9 Repetition: While Statements

While statements are a program construct found in most programming languages. In this chapter, we will take a close look at the use of while statements as a mechanism for repetition. As with loops and recursion, you will find that the while statement is a powerful tool for making segments of code in your programs execute over and over again.

The while program construct corresponds naturally to the way we say and think about certain actions in our everyday conversation. For example, we might say, “While I am driving, I will continue to keep my hands on the wheel and my eyes on the road.” Or, perhaps something like “While I am a member of the baseball team, I will practice batting.” The “while” construct is intuitive, as away to think about actions that repeat while some condition is true.

Another way to think about a while statement is as an alternative to tail recursion. In chapter 8, we learned that tail-recursive methods are those methods that contain only one recursive call (in either the Then part or the Else part of an If-Else statement). And, in tail recursion the recursive call is the last line in the Then part (or the Else part). Any repetitive action that can be written using tail-recursive methods can be written using a while statement instead. For this reason, computer scientists refer to the while statement as a form of syntactic sugar – designed to make writing code a bit easier for programmers.

Section 9-1 introduces the while statement. The chase animation (written tail-recursively in section 8-1) is re-written using a while statement instead of recursion. A second example is presented as well. In section 9-2, we look at infinite loops, where this is a desired behavior.

Notes to the instructor:
1) The example in section 9-1 uses a while loop as an alternative to tail-recursion. The Towers of Hanoi project in the projects section of this chapter can be used to illustrate a broader use of while loops as an alternative to recursion.

2) Our examples in chapters 8 and 9 that use recursion and while statements do so with the code for repetition within methods, and not within questions (even though it is possible to have recursion in questions, as well). Our approach in this book is to use visualization to make programming concepts easier to teach and learn. But, a question is not visible. That is, a question returns a value but does not actually do any animation. So, in a question, the actual repetition is not visible and the student does not see the repetition in action. Thus, we do not see added value in trying to demonstrate the concept of repetition in examples where the repetition occurs in questions.
9- 1 While

Two powerful techniques for repeating blocks of instructions have been presented: the Loop construct for repeating a block of instructions a counted number of times and recursion to repeat instructions when the count was not known. This chapter presents another repetition construct, the while statement.

The while statement is a conditional loop. The condition of a while loop is the same kind of Boolean condition as used in if-else statements. The Boolean condition acts like a gatekeeper at a popular dance club. If the condition is true, entry is gained to the loop and the instructions within the loop are executed, otherwise the loop is skipped. But, the while statement is a loop construct. This means that if the condition is true and the instructions are executed, then (after the instructions are executed) the condition gets checked again. If the condition is still true, the instructions within the loop get repeated. But, if the condition has become false, the loop ends and Alice goes on to execute the next statement in the program.

Revisiting the Chase

As an example of a while loop, let’s borrow the chase animation from chapter 8, seen in Figure 9-1-1. The key concept in the chase method was repetition implemented as a recursive call to chase. As you will recall, the chase program is an example of tail recursion because there is only one recursive call and the call is the last statement in the if-part of the if-else statement. Code that can be written in a tail-recursive style can alternatively be written using a while loop.
In the chase scene, the big fish is hungry and is swimming after the goldfish. If the big fish is more than 1 meter from the goldfish, the big fish moves towards the goldfish a given distance. At the same time the big fish is moving toward the goldfish, the goldfish moves to a random position nearby. (That’s what a chase scene in an action movie is all about – one individual chasing and another fleeing.) If the big fish is still more than a meter from the goldfish we do it all again and again. Eventually, when the big fish finally gets within a meter of the goldfish, the big fish eats the goldfish.

Using while
Let’s plan a method for the chase animation using the while statement. Think about the situation like this:

"WHILE the big fish is more than 1 meter away from the goldfish, move the big fish toward the goldfish and, at the same time, move the goldfish to a random position nearby"

The condition in our while statement will be “the big fish is more than 1 meter away from the goldfish.” If this condition is true, the chase will ensue. A storyboard is shown below.

