9 Arrays

I might repeat to myself slowly and soothingly, a list of quotations beautiful from minds profound—if I can remember any of the damn things.
—Dorothy Parker

In this chapter:
– What is an array?
– Declaring an array
– Initializing an array
– Array operations – using a `for` loop with an array
– Arrays of objects

9-1 Arrays, why do you care?

Let's take a moment to revisit the car example from the previous chapter on object-oriented programming. You may remember I spent a great deal of effort on developing a program that contained multiple instances of a class, that is, two objects.

```
Car myCar1;
Car myCar2;
```

This was indeed an exciting moment in the development of your life as a computer programmer. It's likely, however, that you're contemplating a somewhat obvious question. How could you take this further and write a program with 100 `Car` objects? With some clever copying and pasting, you might write a program with the following beginning:

```
Car myCar1
Car myCar2
Car myCar3
Car myCar4
Car myCar5
Car myCar6
Car myCar7
Car myCar8
Car myCar9
Car myCar10
Car myCar11
Car myCar12
Car myCar13
Car myCar14
Car myCar15
Car myCar16
Car myCar17
Car myCar18
Car myCar19
```

```
Car myCar20
Car myCar21
Car myCar22
Car myCar23
Car myCar24
Car myCar25
Car myCar26
Car myCar27
Car myCar28
Car myCar29
Car myCar30
Car myCar31
Car myCar32
Car myCar33
Car myCar34
Car myCar35
Car myCar36
Car myCar37
Car myCar38
Car myCar39
Car myCar40
Car myCar41
Car myCar42
Car myCar43
Car myCar44
Car myCar45
Car myCar46
Car myCar47
Car myCar48
Car myCar49
Car myCar50
Car myCar51
Car myCar52
Car myCar53
Car myCar54
Car myCar55
Car myCar56
Car myCar57
Car myCar58
Car myCar59
Car myCar60
Car myCar61
Car myCar62
Car myCar63
Car myCar64
Car myCar65
Car myCar66
Car myCar67
```

```
Car myCar68
Car myCar69
Car myCar70
Car myCar71
Car myCar72
Car myCar73
Car myCar74
Car myCar75
Car myCar76
Car myCar77
Car myCar78
Car myCar79
Car myCar80
Car myCar81
Car myCar82
Car myCar83
Car myCar84
Car myCar85
Car myCar86
Car myCar87
Car myCar88
Car myCar89
Car myCar90
Car myCar91
Car myCar92
Car myCar93
Car myCar94
Car myCar95
Car myCar96
Car myCar97
Car myCar98
Car myCar99
Car myCar100
```

If you really want to give yourself a headache, try completing the rest of the program modeled after the above start. It will not be a pleasant endeavor. I am certainly not about to leave you any workbook space in this book to practice.

An array will allow you to take these 100 lines of code and put them into one line. Instead of having 100 variables, an array is *one* thing that contains a *list* of variables.

Any time a program requires multiple instances of similar data, it might be time to use an array. For example, an array can be used to store the scores of four players in a game, a selection of 10 colors in a design program, or a list of fish objects in an aquarium simulation.

Exercise 9-1: Looking at all of the sketches you have created so far, do any merit the use of an array? Why?

9-2 What is an array?

From Chapter 4, you may recall that a variable is a named pointer to a location in memory where data is stored. In other words, variables allow programs to keep track of information over a period of time. An array is exactly the same, only instead of pointing to one singular piece of information, an array points to multiple pieces. See Figure 9-1.

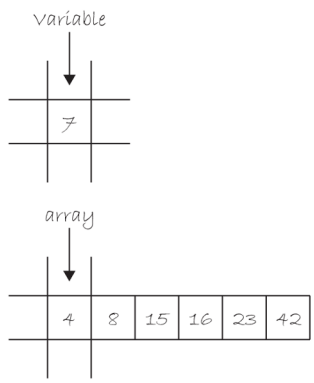

Figure 9-1

You can think of an array as a list of variables. A list, it should be noted, is useful for two important reasons. Number one, the list keeps track of the elements in the list themselves. Number two, the list keeps track of *the order* of those elements (which element is the first in the list, the second, the third, etc.). This is a crucial point since in many programs, the order of information is just as important as the information itself.

