Table of Contents

1 Pixels

A journey of a thousand miles begins with a single step.
—Lao-tzu

In this chapter:
– Specifying pixel coordinates
– Basic shapes: point, line, rectangle, ellipse
– Color: grayscale, RGB
– Color: alpha transparency

Note that you are not doing any programming yet in this chapter! You are just dipping your feet in the water and getting comfortable with the idea of creating onscreen graphics with text-based commands, that is, "code"!

1-1 Graph paper

This book will teach you how to program in the context of computational media, and it will use the development environment Processing (*http://www.processing.org*) as the basis for all discussion and examples. But before any of this becomes relevant or interesting, you must first channel your eighth-grade self, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where you will begin, with two points on that graph paper.

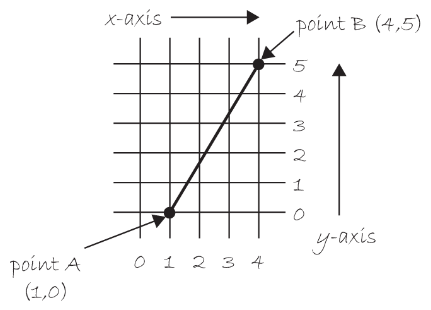


Figure 1-1

Figure 1-1 shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

```
line(1, 0, 4, 5);
```

Congratulations, you have written your first line of computer code! I'll will get to the precise formatting of the above later, but for now, even without knowing too much, it should make a fair amount of sense. I am providing a *command* (which I will refer to as a *function*) named *line* for the machine to follow. In addition, I am specifying some *arguments* for how that line should be drawn, from point A (1,0) to point B (4,5). If you think of that line of code as a sentence, the *function* is a *verb* and the *arguments* are the *objects* of the sentence. The code sentence also ends with a semicolon instead of a period.

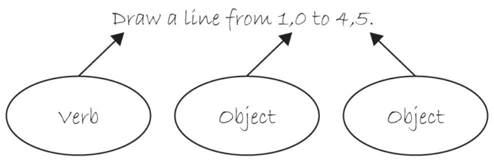


Figure 1-2

The key here is to realize that the computer screen is nothing more than a *fancier* piece of graph paper. Each pixel of the screen is a coordinate — two numbers, an *x* (horizontal) and a *y* (vertical) — that determine the location of a point in space. And it's your job to specify what shapes and colors should appear at these pixel coordinates.

Nevertheless, there is a catch here. The graph paper from eighth grade (*Cartesian coordinate system*) placed (0,0) in the center with the y-axis pointing up and the x-axis pointing to the right (in the positive direction, negative down and to the left). The coordinate system for pixels in a computer window, however, is reversed along the y-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically. See Figure 1-3.

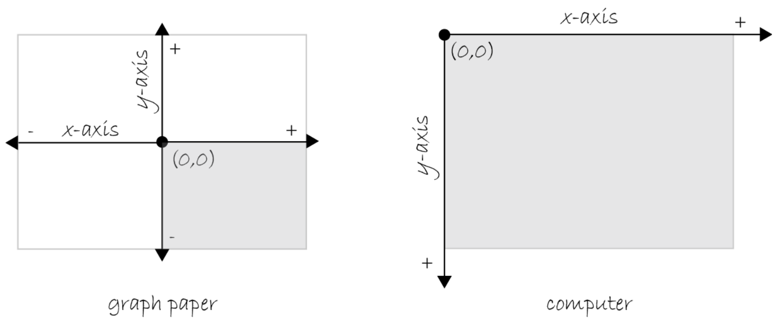


Figure 1-3

Exercise 1-1: Looking at how I wrote the instruction for line — `line(1, 0, 4, 5);` *— how would you guess you would write an instruction to draw a rectangle? A circle? A triangle? Write out the instructions in English and then translate it into code.*

English: ____

Code: ____

English: ____

Code: ____

English: ____

Code: ____

Come back later and see how your guesses matched up with how Processing actually works.

1-2 Simple shapes

The vast majority of the programming examples in this book will be visual in nature. You may ultimately learn to develop interactive games, algorithmic art pieces, animated logo designs, and (insert your own category here) with Processing, but at its core, each visual program will involve setting pixels. The simplest way to get started in understanding how this works is to learn to draw primitive shapes. This is not unlike how you learn to draw in elementary school, only here you do so with code instead of crayons.

I'll start with the four primitive shapes shown in Figure 1-4.

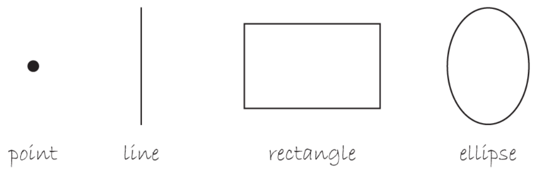


Figure 1-4

For each shape, ask yourself what information is required to specify the location and size (and later color) of that shape and learn how Processing expects to receive that information. In each of the diagrams below (Figure 1-5 through Figure 1-11), assume a window with a width of ten pixels and height of ten pixels. This isn't particularly realistic since when you really start coding you will most likely work with much larger windows (ten by ten pixels is barely a few millimeters of screen space). Nevertheless, for demonstration purposes, it's nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.

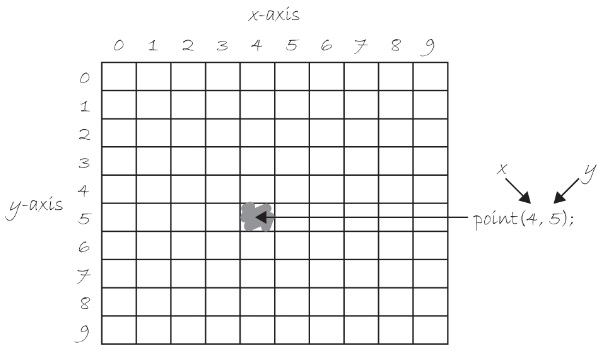


Figure 1-5

A point is the easiest of the shapes and a good place to start. To draw a point, you only need an (x,y) coordinate as shown in Figure 1-5. A line isn't terribly difficult either. A line requires two points, as shown in Figure 1-6.

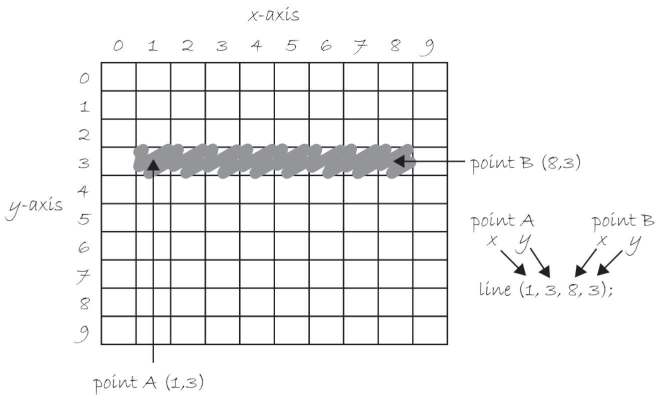


Figure 1-6

Once you arrive at drawing a rectangle, things become a bit more complicated. In Processing, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height (see Figure 1-7).

