6 Loops

"Repetition is the reality and the seriousness of life."
—Soren Kierkegaard

"What's the key to comedy? Repetition. What's the key to comedy? Repetition."
—Anonymous

In this chapter:
- The concept of iteration.
- Two types of loops: "while," and "for." When do we use them?
- Iteration in the context of computer graphics.

6.1 What is iteration? I mean, what is iteration? Seriously, what is iteration?

Iteration is the generative process of repeating a set of rules or steps over and over again. It is a fundamental concept in computer programming and we will soon come to discover that it makes our lives as coders quite delightful. Let's begin.

For the moment, think about legs. Lots and lots of legs on our little Zoog. If we had only read Chapter 1 of this book, we would probably write some code as in Example 6-1.

Example 6-1: Many lines

```
size(200,200);
background(255);

// Legs
stroke(0);
line(50,60,50,80);
line(60,60,60,80);
line(70,60,70,80);
line(80,60,80,80);
line(90,60,90,80);
line(100,60,100,80);
line(110,60,110,80);
line(120,60,120,80);
line(130,60,130,80);
line(140,60,140,80);
line(150,60,150,80);
```

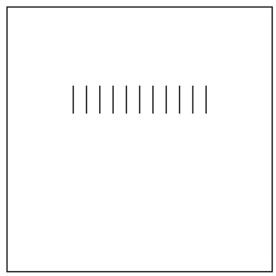


fig. 6.1

In the above example, legs are drawn from $x = 50$ pixels all the way to $x = 150$ pixels, with one leg every 10 pixels. Sure, the code accomplishes this, however, having learned variables in Chapter 4, we can make some substantial improvements and eliminate the hard-coded values.

First, we set up variables for each parameter of our system: the legs' x, y locations, length, and the spacing between the legs. Note that for each leg drawn, only the x value changes. All other variables stay the same (but they could change if we wanted them to!).

Example 6-2: Many lines with variables

```
size(200,200);
background(0);

// Legs
stroke(255);

int y = 80;           // Vertical location of each line
int x = 50;           // Initial horizontal location for first line
int spacing = 10;     // How far apart is each line
int len = 20;         // Length of each line

line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);

x = x + spacing;
line(x,y,x,y+len);
```

Draw the first leg.

Add spacing so the next leg appears 10 pixels to the right.

Continue this process for each leg, repeating it over and over.

Not too bad, I suppose. Strangely enough, although this is technically more efficient (we could adjust the spacing variable, for example, by changing only one line of code), we have taken a step backward, having produced twice as much code! And what if we wanted to draw 100 legs? For every leg, we need two lines of code. That's 200 lines of code for 100 legs! To avoid this dire, carpal-tunnel inducing problem, we want to be able to say something like:

> *Draw one line one hundred times.*

Aha, only one line of code!

Obviously, we are not the first programmers to reach this dilemma and it is easily solved with the very commonly used *control structure*—the *loop*. A loop structure is similar in syntax to a conditional (see Chapter 5). However, instead of asking a yes or no question to determine whether a block of code should be executed one time, our code will ask a yes or no question to determine *how many times* the block of code should be *repeated*. This is known as iteration.

6.2 "WHILE" Loop, the Only Loop You Really Need

There are three types of loops, the ***while*** loop, the ***do-while*** loop, and the ***for*** loop. To get started, we are going to focus on the ***while*** loop for a little while (sorry, couldn't resist). For one thing, the only loop you really need is ***while***. The ***for*** loop, as we will see, is simply a convenient alternative, a great shorthand for simple counting operations. ***Do-while***, however, is rarely used (not one example in this book requires it) and so we will ignore it.

Just as with conditional (***if/else***) structures, a ***while*** loop employs a boolean test condition. If the test evaluates to true, the instructions enclosed in curly brackets are executed; if it is false, we continue on to the next line of code. The difference here is that the instructions inside the ***while*** block continue to be executed over and over again until the test condition becomes false. See Figure 6.2.

