4/Variables

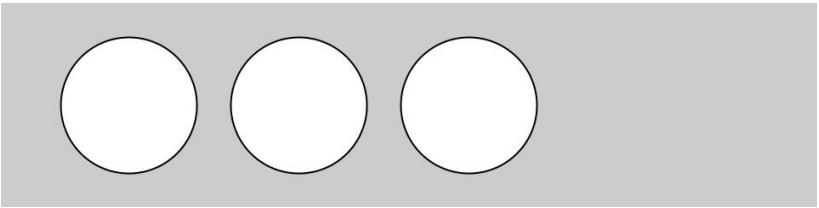
A *variable* stores a value in memory so that it can be used later in a program. The variable can be used many times within a single program, and the value is easily changed while the program is running.

First Variables

One of the reasons we use variables is to avoid repeating ourselves in the code. If you are typing the same number more than once, consider using a variable instead so that your code is more general and easier to update.

Example 4-1: Reuse the Same Values

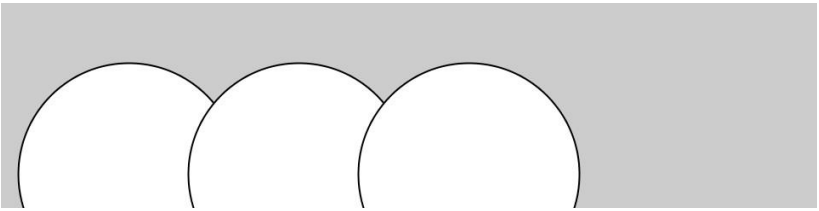
For instance, when you make the *y* coordinate and diameter for the three circles in this example into variables, the same values are used for each ellipse:



```
size(480, 120);
int y = 60;
int d = 80;
ellipse(75, y, d, d);     // Left
ellipse(175, y, d, d);    // Middle
ellipse(275, y, d, d);    // Right
```

Example 4-2: Change Values

Simply changing the *y* and *d* variables alters all three ellipses:



```
size(480, 120);
int y = 100;
int d = 130;
ellipse(75, y, d, d);     // Left
ellipse(175, y, d, d);    // Middle
ellipse(275, y, d, d);    // Right
```

Without the variables, you'd need to change the *y* coordinate used in the code three times and the diameter six times. When comparing Example 4-1 on page 35 and Example 4-2 on page 36, notice how the bottom three lines are the same, and only the middle two lines with the variables are different. Variables allow you to separate the lines of the code that change from the lines that don't, which makes programs easier to modify. For instance, if you place variables that control colors and sizes of shapes in one place, then you can quickly explore different visual options by focusing on only a few lines of code.

Making Variables

When you make your own variables, you determine the *name*, the *data type*, and the *value*. The name is what you decide to call the variable. Choose a name that is informative about what the variable stores, but be consistent and not too verbose. For instance, the variable name "radius" will be clearer than "r" when you look at the code later.

The range of values that can be stored within a variable is defined by its *data type*. For instance, the *integer* data type can store numbers without decimal places (whole numbers). In code, *integer* is abbreviated to int. There are data types to store

each kind of data: integers, floating-point (decimal) numbers, characters, words, images, fonts, and so on.

Variables must first be *declared*, which sets aside space in the computer's memory to store the information. When declaring a variable, you also need to specify its data type (such as `int`), which indicates what kind of information is being stored. After the data type and name are set, a value can be assigned to the variable:

```
int x;  // Declare x as an int variable
x = 12; // Assign a value to x
```

This code does the same thing, but is shorter:

```
int x = 12; // Declare x as an int variable and assign a value
```

The name of the data type is included on the line of code that declares a variable, but it's not written again. Each time the data type is written in front of the variable name, the computer thinks you're trying to declare a new variable. You can't have two variables with the same name in the same part of the program (Appendix D), so the program has an error:

```
int x;       // Declare x as an int variable
int x = 12;  // ERROR! Can't have two variables called x here
```

Each data type stores a different kind of data. For instance, an `int` variable can store a whole number, but it can't store a number with decimal points, called a `float`. The word "`float`" refers to "floating point," which describes the technique used to store a number with decimal points in memory. (The specifics of that technique aren't important here.)

A floating-point number can't be assigned to an `int` because information would be lost. For instance, the value 12.2 is different from its nearest `int` equivalent, the value 12. In code, this operation will create an error:

```
int x = 12.2;  // ERROR! A floating-point value can't fit in
an int
```

However, a `float` variable can store an integer. For instance, the integer value 12 can be converted to the floating-point equivalent 12.0 because no information is lost. This code works without an error:

```
float x = 12;  // Automatically converts 12 to 12.0
```

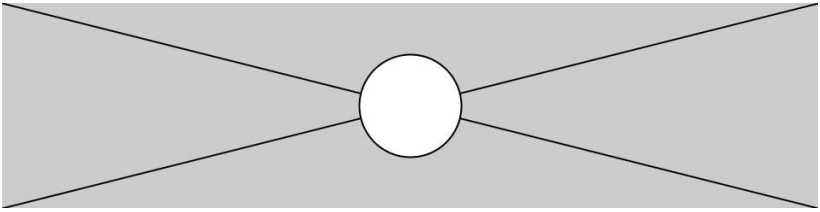
Data types are discussed in more detail in Appendix B.

