5/Response

Code that responds to input from the mouse, keyboard, and other devices has to run continuously. To make this happen, place the lines that update inside a Processing function called `draw()`.

Once and Forever

The code within the `draw()` block runs from top to bottom, then repeats until you quit the program by clicking the Stop button or closing the window. Each trip through `draw()` is called a *frame*. (The default frame rate is 60 frames per second, but this can be changed).

Example 5-1: The draw() Function

To see how `draw()` works, run this example:

```
void draw() {
  // Displays the frame count to the Console
  println("I'm drawing");
  println(frameCount);
}
```

You'll see the following:

```
I'm drawing
1
I'm drawing
2
I'm drawing
3
...
```

In the preceding example program, the `println()` functions write the text "I'm drawing" followed by the current frame count

as counted by the special `frameCount` variable (1, 2, 3, …). The text appears in the Console, the black area at the bottom of the Processing editor window.

Example 5-2: The setup() Function

To complement the looping `draw()` function, Processing has a function called `setup()` that runs just once when the program starts:

```
void setup() {
  println("I'm starting");
}

void draw() {
  println("I'm running");
}
```

When this code is run, the following is written to the Console:

```
I'm starting
I'm running
I'm running
I'm running
...
```

The text "I'm running" continues to write to the Console until the program is stopped.

In a typical program, the code inside `setup()` is used to define the starting values. The first line is always the `size()` function, often followed by code to set the starting fill and stroke colors, or perhaps to load images and fonts. (If you don't include the `size()` function, the Display Window will be 100×100 pixels.)

Now you know how to use `setup()` and `draw()`, but this isn't the whole story. There's one more location to put code—you can also place variables outside of `setup()` and `draw()`. If you create a variable inside of `setup()`, you can't use it inside of `draw()`, so you need to place those variables somewhere else. Such variables are called *global* variables, because they can be used anywhere ("globally") in the program. This is clearer when we list the order in which the code is run:

1. Variables declared outside of `setup()` and `draw()` are created.
2. Code inside `setup()` is run once.
3. Code inside `draw()` is run continuously.

Example 5-3: Global Variable

The following example puts it all together:

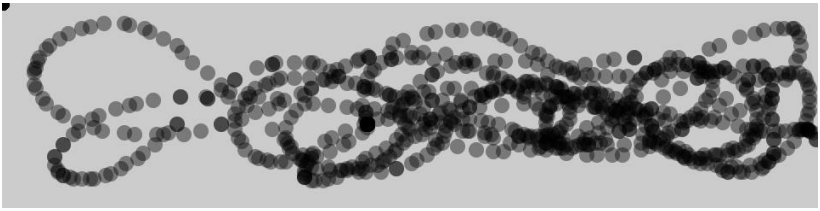
```
int x = 280;
int y = -100;
int diameter = 380;

void setup() {
  size(480, 120);
  fill(102);
}

void draw() {
  background(204);
  ellipse(x, y, diameter, diameter);
}
```

Follow

Now that we have code running continuously, we can track the mouse position and use those numbers to move elements on screen.

Example 5-4: Track the Mouse

The `mouseX` variable stores the *x* coordinate, and the `mouseY` variable stores the *y* coordinate:



```
void setup() {
  size(480, 120);
  fill(0, 102);
  noStroke();
}

void draw() {
  ellipse(mouseX, mouseY, 9, 9);
}
```

In this example, each time the code in the `draw()` block is run, a new circle is drawn to the window. This image was made by moving the mouse around to control the circle's location. Because the fill is set to be partially transparent, denser black areas show where the mouse spent more time and where it moved slowly. The circles that are spaced farther apart show when the mouse was moving faster.