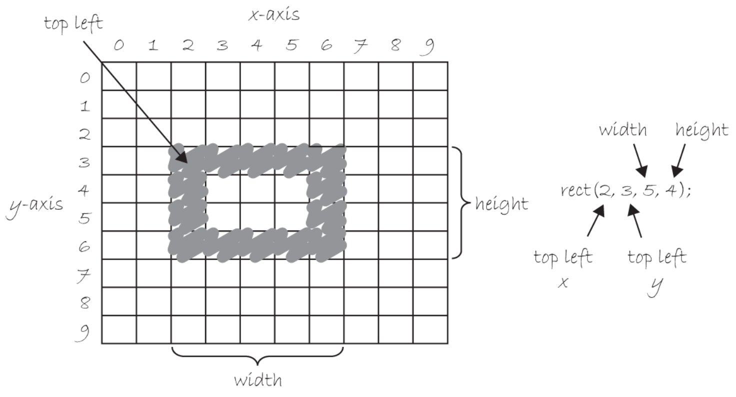


Figure 1-7

However, a second way to draw a rectangle involves specifying the centerpoint, along with width and height as shown in Figure 1-8. If you prefer this method, you first indicate that you want to use the

CENTER mode before the instruction for the rectangle itself. Note that Processing is case-sensitive. Incidentally, the default mode is CORNER, which is how I began as illustrated in Figure 1-7.

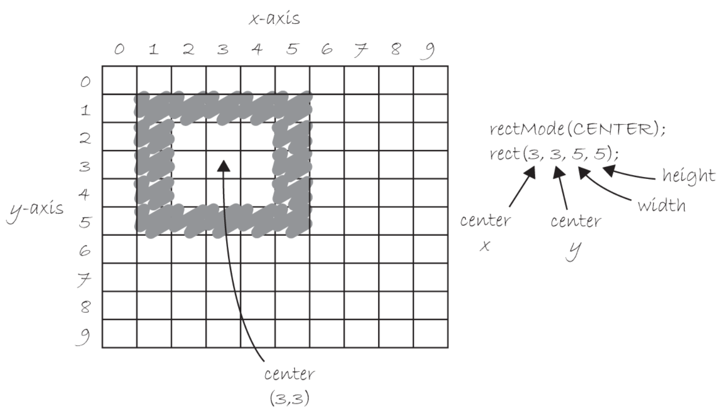


Figure 1-8

Finally, you can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is CORNERS (see Figure 1-9).

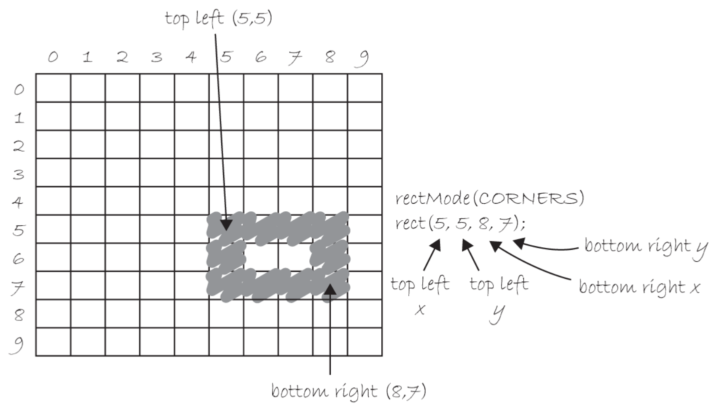


Figure 1-9

Once you have become comfortable with the concept of drawing a rectangle, an ellipse is a snap. In fact, it's identical to rect() with the difference being that an ellipse is drawn where the bounding box[1] (as shown in Figure 1-10) of the rectangle would be. The default mode for ellipse() is CENTER, rather than CORNER as with rect(). See Figure 1-11.

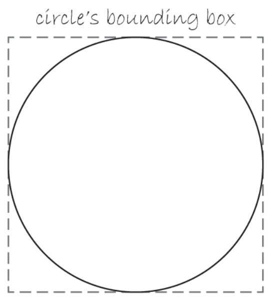


Figure 1-10

[1] A bounding box of a shape in computer graphics is the smallest rectangle that includes all the pixels of that shape. For example, the bounding box of a circle is shown in Figure 1-10.

