5-1 Boolean expressions

What’s your favorite kind of test? Essay format? Multiple choice? In the world of computer programming, one only takes one kind of test: a boolean test — true or false. A boolean expression (named for mathematician George Boole) is an expression that evaluates to either true or false. Let’s look at some common language examples:

• My favorite color is pink. → true
• I am afraid of computer programming. → false
• This book is a hilarious read. → false

In the formal logic of computer science, relationships between numbers are tested.

• 15 is greater than 20 → false
• 5 equals 5 → true
• 32 is less than or equal to 33 → true

In this chapter, I will show how to use a variable in a boolean expression, allowing a sketch to take different paths depending on the current value stored in the variable.

• x > 20 → depends on current value of x
• y == 5 → depends on current value of y
• z <= 33 → depends on current value of z

The following operators can be used in a boolean expression.

**Relational operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
</tbody>
</table>
5-2 Conditionals: if, else, else if

Boolean expressions (often referred to as “conditionals”) operate within the sketch as questions. Is 15 greater than 20? If the answer is yes (i.e., true), you can choose to execute certain instructions (such as draw a rectangle); if the answer is no (i.e., false), those instructions are ignored. This introduces the idea of branching; depending on various conditions, the program can follow different paths.

In the physical world, this might amount to instructions like so:

If I am hungry then eat some food, otherwise if I am thirsty, drink some water, otherwise, take a nap.

In Processing, you might have something more like:

If the mouse is on the left side of the screen, draw a rectangle on the left side of the screen.

Or, more formally, with the output shown in Figure 5-1,
FIGURE 5-1

if (mouseX < width/2) {
  fill(255);
  rect(0, 0, width/2, height);
}

The boolean expression and resulting instructions in the above source code is contained within a block of code with the following syntax and structure:
if (boolean expression) {
  // code to execute if boolean expression is true
}

The structure can be expanded with the keyword else to include code that is executed if the boolean expression is false. This is the equivalent of “otherwise, do such and such.”
if (boolean expression) {
  // code to execute if boolean expression is true
For example, I could say the following, with the output shown in Figure 5-2

```
if (mouseX < width/2) {
    background(255);
} else {
    background(0);
}
```

If the mouse is on the left side of the screen, draw a white background, otherwise draw a black background.

Finally, for testing multiple conditions, you can employ an `else if`. When an `else if` is used, the conditional statements are evaluated in the order presented. As soon as one boolean expression is found to be true, the corresponding code is executed and the remaining boolean expressions are ignored. See Figure 5-3.
if (boolean expression #1) {
    // code to execute if boolean expression #1 is true
} else if (boolean expression #2) {
    // code to execute if boolean expression #2 is true
} else if (boolean expression #n) {
    // code to execute if boolean expression #n is true

FIGURE 5-3

if (A is true)

    No       Yes

     ↓        ↓

    Do this.  And this.

else if (B is true)

    No       Yes

     ↓        ↓

    Do this.  And this.

else if (C is true)

    No       Yes

     ↓        ↓

    Do this.  And this.

else

    Do this.  And this.

Now on to something else....
Taking the simple mouse example a step further, I could say the following, with results shown in Figure 5-4.

If the mouse is on the left third of the window, draw a white background, if it’s in the middle third, draw a gray background, otherwise, draw a black background.

```javascript
if (mouseX < width/3) {
  background(255);
} else if (mouseX < 2*width/3) {
  background(127);
} else {
  background(0);
}
```

**Exercise 5-1**

Consider a grading system where numbers are turned into letters. Fill in the blanks in the following code to complete the boolean expression.
It’s worth pointing out that in Exercise 5-2 on page 71 when I tested for equality I had to use two equal signs. This is because, when programming, asking if something is equal is different from assigning a value to a variable.

**Exercise 5-2**

Examine the following code samples and determine what will appear in the message window. Write down your answer and then execute the code in Processing to compare.
Problem #1: Determine if a number is between 0 and 25, 26 and 50, or greater than 50.

<table>
<thead>
<tr>
<th>int x = 75;</th>
<th>int x = 75;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &gt; 50) {</td>
<td>if (x &gt; 25) {</td>
</tr>
<tr>
<td>println(x + &quot; &gt; 50!&quot;);</td>
<td>println(x + &quot; &gt; 25!&quot;);</td>
</tr>
<tr>
<td>} else if (x &gt; 25) {</td>
<td>} else if (x &gt; 50) {</td>
</tr>
<tr>
<td>println(x + &quot; &gt; 25!&quot;);</td>
<td>println(x + &quot; &gt; 50!&quot;);</td>
</tr>
<tr>
<td>} else {</td>
<td>} else {</td>
</tr>
<tr>
<td>println(x + &quot; &lt;= 25!&quot;);</td>
<td>println(x + &quot; &lt;= 25!&quot;);</td>
</tr>
</tbody>
</table>
Although the syntax is correct, what is problematic about the code in column two above?