Example 5-5: The Dot Follows You

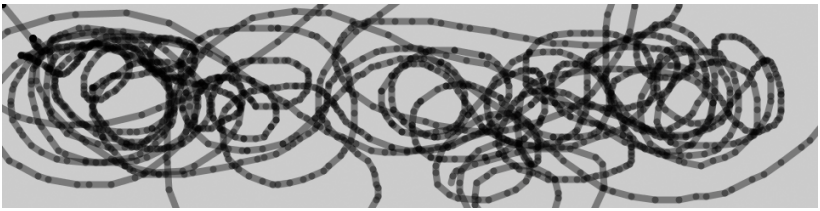
In this example, a new circle is added to the window each time the code in `draw()` is run. To refresh the screen and only display the newest circle, place a `background()` function at the beginning of `draw()` before the shape is drawn:

```
void setup() {
  size(480, 120);
  fill(0, 102);
  noStroke();
}

void draw() { background(204);
  ellipse(mouseX, mouseY, 9, 9);
}
```

The background() function clears the entire window, so be sure to always place it before other functions inside draw(); otherwise, the shapes drawn before it will be erased.

Example 5-6: Draw Continuously

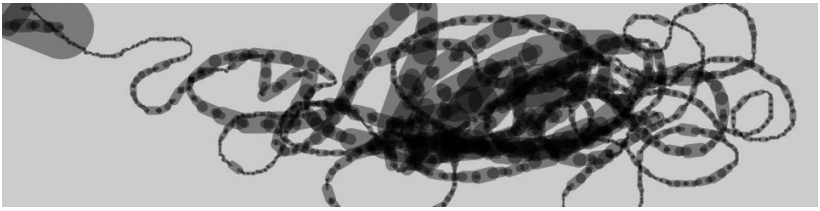
The pmouseX and pmouseY variables store the position of the mouse at the previous frame. Like mouseX and mouseY, these special variables are updated each time draw() runs. When combined, they can be used to draw continuous lines by connecting the current and most recent location:



```
void setup() {
  size(480, 120);
  strokeWeight(4);
  stroke(0, 102);
}

void draw() {
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Example 5-7: Set Line Thickness

The pmouseX and pmouseY variables can also be used to calculate the speed of the mouse. This is done by measuring the distance between the current and most recent mouse location. If the mouse is moving slowly, the distance is small, but if the mouse starts moving faster, the distance grows. A function called dist() simplifies this calculation, as shown in the following example. Here, the speed of the mouse is used to set the thickness of the drawn line:



```
void setup() {
  size(480, 120);
  stroke(0, 102);
}

void draw() {
  float weight = dist(mouseX, mouseY, pmouseX, pmouseY);
  strokeWeight(weight);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Example 5-8: Easing Does It

In Example 5-7 on page 53, the values from the mouse are converted directly into positions on the screen. But sometimes you want the values to follow the mouse loosely—to lag behind to create a more fluid motion. This technique is called *easing*. With easing, there are two values: the current value and the value to move toward (see Figure 5-1). At each step in the program, the current value moves a little closer to the target value:

```
float x;
float easing = 0.01;

void setup() {
  size(220, 120);
}

void draw() {
  float targetX = mouseX;
  x += (targetX - x) * easing;
  ellipse(x, 40, 12, 12);
  println(targetX + " : " + x);
}
```

The value of the x variable is always getting closer to targetX. The speed at which it catches up with targetX is set with the easing variable, a number between 0 and 1. A small value for easing

causes more of a delay than a larger value. With an easing value of 1, there is no delay. When you run Example 5-8 on page 54, the actual values are shown in the Console through the `println()` function. When moving the mouse, notice how the numbers are far apart, but when the mouse stops moving, the `x` value gets closer to `targetX`.