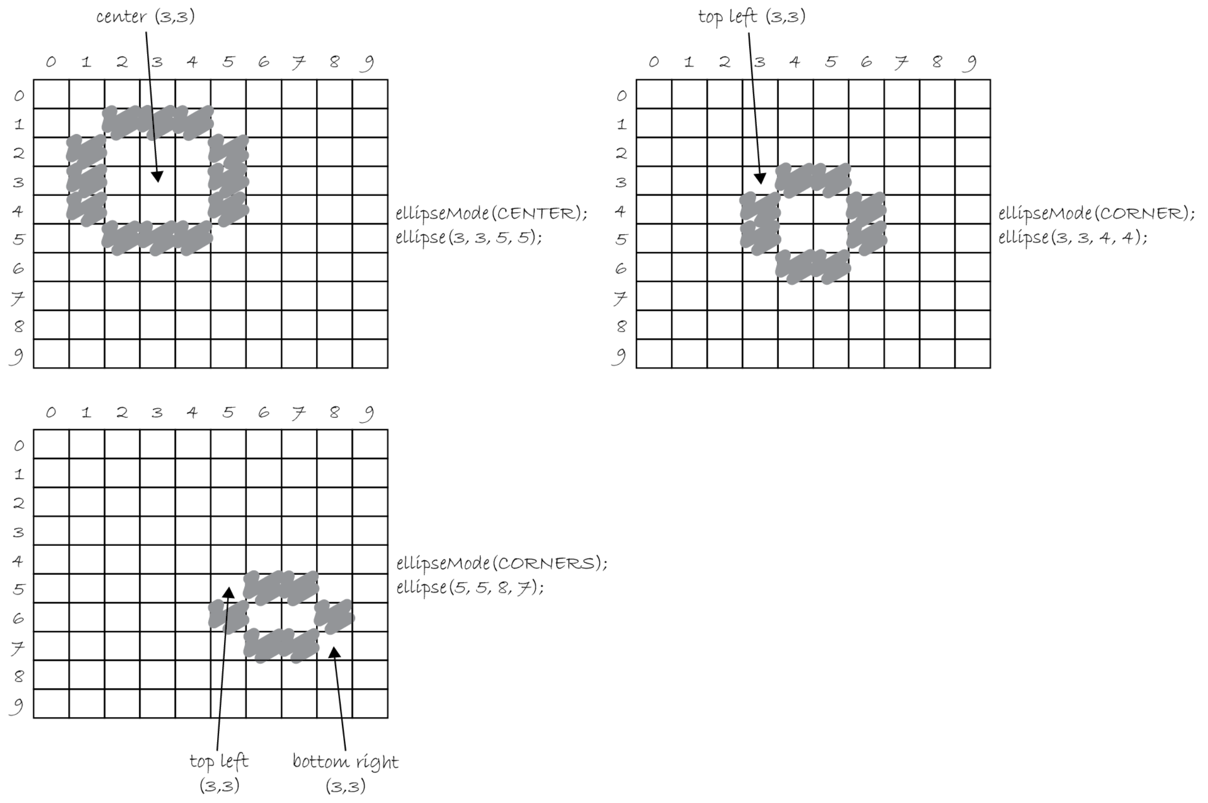


Figure 1-11

It's important to acknowledge that in Figure 1-11, the ellipses do not look particularly circular. Processing has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, you get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, you get a nice round ellipse. Later, you will see that Processing gives you the power to develop your own algorithms for coloring in individual pixels (in fact, you can probably already imagine how you might do this using `point()` over and over again), but for now, it's best to let `ellipse()` do the hard work.

Certainly, point, line, ellipse, and rectangle are not the only shapes available in the Processing library of functions. In Chapter 2, you will see how the Processing reference provides a full list of available drawing functions along with documentation of the required arguments, sample syntax, and imagery. For now, as an exercise, you might try to imagine what arguments are required for some other shapes (Figure 1-12): `triangle()`, `arc()`, `quad()`, `curve()`.

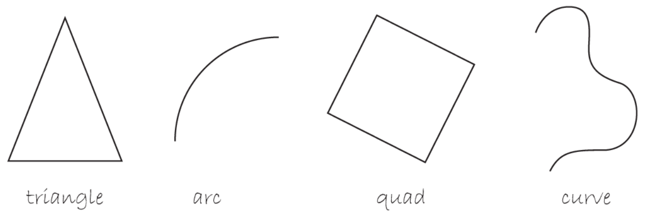
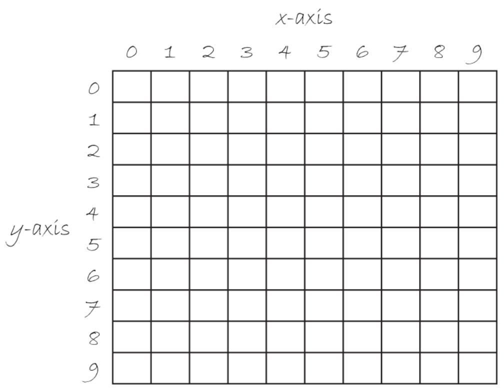


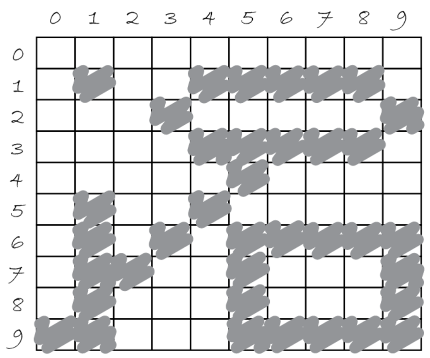
Figure 1-12

Exercise 1-2: Using the blank graph below, draw the primitive shapes specified by the code.

```
line(0, 0, 9, 6);
point(0, 2);
point(0, 4);
rectMode(CORNER);
rect(5, 0, 4, 3);
ellipseMode(CENTER);
ellipse(3, 7, 4, 4);
```



Exercise 1-3: Reverse engineer a list of primitive shape drawing instructions for the diagram below.



Note: There is more than one correct answer!

____

____

____

____

____

1-3 Grayscale color

As you learned in Section 1-2 on page 5, the primary building block for placing shapes onscreen is a pixel coordinate. You politely instructed the computer to draw a shape at a specific location with a specific size. Nevertheless, a fundamental element was missing — color.

In the digital world, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Therefore, color is defined with a range of numbers. I'll start with the simplest case: *black and white* or *grayscale*. To specify a value for grayscale, use the following: 0 means black, 255 means white. In between, every other number — 50, 87, 162, 209, and so on — is a shade of gray ranging from black to white. See Figure 1-13.

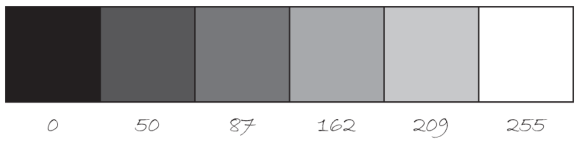


Figure 1-13

Does 0–255 seem arbitrary to you?

Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a *bit*, eight of them together is a *byte*. Imagine if you had eight bits (one byte) in sequence — how many ways can you configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. Processing will use eight bit color for the grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components; see Section 1-4 on page 12).

Understanding how this range works, you can now move to setting specific grayscale colors for the shapes you drew in Section 1-2 on page 5. In Processing, every shape has a `stroke()` or a `fill()` or both. The `stroke()` specifies the color for the outline of the shape, and the `fill()` specifies the color for the interior of that shape. Lines and points can only have `stroke()`, for obvious reasons.

If you forget to specify a color, Processing will use black (0) for the `stroke()` and white (255) for the `fill()` by default. Note that I'm now using more realistic numbers for the pixel locations, assuming a larger window of size 200 × 200 pixels. See Figure 1-14.

```
rect(50, 40, 75, 100);
```

By adding the `stroke()` and `fill()` functions *before* the shape is drawn, you can set the color. It's much like instructing your friend to use a specific pen to draw on the graph paper. You would have to tell your friend *before* he or she starting drawing, not after.

There is also the function `background()`, which sets a background color for the window where shapes will be rendered.

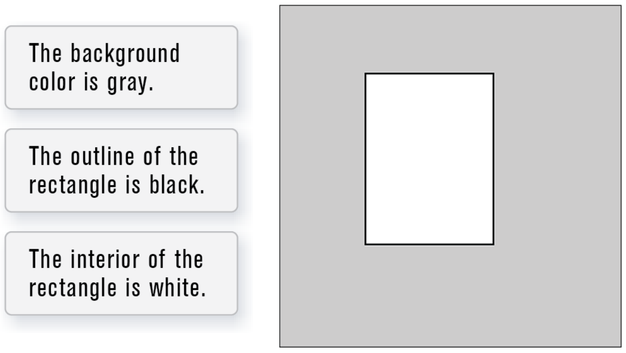

The background color is gray.
The outline of the rectangle is black.
The interior of the rectangle is white.

