11/Arrays

An *array* is a list of variables that share a common name. Arrays are useful because they make it possible to work with more variables without creating a new name for each. This makes the code shorter, easier to read, and more convenient to update.

From Variables to Arrays

When a program needs to keep track of one or two things, it's not necessary to use an array. In fact, adding an array might make the program more complicated than necessary. However, when a program has many elements (for example, a field of stars in a space game or multiple data points in a visualization), arrays make the code easier to write.

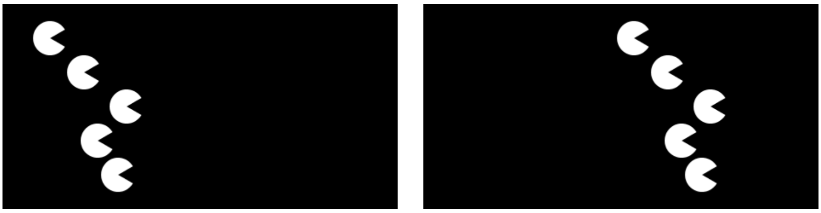
Example 11-1: Many Variables

To see what we mean, refer to Example 8-3 on page 104. This code works fine if we're moving around only one shape, but what if we want to have two? We need to make a new `x` variable and update it within `draw()`:

```
float x1 = -20;
float x2 = 20;

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

void draw() {
  background(0);
  x1 += 0.5;
  x2 += 0.5;
  arc(x1, 30, 40, 40, 0.52, 5.76);
  arc(x2, 90, 40, 40, 0.52, 5.76);
}
```

Example 11-2: Too Many Variables

The code for the previous example is still manageable, but what if we want to have five circles? We need to add three more variables to the two we already have:



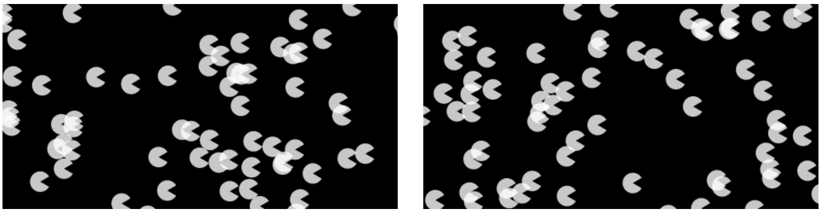
```
float x1 = -10;
float x2 = 10;
float x3 = 35;
float x4 = 18;
float x5 = 30;

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

void draw() {
  background(0);
  x1 += 0.5;
  x2 += 0.5;
  x3 += 0.5;
```

```
    x4 += 0.5;
    x5 += 0.5;
    arc(x1, 20, 20, 20, 0.52, 5.76);
    arc(x2, 40, 20, 20, 0.52, 5.76);
    arc(x3, 60, 20, 20, 0.52, 5.76);
    arc(x4, 80, 20, 20, 0.52, 5.76);
    arc(x5, 100, 20, 20, 0.52, 5.76);
  }
```

This code is starting to get out of control.

Example 11-3: Arrays, Not Variables

Imagine what would happen if you wanted to have 3,000 circles. This would mean creating 3,000 individual variables, then updating each one separately. Could you keep track of that many variables? Would you want to? Instead, we use an array:



```
  float[] x = new float[3000];

  void setup() {
    size(240, 120);
    noStroke();
    fill(255, 200);
    for (int i = 0; i < x.length; i++) {
      x[i] = random(-1000, 200);
    }
  }

  void draw() {
    background(0);
    for (int i = 0; i < x.length; i++) {
      x[i] += 0.5;
      float y = i * 0.4;
      arc(x[i], y, 12, 12, 0.52, 5.76);
    }
  }
```

We'll spend the rest of this chapter talking about the details that make this example possible.

Make an Array

Each item in an array is called an *element*, and each has an *index* value to mark its position within the array. Just like coordinates on the screen, index values for an array start counting from 0. For instance, the first element in the array has the index value 0, the second element in the array has the index value 1, and so on. If there are 20 values in the array, the index value of the last element is 19. Figure 11-1 shows the conceptual structure of an array.