In an array, each element of the list has a unique *index,* an integer value that designates its position in the list (element #1, element #2, etc.). In all cases, the name of the array refers to the list as a whole, while each element is accessed via its position.

Notice how in Figure 9-2, the indices range from 0 to 9. The array has a total of 10 elements, but the first element number is 0 and the last element is 9. You might be tempted to stomp your feet and complain: "Hey, why aren't the elements numbered from 1 to 10? Wouldn't that be easier?"

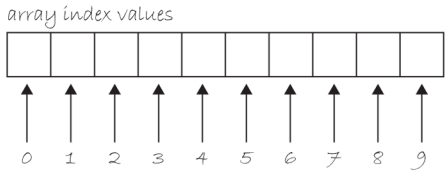

Figure 9-2

While at first, it might intuitively seem like I should start counting at 1 (and some programming languages do), I start at 0 because technically the first element of the array is located at the start of the array, a distance of zero from the beginning. Numbering the elements starting at 0 also makes many *array operations* (the process of executing a line of code for every element of the list) a great deal more convenient. As I continue through several examples, you will begin to believe in the power of counting from zero.

Exercise 9-2: If you have an array with 1,000 elements, what is the range of index values for that array?

Answer: _____ through _____

9-3 Declaring and creating an array

In Chapter 4, you learned that all variables must have a name and a data type. Arrays are no different. The declaration statement, however, does look different. You denote the use of an array by placing empty square brackets (`[]`) after the type declaration. Let's start with an array of primitive values, for example, integers. (You can have arrays of any data type, and I will soon show how you can make an array of objects.) See Figure 9-3.

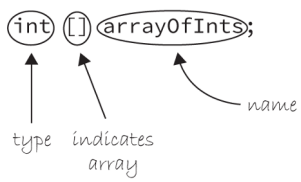

Figure 9-3

The declaration in Figure 9-3 indicates that `arrayOfInts` will store a list of integers. The array name `arrayOfInts` can be absolutely anything you want it to be (I only include the word "array" here to illustrate what you are learning).

One fundamental property of arrays, however, is that they are of fixed size. Once I define the size for an array, it can never change. A list of 10 integers can never *go to 11*. But where in the above code is the size

of the array defined? It is not. The code simply declares the array; I must also make sure I *create* the actual instance of the array with a specified size.

To do this, I use the `new` operator, in a similar manner as I did in calling the constructor of an object. In the object's case, I am saying "Make a *new* Car" or "Make a *new* Zoog." With an array, I am saying "Make a *new* array of integers," or "Make a *new* array of Car objects," and so on. See array declaration in Figure 9-4.

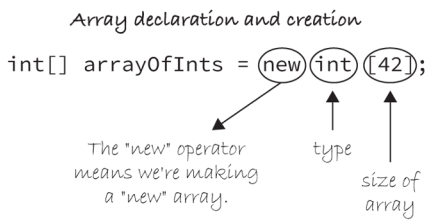

Figure 9-4

The array declaration in Figure 9-4 allows me to specify the array size: how many elements I want the array to hold (or, technically, how much memory in the computer I am asking for to store my beloved data). I write this statement as follows: the `new` operator, followed by the data type, followed by the size of the array enclosed in brackets. This size must be an integer. It can be a hard-coded number, a variable (of type integer), or an expression that evaluates to an integer (like 2 + 2).