Figure 1-14

Example 1-1. Stroke and fill

```
background(255);
stroke(0);
fill(150);
rect(50, 50, 75, 100);
```

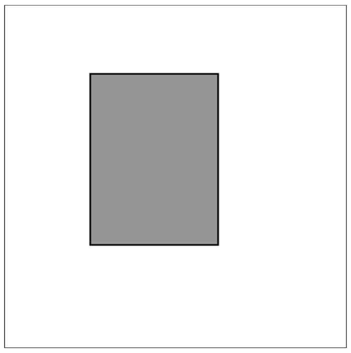


Figure 1-15

`stroke()` or `fill()` can be eliminated with the `noStroke()` or `noFill()` functions. Your instinct might be to say `stroke(0)` for no outline, however, it's important to remember that 0 is not "nothing," but rather denotes the color black. Also, remember not to eliminate both — with `noStroke()` and `noFill()`, nothing will appear!

Example 1-2. noFill()

```
background(255);
stroke(0);
noFill();
ellipse(60, 60, 100, 100);
```

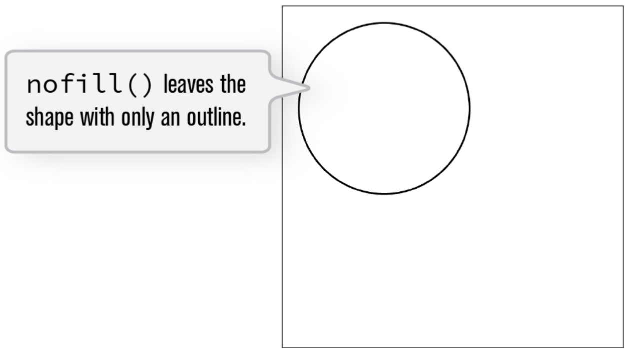


Figure 1-16

When you draw a shape, Processing will always use the most recently specified `stroke()` and `fill()`, reading the code from top to bottom. See Figure 1-17.

```
background(150);
stroke(0);
line(0, 0, 200, 200);
stroke(255);
noFill();
rect(25, 25, 75, 75);
```

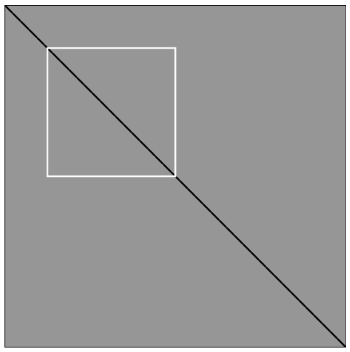


Figure 1-17

Exercise 1-4: Try to guess what the instructions would be for the following screenshot.

____

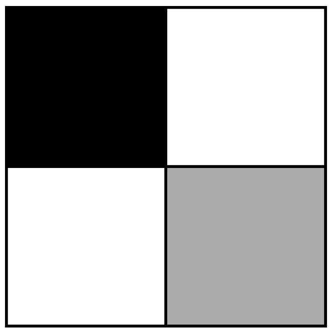
____

____

____

____

____



1-4 RGB color

A nostalgic look back at graph paper helped you to learn the fundamentals for pixel locations and size. Now that it's time to study the basics of digital color, here's another childhood memory to get you started. Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got.

Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are different: red, green, and blue (i.e., "RGB" color). And with color on the screen, you're mixing light, not paint, so the mixing rules are different as well.

- Red + green = yellow
- Red + blue = purple
- Green + blue = cyan (blue-green)
- Red + green + blue = white
- No colors = black

This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple.

While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fingers. And of course you can't say "Mix some red with a bit of blue"; you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order red, green, and blue. You will get the hang of RGB color mixing through experimentation, but next I will cover some code using some common colors.

Note that the print version of this book will only show you black and white versions of each Processing sketch, but all sketches can be seen online in full color at *http://learningprocessing.com*. You can also see a color version of the tutorial on the Processing website.

Example 1-3. RGB color

```
background(255);
noStroke();

fill(255, 0, 0);          Bright red
ellipse(20, 20, 16, 16);

fill(127, 0, 0);          Dark red
ellipse(40, 20, 16, 16);

fill(255, 200, 200);      Pink (pale red).
ellipse(60, 20, 16, 16);
```

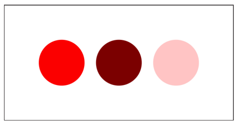


Figure 1-18

Processing also has a color selector to aid in choosing colors. Access this via "Tools" (from the menu bar) → "Color Selector." See Figure 1-19.

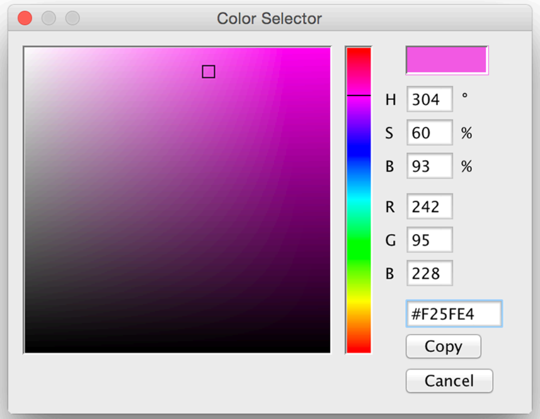


Figure 1-19

Exercise 1-5: Complete the following program. Guess what RGB values to use (you will be able to check your results in Processing after reading the next chapter). You could also use the color selector, shown in Figure 1-19.

```
fill(________,________,________);       Bright blue
ellipse(20, 40, 16, 16);

fill(________,________,________);       Dark purple
ellipse(40, 40, 16, 16);

fill(________,________,________);       Yellow
ellipse(60, 40, 16, 16);
```

Exercise 1-6: What color will each of the following lines of code generate?

```
fill(0, 100, 0);        ______________________________________

fill(100);              ______________________________________

stroke(0, 0, 200);      ______________________________________

stroke(225);            ______________________________________

stroke(255, 255, 0);    ______________________________________

stroke(0, 255, 255);    ______________________________________

stroke(200, 50, 50);    ______________________________________
```

1-5 Color transparency

In addition to the red, green, and blue components of each color, there is an additional optional fourth component, referred to as the color's "alpha." Alpha means opacity and is particularly useful when you want to draw elements that appear partially see-through on top of one another. The alpha values for an image are sometimes referred to collectively as the "alpha channel" of an image.