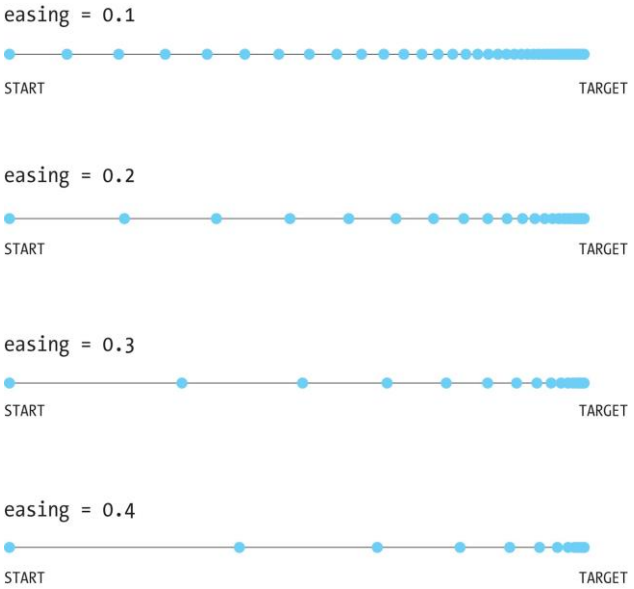
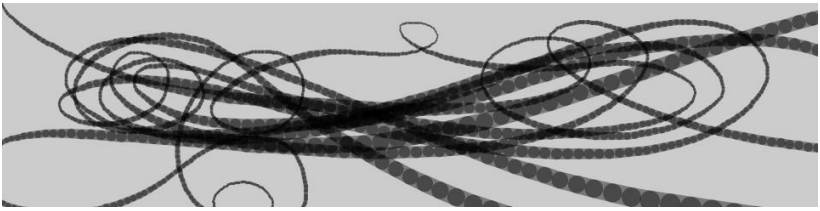


Figure 5-1. *Easing changes the number of steps it takes to move from one place to another*

All of the work in this example happens on the line that begins `x +=`. There, the difference between the target and current value is calculated, then multiplied by the easing variable and added to `x` to bring it closer to the target.

Example 5-9: Smooth Lines with Easing

In this example, the easing technique is applied to Example 5-7 on page 53. In comparison, the lines are more fluid:



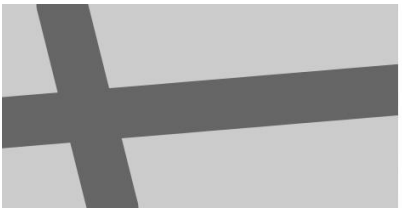
```
float x;
float y;
float px;
float py;
float easing = 0.05;

void setup() {
  size(480, 120);
  stroke(0, 102);
}

void draw() {
  float targetX = mouseX;
  x += (targetX - x) * easing;
  float targetY = mouseY;
  y += (targetY - y) * easing;
  float weight = dist(x, y, px, py);
  strokeWeight(weight);
  line(x, y, px, py);
  py = y;
  px = x;
}
```

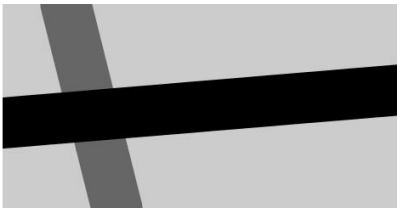
Click

In addition to the location of the mouse, Processing also keeps track of whether the mouse button is pressed. The `mousePressed` variable has a different value when the mouse button is pressed and when it is not. The `mousePressed` variable is a data type called `boolean`, which means that it has only two possible values: `true` and `false`. The value of `mousePressed` is `true` when a button is pressed.

Example 5-10: Click the Mouse

The mousePressed variable is used along with the if statement to determine when a line of code will run and when it won't. Try this example before we explain further:





```
void setup() {
  size(240, 120);
  strokeWeight(30);
}

void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed == true) {
    stroke(0);
  }
  line(0, 70, width, 50);
}
```

In this program, the code inside the if block runs only when a mouse button is pressed. When a button is not pressed, this code is ignored. Like the for loop discussed in "Repetition" on page 40, the if also has a *test* that is evaluated to true or false:

```
if (test) {
  statements
}
```

When the test is true, the code inside the block is run; when the test is false, the code inside the block is not run. The computer determines whether the test is true or false by evaluating the expression inside the parentheses. (If you'd like to refresh your memory, the discussion of relational expressions is in Example 4-6 on page 41.)

The == symbol compares the values on the left and right to test whether they are equivalent. This == symbol is different from the assignment operator, the single = symbol. The == symbol asks, "Are these things equal?" and the = symbol sets the value of a variable.