Example 9-1. Additional array declaration and creation examples

```
float[] scores = new float[4];                  // A list of 4 floating point numbers
Human[] people = new Human[100];                // A list of 100 Human objects
int num = 50;
Car[] cars = new Car[num];                      // Using a variable to specify size
Spaceship[] ships = new Shapeship[num*2 + 3];   // Using an expression to
                                                    specify size
```

Exercise 9-3: Write the declaration statements for the following arrays:

30 integers: _____

100 floating point numbers: _____

56 Zoog objects: _____

Exercise 9-4: Which of the following array declarations are valid and which are invalid (and why)?

```
int[] numbers = new int[10];          ____________

float[] numbers = new float[5 + 6];   ____________

int num = 5;
float[] numbers = new int[num];       ____________

float num = 5.2;
Car[] cars = new Car[num];            ____________

int num = (5 * 6)/2;
float[] numbers = new float[num = 5]; ____________

int num = 5;
Zoog[] zoogs = new Zoog[num * 10];    ____________
```

Things are looking up. Not only did I successfully declare the existence of an array, but I have given it a size and allocated physical memory for the stored data. A major piece is missing, however: the data stored in the array itself!

9-4 Initializing an array

One way to fill an array is to hard-code the values stored in each spot of the array.

Example 9-2. Initializing the elements of an array one at a time

```
int[] stuff = new int[3];

stuff[0] = 8; // The first element of the array equals 8
stuff[1] = 3; // The second element of the array equals 3
stuff[2] = 1; // The third element of the array equals 1
```

As you can see, each element of the array is referred to individually by specifying an index, starting at 0. The syntax for this is the name of the array, followed by the index value enclosed in brackets.

```
arrayName[INDEX]
```

A second option for initializing an array is to manually type out a list of values enclosed in curly braces and separated by commas.

Example 9-3. Initializing the elements of an array all at once

```
int[] arrayOfInts = { 1, 5, 8, 9, 4, 5 } ;
float[] floatArray = { 1.2, 3.5, 2.0, 3.4123, 9.9 } ;
```

Exercise 9-5: Declare an array of three `Zoog` *objects. Initialize each spot in the array with a Zoog object via its index.*

```
Zoog__ zoogs = new _______[___];

_______[_______] = _______ _______(100, 100, 50, 60, 16);

_______[_______] = _______ _______(____________________);

_______[_______] = _______ _______(____________________);
```

Both of these approaches are not commonly used and you will not see them in most of the examples throughout the book. In fact, neither initialization method has really solved the problem posed at the beginning of the chapter. Imagine initializing each element individually with a list of 100 or (gasp) 1,000 or (gasp gasp!) 1,000,000 elements.

The solution to all of your woes involves a means for *iterating* through the elements of the array. Ding ding ding. Hopefully a loud bell is ringing in your head. Loops! (If you're lost, revisit Chapter 6.)

9-5 Array operations

Consider, for a moment, the following problem:

1. Create an array of 1,000 floating point numbers.
2. Initialize every element of that array with a random number between 0 and 10.

Part 1 you already know how to do.

```
float[] values = new float[1000];
```

What I want to avoid is having to do this for Part 2:

```
values[0] = random(0, 10);
values[1] = random(0, 10);
values[2] = random(0, 10);
values[3] = random(0, 10);
values[4] = random(0, 10);
values[5] = random(0, 10);
// etc. etc.
```

Let's describe in English what I want to program.

For every number *n* from 0 to 999, initialize the *nth* element stored in array as a random value between 0 and 10. Translating into code, I have:

```
int n = 0;
values[n] = random(0, 10);
values[n + 1] = random(0, 10);
values[n + 2] = random(0, 10);
values[n + 3] = random(0, 10);
values[n + 4] = random(0, 10);
values[n + 5] = random(0, 10);
```

Unfortunately, the situation has not improved. I have, nonetheless, taken a big leap forward. By using a variable (`n`) to describe an index in the array, I can now employ a `while` loop to initialize every `n` element.

Example 9-4. Using a `while` loop to initialize all elements of an array

```
int n = 0;
while (n < 1000) {
  values[n] = random(0, 10);
  n = n + 1;
}
```

A `for` loop allows you to be even more concise, as Example 9-5 shows.

Example 9-5. Using a `for` loop to initialize all elements of an array

```
for (int n = 0; n < 1000; n++) {
  values[n] = random(0, 10);
}
```

What was once 1,000 lines of code is now three!

I can exploit the same technique for any type of array operation I might like to do beyond simply initializing the elements. For example, I could take the array and double the value of each element. (I will use `i` from now on instead of `n` as it is more commonly used by programmers.)