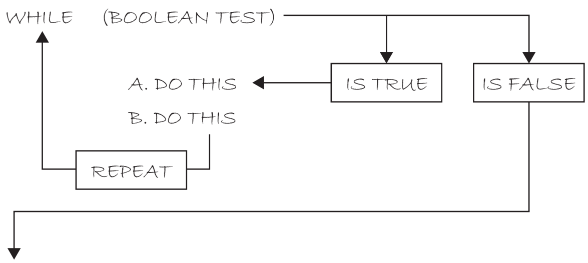


fig. 6.2

Let's take the code from the legs problem. Assuming the following variables…

```
int y = 80;          // Vertical location of each line
int x = 50;          // Initial horizontal location for first line
int spacing = 10;    // How far apart is each line
int len = 20;        // Length of each line
```

… we had to manually repeat the following code:

```
stroke(255);
line(x,y,x,y+len);      // Draw the first leg

x = x + spacing;        // Add "spacing" to x
line(x,y,x,y+len);      // The next leg is 10 pixels to the right


x = x + spacing;        // Add "spacing" to x
line(x,y,x,y+len);      // The next leg is 10 pixels to the right


x = x + spacing;        // Add ''spacing" to x
line(x,y,x,y+len);      // The next leg is 10 pixels to the right

// etc. etc. repeating with new legs
```

Now, with the knowledge of the existence of ***while*** loops, we can rewrite the code as in Example 6-3, adding a variable that tells us when to stop looping, that is, at what pixel the legs stop.

Example 6-3: While loop

```
int endLegs = 150;

stroke(0);
while (x <= endLegs) {
  line(x,y,x,y+len);
  x = x + spacing;
}
```

A variable to mark where the legs end.

Draw each leg inside a ***while*** loop.

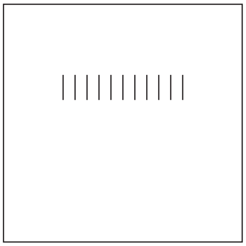


fig. 6.3

Instead of writing "*line(x,y,x,y+len);*" many times as we did at first, we now write it only *once inside of the **while** loop*, saying "as long as *x* is less than 150, draw a line at *x*, all the while incrementing *x*." And so what took 21 lines of code before, now only takes four!

In addition, we can change the spacing variable to generate more legs. The results are shown in Figure 6.4.

```
int spacing = 4;


while (x <= endLegs) {
  line(x,y,x,y+len); // Draw EACH leg
  x = x + spacing;
}
```

A smaller spacing value results in legs closer together.

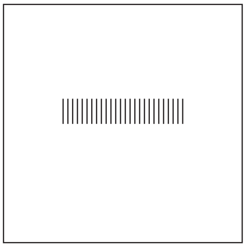


fig. 6.4

Let's look at one more example, this time using rectangles instead of lines, as shown in Figure 6.5, and ask three key questions.

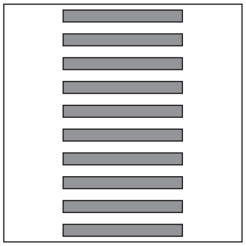


fig. 6.5

1. What is the initial condition for your loop? Here, since the first rectangle is at *y* location 10, we want to start our loop with *y* = 10.

```
int y = 10;
```

2. When should your loop stop? Since we want to display rectangles all the way to the bottom of the window, the loop should stop when *y* is greater than height. In other words, we want the loop to keep going *as long as y is less than height.*

```
while (y < 100) {
  // Loop!
}
```

3. What is your loop operation? In this case, each time through the loop, we want to draw a new rectangle below the previous one. We can accomplish this by calling the ***rect()*** function and incrementing *y* by 20.

```
rect(100,y,100,10);
y = y + 20;
```

Putting it all together:

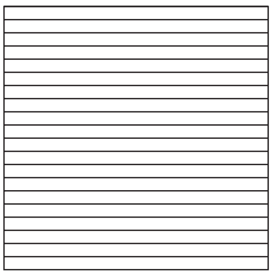
```
int y = 10;

while (y < height) {
  rect(100,y,100,10);


  y = y + 20;
}
```

Initial condition.

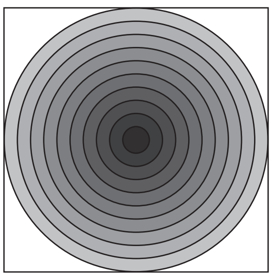
The loop continues while the boolean expression is true. Therefore, the loop stops when the boolean expression is false.

We increment *y* each time through the loop, drawing rectangle after rectangle until *y* is no longer less than height.