<table>
<thead>
<tr>
<th>While the big fish is more than 1 meter from the goldfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do in order</td>
</tr>
<tr>
<td>Point the big fish at the goldfish</td>
</tr>
<tr>
<td>Do together</td>
</tr>
<tr>
<td>Move the goldfish randomly</td>
</tr>
<tr>
<td>Move the big fish towards the goldfish</td>
</tr>
<tr>
<td>The big fish eats the goldfish</td>
</tr>
</tbody>
</table>

The idea of a while loop is that the condition must be true to allow entry into the loop. After running the instructions within the while statement, the condition will be evaluated again, and if the condition is still true, the instructions within the while statement will run again. This process continues until the condition finally evaluates to false. A chase method to implement the storyboard is illustrated in Figure 9-1-2.
This version of the *chase* method is almost exactly the same as the recursive *chase* method presented in section 8-1. The only difference is that the *while* loop statement replaces the *If* statement and the recursive call is not needed. Some programmers prefer using recursion and others prefer using *while*. In some other programming languages the *while* statement runs faster than recursion and so a while loop is preferred. However, in Alice both approaches run equally fast, so it is a matter of choice as to which approach you wish to use.

**A game-like example**

This example is a simple version of a horse race in a carnival game. In the initial scene, shown in Figure 9-1-3, the finish line was created by adding a kayak paddle to the scene and positioning it flat on the ground. (Other objects could be used to simulate a finish line, of course.) Then, 3 colored markers (we used gumdrops) were added for the target location where each horse should run to finish the race. Unlike a traditional horse race, where horses run around an oval-shaped track and each horse breaks to the inside of the track, in a carnival game horses move straight ahead in a mechanical track and you can win a prize if you pick the right horse. We changed the color of each horse’s midsection to match the color of the marker (gumdrop) to which it will run. So, the horse with a red midsection will race to the red marker, the horse with blue midsection to the blue marker, and the same for the yellow marker.

There are at least two choices as to how to run the race. One choice is to have all the horses run continually, perhaps different random amounts, until one of the horses reaches the finish line. (This is left as a lab exercise.) The approach we show here is to generate a random number and then, depending on the value of that random number, move one of the horses forward. An essential step is how to decide when the game will end. Well, as soon as one of the horses reaches the finish line the game is over.
While all of the horses have not yet reached the finish line, the horses should move forward toward the finish line. A while loop can be used to implement the race. To write the code for the race, we have two problems:

1) How to determine when the race is over.
2) How to randomly select one of the horses to move forward for this iteration of the while loop.

**How to determine when the race is over**

The race is over when one of the horses gets close enough to its marker on the finish line to be declared a winner. The built-in questions in Alice offer a number of possible choices ways to find out how close a horse is to its marker. In this example, let’s use the *at least threshold away from* question, where threshold is some minimal distance between the horse and its marker on the finish line. *(In our world, we found the winning distance between horse and finish line to be 1.1 meters. But, you will have to experiment in your world to find the best distance.)*

Writing a Boolean condition for a while statement is similar to writing a condition for an if-else statement. A condition is needed to determine whether the loop will execute and continue executing. In this example, it is not enough just to test “Horse is at least 1.1 meters from gumdrop” and stop as soon as it becomes less than 1.1. This is because we have three different objects. So, we must test the condition for all three horses and stop the race as soon as any one horse gets within 1.1 meters. Clearly, the test condition must have three parts combined by using the world-level logical operator *and*. *(We used a similar condition with the if-else statement in the Zeus world in section 6-1.)* The condition for the horse race is shown in Figure 9-1-4:

![Figure 9-1-4. A multiple condition to end the race](image-url)
Random selection

Now, we can tackle the task of deciding which horse moves next. This is where random selection is needed. In section 8-1 and other examples in previous chapters, we have used the world-level random number question. But, in this problem we need to randomly select a horse to move forward in such a way as to give each horse a “fighting chance” to win the race. (After all, we don’t want to “fix” the race.) So, instead of using the random number question, let’s use the world-level question “choose true n of the time.”

It seems reasonable to use a value of $1/3$ for $n$, so that each horse will be chosen $1/3$ of the time. The other $2/3$ of the time, that horse will not move. As shown in Figure 9-1-5, we start the random selection with the first horse, whose name is Horse. Horse will be selected $1/3$ of the time. But, what about when Horse is not selected? When Horse is not selected, the Else-part takes over and a selection must be made from Horse2 or Horse3. Once again, to decide which of these horses should move, “choose true n of the time” is used. But, this time a value of $1/2$ is entered for $n$ because we want the two horses to each have an equal chance to be selected.