It's a common mistake, even for experienced programmers, to write = in your code when you mean to write ==. The Processing software won't always warn you when you do this, so be careful.

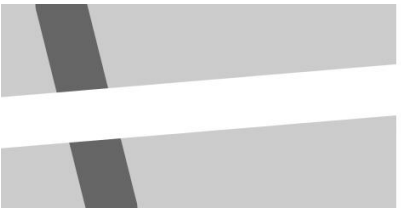
Alternatively, the test in `draw()` in can be written like this:

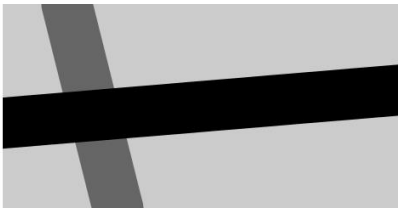
```
if (mousePressed) {
```

Boolean variables, including `mousePressed`, don't need the explicit comparison with the == operator, because they can be only `true` or `false`.

Example 5-11: Detect When Not Clicked

A single `if` block gives you the choice of running some code or skipping it. You can extend an `if` block with an `else` block, allowing your program to choose between two options. The code inside the `else` block runs when the value of the `if` block test is `false`. For instance, the stroke color for a program can be white when the mouse button is not pressed, and can change to black when the button is pressed:





```
void setup() {
  size(240, 120);
  strokeWeight(30);
}

void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed) {
    stroke(0);
  } else {
    stroke(255);
  }
  line(0, 70, width, 50);
}
```

Example 5-12: Multiple Mouse Buttons

Processing also tracks which button is pressed if you have more than one button on your mouse. The mouseButton variable can be one of three values: LEFT, CENTER, or RIGHT. To test which button was pressed, the == operator is needed, as shown here:

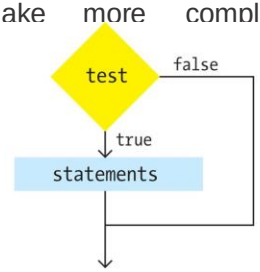
```
void setup() {
  size(120, 120);
  strokeWeight(30);
}

void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed) {
    if (mouseButton == LEFT) {
      stroke(255);
```

```
    } else {
      stroke(0);
    }
    line(0, 70, width, 50);
  }
}
```

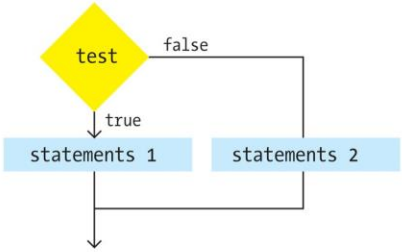
A program can have many more `if` and `else` structures (see Figure 5-2) than those found in these short examples. They can be chained together into a long series with each testing for

something different, and `if` blocks can be embedded inside of other `if` blocks to make more complex decisions.

```
if (test) {
  statements
}
```




```
if (test) {
  statements 1
} else {
  statements 2
}
```




```
if (test 1) {
  statements 1
} else if (test 2) {
  statements 2
}
```

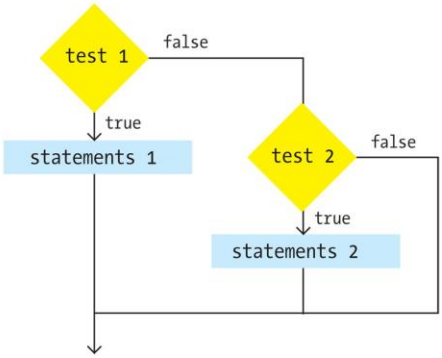


Figure 5-2. *The if and else structure makes decisions about which blocks of code to run*

Location

An `if` structure can be used with the `mouseX` and `mouseY` values to determine the location of the cursor within the window.