Exercise 6-1: Fill in the blanks in the code to recreate the following screenshots.



```
size(200,200);
background(255);
  int y = 0;
  while (________) {
    stroke(0);
    line(_______,_______,_______,_______);
    y = ________;
  }
```



```
size(200,200);
background(255);
  float w = ________;
  while (________) {
    stroke(0);
    fill(________);
    ellipse(_______,_______,_______,_______);
    __________20;
  }
```

6.3 "Exit" Conditions

Loops, as you are probably starting to realize, are quite handy. Nevertheless, there is a dark, seedy underbelly in the world of loops, where nasty things known as *infinite loops* live. See Figure 6.6.

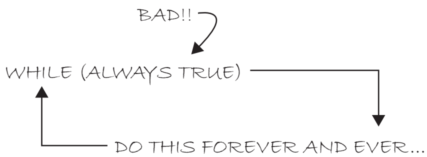


fig. 6.6

Examining the "legs" in Example 6-3, we can see that as soon as *x* is greater than 150, the loop stops. And this always happens because *x* increments by "spacing", which is always a positive number. This is not an accident; whenever we embark on programming with a loop structure, we must make sure that the exit condition for the loop will eventually be met!

Processing will not give you an error should your exit condition never occur. The result is Sisyphean, as your loop rolls the boulder up the hill over and over and over again to infinity.

Example 6-4: Infinite loop. Don't do this!

```
int x = 0;
while (x < 10){
  println(x);
  x = x - 1;
}
```

Decrementing *x* results in an infinite loop here because the value of *x* will never be 10 or greater. Be careful!

For kicks, try running the above code (make sure you have saved all your work and are not running some other mission-critical software on your computer). You will quickly see that *Processing* hangs. The only way out of this predicament is probably to force-quit *Processing*. Infinite loops are not often as obvious as in Example 6-4. Here is another flawed program that will *sometimes* result in an infinite loop crash.

Example 6-5: Another infinite loop. Don't do this!

```
int y = 80;                 // Vertical location of each line
int x = 0;                  // Horizontal location of first line
int spacing = 10;           // How far apart is each line
int len = 20;               // Length of each line
int endLegs = 150;          // Where should the lines stop?

void setup() {
  size(200,200);
}

void draw() {
  background(0);
  stroke(255);

  x = 0;
  spacing = mouseX / 2;
```

The spacing variable, which sets the distance in between each line, is assigned a value equal to ***mouseX*** divided by two.

```
    while (x <= endLegs) {
      line(x,y,x,y+len);

      x = x + spacing;
    }
  }
```

Exit Condition — when *x* is greater than ***endlegs.***

Incrementation of *x*. *x* always increases by the value of spacing. What is the range of possible value for spacing?

Will an infinite loop occur? We know we will be stuck looping forever if *x* never is greater than 150. And since *x* increments by spacing, if spacing is zero (or a negative number) *x* will always remain the same value (or go down in value.)

Recalling the ***constrain()*** function described in Chapter 4, we can guarantee no infinite loop by constraining the value of spacing to a positive range of numbers:

```
int spacing = constrain(mouseX/2, 1, 100);
```

Using ***constrain()*** to ensure the exit condition is met.

Since spacing is directly linked with the necessary exit condition, we enforce a specific range of values to make sure no infinite loop is ever reached. In other words, in pseudocode we are saying: *"Draw a series of lines spaced out by N pixels where N can never be less than 1!"*

This is also a useful example because it reveals an interesting fact about ***mouseX***. You might be tempted to try putting ***mouseX*** directly in the incrementation expression as follows:

```
while (x <= endLegs) {
  line(x,y,x,y+len);
  x = x + mouseX / 2;
}
```

Placing ***mouseX*** inside the loop is not a solution to the infinite loop problem.

Wouldn't this solve the problem, since even if the loop gets stuck as soon as the user moves the mouse to a horizontal location greater than zero, the exit condition would be met? It is a nice thought, but one that is sadly quite flawed. ***mouseX*** and ***mouseY*** are updated with new values at the beginning of each cycle through ***draw()***. So even if the user moves the mouse to *X* location 50 from location 0, ***mouseX*** will never know this new value because it will be stuck in its infinite loop and not able to get to the next cycle through ***draw()***.

