With this insight, the instruction in Figure 8-2-6 can be written:

Figure 8-2-6. Moving the disk the appropriate amount forward/back

The last step in the *Moveit* action is to move the disk back down onto the tower. This instruction should simply do the opposite of what was done in step 1. The complete *Moveit* method appears in Figure 8-2-7.

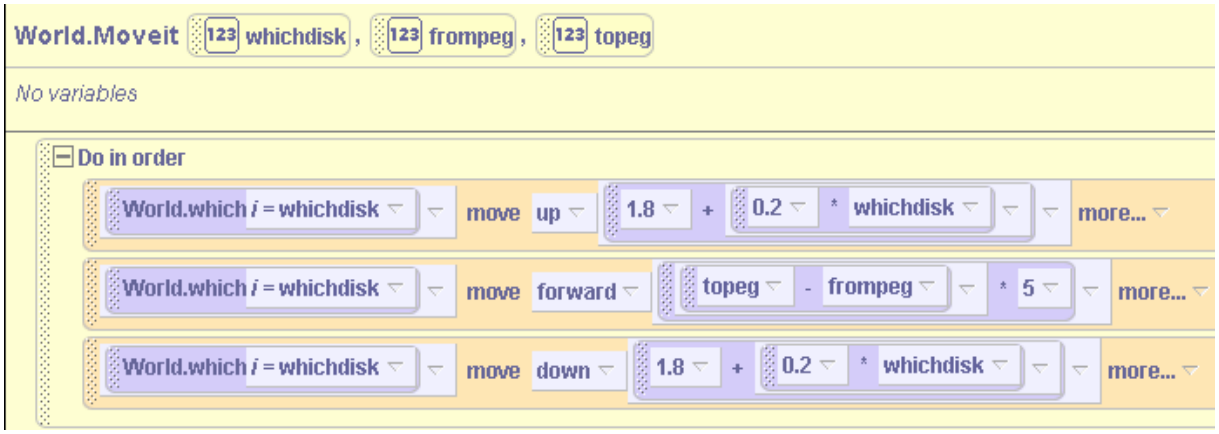


Figure 8-2-7. The code for *Moveit*

The *Towers* method was presented earlier in Figure 8-2-3. Now, with the *Moveit* method completed, all that remains is to call the *Towers* method when the world starts. The following link in the Events Editor will work:

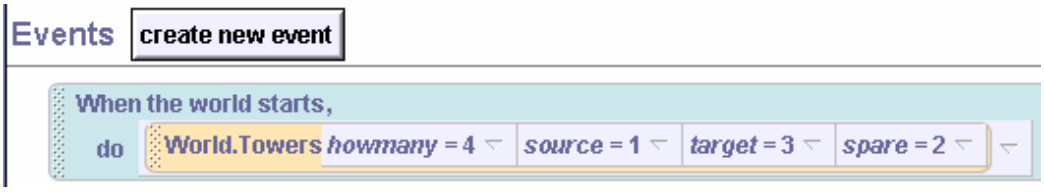


Figure 8-2-8. Invoking the Towers method

Exercise:

1) Create the Towers of Hanoi puzzle as described in the reading for this section. When you have it working, modify the event-trigger link that calls the Towers method so that it moves the disks from tower 1 to tower 2 (instead of tower 3).

8 Summary

In this chapter, the concept of using recursion as a mechanism for repetition was introduced. Recursion is a powerful tool for building more complex and interesting worlds. Throughout the rest of the book, example worlds will be significantly richer and more interesting than they have been in the past!

A chase scene example was used to demonstrate how to combine a *move to* instruction with the world-level *random number* question to enable *random motion*. Moving objects randomly makes it easier to build many different kinds of fun worlds. And, random motion makes it easier to understand how recursion works and why we want to use it. The chase scene example in this chapter demonstrates a kind of recursion said to be *generative*, because more executions are "generated" each time the result of the previous decision is true.

The famous Towers of Hanoi puzzle provided an example of the use of *structural* recursion, allowing us to compare generative recursion to structural recursion.