![Figure 9-1-5. Random selection of a horse](image)

Now that we have figured out how to write the Boolean condition for the while statement and how to randomly select one of the horses to move forward each iteration, we can put all of the code together, as seen in Figure 9-1-6.
At the end of the program shown in Figure 9-1-6 (after the while loop) is an *If*-else statement to determine which horse has won the race. The winner simply moves up 0.50 meters into the air. This is not a very realistic end to a race, but at least it is easy to see which horse won! Because of the random selection used in this program, different horses should win the race if the program is run several times.

Figure 9-1-6. Horse Race Animation
9-1 Exercises

1. Horse Race 1
Create and modify the horserace world in this section. The modified horse race should:
   (a) The horses should move in a more realistic animation, rather than simply sliding along
       the ground as in the example world.
   (b) All 3 horses should move simultaneously forward, but a different random distance.

2. Horse Race 2
As stated in this section, any code that uses tail-recursion can be written using a while loop. The
reverse is also true. Create a horse race implementation that uses tail-recursion instead of the
while loop.

3. Robot and scooter
Create a world with gorilla robot and a scooter, as illustrated below. Write an interactive
program to allow the user to drag the scooter around the scene. (Use a let the mouse move event
that allows the user to move the scooter) As the scooter is dragged around the scene, make the
robot chase after it by moving to the scooter without colliding with the scooter.

   Hint: The while something is true event can be used. The "something" is the distance the
   scooter is in front of the gorilla robot. See Tips & Techniques 7 for further details on using BDE
   as part of while something is true.)

4. Rabbit and Butterfly 2
   In section 8-1, an exercise was to create an animation named Rabbit and Butterfly, where the
   White Rabbit chases a butterfly that is moving randomly around the world. Write an animation
   for this world using a while loop.
9-2 Infinite Repetition

Our goal in this section is to illustrate repetitive motions where the motion should continue forever – or at least until the world is stopped. These techniques are useful in animations where some action should be occurring continuously in the background or where some animation is part of a game and should be continued until the game is won or lost.

Mutually invoked methods

The first technique to be presented is a fairly sophisticated programming concept where methods mutually invoke one another. Its power will be illustrated through the use of an example. Suppose a virtual world with a helicopter in the initial scene, as in Figure 9-2-1.

The idea in this animation is to make the top shaft rotate once in a clockwise direction and then shift direction and rotate once in a counter-clockwise direction and then repeat these actions over and over again, as long as the world is running. In the world shown in Figure 9-2-1, we can view clockwise as the rotor blade turning right and counterclockwise as the rotor blade turning left. A turnRight rotation method is shown in Figure 9-2-2(a) and a turnLeft rotation in Figure 9-2-2(b).

![Figure 9-2-1. Initial helicopter world](image)

![Figure 9-2-2a. The turnRight method](image)

Figure 9-2-2a. The turnRight method
So, how do we make these two methods repeat over and over again as long as the world is running? Actually there are several ways to make this happen. In this example, the technique we will use is: the `turnLeft` method (after rotating the blade to the left) will call the `turnRight` method and then the `turnRight` method (after rotating the blade to the right) will call the `turnRight` method. Thus a loop is created through the use of mutually invoked methods! Figures 9-2-3(a) and 9-2–3(b) illustrate the mutually invoked methods.
**Infinite recursion**

In some games, points are scored when some event occurs. But, that is not the end of the game. The game is reset in some way and the game continues. In the big fish chasing the goldfish example presented in sections 8-1 and 9-1, the animation stopped when the big fish caught up to and ate the goldfish. But, let’s extend this chase scene to a more game-like scenario where the big fish continues to catch a goldfish as part of some sort of game. When the big fish catches and eats the goldfish, the goldfish will disappear and after a short delay a new goldfish will appear in a different location on the screen and the game will continue.