**Problem #2:** If a number is 5, change it to 6. If a number is 6, change it to 5.

<table>
<thead>
<tr>
<th>int x = 5;</th>
<th>int x = 5;</th>
</tr>
</thead>
<tbody>
<tr>
<td>println(&quot;x is now: &quot; + x);</td>
<td>println(&quot;x is now: &quot; + x);</td>
</tr>
<tr>
<td>if (x == 5) {</td>
<td>if (x == 5) {</td>
</tr>
<tr>
<td>x = 6;</td>
<td>x = 6;</td>
</tr>
<tr>
<td>}</td>
<td>} else if (x == 6) {</td>
</tr>
<tr>
<td>if (x == 6) {</td>
<td>}</td>
</tr>
<tr>
<td>x = 5;</td>
<td>x = 5;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
Although the syntax is correct, what is problematic about the code in column one above?

```java
if (x == y) {
    "Is x equal to y?” Use double equals!
    x = y;
    "Set x equal to y.” Use single equals!
}
```

### 5-3 Conditionals in a sketch

Let’s look at a very simple example of a program that performs different tasks based on the result of certain conditions. The pseudocode is below.

**Step 1.** Create variables to hold on to red, green, and blue color components. Call them r, g, and b.

**Step 2.** Continuously draw the background based on those colors.

**Step 3.** If the mouse is on the right-hand side of the screen, increment the value of r, if it’s on the left-hand side decrement the value of r.

**Step 4.** Constrain the value r to be within 0 and 255.

This pseudocode is implemented in Processing in Example 5-1.
float r = 150;
float g = 0;
float b = 0;

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

void draw() {
    background(r, g, b);
    stroke(255);
    line(width/2, 0, width/2, height);

    if (mouseX > width/2) {
        r = r + 1;
    } else {
        r = r - 1;
    }

    if (r > 255) {
        r = 255;
    } else if (r < 0) {
        r = 0;
    }
}
Constraining the value of a variable, as in Step 4, is a common problem. Here, I do not want color values to increase to unreasonable extremes. In other examples, you might want to constrain the size or location of a shape so that it does not get too big or too small, or wander off the screen.

While using if statements is a perfectly valid solution to the constrain problem, Processing does offer a function entitled `constrain()` that will get you the same result in one line of code.

```java
if (r > 255) {
   r = 255;
} else if (r < 0) {
   r = 0;
}

r = constrain(r, 0, 255);
```
constrain() takes three arguments: the value you intend to constrain, the minimum limit, and the maximum limit. The function returns the “constrained” value and is assigned back to the variable r. (Remember what it means for a function to return a value? See the discussion of random().)

Getting into the habit of constraining values is a great way to avoid errors; no matter how sure you are that your variables will stay within a given range, there are no guarantees other than constrain() itself. And someday, as you work on larger software projects with multiple programmers, functions such as constrain() can ensure that sections of code work well together. Handling errors before they happen in code is emblematic of good style.

Let’s make my first example a bit more advanced and change all three color components according to the mouse location and click state. Note the use of constrain() for all three variables. The system variable mousePressed is true or false depending on whether the user is holding down the mouse button.

Example 5-2

More conditionals
float r = 0;
float b = 0;
float g = 0;

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

void draw() {
    background(r, g, b);
    stroke(0);

    line(width/2, 0, width/2, height);
    line(0, height/2, width, height/2);

    if (mouseX > width/2) {
        r = r + 1;
    } else {
        r = r - 1;
    }

    if (mouseY > height/2) {
        b = b + 1;
    } else {
        b = b - 1;
    }

    if (mousePressed) {
        g = g + 1;
    } else {
        g = g - 1;
    }

    r = constrain(r, 0, 255);
    g = constrain(g, 0, 255);
    b = constrain(b, 0, 255);
}

Three variables for the background color.

Color the background and draw lines to divide the window into quadrants.

If the mouse is on the right-hand side of the window, increase red. Otherwise, it is on the left-hand side and decrease red.

If the mouse is on the bottom of the window, increase blue. Otherwise, it is on the top and decrease blue.

If the mouse is pressed (using the system variable mousePressed) increase green.

Constrain all color values to between 0 and 255.
Move a rectangle across a window by incrementing a variable. Start the shape at x coordinate 0 and use an if statement to have it stop at coordinate 100. Rewrite the sketch to use `constrain()` instead of the `if` statement. Fill in the missing code.
5-4 Logical operators

You have conquered the simple if statement:

*If my temperature is greater than 98.6, then take me to the doctor.*

Sometimes, however, simply performing a task based on one condition is not enough. For example:

*If my temperature is greater than 98.6 OR I have a rash on my arm, take me to the doctor.*

*If I am stung by a bee AND I am allergic to bees, take me to the doctor.*

The same idea applies in programming.
If the mouse is on the right side of the screen AND the mouse is on the bottom of the screen, draw a rectangle in the bottom right corner.

Your first instinct might be to write the above code using a nested if statement, like so:

```java
if (mouseX > width/2) {
    if (mouseY > height/2) {
        fill(255);
        rect(width/2, height/2, width/2, height/2);
    }
}
```

In other words, you would have to get a true answer for two if statements before the code is executed. This works, yes, but can be accomplished in a simpler way using what is called a “logical and,” written as two ampersands ("&&"). A single ampersand (“&”) means something else in Processing so make sure you include two!

```
| | (logical OR)
&& (logical AND)
! (logical NOT)
```

A “logical or” is two vertical bars (a.k.a. two “pipes”) “||”. If you can’t find the pipe, it’s typically on the keyboard as shift-backslash.

```java
if (mouseX > width/2 || mouseY > height/2) {
    fill(255);
    rect(width/2, height/2, width/2, height/2);
}
```

In addition to && and ||, you also have access to the logical operator “not,” written as an exclamation point: !