Example 9-6. An array operation

```
for (int i = 0; i < 1000; i++) {
  values[i] = values[i] * 2;
}
```

There is one problem with Example 9-6: the use of the hard-coded value 1,000. Striving to be better programmers, you should always question the existence of a hard-coded number. In this case, what if you wanted to change the array to have 2,000 elements? If your program was very long with many array operations, you would have to make this change everywhere throughout your code. Fortunately, Processing offers a nice means for accessing the size of an array dynamically, using the dot syntax you learned for objects in Chapter 8. `length` is a property of every array and you can access it by saying:

arrayName dot length

Let's use `length` while clearing an array. This will involve resetting every value to 0.

Example 9-7. An array operation using dot length

```
for (int i = 0; i < values.length; i++) {
  values[i] = 0;
}
```

Exercise 9-6: Assuming an array of 10 integers, that is,

```
int[] nums = { 5, 4, 2, 7, 6, 8, 5, 2, 8, 14 };
```

Write code to perform the following array operations (Note that the number of clues vary, just because a `[____]` is not explicitly written in does not mean there should not be brackets).

Square each number (i.e., multiply each by itself)	`for (int i ________; i < ________; i++) {`
	
	`  _______[i] = ____________*____________;`
	`}`

Add a random number between zero and 10 to each number.	`_____________________________________________`
	
	`  __________ += int(___________________);`
	
	`___`

Add to each number the number that follows in the array. Skip the last value in the array.	`for (int i = 0; i < ___________; i++) {`
	
	`  __________ += ________[_______];`
	`}`

Calculate the sum of all the numbers.	`_______ ___________ = ______;`
	`for (int i = 0; i < nums.length; i++) {`
	
	`  __________ += _________________;`
	`}`

9-6 Simple array example: the snake

A seemingly trivial task, programming a trail following the mouse, is not as easy as it might initially appear. The solution requires an array, which will serve to store the history of mouse locations. I will use two arrays, one to store horizontal mouse locations, and one for vertical. Let's say, arbitrarily, that I want to store the last 50 mouse locations.

First, I declare the two arrays.

```
int[] xpos = new int[50];
int[] ypos = new int[50];
```

Second, in `setup()`, I must initialize the arrays. Since at the start of the program there has not been any mouse movement, I will just fill the arrays with 0's.

```
for (int i = 0; i < xpos.length; i++) {
  xpos[i] = 0;
  ypos[i] = 0;
}
```

Each time through the main `draw()` loop, I want to update the array with the current mouse location. Let's choose to put the current mouse location in the last spot of the array. The length of the array is 50, meaning index values range from 0–49. The last spot is index 49, or the length of the array minus one.

```
xpos[xpos.length - 1] = mouseX;
ypos[ypos.length - 1] = mouseY;
```

The last spot in an array is length minus one.

Now comes the hard part. I want to keep only the last 50 mouse locations. By storing the current mouse location at the end of the array, I am overwriting what was previously stored there. If the mouse is at (10,10) during one frame and (15,15) during another, I want to put (10,10) in the second to last spot and (15,15) in the last spot. A solution is to shift all of the elements of the array down one spot before updating the current location. This is shown in Figure 9-5.

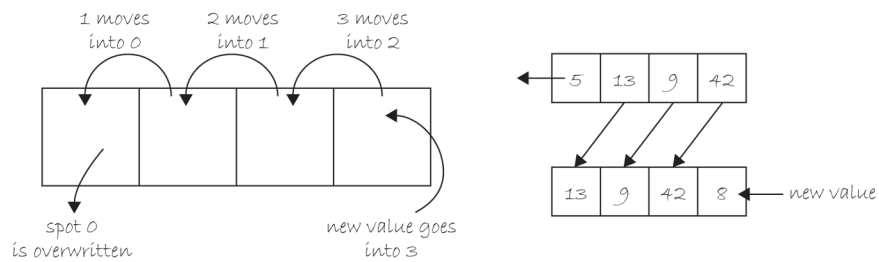

Figure 9-5

Element index 49 moves into spot 48, 48 moves into spot 47, 47 into 46, and so on. I can do this by looping through the array and setting each element index `i` to the value of element `i+1`. Note I must stop at the second to last value since for element 49 there is no element 50 (49 plus 1). In other words, instead of having an exit condition

```
i < xpos.length;
```