Example 5-13: Find the Cursor

For instance, this example tests to see whether the cursor is on the left or right side of a line and then moves the line toward the cursor:

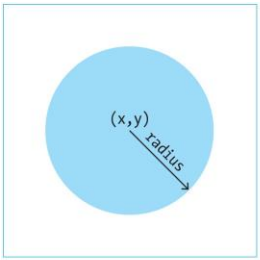
```
float x;
int offset = 10;

void setup() {
  size(240, 120);
  x = width/2;
}

void draw() {
  background(204);
  if (mouseX > x) {
    x += 0.5;
    offset = -10;
  }
  if (mouseX < x) {
    x -= 0.5;
    offset = 10;
  }
  // Draw arrow left or right depending on "offset" value
  line(x, 0, x, height);
  line(mouseX, mouseY, mouseX + offset, mouseY - 10);
  line(mouseX, mouseY, mouseX + offset, mouseY + 10);
  line(mouseX, mouseY, mouseX + offset*3, mouseY);
}
```

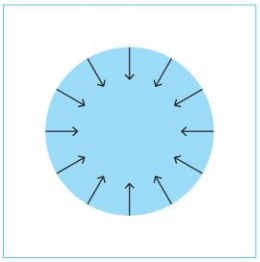
To write programs that have graphical user interfaces (buttons, checkboxes, scrollbars, etc.), we need to write code that knows when the cursor is within an enclosed area of the screen. The following two examples introduce how to check whether the cursor is inside a circle and a rectangle. The code is written in a modular way with variables, so it can be used to check for *any* circle and rectangle by changing the values.

Example 5-14: The Bounds of a Circle

For the circle test, we use the `dist()` function to get the distance from the center of the circle to the cursor, then we test to see if that distance is less than the radius of the circle (see Figure 5-3). If it is, we know we're inside. In this example, when the cursor is within the area of the circle, its size increases:

```
int x = 120;
int y = 60;
int radius = 12;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(204);
  float d = dist(mouseX, mouseY, x, y);
  if (d < radius) {
    radius++;
    fill(0);
  } else {
    fill(255);
  }
  ellipse(x, y, radius, radius);
}
```



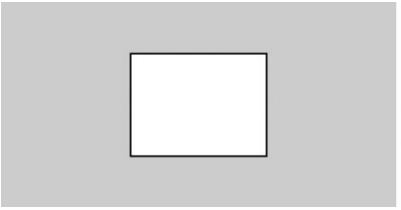


```
dist(x, y, mouseX, mouseY) < radius
```

Figure 5-3. *Circle rollover test. When the distance between the mouse and the circle is less than the radius, the mouse is inside the circle.*

Example 5-15: The Bounds of a Rectangle

We use another approach to test whether the cursor is inside a rectangle. We make four separate tests to check if the cursor is on the correct side of each edge of the rectangle, then we compare each test and if they are all `true`, we know the cursor is inside. This is illustrated in Figure 5-4. Each step is simple, but it looks complicated when it's all put together:



```
int x = 80;
int y = 30;
```

```
int w = 80;
int h = 60;

void setup() {
  size(240, 120);
}

void draw() {
  background(204);
  if ((mouseX > x) && (mouseX < x+w) &&
      (mouseY > y) && (mouseY < y+h)) {
    fill(0);
  } else {
    fill(255);
  }
  rect(x, y, w, h);
}
```

The test in the `if` statement is a little more complicated than we've seen. Four individual tests (e.g., `mouseX > x`) are combined with the logical AND operator, the `&&` symbol, to ensure that every relational expression in the sequence is `true`. If one of them is `false`, the entire test is `false` and the fill color won't be set to black. This is explained further in the reference entry for &&.