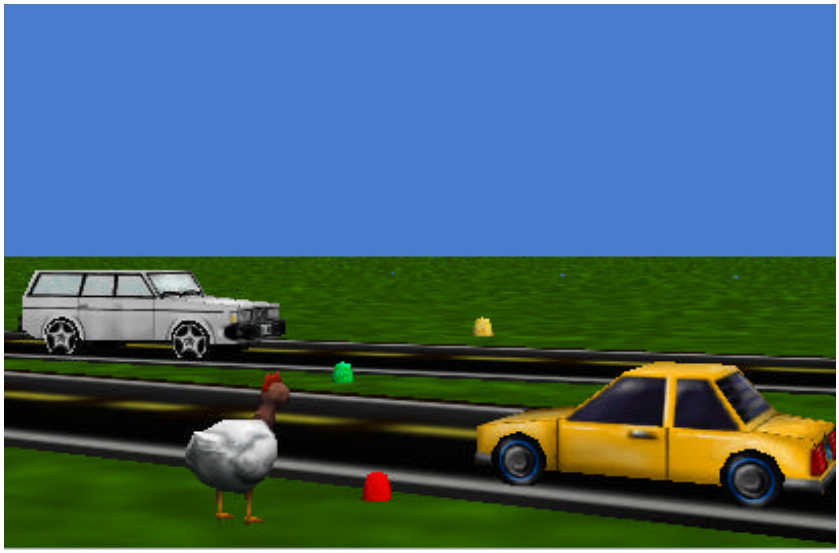
Important concepts in this chapter

- Recursion is most useful when we cannot use a Loop instruction because we do not know (at the time we are writing the program) how many loop iterations will be needed.
- Recursion is when a method calls itself.
- Recursion enables a method to be repeatedly invoked.
- Any recursive method must have at least one *base case*, where no recursive call is made. When the base case occurs, the recursion stops.
- Recursion may be *generative* or *structural.*

8 Projects

1. Why did the chicken cross the road?

A popular child's riddle is "Why did the chicken cross the road?" Of course, there are many answers to this riddle. In this project, the chicken (who has a real sweet-tooth) crosses the road to eat the gumdrops along the way. Write a game animation where the player guides the chicken across the road to get to the gumdrops. Cars and other vehicles should move in both directions as the chicken tries to cross the road to where the gumdrops are located. Use arrow keys to make the chicken jump left, right, forward and back. Use the space bar to have the chicken peck at the gum drop. When the gum drop is pecked, it should disappear.



A recursive method is used to control the play of the game. If the chicken gets hit by a vehicle, the game is over (squish!). The game continues as long as the chicken has not managed to peck all the gumdrops and the chicken has not yet been squished by a vehicle. If the chicken manages to cross the road and peck at all the gumdrops along the way, the player wins the game. Signal the player's success by making 3D text "You Win" appear or by playing some triumphant sound.

Extra: Create three buttons that allow the player select the speed at which the vehicles move across the screen (slow, medium, and fast). The buttons should be visible at the beginning of the game. Once the player clicks on one of the buttons, the buttons disappear and the game begins.

2. Reversal

In the world below, the row of skeletons (Graveyard folder on the web gallery) are guarding the gate. Every so often in this world, the row of skeletons is to reverse order. This project is to animate the skeleton chain reversal using structural recursion. The storyboard goes something like the following:

Reverse method:
    If the row of Skeletons is not yet reversed is more than one then
        Reverse the row of Skeletons starting with the 2nd Skeleton (by
                recursively calling Reverse
        Move the Head Skeleton to the end of the list

The base case is when there is just one Skeleton in the row (that has not yet been reversed). Of course, a row of 1 Skeleton is already reversed!

The recursive case (for n Skeletons, where n is larger than 1) says to first reverse the last n-1 Skeletons, and then move the first Skeleton to the end of the list.

Implement the Skeleton reversal storyboard given above. The program you will write should be quite similar to the Towers of Hanoi program, including the *which* method.

3. MidasTouchGame. An exercise in section 8-1 was named MidasTouch2. If you have already completed this exercise, begin with that world and modify it for this project. Otherwise, begin by creating the world for the MidasTouch2 exercise and then modify the animation for this project. In this project, convert the interactive MidasTouch world to work as a primitive kind of game. Add several candy canes to the scene. To win the game, the user has to guide Alice around the screen to touch each candy cane. When all have been touched and turned to gold, the game is over. Once a candy cane has turned to gold, it should become inactive. That is, Alice should not bend over to touch it again – even if she gets close enough to do so. The CheckCandy method should be modified to accept a parameter that specifies which candy cane is to be checked.