It's important to realize that pixels are not literally transparent; this is simply a convenient illusion that is accomplished by blending colors. Behind the scenes, Processing takes the color numbers and adds a percentage of one to a percentage of another, creating the optical perception of blending. (If you're interested in programming "rose-colored" glasses, this is where you would begin.)

Alpha values also range from 0 to 255, with 0 being completely transparent (i.e., zero percent opaque) and 255 completely opaque (i.e., 100 percent opaque). Example 1-4 shows a code example that is displayed in Figure 1-20.

Example 1-4. Opacity

```
background(0);
noStroke();

fill(0, 0, 255);          // No fourth argument means 100% opacity.
rect(0, 0, 100, 200);

fill(255, 0, 0, 255);     // 255 means 100% opacity.
rect(0, 0, 200, 40);

fill(255, 0, 0, 191);     // 75% opacity
rect(0, 50, 200, 40);

fill(255, 0, 0, 127);     // 50% opacity
rect(0, 100, 200, 40);

fill(255, 0, 0, 63);      // 25% opacity
rect(0, 150, 200, 40);
```

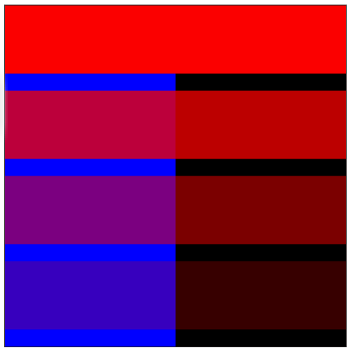


Figure 1-20

1-6 Custom color ranges

RGB color with ranges of 0 to 255 is not the only way you can handle color in Processing. Behind the scenes in the computer's memory, color is *always* talked about as a series of 24 bits (or 32 in the case of colors with an alpha). However, Processing will let you think about color any way you like, and translate any values into numbers the computer understands. For example, you might prefer to think of color as ranging from 0 to 100 (like a percentage). You can do this by specifying a custom `colorMode()`.

```
colorMode(RGB, 100);      // With colorMode() you can set your own color range.
```

The above function says: "OK, I want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."

Although it's rarely convenient to do so, you can also have different ranges for each color component:

```
colorMode(RGB, 100, 500, 10, 255);
```

Now I am saying "Red values range from 0 to 100, green from 0 to 500, blue from 0 to 10, and alpha from 0 to 255."

Finally, while you will likely only need RGB color for all of your programming needs, you can also specify colors in the HSB (hue, saturation, and brightness) mode. While HSB values also default to a range of 0 to 255, a common set of ranges (with brief explanation) are as follows:

- **Hue** — The shade of color itself (red, blue, orange, etc.) ranging from 0 to 360 (think of 360° on a color "wheel").
- **Saturation** — The vibrancy of the color, 0 to 100 (think of 50%, 75%, etc.).
- **Brightness** — The, well, brightness of the color, 0 to 100.

Exercise 1-7: Design a creature using simple shapes and colors. Draw the creature by hand using only points, lines, rectangles, and ellipses. Then attempt to write the code for the creature, using the Processing commands covered in this chapter: `point()`, `line()`, `rect()`, `ellipse()`, `stroke()`, *and* `fill()`. *In the next chapter, you will have a chance to test your results by running your code in Processing.*

Example 1-5 shows my version of Zoog, with the outputs shown in Figure 1-21.

Example 1-5. Zoog

```
background(255);
ellipseMode(CENTER);
rectMode(CENTER);
stroke(0);
fill(150);
rect(100, 100, 20, 100);
fill(255);
ellipse(100, 70, 60, 60);
fill(0);
ellipse(81, 70, 16, 32);
ellipse(119, 70, 16, 32);
stroke(0);
line(90, 150, 80, 160);
line(110, 150, 120, 160);
```

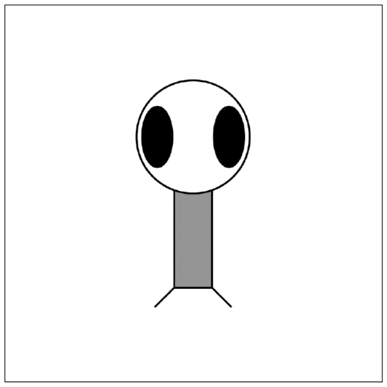


Figure 1-21

The sample answer is my Processing-born being, named Zoog. Over the course of the first nine chapters of this book, I will follow the course of Zoog's childhood. The fundamentals of programming will be demonstrated as Zoog grows up. You will first learn to display Zoog, then to make an interactive Zoog and animated Zoog, and finally to duplicate Zoog in a world of many Zoogs.

I suggest you design your own "thing" (note that there is no need to limit yourself to a humanoid or creature-like form; any programmatic pattern will do) and recreate all of the examples throughout the first nine chapters with your own design. Most likely, this will require you to change only a small portion (the shape rendering part) of each example. This process, however, should help solidify your understanding of the basic elements required for computer programs — *variables*, *conditionals*, *loops*, *functions*, *objects*, and *arrays* — and prepare you for when Zoog matures, leaves the nest, and ventures off into the more advanced topics from Chapter 10 onwards in this book.

18 Data Input

A million monkeys were given a million typewriters. It's called the Internet.
—Simon Munnery

In this chapter:
– Manipulating strings
– Reading and writing text files
– Tabular data
– Word counting and text analysis
– XML and JSON data
– Threads
– Using data from APIs

This chapter will move beyond displaying text and examine how to use `String` objects as the basis for reading and writing data. I'll start by covering more sophisticated methods for manipulating strings, searching in them, chopping them up, and joining them together. Afterward, I will show how these skills allow you to use input from data sources such as text files, web pages, XML feeds, JSON data, and third party APIs, and you'll also take a step into the world of data visualization.

18-1 Manipulating strings

In Chapter 17, I touched on a few of the basic functions available in the Java `String` class, such as `charAt()`, `toUpperCase()`, `equals()`, and `length()`. These functions are documented on the Processing reference page for `String`. Nevertheless, in order to perform some more advanced data parsing techniques, you'll will need to explore some additional string manipulation functions documented on the Java website. (I'll demonstrate more about the Java API itself in Chapter 23.)

Let's take a closer look at the following two functions: `indexOf()` and `substring()`.

`indexOf()` locates a sequence of characters within a string. It takes one argument — a search string — and returns a numeric value that corresponds to the first occurrence of the search string inside of the `String` object being searched.

```
String search = "def";
String toBeSearched = "abcdefghi";
int index = toBeSearched.indexOf(search);
```

The value of index in this example is 3.

Strings are just like arrays, in that the first character is index number zero and the last character is the length of the string minus one.