*If my temperature is NOT greater than 98.6, I won’t call in sick to work.*
If I am stung by a bee **AND** I am **NOT** allergic to bees, I’ll be fine!

A Processing example is:

*If the mouse is **NOT** pressed, draw a circle, otherwise draw a square.*

Notice this example could also be written omitting the *not*, saying:

*If the mouse is pressed, draw a square, otherwise draw a circle.*

### Exercise 5-4

**Are the following boolean expressions true or false?**

Assume variables `int x = 5` and `int y = 6`.

```
! (x > 6) ____________________________ 
(x == 6 && x == 5) ____________________ 
(x == 6 || x == 5) ____________________ 
(x > −1 && y < 10) ____________________ 
```

Although the syntax is correct, what is flawed about the following boolean expression?

```
(x > 10 && x < 5) ____________________________ 
```

### 5-5 Multiple rollovers

Let’s solve a simple problem together, a slightly more advanced version of Exercise 5-5 on page 77. Consider the four screenshots shown in Figure 5-
from one single sketch. A black square is displayed in one of four quadrants, according to the mouse location.

### Exercise 5-5

Write a program that implements a simple rollover. In other words, if the mouse is over a rectangle, the rectangle changes color. Here is some code to get you started. (How might you do this for a circle?)

```java
int x = 50;
int y = 50;
int w = 100;
int h = 75;
void setup() {
  size(200, 200);
}
void draw() {
  background(255);
  stroke(0);
  if _________ && _________ && _________ && _________) {
    ______________________
  }
  _________ {
    __________
  }
  rect(x, y, w, h);
}
```
Let’s first write the logic of the program in pseudocode (i.e., English).

**Setup:**
1. Set up a window of 200 × 200 pixels.

**Draw:**
1. Draw a white background.
2. Draw horizontal and vertical lines to divide the window in four quadrants.
3. If the mouse is in the top left corner, draw a black rectangle in the top left corner.
4. If the mouse is in the top right corner, draw a black rectangle in the top right corner.
5. If the mouse is in the bottom left corner, draw a black rectangle in the bottom left corner.
6. If the mouse is in the bottom right corner, draw a black rectangle in the bottom right corner.

For instructions 3-6, I’ll ask the question: “How do you know if the mouse is in a given corner?” To accomplish this, you need to develop a more specific `if` statement. For example, you might say: “If the `mouseX` location is greater than 100 pixels and the `mouseY` location is greater than 100 pixels, draw a black rectangle in the bottom right corner. As an exercise, you may want to try writing this program yourself based on the above pseudocode. The answer, for your reference, is given in Example 5-3.

**Example 5-3**

Rollovers
Exercise 5-6: Rewrite Example 5-3 so that the squares fade from white to black when the mouse leaves their area. Hint: you need four variables, one for each rectangle’s color.

5-6 Boolean variables

The natural next step up from programming a rollover is a button. After all, a button is just a rollover that responds when clicked. Now, it may feel ever so slightly disappointing to be programming rollovers and buttons. Perhaps you’re thinking: “Can’t I just select ‘Add Button’ from the menu or something?” For us, right now, the answer is no. Yes, I will eventually cover how to use code from a library (and you might use a library to make buttons in your sketches more easily), but there is value in learning how to program GUI (graphical user interface) elements from scratch.

For one, practicing programming buttons, rollovers, and sliders is an excellent way to learn the basics of variables and conditionals. And two, using the same old buttons and rollovers that every program has is not terribly exciting. If you care about and are interested in developing new interfaces, understanding how to build an interface from scratch is a skill you will need.

OK, with that out of the way, I am going to look at how to use a boolean variable to program a button. A boolean variable (or a variable of type boolean) is

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

void draw() {
  background(255);
  stroke(0);
  line(100, 0, 100, 200);
  line(0, 100, 200, 100);

  noStroke();
  fill(0);
  if (mouseX < 100 && mouseY < 100) {
    rect(0, 0, 100, 100);
  } else if (mouseX > 100 && mouseY < 100) {
    rect(100, 0, 100, 100);
  } else if (mouseX < 100 && mouseY > 100) {
    rect(0, 100, 100, 100);
  } else if (mouseX > 100 && mouseY > 100) {
    rect(100, 100, 100, 100);
  }
}
```
a variable that can only be true or false. Think of it as a switch. It is either on or off. Press the button, turn the switch on. Press the button again, turn it off. I just used a boolean variable in Example 5-2: the built-in variable `mousePressed`. `mousePressed` is true when the mouse is pressed and false when the mouse is not.

And so my button example will include one boolean variable with a starting value of false (the assumption being that the button starts in the off state).

```java
boolean button = false;  // A boolean variable is either true or false.
```

In the case of a rollover, any time the mouse hovered over the rectangle, it turned white. The sketch will turn the background white when the button is pressed and black when it is not.