I must instead say:

```
i < xpos.length - 1;
```

The full code for performing this array shift is as follows:

```
for (int i = 0; i < xpos.length - 1; i++) {
  xpos[i] = xpos[i + 1];
  ypos[i] = ypos[i + 1];
}
```

Finally, I can use the history of mouse locations to draw a series of circles. For each element of the xpos array and ypos array, draw an ellipse at the corresponding values stored in the array.

```
for (int i = 0; i < xpos.length; i++) {
  stroke(0);
  fill(175);
  ellipse(xpos[i], ypos[i], 32, 32);
}
```

Making this a bit fancier, you might choose to link the brightness of the circle as well as the size of the circle to the location in the array, that is, the earlier (and therefore older) values will be bright and small and the later (newer) values will be darker and bigger. This is accomplished by using the counting variable `i` to evaluate color and size.

```
for (int i = 0; i < xpos.length; i++) {
   noStroke();
   fill(255 - i * 5);
   ellipse(xpos[i], ypos[i], i, i);
}
```

Putting all of the code together, I have the following example, with the output shown in Figure 9-6.

Example 9-8. A snake following the mouse

```
// x and y positions
int[] xpos = new int[50];
int[] ypos = new int[50];

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

  // Initialize
  for (int i = 0; i < xpos.length; i++) {
    xpos[i] = 0;
    ypos[i] = 0;
  }
}

void draw() {
  background(255);

  // Shift array values
  for (int i = 0; i < xpos.length - 1; i++) {
    xpos[i] = xpos[i + 1];
    ypos[i] = ypos[i + 1];
  }

  // New location
  xpos[xpos.length - 1] = mouseX;
  ypos[ypos.length - 1] = mouseY;

  // Draw everything
  for (int i = 0; i < xpos.length; i++) {
    noStroke();
    fill(255 - i*5);
    ellipse(xpos[i], ypos[i], i, i);
  }
}
```

Declare two arrays with 50 elements.

Initialize all elements of each array to zero.

Shift all elements down one spot. xpos[0] = xpos[1], xpos[1] = xpos = [2], and so on. Stop at the second to last element.

Update the last spot in the array with the mouse location.

Draw an ellipse for each element in the arrays. Color and size are tied to the loop's counter: `i`.

Figure 9-6

Exercise 9-7: Rewrite the snake example in an object-oriented fashion with a `Snake` *class. Can you make snakes with slightly different looks (different shapes, colors, sizes)? (For an advanced problem, create a* `Point` *class that stores an x and y coordinate as part of the sketch. Each snake object will have an array of Point objects, instead of two separate arrays of x and y values. This involves arrays of objects, covered in the next section.)*

9-7 Arrays of objects

I know, I know. I still have not fully answered the question. How can you write a program with 100 car objects?

One of the nicest features of combining object-oriented programming with arrays is the simplicity of transitioning a program from one object to 10 objects to 10,000 objects. In fact, if I have been careful, I will not have to change the `Car` class whatsoever. A class does not care how many objects are made from

it. So, assuming I keep the identical `Car` class code, let's look at how to expand the main program to use an array of objects instead of just one.

Let's revisit the main program for one `Car` object.

```
Car myCar;

void setup() {
  myCar = new Car(color(255, 0, 0), 0, 100, 2);
}

void draw() {
  background(255);
  myCar.move();
  myCar.display();
}
```

There are three steps in the above code and each one needs to be changed to account for an array.