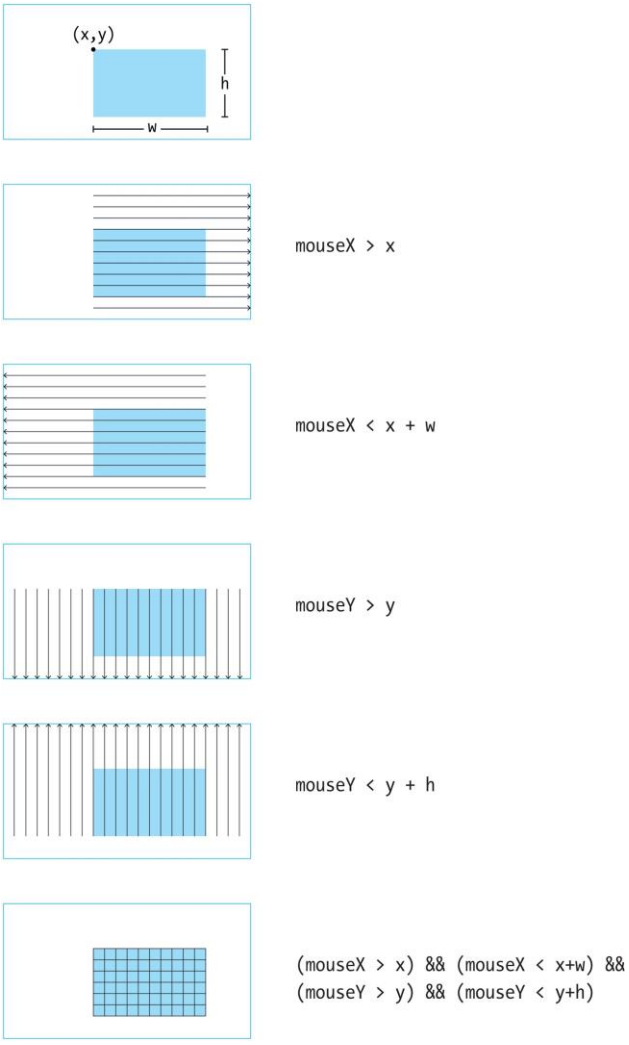


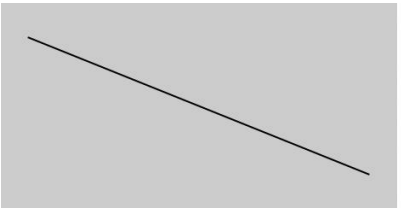
Figure 5-4. *Rectangle rollover test. When all four tests are combined and true, the cursor is inside the rectangle.*

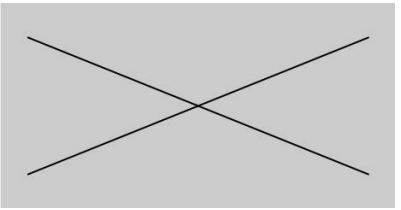
Type

Processing keeps track of when any key on a keyboard is pressed, as well as the last key pressed. Like the `mousePressed` variable, the `keyPressed` variable is `true` when any key is pressed, and `false` when no keys are pressed.

Example 5-16: Tap a Key

In this example, the second line is drawn only when a key is pressed:





```
void setup() {
  size(240, 120);
}

void draw() {
  background(204);
  line(20, 20, 220, 100);
  if (keyPressed) {
    line(220, 20, 20, 100);
  }
}
```

The key variable stores the most recent key that has been pressed. The data type for key is char, which is short for "character" but usually pronounced like the first syllable of "charcoal." A char variable can store any single character, which includes letters of the alphabet, numbers, and symbols. Unlike a string value (see Example 7-8 on page 97), which is distinguished by double quotes, the **char** data type is specified by single quotes. This is how a char variable is declared and assigned:

```
char c = 'A';  // Declares and assigns 'A' to the variable c
```

And these attempts will cause an error:

```
char c = "A";  // Error! Can't assign a String to a char
char h = A;    // Error! Missing the single quotes from 'A'
```

Unlike the boolean variable keyPressed, which reverts to false each time a key is released, the key variable keeps its value until the next key is pressed. The following example uses the value of key to draw the character to the screen. Each time a new key is pressed, the value updates and a new character draws. Some

keys, like Shift and Alt, don't have a visible character, so when you press them, nothing is drawn.