```java
if (button) {
    background(255);
} else {
    background(0);
}
```

If the value of `button` is true, the background is white. If it is false, black.

I can then check to see if the mouse location is inside the rectangle and if the mouse is pressed, setting the value of `button` to true or false accordingly. Here is the full example:

Example 5-4

**Hold down the button**
This example simulates a button connected to a light that is only on when the button is pressed. As soon as you let go, the light goes off. While this might be a perfectly appropriate form of interaction for some instances, it’s not what I am really going for in this section. What I want is a button that operates like a switch; when you flip the switch (press the button), if the light is off, it turns on. If it’s on, it turns off.

For this to work properly, I must check to see if the mouse is located inside the rectangle inside `mousePressed()` rather than as above in `draw()`.

By definition, when the user clicks the mouse, the code inside `mousePressed()` is executed once and only once (see Section 3-4 on page 42). When the mouse is clicked, I want the switch to turn on or off (once and only once).

I now need to write some code that “toggles” the switch, changes its state from on to off, or off to on. This code will go inside `mousePressed()`.

If the variable `button` equals true, it should be set to false. If it is false, it should be set to true.

```java
boolean button = false;
int x = 50;
int y = 50;
int w = 100;
int h = 75;

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

void draw() {
    if (mouseX > x && mouseX < x+w && mouseY > y && mouseY < y+h && mousePressed) {
        button = true;
    } else {
        button = false;
    }

    if (button) {
        background(255);
        stroke(0);
    } else {
        background(0);
        stroke(255);
    }

    fill(175);
    rect(x, y, w, h);
}
```

The button is pressed if (mouseX, mouseY) is inside the rectangle and mousePressed is true.

The explicit way to toggle a boolean variable. If the value of button is true, set it equal to false. Otherwise, it must be false, so set it equal to true.
There is a simpler way to go which is the following:

```java
button = !button;
```

Here, the value of `button` is set to *not* itself. In other words, if the button is true then I set it to *not true* (false). If it is false then I set it to *not false* (true). Armed with this odd but effective line of code, you are ready to look at the button in action in Example 5-5.

**Example 5-5**

```
Button as switch

boolean button = false;
int x = 50;
int y = 50;
int w = 100;
int h = 75;

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

void draw() {
  if (button) {
    background(255);
    stroke(0);
  } else {
    background(0);
    stroke(255);
  }

  fill(175);
  rect(x, y, w, h);
}

void mousePressed() {
  if (mouseX > x & & mouseY < x + w & & mouseY > y & & mouseY < y + h) {
    button = !button;
  }
}
```

When the mouse is pressed, the state of the button is toggled. Try moving this code to `draw()` like in the rollover example. (See Exercise 5-7 below.)
Exercise 5-7

Why doesn’t the following code work properly when it’s moved to `draw()`?

```java
if (mouseX > x && mouseX < x+w &&
     mouseY > y && mouseY < y+h && mousePressed) {
    button = !button;
}
```

Exercise 5-8

Example 4-3 in the previous chapter moved a circle across the window. Change the sketch so that the circle only starts moving once the mouse has been pressed. Use a boolean variable.
5-7 A bouncing ball

It’s time again to return to my friend Zoog. Let’s review what you have done so far. First, you learned to draw Zoog with shape functions available from the Processing reference. Afterward, you realized you could use variables instead of hard-coded values. Having these variables allowed you to move Zoog. If Zoog’s location is x, draw it at x, then at x + 1, then at x + 2, and so on. It was an exciting, yet sad moment. The pleasure you experienced from discovering the motion was quickly replaced by the lonely feeling of watching Zoog leave the screen. Fortunately, conditional statements are here to save the day, allowing me to ask the question: Has Zoog reached the edge of the screen? If so, turn Zoog around!
To simplify things, let’s start with a simple circle instead of Zoog’s entire pattern.

Write a program where Zoog (a simple circle) moves across the screen horizontally from left to right. When it reaches the right edge it reverses direction.

From the previous chapter on variables, you know you need global variables to keep track of Zoog’s location.

```c
int x = 0;
```

Is this enough? No. In the previous example Zoog always moved one pixel.

```c
x = x + 1;
```

This tells Zoog to move to the right. But what if I want it to move to the left? Easy, right?

```c
x = x - 1;
```

In other words, sometimes Zoog moves with a speed of “+1” and sometimes “−1.”

The speed of Zoog varies. Yes, bells are ringing. In order to switch the direction of Zoog’s speed, I need another variable: speed.

```c
int x = 0;
int speed = 1;
```

A variable for Zoog’s speed. When speed is positive Zoog moves to the right, when speed is negative Zoog moves to the left.

Now that I have variables, I can move on to the rest of the code.

Assuming `setup()` sets the size of the window, I can go directly to examining the steps required inside of `draw()`. I can also refer to Zoog as a ball in this instance since I am just going to draw a circle.

```c
background(255);
stroke(0);
fill(100);
ellipse(x, 100, 32, 32);
```

For simplicity, Zoog is just a circle.

Elementary stuff. Now, in order for the ball to move, the value of its \(x\) location should change each cycle through `draw()`.