Before	After
`// Declare the car` `Car myCar;`	`// Declare the car array` `Car[] cars = new Car[100];`

`// Initialize the car`
`myCar = new Car(color(255), 0, 100, 2);`
	`// Initialize each element of the array`
	`for (int i = 0; i < cars.length; i++) {`
	`  cars[i] = new Car(color(i*2), 0, i*2, i);`
	`}`

`// Run the car by calling methods`
`myCar.move();`
`myCar.display();`
	`// Run each element of the array`
	`for (int i = 0; i < cars.length; i++) {`
	`  cars[i].move();`
	`  cars[i].display();`
	`}`

This leaves you with Example 9-9. Note how changing the number of cars present in the program requires only altering the array definition. Nothing else anywhere has to change!

Example 9-9. An array of Car objects

```
Car[] cars = new Car[100];

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

  for (int i = 0; i < cars.length; i++) {
    cars[i] = new Car(color(i*2), 0, i*2, i/20.0);
  }
}


void draw() {

  background(255);

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

class Car {

  color c;
  float xpos;
  float ypos;
  float xspeed;

  Car(color c_, float xpos_, float ypos_, float xspeed_) {
    c = c_;
    xpos = xpos_;
    ypos = ypos_;
    xspeed = xspeed_;
  }

  void display() {
    rectMode(CENTER);
    stroke(0);
    fill(c);
    rect(xpos, ypos, 20, 10);
  }

  void move() {
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }
  }

}
```

An array of 100 `Car` objects!

Initialize each car using a for loop.

Run each car using a for loop.

The `Car` class does not change whether you are making one car, 100 cars or 1,000 cars!

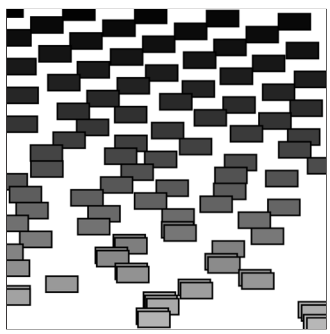

Figure 9-7

9-8 Interactive objects

When you first learned about variables (Chapter 4) and conditionals (Chapter 5), you programmed a simple rollover effect. A rectangle appears in the window and is one color when the mouse is on top and another color when the mouse is not. The following is an example that takes this simple idea and puts it into a `Stripe` class. Even though there are 10 stripes, each one individually responds to the mouse by having its own `rollover()` function.

```
void rollover(int mx, int my) {
  if (mx > x & & mx < x + w) {
    mouse = true;
  } else {
    mouse = false;
  }
}
```

This function checks to see if a point (`mx`,`my`) is contained within the vertical stripe. Is it greater than the left edge and less than the right edge? If so, a boolean variable `mouse` is set to true. When designing your classes, it's often convenient to use a boolean variable to keep track of properties of an object that resemble a switch. For example, a `Car` object could be running or not running. Zoog could be happy or not happy.

This boolean variable is used in a conditional statement inside of the `Stripe` object's `display()` function to determine the stripe's color.

```
void display() {
  if (mouse) {
    fill(255);
  } else {
    fill(255, 100);
  }
  noStroke();
  rect(x, 0, w, height);
}
```

When I call the `rollover()` function on that object, I can then pass in `mouseX` and `mouseY` as arguments.

```
stripes[i].rollover(mouseX, mouseY);
```

Even though I could have accessed `mouseX` and `mouseY` directly inside of the `rollover()` function, it's better to use arguments. This allows for greater flexibility. The `Stripe` object can check and determine if any (x,y) coordinate is contained within its rectangle. Perhaps later, I will want the stripe to turn white when another object, rather than the mouse, is over it.

Here is the full "interactive stripes" example.

Example 9-10. Interactive stripes

```
Stripe[] stripes = new Stripe[10];

void setup() {
  size(200, 200);
  for (int i = 0; i < stripes.length; i++) {
    stripes[i] = new Stripe();
  }
}

void draw() {
  background(100);
  // Move and display all stripes
  for (int i = 0; i < stripes.length; i++) {
    stripes[i].rollover(mouseX, mouseY);
    stripes[i].move();
    stripes[i].display();
  }
}

class Stripe {
  float x;        // horizontal location of stripe
  float speed;    // speed of stripe
  float w;        // width of stripe
  boolean mouse;  // Is the mouse over the stripe?

  Stripe() {
    x = 0;               // All stripes start at 0
    speed = random(1);   // All stripes have a random positive speed
    w = random(10, 30);
    mouse = false;
  }

  void display() {
    if (mouse) {
      fill(255);
    } else {
      fill(255, 100);
    }
    noStroke();
    rect(x, 0, w, height);
  }

  void move() {
    x += speed;
    if (x > width + 20) x = -20;
  }

  void rollover(int mx, int my) { {
    // Left edge is x, right edge is x + w
    if (mx > x & & mx < x + w)
      mouse = true;
    } else {
      mouse = false;
    }
  }
}
```