Processing Variables

Processing has a series of special variables to store information about the program while it runs. For instance, the width and height of the window are stored in variables called `width` and `height`. These values are set by the `size()` function. They can be used to draw elements relative to the size of the window, even if the `size()` line changes.

Example 4-3: Adjust the Size, See What Follows

In this example, change the parameters to `size()` to see how it works:



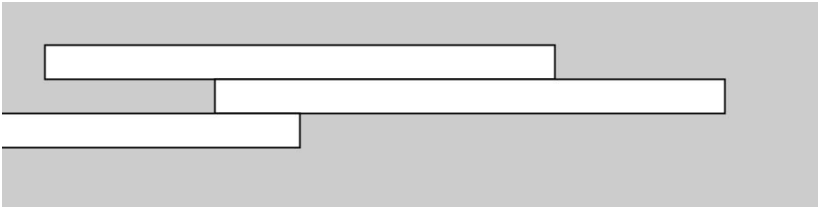
```
size(480, 120);
line(0, 0, width, height); // Line from (0,0) to (480, 120)
line(width, 0, 0, height); // Line from (480, 0) to (0, 120)
ellipse(width/2, height/2, 60, 60);
```

Other special variables keep track of the status of the mouse and keyboard values and much more. These are discussed in Chapter 5.

A Little Math

People often assume that math and programming are the same thing. Although knowledge of math can be useful for certain types of coding, basic arithmetic covers the most important parts.

Example 4-4: Basic Arithmetic



```
size(480, 120);
int x = 25;
int h = 20;
int y = 25;
rect(x, y, 300, h);          // Top
x = x + 100;
rect(x, y + h, 300, h);      // Middle
x = x - 250;
rect(x, y + h*2, 300, h);    // Bottom
```

In code, symbols like +, –, and * are called *operators*. When placed between two values, they create an *expression*. For instance, 5 + 9 and 1024 – 512 are both expressions. The operators for the basic math operations are:

+	Addition
–	Subtraction
*	Multiplication
/	Division
=	Assignment

Processing has a set of rules to define which operators take precedence over others, meaning which calculations are made first, second, third, and so on. These rules define the order in which the code is run. A little knowledge about this goes a long way toward understanding how a short line of code like this works:

```
int x = 4 + 4 * 5; // Assign 24 to x
```

The expression 4 * 5 is evaluated first because multiplication has the highest priority. Second, 4 is added to the product of 4 * 5 to yield 24. Last, because the *assignment operator* (the equals sign) has the lowest precedence, the value 24 is assigned

to the variable *x*. This is clarified with parentheses, but the result is the same:

```
int x = 4 + (4 * 5); // Assign 24 to x
```

If you want to force the addition to happen first, just move the parentheses. Because parentheses have a higher precedence than multiplication, the order is changed and the calculation is affected:

```
int x = (4 + 4) * 5; // Assign 40 to x
```

An acronym for this order is often taught in math class: PEMDAS, which stands for Parentheses, Exponents, Multiplication, Division, Addition, Subtraction, where parentheses have the highest priority and subtraction the lowest. The complete order of operations is found in Appendix C.

Some calculations are used so frequently in programming that shortcuts have been developed; it's always nice to save a few keystrokes. For instance, you can add to a variable, or subtract from it, with a single operator:

```
x += 10; // This is the same as x = x + 10
y -= 15; // This is the same as y = y - 15
```

It's also common to add or subtract 1 from a variable, so shortcuts exist for this as well. The ++ and — operators do this:

```
x++; // This is the same as x = x + 1
y--; // This is the same as y = y - 1
```

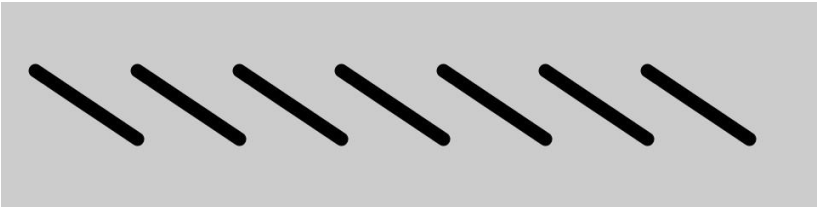
More shortcuts can be found in the *Processing Reference*.