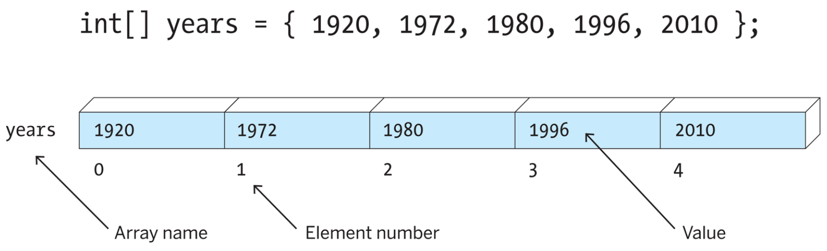


Figure 11-1. *An array is a list of one or more variables that share the same name*

Using arrays is similar to working with single variables; it follows the same patterns. As you know, you can make a single integer variable called `x` with this code:

```
int x;
```

To make an array, just place brackets after the data type:

```
int[] x;
```

The beauty of creating an array is the ability to make 2, 10, or 100,000 variable values with only one line of code. For instance, the following line creates an array of 2,000 integer variables:

```
int[] x = new int[2000];
```

You can make arrays from all Processing data types: `boolean`, `float`, `String`, `PShape`, and so on, as well as any user-defined `class`. For example, the following code creates an array of 32 `PImage` variables:

```
PImage[] images = new PImage[32];
```

To make an array, start with the name of the data type, followed by the brackets. The name you select for the array is next, followed by the assignment operator (the equal symbol), followed by the `new` keyword, followed by the name of the data type again, with the number of elements to create within the brackets. This pattern works for arrays of all data types.

Each array can store only one type of data (`boolean`, `int`, `float`, `PImage`, etc.). You can't mix and match different types of data within a single array. If you need to do this, work with objects instead.

Before we get ahead of ourselves, let's slow down and talk about working with arrays in more detail. Like making an object, there are three steps to working with an array:

1. Declare the array and define the data type.
2. Create the array with the keyword `new` and define the length.
3. Assign values to each element.

Each step can happen on its own line, or all the steps can be compressed together. Each of the three following examples shows a different technique to create an array called `x` that stores two integers, 12 and 2. Pay close attention to what happens before `setup()` and what happens within `setup()`.

Example 11-4: Declare and Assign an Array

First, we'll declare the array outside of `setup()` and then create and assign the values within. The syntax `x[0]` refers to the first element in the array and `x[1]` is the second:

```
int[] x;            // Declare the array

void setup() {
  size(200, 200);
  x = new int[2];   // Create the array
  x[0] = 12;        // Assign the first value
  x[1] = 2;         // Assign the second value
}
```

Example 11-5: Compact Array Assignment

Here's a slightly more compact example, in which the array is both declared and created on the same line, then the values are assigned within `setup()`:

```
int[] x = new int[2];  // Declare and create the array

void setup() {
  size(200, 200);
  x[0] = 12;           // Assign the first value
  x[1] = 2;            // Assign the second value
}
```

Example 11-6: Assigning to an Array in One Go

You can also assign values to the array when it's created, if it's all part of a single statement:

```
int[] x = { 12, 2 };  // Declare, create, and assign

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

Avoid creating arrays within `draw()`, because creating a new array on every frame will slow down your frame rate.

Example 11-7: Revisiting the First Example

As a complete example of how to use arrays, we've recoded Example 11-1 on page 149 here. Although we don't yet see the full benefits revealed in Example 11-3 on page 151, we do see some important details of how arrays work:

```
float[] x = {-20, 20};

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

void draw() {
  background(0);
  x[0] += 0.5;  // Increase the first element
  x[1] += 0.5;  // Increase the second element
  arc(x[0], 30, 40, 40, 0.52, 5.76);
  arc(x[1], 90, 40, 40, 0.52, 5.76);
}
```

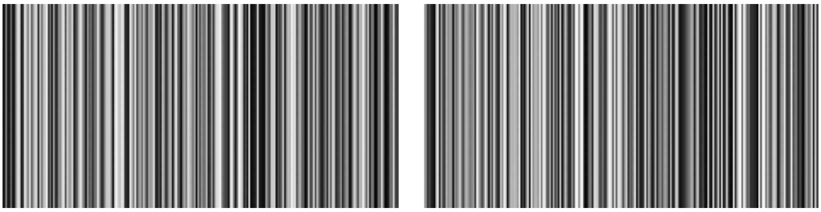
Repetition and Arrays

The `for` loop, introduced in "Repetition" on page 40, makes it easier to work with large arrays while keeping the code concise. The idea is to write a loop to move through each element of the array one by one. To do this, you need to know the length of the array. The `length` field associated with each array stores the number of elements. We use the name of the array with the dot operator (a period) to access this value. For instance:

```
int[] x = new int[2];      // Declare and create the array
println(x.length);         // Prints 2 to the Console

int[] y = new int[1972];   // Declare and create the array
println(y.length);         // Prints 1972 to the Console
```

Example 11-8: Filling an Array in a for Loop

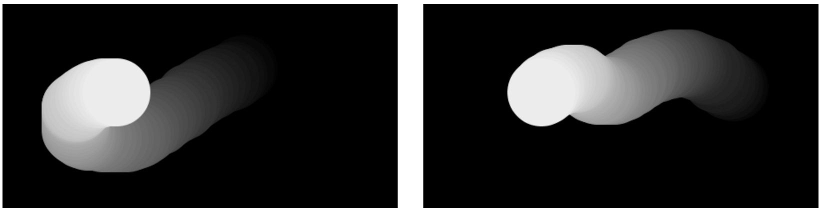
A `for` loop can be used to fill an array with values, or to read the values back out. In this example, the array is first filled with random numbers inside `setup()`, and then these numbers are used to set the stroke value inside `draw()`. Each time the program is run, a new set of random numbers is put into the array:



```
float[] gray;

void setup() {
  size(240, 120);
  gray = new float[width];
  for (int i = 0; i < gray.length; i++) {
    gray[i] = random(0, 255);
 }
}

void draw() {
  for (int i = 0; i < gray.length; i++) {
    stroke(gray[i]);
    line(i, 0, i, height);
  }
}
```

Example 11-9: Track Mouse Movements

In this example, there are two arrays to store the position of the mouse—one for the *x* coordinate and one for the *y* coordinate. These arrays store the location of the mouse for the previous 60 frames. With each new frame, the oldest *x* and *y* coordinate values are removed and replaced with the current `mouseX` and `mouseY` values. The new values are added to the first position of the array, but before this happens, each value in the array is moved one position to the right (from back to front) to make room for the new numbers. This example visualizes this action. Also, at each frame, all 60 coordinates are used to draw a series of ellipses to the screen:



```
int num = 60;
int[] x = new int[num];
int[] y = new int[num];

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

void draw() {
  background(0);
  // Copy array values from back to front
  for (int i = x.length-1; i > 0; i--) {
    x[i] = x[i-1];
    y[i] = y[i-1];
  }
  x[0] = mouseX;  // Set the first element
  y[0] = mouseY;  // Set the first element
  for (int i = 0; i < x.length; i++) {
    fill(i * 4);
    ellipse(x[i], y[i], 40, 40);
  }
}
```

The technique for storing a shifting buffer of numbers in an array shown in this example and Figure 11-2 is less efficient than an alternative technique that uses the % (modulo) operator. This is explained in the Examples → Basics → Input → StoringInput example included with Processing.

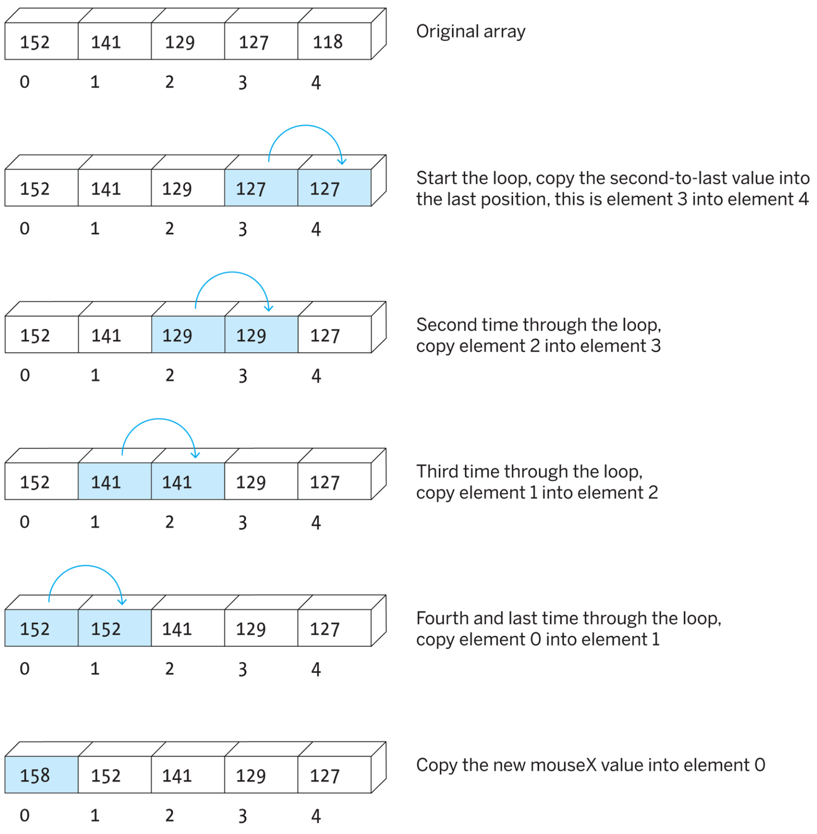

152
141
129
127
118
0
1
2
3
4
Original array
152
141
129
127
127
Start the loop, copy the second-to-last value into the last position, this is element 3 into element 4
152
141
129
129
127
Second time through the loop, copy element 2 into element 3
152
141
141
129
127
Third time through the loop, copy element 1 into element 2
152
152
141
129
127
Fourth and last time through the loop, copy element 0 into element 1
158
152
141
129
127
Copy the new mouseX value into element 0

Figure 11-2. *Shifting the values in an array one place to the right*

Arrays of Objects

The two short examples in this section bring together every major programming concept in this book: variables, iteration, conditionals, functions, objects, and arrays. Making an array of objects is nearly the same as making the arrays we introduced on the previous pages, but there's one additional consideration: because each array element is an object, it must first be created with the keyword `new` (like any other object) before it is assigned to the array. With a custom-defined `class` such as `JitterBug` (see Chapter 10), this means using `new` to set up each element before it's assigned to the array. Or, for a built-in Processing class such as PImage, it means using the loadImage() function to create the object before it's assigned.

Example 11-10: Managing Many Objects

This example creates an array of 33 JitterBug objects and then updates and displays each one inside draw(). For this example to work, you need to add the JitterBug class to the code:

```
JitterBug[] bugs = new JitterBug[33];

void setup() {
  size(240, 120);
  for (int i = 0; i < bugs.length; i++) {
    float x = random(width);
    float y = random(height);
    int r = i + 2;
    bugs[i] = new JitterBug(x, y, r);
  }
}

void draw() {
  for (int i = 0; i < bugs.length; i++) {
    bugs[i].move();
    bugs[i].display();
  }
}

// Insert JitterBug class from Example 10-1
```

Example 11-11: A New Way to Manage Objects

When working with arrays of objects, there's a different kind of loop to use called an "enhanced" for loop. Instead of creating a new counter variable, such as the `i` variable in Example 11-10 on page 159, it's possible to iterate over the elements of an array or list directly. In the following example, each object in the `bugs` array of `JitterBug` objects is assigned to `b` in order to run the `move()` and `display()` methods for all objects in the array.

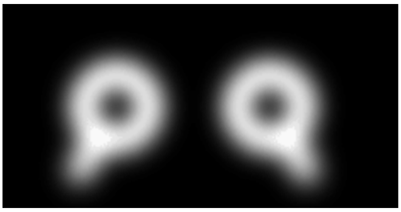
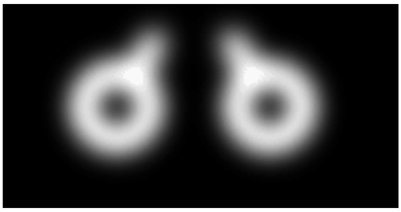
The enhanced `for` loop is often tidier than looping with a number, although in this example, we didn't use it inside `setup()` because `i` was needed in two places inside the loop, demonstrating how sometimes it's helpful to have the number around:

```
JitterBug[] bugs = new JitterBug[33];

void setup() {
  size(240, 120);
  for (int i = 0; i < bugs.length; i++) {
    float x = random(width);
    float y = random(height);
    int r = i + 2;
    bugs[i] = new JitterBug(x, y, r);
  }
}

void draw() {
  for (JitterBug b : bugs) {
    b.move();
    b.display();
  }
}

// Insert JitterBug class from Example 10-1
```

The final array example loads a sequence of images and stores each as an element within an array of `PImage` objects.

Example 11-12: Sequences of Images

To run this example, get the images from the *media.zip* file as described in Chapter 7. The images are named sequentially (*frame-0000.png*, *frame-0001.png*, and so forth), which makes it possible to create the name of each file within a `for` loop, as seen in the eighth line of the program:

```
int numFrames = 12;  // The number of frames
PImage[] images = new PImage[numFrames];  // Make the array
int currentFrame = 0;

void setup() {
  size(240, 120);
  for (int i = 0; i < images.length; i++) {
    String imageName = "frame-" + nf(i, 4) + ".png";
    images[i] = loadImage(imageName);  // Load each image
  }
  frameRate(24);
}


void draw() {
  image(images[currentFrame], 0, 0);
  currentFrame++;       // Next frame
  if (currentFrame >= images.length) {
    currentFrame = 0;  // Return to first frame
  }
}
```

The `nf()` function formats numbers so that `nf(1, 4)` returns the string “0001” and `nf(11, 4)` returns “0011”. These values are concatenated with the beginning of the filename (*frame-*) and the end (*.png*) to create the complete filename as a `String` variable. The files are loaded into the array on the following line. The images are displayed to the screen one at a time in `draw()`. When the last image in the array is displayed, the program returns to the beginning of the array and shows the images again in sequence.