6.4 "FOR" Loop

A certain style of ***while*** loop where one value is incremented repeatedly (demonstrated in Section 6.2) is particularly common. This will become even more evident once we look at arrays in Chapter 9. The ***for*** loop is a nifty shortcut for commonly occurring ***while*** loops. Before we get into the details, let's talk through some common loops you might write in *Processing* and how they are written as a ***for*** loop.

Start at 0 and count up to 9.	`for (int i = 0; i < 10; i = i + 1)`
Start at 0 and count up to 100 by 10.	`for (int i = 0; i < 101; i = i + 10)`
Start at 100 and count down to 0 by 5.	`for (int i = 100; i >= 0; i = i - 5)`

Looking at the above examples, we can see that a ***for*** loop consists of three parts:

- **Initialization**—Here, a variable is declared and initialized for use within the body of the loop. This variable is most often used inside the loop as a counter.
- **Boolean Test**—This is exactly the same as the boolean tests found in conditional statements and ***while*** loops. It can be any expression that evaluates to true or false.
- **Iteration Expression**—The last element is an instruction that you want to happen with each loop cycle. Note that the instruction is executed at the end of each cycle through the loop. (You can have multiple iteration expressions, as well as variable initializations, but for the sake of simplicity we will not worry about this now.)

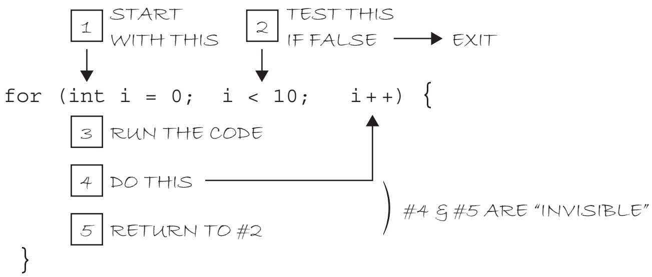


fig. 6.7

In English, the above code means: repeat this code 10 times. Or to put it even more simply: count from zero to nine!

To the machine, it means the following:

- Declare a variable *i*, and set its initial value to 0.
- While *i* is less than 10, repeat this code.
- At the end of each iteration, add one to *i*.

A ***for*** loop can have its own variable just for the purpose of counting. A variable not declared at the top of the code is called a ***local variable.*** We will explain and define it shortly.

Increment/Decrement Operators

The shortcut for adding or subtracting one from a variable is as follows:

`x++;` is equivalent to: `x = x + 1;` meaning: "increment *x* by 1" or "add 1 to the current value of *x*"

`x--;` is equivalent to: `x = x - 1;`

We also have:

`x+=2;` same as `x=x+2;`

`x*=3;` same as `x=x*3;`

and so on.

The same exact loop can be programmed with the *while* format:

```
int i = 0;
while (i < 10) {
  i++;
  //lines of code to execute here
}
```

This is the translation of the ***for*** loop, using a ***while*** loop.

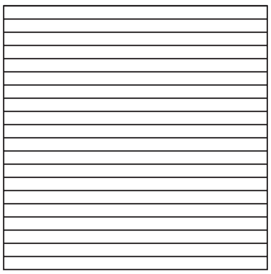
Rewriting the leg drawing code to use a *for* statement looks like this:

Example 6-6: Legs with a *for* loop

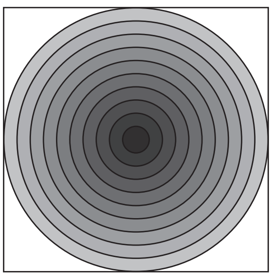
```
int y = 80;                // Vertical location of each line
int spacing = 10;          // How far apart is each line
int len = 20;              // Length of each line

for (int x = 50; x <= 150; x += spacing) {
  line(x,y,x,y+len);
}
```

Translation of the legs ***while*** loop to a ***for*** loop.

*Exercise 6-2: Rewrite Exercise 6-1 using a **for** loop.*



```
size(200,200);
background(255);

 for (int y =________;________;________) {
   stroke(0);
   line(________,________,________,________);
 }
```



```
size(200,200);
background(255);
 for (________;________;________-=20) {
   stroke(0);
   fill(________);
   ellipse(________,________,________,
 ________);
   ellipse(________,________,________,
 ________);
 }
```