```c
x = x + speed;
```

If you ran the program now, the circle would start on the left side of the window, move toward the right, and continue off the edge of the screen — this is the result I achieved in Chapter 4. In order for it to turn around, I need a conditional statement.

*If the ball goes off the edge, turn the ball around.*

Or more formally…
If $x$ is greater than width, reverse speed.

```java
if (x > width) {
    speed = speed * -1;  // Multiplying by -1 reverses the speed.
}
```

**Reversing the polarity of a number**

When I want to reverse the polarity of a number, I mean that I want a positive number to become negative and a negative number to become positive. This is achieved by multiplying by $-1$. Remind yourself of the following:

- $-5 \times -1 = 5$
- $5 \times -1 = -5$
- $1 \times -1 = -1$
- $-1 \times -1 = 1$

Running the sketch, I now have a circle that turns around when it reaches the right-most edge, but runs off the left-most edge of the screen. I’ll need to revise the conditional slightly.

*If the ball goes off either the right or left edge, turn the ball around.*

Or more formally…

*If $x$ is greater than width or if $x$ is less than zero, reverse speed.*

```java
if ((x > width) || (x < 0)) {
    speed = speed * -1;  // Remember, || means “or.”
}
```

Example 5-6 puts it all together.

**Example 5-6**

**Bouncing ball**
Exercise 5-9: Rewrite Example 5-6 so that the ball not only moves horizontally, but vertically as well. Can you implement additional features, such as changing the size or color of the ball based on certain conditions? Can you make the ball speed up or slow down in addition to changing direction?

The “bouncing ball” logic of incrementing and decrementing a variable can be applied in many ways beyond the motion of shapes onscreen. For example, just as a square moves from left to right, a color can go from less red to more red. Example 5-7 takes the same bouncing ball algorithm and applies it to changing color.

Example 5-7

“Bouncing” color
float c1 = 8;
float c2 = 255;
float c1Change = 1;
float c2Change = -1;

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

void draw() {
  noStroke();

  // Draw rectangle on left
  fill(c1, 0, c2);
  rect(0, 0, 100, 200);

  // Draw rectangle on right
  fill(c2, 0, c1);
  rect(100, 0, 100, 200);

  // Adjust color values
  c1 = c1 + c1Change;
  c2 = c2 + c2Change;

  // Reverse direction of color change
  if (c1 < 0 || c1 > 255) {
    c1Change *= -1;
  }
  if (c2 < 0 || c2 > 255) {
    c2Change *= -1;
  }
}
Having the conditional statement in your collection of programming tools allows for more complex motion. For example, consider a rectangle that follows the edges of a window. One way to solve this problem is to think of the rectangle’s motion as having four possible states, numbered 0 through 3. See Figure 5-10.
• State #0: left to right.
• State #1: top to bottom.
• State #2: right to left.
• State #3: bottom to top.
I can use a variable to keep track of the state number and adjust the (x,y) coordinate of the rectangle according to that state. For example: “If the state equals 2, set x equal to itself minus 1.”
Once the rectangle reaches the endpoint for that state, I can change the state variable. “If the state equals 2: (a) set x equal to itself minus 1, (b) if x is less than zero, set state equal to 3.”
The following example implements this logic.
Example 5-8

Square following edge, uses a “state” variable
For me, one of the happiest moments of my programming life was the moment I realized I could code gravity. And in fact, armed with variables and conditionals, you are now ready for this moment.

The bouncing ball sketch demonstrated that an object moves by altering its location according to speed.

\[ \text{location} = \text{location} + \text{speed} \]

Gravity is a force of attraction between all masses. When you drop a pen, the force of gravity from the earth (which is overwhelmingly larger than the pen) causes the
pen to accelerate toward the ground. What I must add to the bouncing ball is the concept of “acceleration” (which is caused by gravity, but could be caused by any number of forces). Acceleration increases (or decreases) speed. In other words, acceleration is the rate of change of speed. And speed is the rate of change of location. So I just need another line of code:

\[ speed = speed + \text{acceleration} \]

And now I have a simple gravity simulation.
Exercise 5-10

Continue with your design and add some of the functionality demonstrated in this chapter. Some options:

- Make parts of your design rollovers that change color when the mouse is over certain areas.
- Move it around the screen. Can you make it bounce off all edges of the window?
- Fade colors in and out.

Here is a simple version with Zoog.
Example 5-10

**Zoog and conditionals**

```java
float x = 100;
float y = 100;
float w = 60;
float h = 60;
float eyeSize = 16;
float xspeed = 3;
float yspeed = 1;

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

void draw() {
  // Change the location of Zoog by speed
  x = x + xspeed;
  y = y + yspeed;

  if ((x > width) || (x < 0)) {
    xspeed = -xspeed * -1;
  }

  if ((y > height) || (y < 0)) {
    yspeed = -yspeed * -1;
  }

  background(255);
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(150);
  rect(x, y, w/6, h*2);

  // Draw Zoog's head
  fill(255);
  ellipse(x, y-h/2, w, h);

  // Draw Zoog's eyes
  fill(0);
  ellipse(x-w/3, y-h/2, eyeSize, eyeSize*2);
  ellipse(x+w/3, y-h/2, eyeSize, eyeSize*2);