Exercise 18-1: Predict the result of the code below.

```
String sentence = "The quick brown fox jumps over the lazy dog.";

println(sentence.indexOf("quick")); ________

println(sentence.indexOf("fo")); ________

println(sentence.indexOf("The")); ________

println(sentence.indexOf("blah blah")); ________
```

If you're stuck on the last line of Exercise 18-1, it's because there is no way for you to know the answer without consulting the Java reference (or making an educated guess). If the search string cannot be found, `indexOf()` returns `-1`. This is a good choice because `-1` is not a legitimate index value in the string itself, and therefore can indicate "not found." There are no *negative* indices in a string of characters or in an array.

After finding a search phrase within a string, you might want to separate out part of the string, saving it in a different variable. A part of a string is known as a *substring* and substrings are made with the `substring()` function which takes two arguments, a start index and an end index. `substring()` returns the substring in between the two indices.

```
String alphabet = "abcdefghi" ;
String sub = alphabet.substring(3, 6);
```

The string sub is now "def".

Note that the substring begins at the specified *start index* (the first argument) and extends to the character at *end index* (the second argument) *minus one*. I know, I know. Wouldn't it have been easier to just take the *substring* from the start index all the way to the end index? While this might initially seem true, it's actually quite convenient to stop at end index minus one. For example, if you ever want to make a substring that extends to the end of a string, you can simply go all the way to `thestring.length()`. In addition, with end index minus one marking the end, the length of the substring is easily calculated as *end index minus begin index*.

Exercise 18-2: Fill in the blanks to get the substring "fox jumps over the lazy dog" (without the period).

```
String sentence = "The quick brown fox jumps over the lazy dog.";

int foxIndex = sentence.indexOf(________);
int periodIndex = sentence.indexOf(".");

String sub = ________.________(________,________);
```

Exercise 18-3: Write your own "substring" function that receives three arguments: a string, a starting position, and a total number of characters. The function should return the corresponding substring starting at the position and with a length matching the number. Here's some code to get you started.

```
void substring(String txt, int start, int num) {

  return __________________________________;
}
```

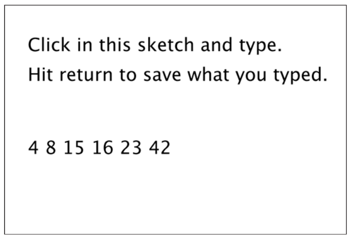
18-2 Splitting and joining

In Chapter 17, you saw how strings can be joined together (referred to as "concatenation") using the "+" operator. Let's review with an example that uses concatenation to get user input from a keyboard.

Example 18-1. User input

```
PFont f;

// Variable to store text currently being typed
String typing = "";
// Variable to store saved text when return is hit
String saved = "";
```

For keyboard input, I use two variables. One will store the text as it is being typed. Another will keep a copy of the typed text once the Enter key is pressed.



Click in this sketch and type.
Hit return to save what you typed.

4 8 15 16 23 42

Figure 18-1

```
void setup() {
  size(300, 200);
  f = createFont("Arial", 16);
}

void draw() {
  background(255);
  int indent = 25;

  // Set the font and fill for text
  textFont(f);
  fill(0);

  // Display everything
  text("Click in this sketch and type. \nHit return to save what you typed.",
indent, 40);
  text(typing, indent, 90);
  text(saved, indent, 130);
}

void keyPressed() {
  // If the return key is pressed, save the String and clear it
  if (key == '\n') {
    saved = typing;
    typing = "";
```

A string can be cleared by setting it equal to `""`.

```
  // Otherwise, concatenate the String
  } else {
    typing = typing + key;
  }
}
```

Each character typed by the user is added to the end of the string.

Exercise 18-4: Create a sketch that chats with the user. For example, if a user enters "cats" the sketch might reply, "How do cats make you feel?"

Processing has two additional functions that make joining strings (or the reverse, splitting them up) easy. In sketches that involve parsing data from a file or the web, you might get hold of that data in the form of an array of strings or as one long string. Depending on what you want to accomplish, it's useful to know how to switch between these two modes of storage. This is where these two new functions, `split()` and `join()`, will come in handy.

```
"one long string or array of strings" ← → { "one", "long", "string", "or", "array",
"of", "strings" }
```

Let's take a look at the split() function. split() separates a longer string into an array of strings, based on a split character known as the *delimiter*. It takes two arguments, the String object to be split and the delimiter. (The delimiter can be a single character or a string.)

```
// Splitting a string based on spaces
String spaceswords = "The quick brown fox jumps over the lazy dog.";

String[] list = split(spaceswords, " " );

printArray(list);
```

Note how printArray() can be used to print the contents of an array and their corresponding indices to the message console.

This period is not set as a delimiter and therefore will be included in the last string in the array: "dog."

Here is an example using a comma as the delimiter (this time passing in a single character: ','.)

```
// Splitting a string based on commas
String commaswords = "The,quick,brown,fox,jumps,over,the,lazy,dog.";
String[] list = split(commaswords, ',');
printArray(list);
```

If you want to use more than one delimiter to split up a text, you must use the Processing function splitTokens(). splitTokens() works identically as split() with one exception: any character that appears in the passed string qualifies as a delimiter.

```
// Splitting a String based on multiple delimiters
String stuff = "hats & apples, cars + phones % elephants dog.";
String[] list = splitTokens(stuff, " &,+." );
printArray(list);
```

The period is specified as a delimiter and therefore will not be included in the last string in the array: "dog".

Exercise 18-5: Fill in what the above code will print in the Processing message window:

```
hats___________
__________________
__________________
__________________
__________________
```

If you are splitting numbers in a string, the resulting array can be converted into an integer array with Processing's int() function.

```
// Calculate sum of a list of numbers in a String
String numbers = "8,67,5,309";
// Converting the String array to an int array
int[] list = int(split(numbers, ','));
```

Numbers in a string are not numbers and cannot be used in mathematical operations unless you convert them first.

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

The reverse of `split()` is `join()`. `join()` takes an array of strings and joins them together into one long `String` object. The `join()` function also takes two arguments, the array to be joined and a *separator*. The separator can either be a single character or a string of characters.