array of `Stripe` objects

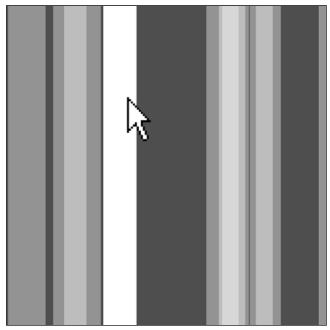

Figure 9-8

Check in the mouse is over a stripe by passing the the mouse coordinates into the `Stripe` class `rollover()` function.

A boolean variable keeps track of the object's state.

That boolean variable determines stripe color.

This function checks to see if the point (mx,my) is inside the stripe (returning `true`) or outside (returning `false`).

Exercise 9-8: Write a `Button` *class (see Example 5-5 for a non-object-oriented button). The* `Button` *class should register when a mouse is pressed over the button and change color. Create buttons of different sizes and locations using an array. Before writing the main program, sketch out the* `Button` *class. Assume the button is off when it first appears. Here is a code framework:*

```
class Button {
  float x;
  float y;
  float w;
  float h;
  boolean on;

  Button(float tempX, float tempY, float tempW, float tempH) {
    x = tempX;
    y = tempY;
    w = tempW;
    h = tempH;

    on = __________;
  }

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------
}
```

9-9 Processing's array functions

OK, so I have a confession to make. I lied. Well, sort of. See, earlier in this chapter, I made a very big point of emphasizing that once you set the size of an array, you can never change that size. Once you have made 10 Button objects, you can't make an 11th.

And I stand by those statements. Technically speaking, when you allocate 10 spots in an array, you have told Processing exactly how much space in memory you intend to use. You can't expect that block of memory to happen to have more space next to it so that you can expand the size of your array.

However, there is no reason why you couldn't just make a new array (one that has 11 spots in it), copy the first 10 from your original array, and pop a new Button object in the last spot. Processing, in fact, offers a set of array functions that manipulate the size of an array by managing this process for you. They are: `shorten()`, `concat()`, `subset()`, `append()`, `splice()`, and `expand()`. In addition, there are functions for changing the order in an array, such as `sort()` and `reverse()`.

Details about all of these functions can be found in the reference. Let's look at one example that uses `append()` to expand the size of an array. This example (which includes an answer to Exercise 8-5 on page 154) starts with an array of one object. Each time the mouse is pressed, a new object is created and appended to the end of the original array.

Example 9-11. Resizing an array using `append()`

```
Ball[] balls = new Ball[1];
float gravity = 0.1;

void setup() {
  size(200, 200);
  // Initialize ball index 0
  balls[0] = new Ball(50, 0, 16);
}

void draw() {
  background(100);
  // Update and display all balls
  for (int i = 0; i < balls.length; i++) {
    balls[i].gravity();
    balls[i].move();
    balls[i].display();
  }
}

void mousePressed() {
  // A new ball object
  Ball b = new Ball(mouseX, mouseY, 10);

     // Append to array
     balls = (Ball[]) append(balls, b);
   }

   class Ball {
     float x;
```

I start with an array with just one element.

Whatever the length of that array, update and display all of the objects.

Make a new object at the mouse location.

The function `append()` adds an element to the end of the array. `append()` takes two arguments. The first is the array you want to append to, and the second is the thing you want to append. You have to reassign the result of the `append()` function to the original array. In addition, the `append()` function requires that you explicitly state the data type for the array again by putting the data type in parentheses: `(Ball[])`. This is known as casting.