Repetition

As you write more programs, you'll notice that patterns occur when lines of code are repeated, but with slight variations. A code structure called a *for loop* makes it possible to run a line of code more than once to condense this type of repetition into fewer lines. This makes your programs more modular and easier to change.

Example 4-5: Do the Same Thing Over and Over

This example has the type of pattern that can be simplified with a for loop:



```
size(480, 120);
strokeWeight(8);
line(20, 40, 80, 80);
line(80, 40, 140, 80);
line(140, 40, 200, 80);
line(200, 40, 260, 80);
line(260, 40, 320, 80);
line(320, 40, 380, 80);
line(380, 40, 440, 80);
```

Example 4-6: Use a for Loop

The same thing can be done with a for loop, and with less code:

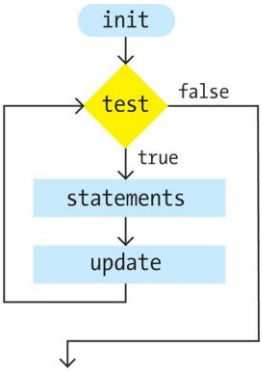
```
size(480, 120);
strokeWeight(8);
for (int i = 20; i < 400; i += 60) {
  line(i, 40, i + 60, 80);
}
```

The for loop is different in many ways from the code we've written so far. Notice the braces, the { and } characters. The code between the braces is called a *block*. This is the code that will be repeated on each iteration of the for loop.

Inside the parentheses are three statements, separated by semicolons, that work together to control how many times the code inside the block is run. From left to right, these statements are referred to as the *initialization* (init), the *test*, and the *update*:

```
for (init; test; update) {
  statements
}
```

The `init` sets the starting value, often declaring a new variable to use within the `for` loop. In the earlier example, an integer named `i` was declared and set to 20. The variable name `i` is frequently used, but there's really nothing special about it. The *test* evaluates the value of this variable (here, it checks whether `i` still less than 400), and the *update* changes the variable's value (adding 60 before repeating the loop). Figure 4-1 shows the order in which they run and how they control the code statements inside the block.




```
for (init; test; update) {
  statements
}
```

Figure 4-1. *Flow diagram of a for loop*

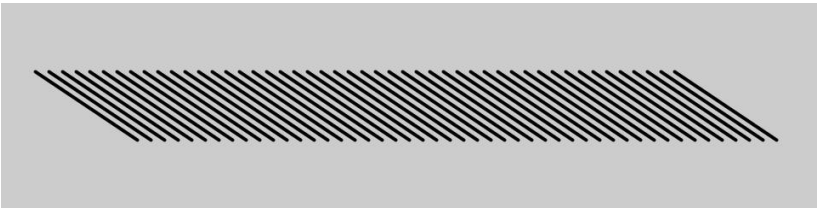
The *test* statement requires more explanation. It's always a *relational expression* that compares two values with a *relational operator*. In this example, the expression is "i < 400" and the operator is the < (less than) symbol. The most common relational operators are:

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

The relational expression always evaluates to `true` or `false`. For instance, the expression `5 > 3` is `true`. We can ask the question, "Is five greater than three?" Because the answer is "yes," we say the expression is `true`. For the expression `5 < 3`, we ask, "Is five less than three?" Because the answer is "no," we say the expression is `false`. When the evaluation is `true`, the code inside the block is run, and when it's `false`, the code inside the block is not run and the `for` loop ends.

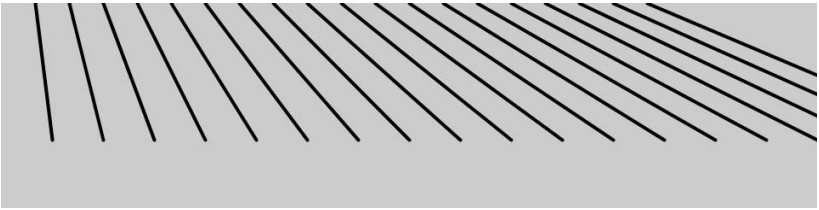
Example 4-7: Flex Your for Loop's Muscles

The ultimate power of working with a `for` loop is the ability to make quick changes to the code. Because the code inside the block is typically run multiple times, a change to the block is magnified when the code is run. By modifying Example 4-6 on page 41 only slightly, we can create a range of different patterns:



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 8) {
  line(i, 40, i + 60, 80);
}
```

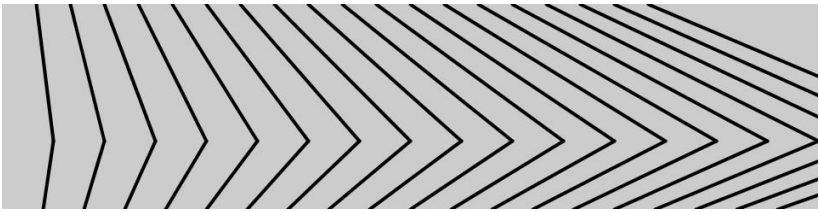
Example 4-8: Fanning Out the Lines