  // Draw Zoog's legs
  stroke(0);
  line(x-w/12, y+h, x-w/4, y+h+10);
  line(x+w/12, y+h, x+w/4, y+h+10);
}
```

Zoog has variables for speed in the horizontal and vertical direction.

An `if` statement with a logical `or` determines if Zoog has reached either the right or left edges of the screen. When this is true, multiply the speed by `-1`, reversing Zoog's direction!

Identical logic is applied to the `y` direction as well.
1 “&” or “|” are reserved for bitwise operations in Processing. A bitwise operation compares each bit (0 or 1) of the binary representations of two numbers. It’s used in rare circumstances where you require low-level access to bits.

### 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’s a fundamental concept in computer programming and you will soon come to discover that it makes your life as a coder quite delightful. Let’s begin. For the moment, think about legs. Lots and lots of legs on little Zoog. If you had only read Chapter 1 of this book, you would probably write some code as in Example 6-1.

#### Example 6-1

<table>
<thead>
<tr>
<th>Many lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(200, 200);</td>
</tr>
<tr>
<td>background(255);</td>
</tr>
<tr>
<td>// Legs</td>
</tr>
<tr>
<td>stroke(0);</td>
</tr>
<tr>
<td>line(50, 60, 50, 80);</td>
</tr>
<tr>
<td>line(60, 60, 60, 80);</td>
</tr>
<tr>
<td>line(70, 60, 70, 80);</td>
</tr>
<tr>
<td>line(80, 60, 80, 80);</td>
</tr>
<tr>
<td>line(90, 60, 90, 80);</td>
</tr>
<tr>
<td>line(100, 60, 100, 80);</td>
</tr>
<tr>
<td>line(110, 60, 110, 80);</td>
</tr>
<tr>
<td>line(120, 60, 120, 80);</td>
</tr>
<tr>
<td>line(130, 60, 130, 80);</td>
</tr>
<tr>
<td>line(140, 60, 140, 80);</td>
</tr>
<tr>
<td>line(150, 60, 150, 80);</td>
</tr>
</tbody>
</table>
In the above example, legs are drawn from x position 50 all the way to x position 150, with one leg every 10 pixels. Although the code accomplishes this, I can make some substantial improvements and eliminate the hard-coded values by using the variables I discussed in Chapter 4.

First, I’ll set up variables for each parameter of the system: the legs’ (x,y) positions, 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 you wanted them to!).

Example 6-2

*Many lines with variables*
size(200, 200);
background(255);

// Legs
stroke(0);

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);
\[\text{Draw the first leg.}\]

\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[\text{Add spacing so the next leg appears 10 pixels to the right.}\]

\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[\text{Continue this process for each leg, repeating it over and over.}\]

\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);
\[x = x + \text{spacing};\]
line(x, y, x, y+len);

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

\textit{Draw one line one hundred times.}

Aha, only one line of code!
Obviously, you are not the first programmer to reach this dilemma and it’s easily solved with the very commonly used control structure — the \textit{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, the 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 The **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, I yam 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 you will see, is simply a convenient alternative, a great shorthand for the majority of counting operations. **do-while**, however, is rarely used (not one example in this book requires it) and so I 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, the sketch continues 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.

![FIGURE 6-2](image)

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

```java
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

… I 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, I can rewrite the code as in Example 6-3, adding a variable that specifies when to stop looping, that is, at what pixel the legs stop.

**Example 6-3**

**While loop**

```c
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.
Instead of writing `line(x, y, x, y+len);` many times as I did at first, I 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, I can change the spacing variable to generate more legs. The results are shown in Figure 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.

```cpp
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.
```
1. What is the initial condition for your loop? Here, since the first rectangle is at y location 10, you want to start your loop with \( y = 10 \).
\[
\text{int } y = 10;
\]

2. When should your loop stop? Since you want to display rectangles all the way to the bottom of the window, the loop should stop when \( y \) is greater than \( \text{height} \). In other words, you want the loop to keep going as long as \( y \) is less than \( \text{height} \).
\[
\text{while } (y < \text{height}) \{ \\
\quad \text{// Loop!} \\
\}
\]

3. What is your loop operation? In this case, each time through the loop, you want to draw a new rectangle below the previous one. You can accomplish this by calling the \text{rect()} \ function and incrementing \( y \) by 20.
\[
\text{rect}(100, \, y, \, 100, \, 10); \\
\text{y} = \text{y} + 20;
\]
Putting it all together:

```java
int y = 10;  // Initial condition.
while (y < height) {
    rect(100, y, 100, 10);
    y = y + 20;  // y increments by 20 each time through the loop, drawing rectangle after rectangle until y is no longer less than height.
    stroke(0);
    line(_______,_______,_______,_______);  // The loop continues while the boolean expression is true. Therefore, the loop stops when the boolean expression is false.
}

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

Exercise 6-1

*Fill in the blanks in the code to recreate the following screenshots.*
size(200, 200);
background(255);
float w = ________;
while (________) {
    stroke(0);
    fill(________);
    ellipse(_______,_______,_______,_______);
    ________20;
}
6-3 “Exit” conditions

Loops, as you’re 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.