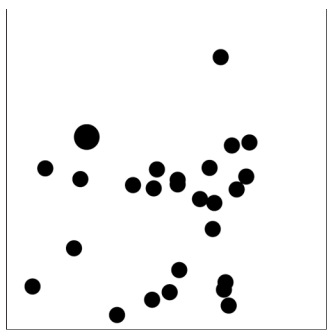

Figure 9-9

```
    float y;
    float speed;
    float w;

    Ball(float tempX, float tempY, float tempW) {
     x = tempX;
     y = tempY;
     w = tempW;
     speed = 0;
    }

    void gravity() {
      // Add gravity to speed
      speed = speed + gravity;
    }

    void move() {
     // Add speed to y location
     y = y + speed;
     // If square reaches the bottom
     // Reverse speed
     if (y > height) {
       speed = speed * -0.95;
       y = height;
     }
    }

    void display() {
    // Display the circle
    fill(255);
    noStroke();
    ellipse(x, y, w, w);
  }
}
```

Another means of having a resizable array is through the use of a special object known as an `ArrayList`, which will be covered in Chapter 23.

9-10 One thousand and one Zoogs

It's time to complete Zoog's journey and look at how to move from one Zoog to many. In the same way that I generated the `Car` array or `Stripe` array example, I can simply copy the exact `Zoog` class created in Example 8-3 and implement an array.

Example 9-12. 200 Zoog objects in an array

```
Zoog[] zoogies = new Zoog[200];

void setup() {
  size(400, 400);
  for (int i = 0; i < zoogies.length; i++) {
    zoogies[i] = new Zoog(random(width), random(height), 30, 30, 8);
  }
```

The only difference between this example and the previous chapter is the use of an array for multiple `Zoog` objects.

```
}

void draw() {
  background(255);
  for (int i = 0; i < zoogies.length; i++) {
    zoogies[i].display();
    zoogies[i].jiggle();
  }
}

class Zoog {

  // Zoog's variables
  float x;
  float y;
  float w;
  float h;
  float eyeSize;

  // Zoog constructor
  Zoog(float tempX, float tempY, float tempW, float tempH, float tempEyeSize) {
    x = tempX;
    y = tempY;
    w = tempW;
    h = tempH;
    eyeSize = tempEyeSize;
  }

  void jiggle() {
    // Change the location
    x = x + random(-1, 1);
    y = y + random(-1, 1);

    // Constrain Zoog to window
    x = constrain(x, 0, width);
    y = constrain(y, 0, height);
  }

  // Display Zoog
  void display() {


    // Draw Zoog's arms with a for loop
    for (float i = y - h/3; i < y + h/2; i += 10) {
      stroke(0);
      line(x - w/4, i, x + w/4, i);
    }

    // Set ellipses and rects to CENTER mode
    ellipseMode(CENTER);
    rectMode(CENTER);

    // Draw Zoog's body
    stroke(0);
    fill(175);
    rect(x, y, w/6, h);

    // Draw Zoog's head
```

For simplicity I have also removed the `speed` parameter from the `jiggle()` function. Try adding it back in as an exercise.

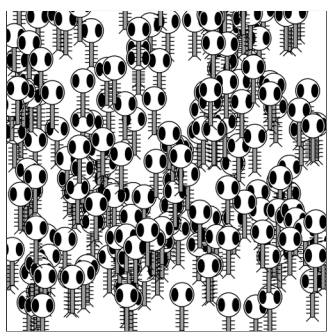

Figure 9-10

```
    stroke(0);
    fill(255);
    ellipse(x, y - h, w, h);

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

    // Draw Zoog's legs
    stroke(0);
    line(x - w/12, y + h/2, x - w/4, y + h/2 + 10);
    line(x + w/12, y + h/2, x + w/4, y + h/2 + 10);
  }
}
```

Lesson Four Project

1. Take the Class you made in Lesson Three and make an array of objects from that class.
2. Can you make the objects react to the mouse? Try using the `dist()` function to determine the object's proximity to the mouse. For example, could you make each object jiggle more the closer it is to the mouse?

How many objects can you make before the sketch runs too slow?

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