Consider the following array:

```
String[] lines = {"It", "was", "a", "dark", "and", "stormy", "night."};
```

Using the "+" operator along with a `for` loop, you can join a string together as follows:

```
// Manual Concatenation
String onelongstring = "";
for (int i = 0; i < lines.length; i++) {
   onelongstring = onelongstring + lines[i] + " ";
}
```

The `join()` function, however, allows you to bypass this process, achieving the same result in only one line of code.

```
// Using Processing's join()
String onelongstring = join(lines, " ");
```

Exercise 18-6: Split the following string into an array of floating point numbers and calculate the average. Note that the dot should not be considered a delimiter as it's part of the floating point value itself.

```
String floats = "5023.23:52.3:10.4:5.9, 901.3---2.3";

float[] numbers = ________(________(________, "________"));
float total = 0;
for (int i = 0; i < numbers.length; i++) {
  ________ += ________;
}

float avg = ____________________;
```

18-3 Dealing with data

Data can come from many different places: websites, news feeds, spreadsheets, databases, and so on. Let's say you've decided to make a map of the world's flowers. After searching online you might find a PDF version of a flower encyclopedia, or a spreadsheet of flower genera, or a JSON feed of flower data, or a

REST API that provides geolocated lat/lon coordinates, or some web page someone put together with beautiful flower photos, and so on and so forth. The question inevitably arises: "I found all this data; which should I use, and how do I get it into Processing?"

If you are really lucky, you might find a Processing library that hands data to you directly with code. Maybe the answer is to just download this library and write some code like:

```
import flowers.*;

void setup() {
  FlowerDatabase fdb = new FlowerDatabase();
  Flower sunflower = fdb.findFlower("sunflower");
  float h = sunflower.getAverageHeight();
}
```

In this case, someone else has done all the work for you. They've gathered data about flowers and built a Processing library with a set of functions that hands you the data in an easy-to-understand format. This library, sadly, does not exist (not yet), but there are some that do. For example, YahooWeather is a library by Marcel Schwittlick that grabs weather data from Yahoo for you, allowing you to write code like `weather.getWindSpeed()` or `weather.getSunrise()` and more. There is still plenty of work to do in the case of using a library.

Let's take another scenario. Say you're looking to build a visualization of Major League Baseball statistics. You can't find a Processing library to give you the data but you do see everything you're looking for at mlb.com. If the data is online and your web browser can show it, shouldn't you be able to get the data in Processing? Passing data from one application (like a web application) to another (say, your Processing sketch) is something that comes up again and again in software engineering. A means for doing this is an API or "application programming interface": a means by which two computer programs can talk to each other. Now that you know this, you might decide to search online for "MLB API". Unfortunately, mlb.com does not provide its data via an API. In this case you would have to load the raw source of the website itself and manually search for the data you're looking for. While possible, this solution is much less desirable given the considerable time required to read through the HTML source as well as program algorithms for parsing it.

The goal of this chapter is to give you an overview of techniques, ranging from the more difficult manual parsing of data, to the parsing of standardized formats, to the use of an API designed specifically for Processing itself. Each means of getting data comes with its own set of challenges. The ease of using a Processing library is dependent on the existence of clear documentation and examples. But in just about all cases, if you can find your data in a format designed for a computer (spreadsheets, XML, JSON, etc.), you'll be able to save some time in the day for a nice walk outside.

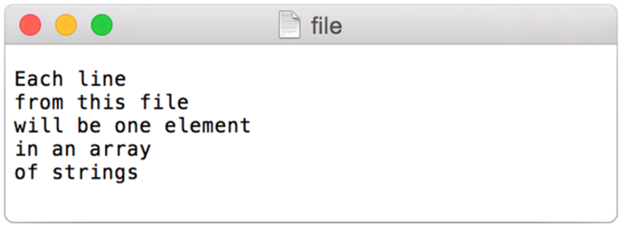
One other note worth a mention about working with data. When developing an application that involves a data source, such as a data visualization, it's sometimes useful to develop with "dummy" or "fake" data. You don't want to be debugging your data retrieval process at the same time as solving problems related to algorithms for drawing. In keeping with my *one-step-at-a-time* mantra, once the meat of the program is completed with dummy data, you can then focus solely on how to retrieve the actual data from the real source. You can always use random or hard-coded numbers into your code when you're experimenting with a visual idea and connect the real data later.

18-4 Working with text files

Let's begin by working with the simplest means of data retrieval: reading from a text file. Text files can be used as a very simple database (you could store settings for a program, a list of high scores, numbers for a graph, etc.) or to simulate a more complex data source.

In order to create a text file, you can use any simple text editor. Windows Notepad or Mac OS X TextEdit will do; just make sure you format the file as "plain text." It is also advisable to name the text files with the ".txt" extension, to avoid any confusion. And just as with image files in Chapter 15, these text files should be placed in the sketch's "data" directory in order for them to be recognized by the Processing sketch.

Once the text file is in place, Processing's `loadStrings()` function is used to read the content of the file into a `String` array. The individual lines of text in the file (see Figure 18-2) each become an individual element in the array.



```
Each line
from this file
will be one element
in an array
of strings
```

Figure 18-2

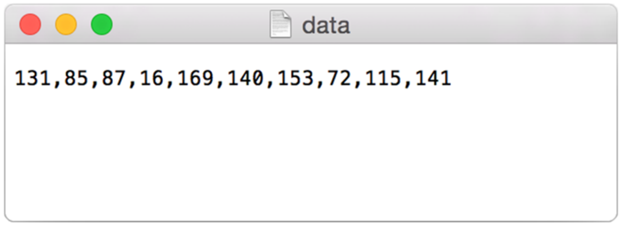
```
String[] lines = loadStrings("file.txt");
println("There are " + lines.length + " lines.");
printArray(lines);
```

This code will print all the lines from the source text file shown in Figure 18-2.

To run the code, create a text file called "file.txt," type a bunch of lines in that file, and place it in your sketch's data directory.

Exercise 18-7: Rewrite Example 17-3 so that it loads the headlines from a text file.

Text from a data file can be used to generate a simple visualization. Example 18-2 loads the data file shown in Figure 18-3. Note the use of the file extension ".csv" to indicate a file with *comma separated values*. The results of visualizing this data are shown in Figure 18-3.



```
131,85,87,16,169,140,153,72,115,141
```

Figure 18-3 contents of "data.csv"

Example 18-2. Graphing comma-separated numbers from a text file

```
int[] data;

void setup() {
  size(200, 200);
  // Load text file as a String
  String[] stuff = loadStrings("data.csv");
  data = int(split(stuff[0], ','));
}
```

The text from the file is loaded into an array. This array has one element because the file only has one line. That element is then split into an array of strings using `','` as a delimiter. Finally, the array is converted an array of integers using `int()`.

```
void draw() {
  background(255);
  stroke(0);
  for (int i = 0; i < data.length; i++) {
    fill(data[i]);
    rect(i*20, 0, 20, data[i]);
  }
  noLoop();
}
```

The array of ints is used to set the color and height of each rectangle.