Two actions must be considered. The first action is that of creating a new goldfish. Actually, this is an illusion. We simply move the invisible goldfish to a new, random location and then, after a short delay, set the goldfish isShowing property back to true. When the goldfish reappears elsewhere in the world, it gives the viewer the impression that it is a different goldfish.

Another reason for moving the goldfish to a new location before making it visible again is so the chase can continue. If the goldfish were made visible again at the same location where it disappeared, the big fish would be waiting at this location, and always immediately “catch” the goldfish again. To move the goldfish to a new location, the move to instruction and the random number question are used for random motion, as seen in Figure 9-2-4. Random motion is a common operation used in many programs, as was explained in section 8-1. When random motion is used, limits must be placed on the amount of movement so as to keep the object in view of the camera and other objects in the scene. In this example, number values (-3 to 3) are used to limit the random motion of the goldfish in right/left and forward/back directions and (2 to 4) for the up/down direction. So, the new goldfish will start out somewhere in the world between 3 units forward and 3 units backward from the center of the world, between 2 and 4 units in the air, and between 3 units to the left and 3 units to the right of center. The purpose of these limitations is to make sure the goldfish reappears above the ground and within reasonable view of the big fish. Although the ground has been made invisible in this water world simulation, it is still there and objects that move below the ground cannot be seen.

![Figure 9-2-4. Moving the goldfish to a new random location](image)

Now, when the big fish eats the goldfish, the goldfish will be moved to a new random location and reappear. The camera will be set to point at the goldfish, and the chase method will be called to restart the chase.
The infinite chase method is shown in Figure 9-2-5. Notice the modification of the Else part of the if–else construct. The way the method now works is:

- If the goldfish is more than 1 meter away from the big fish, then the chase continues—the big fish points towards the goldfish, the big fish and the goldfish both move, and the chase method is called again.
- Otherwise, the Else-part of the if-structure kicks in where the big fish eats the goldfish. Then, the invisible goldfish gets moved to a new location and is made visible again. Finally, the camera is set to point at the goldfish and the chase starts again.

**Infinite Loop and while statements**

In Figure 9-2-6, we show a close-up shot of a carousel. The carousel is actually part of a much larger amusement park world. A variety of objects are available for an amusement park scene in the Amusement park folder on the web gallery. Other actions will be taking place in front of the carousel while the carousel turns round and round to music. Of course, the horses in the carousel go up and down, up and down. The action of the carousel will continue as long as the world is running.
The instructions to move the carouse are placed in a Do together block. The instructions move the horses up and down as the carousel goes round and round. The code for the carousel animation is shown in Figure 9-2-7 and the horse animation (to make the individual horses go up and down) in Figure 9-2-8.

```
World.carouselAnimation; No parameters

No variables

Do together

- CarouselBase turn left 1 revolution duration = 10 seconds style = gently more...

Do in order

- Loop 5 times

- Do together

  - World;horseAnimation horseStartUp = CarouselBase.Horse2 = horseStartDown = CarouselBase.Horse1

- World.horseAnimation horseStartUp = CarouselBase.Horse4 = horseStartDown = CarouselBase.Horse3

- World.horseAnimation horseStartUp = CarouselBase.Horse6 = horseStartDown = CarouselBase.Horse5
```

Figure 9-2-7. Code for Carousel Animation
One way to make the carousel go around forever in an infinite loop is to use a Loop instruction with *infinity* as the count, as shown in the `carouselAnimationLoop` method in Figure 9-2-9.

A second technique is to use a while loop with a Boolean condition “true.” Figure 9-2-10 shows a `carouselAnimationWhile` method. When the conditional test of a while loop is true, it can never become false. The action of the while loop will continue for as long as the animation is running.

A third technique is to use a while *something is true* event with “true” substituted for “something,” as in Figure 9-2-11. The advantage of this approach is that other events can occur while the carousel goes around and around. There is no danger of causing Alice to get stuck just running a while loop.
Figure 9-2-11. While true – an infinite loop using an event

Note that only the "During" phase of a "Begin…During…End" construct is used. This is because the carousel motion is to occur all the time – not just at the beginning, the middle, or the end of some sequence of actions. The "Begin…During…End" construct was discussed in detail in Tips & Techniques 7.