![Figure 6-6: Infinite Loop Diagram](image)

**Figure 6-6**
Examining the legs in Example 6-3, you can see that as soon as \( x \) is greater than 150, the loop stops. And this always happens because \( x \) increments by \textit{spacing}, which is always a positive number. This is not an accident; whenever you embark on programming with a loop structure, you 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

\textbf{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

\textbf{Another infinite loop. Don’t do this!}
Will an infinite loop occur? You know you 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, you can guarantee no infinite loop by constraining the value of spacing to a positive range of numbers:

```c
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, I can enforce a specific range of values to make sure no infinite loop is ever reached. In other words, in pseudocode I would say: “Draw a series of lines spaced out by n pixels where n can never be less than one!”

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:

```c
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’s 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 a 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 The for loop

A certain style of `while` loop where one value is incremented repeatedly (demonstrated in Section 6-2 on page 95) is particularly common. This will become even more evident once you look at arrays in Chapter 9. The `for` loop is a nifty shortcut for commonly occurring `while` loops. Before I 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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Loop Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start at 0 and count up to 9.</td>
<td><code>for (int i = 0; i &lt; 10; i = i + 1)</code></td>
</tr>
<tr>
<td>Start at 0 and count up to 100 by 10.</td>
<td><code>for (int i = 0; i &lt; 101; i = i + 10)</code></td>
</tr>
<tr>
<td>Start at 100 and count down to 0 by 5.</td>
<td><code>for (int i = 100; i &gt;= 0; i = i - 5)</code></td>
</tr>
</tbody>
</table>

Looking at the above examples, you 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 let’s not worry about this now.)
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**. I will explain and define it shortly.

**Increment and 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;`

There is also:

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

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

Example 6-6

**Legs with a for loop**

```java
int y = 88; // 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) {  // Translation of the legs while loop to a for loop.
    line(x, y, x, y + len);
}
```

Exercise 6-2

Rewrite Exercise 6-1 on page 98 using a `for` loop.

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

for (int y = _______; _______; _______) {
    stroke(0);
    line(_______,_______,_______,_______);
}
```
size(200, 200);
background(255);
for (________;________;________ -= 20) {
  stroke(0);
  fill(________);
  ellipse(_______,_______,_______,_______);
}
Exercise 6-3

Following are some additional examples of loops. Match the appropriate screenshot with the loop structure. Each example assumes the same four lines of initial code.
6-5 Local vs. global variables (a.k.a. “variable scope”)

Up until this moment, any time that I used a variable, I declared it at the top of the sketch above setup().

```cpp
int x = 0;  // I have always declared variables at the top of the code.

void setup() {
  size(200, 200);
}
```
This was a nice simplification and allowed me to focus on the fundamentals of declaring, initializing, and using variables. Variables, however, can be declared anywhere within a program, and I will now look at what it means to declare a variable somewhere other than the top and how you might go about choosing the right location for declaring a variable.

Imagine, for a moment, that a computer program is running your life. And in this life, variables are pieces of data written on post-its that you need to remember. One post-it might have the address of a restaurant for lunch. You write it down in the morning and throw it away after enjoying a nice kale burger. But another post-it might contain crucial information (such as a bank account number), and you save it in a safe place for years on end. This is the concept of *scope*. Some variables exist (i.e., are accessible) throughout the entire course of a program’s life — *global variables* — and some live temporarily, only for the brief moment when their value is required for an instruction or calculation — *local variables*.

In Processing, global variables are declared at the top of the program, outside of both `setup()` and `draw()`. These variables can be used in any line of code anywhere in the program. This is the easiest way to use a variable since you do not have to remember when you can and cannot use that variable. You can *always* use that variable (and this is why I started with global variables only).

Local variables are variables declared within a block of code. So far, you have seen many different examples of blocks of code: `setup()`, `draw()`, `mousePressed()`, and `keyPressed()`, if statements, and `while` and `for` loops.

*A local variable declared within a block of code is only available for use inside that specific block of code where it was declared.* If you try to access a local variable outside of the block where it was declared, you will get this error: The variable “variableName” doesn’t exist.

This is the same exact error you would get if you did not bother to declare the variable `variableName` at all. Processing does not know what it is because no variable with that name exists within the block of code you happen to be in.

Here is an example where a local variable is used inside of `draw()` for the purpose of executing a while loop.

---

**Example 6-7**

*Local variable*
Why bother? Couldn’t I just have declared \( x \) as a global variable? While this is true, since I am only using \( x \) within the `draw()` function, it’s wasteful to have it as a global variable. It’s more efficient and ultimately less confusing when programming to declare variables only within the scope of where they are necessary. Certainly, many variables need to be global, but this is not the case here.

A `for` loop offers up a spot for a local variable within the “initialization” part:

```java
for (int i = 0; i < 100; i += 10) {
    stroke(255);
    fill(i);
    rect(i, 0, 10, height);
}
```

It’s not required to use a local variable in the `for` loop, however, it’s usually convenient to do so.

It’s theoretically possible to declare a local variable with the same name as a global variable. In this case, the program will use the local variable within the current scope and the global variable outside of that scope.

**Exercise 6-4**

Predict the result of the following two sketches. After 100 frames, what will the screen look like? Test your theory by running them.
#1: Global “count”
float count = 0; Declared globally.
void setup() {
    size(200, 200);
}
void draw() {
    count = count + 1;
    background(count);
}

#2: Local “count”
void setup() {
    size(200, 200);
}
void draw() {
    float count = 0; Declared locally.
    count = count + 1;
    background(count);
}

6-6 Loop inside the draw() loop
The distinction between local and global variables moves you one step further toward successfully integrating a loop structure into Zoog. Before you finish this chapter, I want to take a look at one of the most common points of confusion that comes with writing your first loop in the context of a “dynamic” Processing sketch. Consider the following loop (which happens to be the answer to Exercise 6-2 on page 102). The outcome of the loop is shown in Figure 6-8.