```
size(480, 120);
strokeWeight(2);
```

```
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
}
```

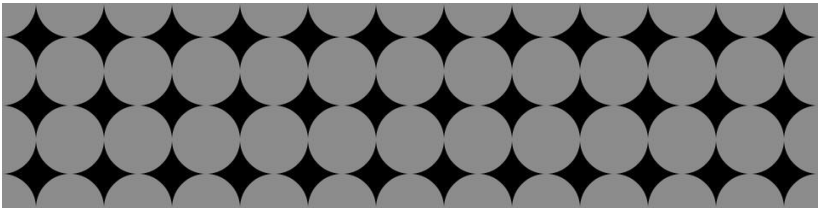
Example 4-9: Kinking the Lines



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
  line(i + i/2, 80, i*1.2, 120);
}
```

Example 4-10: Embed One for Loop in Another

When one `for` loop is embedded inside another, the number of repetitions is multiplied. First, let's look at a short example, and then we'll break it down in Example 4-11 on page 45:

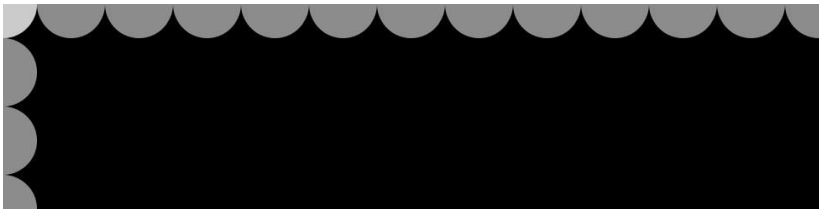


```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y <= height; y += 40) {
  for (int x = 0; x <= width; x += 40) {
    fill(255, 140);
    ellipse(x, y, 40, 40);
```

```
  }
}
```

Example 4-11: Rows and Columns

In this example, the `for` loops are adjacent, rather than one embedded inside the other. The result shows that one `for` loop is drawing a column of 4 circles and the other is drawing a row of 13 circles:

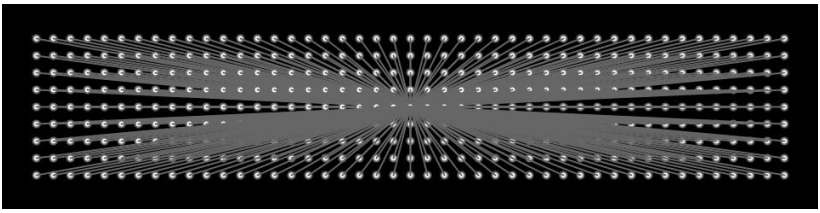


```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y < height+45; y += 40) {
  fill(255, 140);
  ellipse(0, y, 40, 40);
}
for (int x = 0; x < width+45; x += 40) {
  fill(255, 140);
  ellipse(x, 0, 40, 40);
}
```

When one of these `for` loops is placed inside the other, as in Example 4-10 on page 44, the 4 repetitions of the `first` loop are compounded with the 13 of the second in order to run the code inside the embedded block 52 times (4×13 = 52).

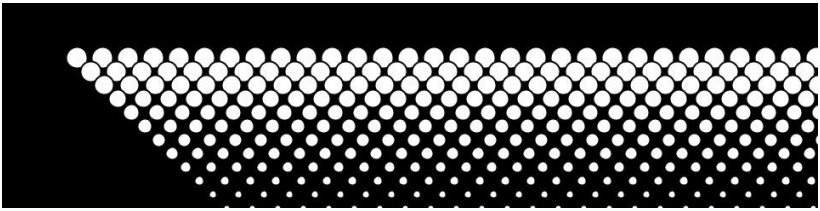
Example 4-10 on page 44 is a good base for exploring many types of repeating visual patterns. The following examples show a couple of ways that it can be extended, but this is only a tiny sample of what's possible. In Example 4-12 on page 46, the code draws a line from each point in the grid to the center of the screen. In Example 4-13 on page 46, the ellipses shrink with each new row and are moved to the right by adding the *y* coordinate to the *x* coordinate.

Example 4-12: Pins and Lines



```
size(480, 120);
background(0);
fill(255);
stroke(102);
for (int y = 20; y <= height-20; y += 10) {
  for (int x = 20; x <= width-20; x += 10) {
    ellipse(x, y, 4, 4);
    // Draw a line to the center of the display
    line(x, y, 240, 60);
  }
}
```

Example 4-13: Halftone Dots



```
size(480, 120);
background(0);
for (int y = 32; y <= height; y += 8) {
  for (int x = 12; x <= width; x += 15) {
    ellipse(x + y, y, 16 - y/10.0, 16 - y/10.0);
  }
}
```