9-2 Exercises
1. StepAfterStep
Create a set of realistic walking motions for any character of your choice from the People gallery. Use mutually invoked methods to transfer from the right step to the left step.

2. StopAndStart
Choose a ride object other than the carousel from the Amusement Park gallery that moves in a circular pattern. Circular pattern means that it moves in a round-and-round manner (e.g., the Ferris Wheel). Create a method that performs an animation appropriate for the ride object selected. Then, create a way to start and stop the ride using the While <something> is true event. For example, you could add a switch to your world that could be clicked to start and stop the ride.

3. SpeedControl
Place a fan (from Objects in the Gallery) in a new world. The fan has four buttons – high, medium, low, and off. Create a method that controls the speed at which the fan blades rotate, depending on whether the high, medium, or low button is clicked. (You may want to use several methods instead of just one method.) The fan should continue running until the animation stops running or the user clicks the off button.
4. OldTimeRockAndRoll
Create a world with an old fashioned phonograph in it. (See the Objects folder in the Gallery.) Create methods to turn the crank and turn the record. Then, create a *While <true> is true* event and use the BDE control mechanism (Tips & Techniques 7) to call a method that plays the record at the same time the crank is turning.

5. Windup Penguin
Create a world with a windup penguin. This is actually a penguin( from the Animals gallery) with a windup key (from the Objects gallery) positioned against its back. The key’s *vehicle* property has been set to the penguin. In this world, make the penguin waddle (or walk) around the world continuously while its wind-up key turns. Use a "while..." event to make this an infinite loop.

*Hint:*
A simple Penguin waddle can be simulated by moving the penguin forward 0.12 meters at the same time as doing the following in order: roll the penguin to the left 0.01 revolutions, roll him to the right 0.02 revolutions and roll him to the left again 0.01 revolutions.

6. OnTheHour
Place a cuckoo clock in a new world. The idea of this animation is to have the cuckoo clock keep time (not in real-time, of course). The minute hand should go around on the face of the clock (perhaps 1 complete revolution should take about 30 seconds in real-time) and the pendulum should swing back and forth. When the minute hand has made 1 revolution (from 12 back to 12 on the face of the clock), then the hour hand should advance to the next hour, the doors should open, and the cuckoo bird (on the clock arm) should come out and chirp once for each hour of time that has elapsed. Then the bird should retreat inside the clock and the doors should close until the next hour has gone by. “All is well” as long as the clock is running -- which should continue until the user stops the animation.

8. Alice-in-the-box
Animations are popular in computer software created for children. For example, in a child’s game, a click on an object labeled with a letter of the alphabet causes the letter to be “spoken” and some animation occurs on screen. This world illustrates an example of an Alice-in-the-box animation for children’s software. As illustrated in the picture below left, the initial scene shows a closed box (composed of a cube with a square as its top). Inside the box, but hidden from view, is Alice and a spring. Extending from the side of the box is a "crank" made from a baseball bat.
At the beginning of the animation, the box is closed. The crank continuously moves around and around and a "clinky" kind of sound could be playing throughout the running of the animation. Every 5 seconds, the box top opens and Alice on her spring pops out, bouncing around a few times (Note that a black square was used in the example above to simulate the depth of the box – although this is not necessary). When Alice stops bouncing around, she and the spring are lowered back into the box and the top closes.
9 Summary

This chapter introduced the *while* statement as a third mechanism for repetition in Alice. Two examples were presented in section 9-1. In the first example, the chase scene was borrowed from section 8-1 and implemented using a while loop. The second example, the horse race, was a more complex, game-like animation using *logical operators* in the condition and *random selection* within the while statement.

In section 9-2, we looked at several situations where infinite repetition is helpful for continuing actions during the animation. Examples illustrated several different techniques, including the use of *mutually invoked methods, infinite recursion, an infinity count* for a Loop instruction, a *true* condition for a while statement, and the *BDE control mechanism* in a “while…” event.