```java
for (int y = 0; y < height; y += 10) {
    stroke(0);
    line(0, y, width, y);
}
```

Let’s say you want to take the above loop and display each line one at a time so that you see the lines appear animated from top to bottom. Your first thought might be to take the above loop and bring it into a dynamic Processing sketch with `setup()` and `draw()`.

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

void draw() {
    background(255);
    for (int y = 0; y < height; y += 10) {
        stroke(0);
        line(0, y, width, y);
    }
}
```
If you read the code, it seems to make sense that you would see each line appear one at a time. “Set up a window of size 200 by 200 pixels. Draw a black background. Draw a line at y position 0. Draw a line at y position 10. Draw a line at y position 20.”

Referring back to Chapter 2, however, you may recall that Processing does not actually update the display window until the end of `draw()` is reached. This is crucial to remember when using `while` and `for` loops.

These loops serve the purpose of repeating something in the context of one cycle through `draw()`. They are a loop inside of the sketch’s main loop, `draw()`. Displaying the lines one at a time is something you can do with a global variable in combination with the very looping nature of `draw()` itself.

**Example 6-8**
The logic of this sketch is identical to Example 4-3, this book’s first motion sketch with variables. Instead of moving a circle across the window horizontally, I am moving a line vertically (but not clearing the background for each frame).

Exercise 6-5

It’s possible to achieve the effect of rendering one line at a time using a for loop. See if you can figure out how this is done. Part of the code is below.

```cpp
int endY;
void setup() {
  size(200, 200);
  frameRate(5);
  endY = ________;
}
void draw() {
  background(0);
  for (int y = ________; ________; ________) {
    stroke(255);
```
Using a loop inside `draw()` also opens up the possibility of interactivity. Example 6-9 displays a series of rectangles (from left to right), each one colored with a brightness according to its distance from the mouse.

**Example 6-9**

*Simple while loop with interactivity*
Exercise 6-6: Rewrite Example 6-9 using a for loop.

6-7 Zoog grows arms

I last left Zoog bouncing back and forth in the Processing window. This new version of Zoog comes with one small change. Example 6-10 uses a for loop to add a series of lines to Zoog’s body, resembling arms.

Example 6-10

Zoog with arms
I can also use a loop to draw multiple instances of Zoog by placing the code for Zoog’s body inside of a for loop. See Example 6-11.

**Example 6-11**

**Multiple Zoogs**
FIGURE 6-11

```cpp
int w = 60;
int h = 60;
int eyeSize = 16;

void setup() {
    size(400, 200);
};

void draw() {
    background(255);
    ellipseMode(CENTER);
    rectMode(CENTER);

    int y = height/2;

    // Multiple versions of Zoog
    for (int x = 88; x < width; x += 88) {
        // Draw Zoog's body
        stroke(0);
        fill(175);
        rect(x, y, w/6, h*2);

        // Draw Zoog's head
        fill(255);
        ellipse(x, y-h/2, w, h);

        // Draw Zoog's eyes
        fill(0);
        ellipse(x-w/3, y-h/2, eyeSize, eyeSize*2);
        ellipse(x+w/3, y-h/2, eyeSize, eyeSize*2);

        // Draw Zoog's legs
        stroke(0);
        line(x-w/12, y+h, x-w/4, y+h+10);
        line(x+w/12, y+h, x+w/4, y+h+10);
    }
};
```

The variable `x` is now included in a for loop, in order to iterate and display multiple Zoogs!
Exercise 6-7: Add something to your design using a for or while loop. Is there anything you already have that could be made more efficient with a loop?

Exercise 6-8

Create a grid of squares (each colored randomly) using a for loop. (Hint: You will need two for loops!) Recode the same pattern using a while loop instead of for.
Lesson Two Project

1. Take your Lesson One design and rewrite it with variables instead of hard-coded values. Consider using a for loop in the creation of your design.

2. Write a series of assignment operations that alter the values of those variables and make the design dynamic. You might also use system variables, such as width, height, mouseX, and mouseY.
3. Using conditional statements, alter the behavior of your design based on certain conditions. What happens if it touches the edge of the screen, or if it grows to a certain size? What happens if you move the mouse over elements in your design?

If your original design was extraordinarily complex with lots of code, you might consider changing it to something very simple, possibly even just starting over from scratch with a single circle or rectangle, to be able to more easily focus on animation behaviors that use variables, conditionals, and loops.

Use the space provided below to sketch designs, notes, and pseudocode for your project.