Example 5-17: Draw Some Letters

This example introduces the `textSize()` function to set the size of the letters, the `textAlign()` function to center the text on its *x* coordinate, and the `text()` function to draw the letter. These functions are discussed in more detail in "Fonts" on page 94:


```
void setup() {
  size(120, 120);
  textSize(64);
  textAlign(CENTER);
}

void draw() {
  background(0);
  text(key, 60, 80);
}
```

By using an `if` structure, we can test to see whether a specific key is pressed and choose to draw something on screen in response.

Example 5-18: Check for Specific Keys

In this example, we test for an H or N to be typed. We use the comparison operator, the == symbol, to see if the `key` value is equal to the characters we're looking for:

```
void setup() {
  size(120, 120);
}

void draw() {
  background(204);
  if (keyPressed) {
    if ((key == 'h') || (key == 'H')) {
      line(30, 60, 90, 60);
    }
    if ((key == 'n') || (key == 'N')) {
      line(30, 20, 90, 100);
    }
  }
  line(30, 20, 30, 100);
  line(90, 20, 90, 100);
}
```

When we watch for H or N to be pressed, we need to check for both the lowercase and uppercase letters in the event that someone hits the Shift key or has the Caps Lock set. We combine the two tests together with a logical OR, the || symbol. If we translate the second `if` statement in this example into plain language, it says, "If the 'h' key is pressed OR the 'H' key is pressed." Unlike with the logical AND (the && symbol), only one of these expressions need be `true` for the entire test to be `true`.

Some keys are more difficult to detect, because they aren't tied to a particular letter. Keys like Shift, Alt, and the arrow keys are coded and require an extra step to figure out if they are pressed. First, we need to check if the key that's been pressed is a coded key, then we check the code with the `keyCode` variable to see which key it is. The most frequently used `keyCode` values are `ALT`, `CONTROL`, and `SHIFT`, as well as the arrow keys, `UP`, `DOWN`, `LEFT`, and `RIGHT`.

Example 5-19: Move with Arrow Keys

The following example shows how to check for the left or right arrow keys to move a rectangle:

```
int x = 215;

void setup() {
  size(480, 120);
}

void draw() {
  if (keyPressed && (key == CODED)) {  // If it's a coded key
    if (keyCode == LEFT) {  // If it's the left arrow
      x--;
    } else if (keyCode == RIGHT) {  // If it's the right arrow
      x++;
    }
  }
  rect(x, 45, 50, 50);
}
```

Map

The numbers that are created by the mouse and keyboard often need to be modified to be useful within a program. For instance, if a sketch is 1920 pixels wide and the mouseX values are used to set the color of the background, the range of 0 to 1920 for mouseX might need to move into a range of 0 to 255 to better control the color. This transformation can be done with an equation or with a function called map().

Example 5-20: Map Values to a Range

In this example, the location of two lines are controlled with the mouseX variable. The gray line is synchronized to the cursor position, but the black line stays closer to the center of the screen to move further away from the white line at the left and right edges:

```
void setup() {
  size(240, 120);
  strokeWeight(12);
}

void draw() {
  background(204);
  stroke(102);
  line(mouseX, 0, mouseX, height);  // Gray line
  stroke(0);
  float mx = mouseX/2 + 60;
  line(mx, 0, mx, height);  // Black line
}
```

The map() function is a more general way to make this type of change. It converts a variable from one range of numbers to another. The first parameter is the variable to be converted, the second and third parameters are the low and high values of that variable, and the fourth and fifth parameters are the desired low and high values. The map() function hides the math behind the conversion.

Example 5-21: Map with the map() Function

This example rewrites Example 5-20 on page 70 using map():

```
void setup() {
  size(240, 120);
  strokeWeight(12);
}

void draw() {
  background(204);
  stroke(102);
  line(mouseX, 0, mouseX, height);  // Gray line
  stroke(0);
```

```
    float mx = map(mouseX, 0, width, 60, 180);
    line(mx, 0, mx, height);  // Black line
  }
```

The map() function makes the code easy to read, because the minimum and maximum values are clearly written as the parameters. In this example, mouseX values between 0 and width are converted to a number from 60 (when mouseX is 0) up to 180 (when mouseX is width). You'll find the useful map() function in many examples throughout this book.