**Important concepts in this chapter**

- The *while* statement is a *conditional loop*.
- The Boolean condition used for entry into a while statement is the same kind of Boolean condition as used in *if-else* statements. The difference is the while statement is a loop and the condition is checked again after the instructions within the while statement have been executed. If the condition is still true, the loop repeats.
- *If-else* statements can be combined with *while* statements. In some programs, a while statement is nested within an if-else statement. In other situations, an if-else statement may be nested with a while loop.
9 Projects

1. Whack-a-mole
Let us design a game where the goal is to click on an object is appearing and then disappearing on the screen. If the user successfully clicks on the object, some visual action should happen so the user will know he/she has managed to “whack” the object with the mouse. In our pseudocode below, a hamster is the object and a windup key is used to signal success. (Of course, you can creatively choose a different object and a different way to signal the fact that the user has clicked on the object.)

We assume the hamster and WindupKey objects have been added to the world and both have been made invisible by setting their isShowing property to false. The code will look something like this:

while the WindupKey is invisible
    move the hamster to a random location on the screen
    make the hamster visible
    wait a short period of time, say 0.25 seconds
    make the hamster invisible
    move the hamster off the screen (perhaps by moving it down 10 meters) so the user can no longer click on it)

When the user finally manages to click the mouse on the object, a method should be invoked that signals success. In our example, the WindupKey would become visible. Once the key is visible, the game ends.

Hints: The Boolean condition for the while loop checks the isShowing property like this:

The most challenging part of this project is to move the object to a random location on the x-z plane (between say –3 and 3 in each dimension). We suggest an instruction similar to:

You will find it helpful to experiment with a wait instruction find a period of time reasonable for your computer (not too fast, not too slow).

2. TowersOfHanoi (*challenging*)
Write the Towers of Hanoi animation program presented as an example section 8-2 using while loops instead of recursion. (Hint: You will need 2 while loops, one located within the other one.)
3. LighthouseWarning
Set up a world that contains a lighthouse, a spotlight, a lightbulb, a stars skydome and a number of boats, as seen in the image below. The lighthouse sits in the harbor to warn ships of shallow water. In this simulation, the light of the lighthouse is to rotate around, as in real life. You should be able to see the light on the side of the boats as the light of the lighthouse swings around.

To create a dramatic lighting effect in this animation, place a light bulb inside of the spotlight. The lightbulb and spotlight will be moved together to create a brighter beam of light. (Hint: set the lightbulb’s vehicle property to be the spotlight.) Then, place the spotlight inside the windowed area at the top of the lighthouse. Finally, make both the lightbulb and the spotlight invisible. Since it is inside the lighthouse, a composite shot of the light is shown below.

Use repetitive world methods to continuously move each boat forward past the lighthouse and to continuously rotate the spotlight. (Hint: You may want to set the duration of the rotation to be 3 seconds or so.) In “my first method”, make the world dark by setting the brightness property of the world’s light to be 0. Then, invoke the methods to create the lighting effect.

4. BumperCars
Create a simulation of the bumper car ride where the cars move continuously around within the bumper arena. The Bumper object is available in the Amusement Park gallery. Use the one-shot copy instruction to create additional bumper cars so there are 4 cars altogether inside the bumper arena. In this animation, each car should be moving forward a small amount until it gets too close to another car or to the wall. If the car gets too close to a another car or to the wall, the car
should turn slightly to get a different direction for moving forward. Use a switch control object (Switches folder in the web gallery) to stop and start the ride. As long as the switch is on, the ride should continue.

5. NeverEndingSlideShow
(This project makes use of the technique of changing the skin of an object. See Tips & Techniques 7. ) In this world, you are to create a never-ending slide show. Set up a new world with a square (from the Shapes gallery) as the screen, the slide projector on a table, and a lightbulb in front of the screen, as shown in the image below left. The lightbulb is for the purpose of lighting up the screen (simulating light from the projector). Then, make the lightbulb invisible by setting its isShowing property to false. The resulting initial scene should appear as shown in the image below right.

For an added sense of realism, start with the world’s light on and then dim the light before turning on the projector. (Dimming the lights was discussed in the BugBand world presented in section 3-2.) The slide show in your project must display at least 4 different slides. Numbering the slides and using an if statement may help you create a method to change slides. Between each slide, the screen must go blank, the projector light must flicker (change the color of the
light), and the slide projector’s tray of slides must rotate. The slides are to change continuously (meaning once all 4 have been seen, the show should go back to slide 1 and start the show over).

The sequence of images below show a sample side show.