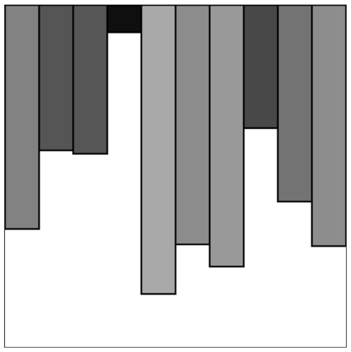
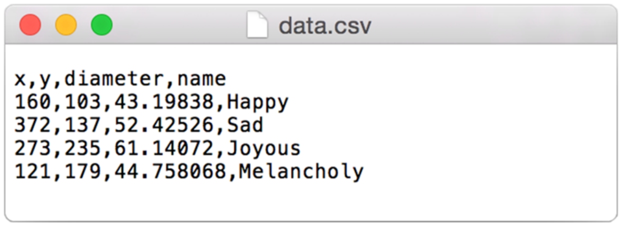


Figure 18-4

Looking at how to parse a csv file with `split()` was a nice learning exercise. In truth, dealing with csv files (which can easily be generated from spreadsheet software such as Google docs) is such a common activity that Processing has an entire built-in class called `Table` to handle the parsing for you.

18-5 Tabular data

A table consists of data arranged as a set of rows and columns, also called "tabular data." If you've ever used a spreadsheet, this is tabular data. Processing's `loadTable()` function takes comma-separated (csv) or tab-separated (tsv) values and automatically places the contents into a `Table` object storing the data in columns and rows. This is a great deal more convenient than struggling to manually parse large data files with `split()`. It was pretty simple in Example 18-2 but with larger files it can get complex quite quickly. Let's say you have a data file that looks like:



```
x,y,diameter,name
160,103,43.19838,Happy
372,137,52.42526,Sad
273,235,61.14072,Joyous
121,179,44.758068,Melancholy
```

Figure 18-5 Each line is a row of a table.

Instead of saying:

```
String[] stuff = loadStrings("data.csv");
```

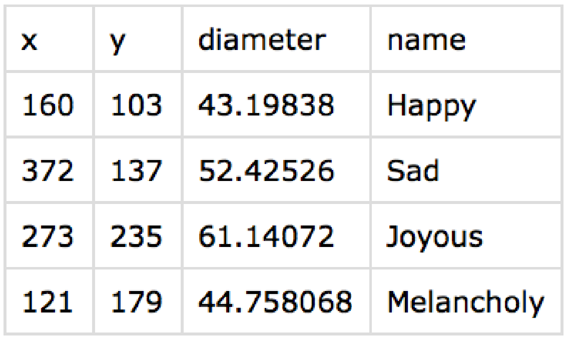
I can say:

```
Table table = loadTable("data.csv");
```

Now I've missed an important detail. Take a look again at Figure 18-5 above. Notice how the first line of text is not the data itself, but rather a *header row*. This row includes labels that describe the data included in each subsequent row. The good news is that Processing can automatically interpret and store the headers for you, if you pass in the option `"header"` when loading the table. (In addition to `"header"`, there are other options you can specify. For example, if your file is called data.txt but is comma separated data you can pass in the option `"csv"`. If it also has a header row, then you can specifiy both options like so: `"header,csv"`). A full list of options can be found on the `loadTable()` documentation page.

```
Table table = loadTable("data.csv", "header");
```

Now that the table is loaded, I can show how you grab individual pieces of data or iterate over the entire table. Let's look at the data visualized as a grid.



x	y	diameter	name
160	103	43.19838	Happy
372	137	52.42526	Sad
273	235	61.14072	Joyous
121	179	44.758068	Melancholy

Figure 18-6

In the above grid you can see that the data is organized in terms of rows and columns. One way to access the data, therefore, would be to request a value by its numeric row and column location (with zero being the first row or first column). This is similar to accessing a pixel color at a given (x,y) location, though in this case the y position (row) comes first. The following code requests a piece of data at a given (row, column) location.

```
int val1 = table.getInt(2, 1);
```

235

```
float val2 = table.getFloat(3, 2);
```

44.758068

```
String s = table.getString(0, 3);
```

"Happy"

While the numeric index is sometimes useful, it's generally going to be more convenient to access each piece of data by the column name. For example, I could pull out a specific row from the `Table`.

```
TableRow row = table.getRow(2);
```

Gets the third row (index 2)

Note in the above line of code that a `Table` object refers to the entire table of data while a `TableRow` object handles an individual row of data within the `Table`.

Once I have the `TableRow` object, I can ask for data from some or all of the columns.

```
int x = row.getInt("x");
```

273

```
int y = row.getInt("y");
```

235

```
float d = row.getFloat("diameter");
```

61.14072

```
String s = row.getString("name");
```

"Joyous"

The method `getRow()` returns a single row from the table. If you want to grab all the rows and iterate over them you can do so in a loop with a counter accessing each row one at a time. The total number of available rows can be retrieved with `getRowCount()`.

```
for (int i = 0; i < table.getRowCount(); i++) {

  TableRow row = table.getRow(i);
  float x = row.getFloat("x");
  float y = row.getFloat("y");
  float d = row.getFloat("diameter");
  String n = row.getString("name");

  // Do something with the data of each row

}
```

Here, I access each row of the table one at a time, in a loop.

If you want to search for a select number of rows within the table, you can do so with `findRows()` and `matchRows()`.

In addition to being read, `Table` objects can be altered or created on the fly while a sketch is running. Cell values can be adjusted, rows can be removed, and new rows can be added. For example, to set new values in a cell there are functions `setInt()`, `setFloat()`, and `setString()`.

```
row.setInt("x", mouseX);
```

Update the value of column "x" to mouseX in a given TableRow.

To add a new row to a Table, simply call the method addRow() and set the values of each column.

```
TableRow row = table.addRow();

row.setFloat("x", mouseX);
row.setFloat("y", mouseY);
row.setFloat("diameter", random(40, 80));
row.setString("name", "new label");
```

Create a new row.

Set the values of all columns in that row.

To delete a row, simply call the method removeRow() and pass in the numeric index of the row you would like removed. For example, the following code removes the first row whenever the size of the table is greater than ten rows.

```
// If the table has more than 10 rows
if (table.getRowCount() > 10) {

  table.removeRow(0);
}
```

Delete the first row (index 0).

The following example puts all of the above code together. Notice how each row of the table contains the data for a Bubble object.

Example 18-3. Loading and Saving Tabular Data

```
Table table;
Bubble[] bubbles;

void setup() {
  size(480, 360);
  loadData();
}

void draw() {
  background(255);
  // Display all bubbles
  for (int i = 0; i < bubbles.length; i++) {
    bubbles[i].display();
  }
}

void loadData() {
  table = loadTable("data.csv", "header");
  bubbles = new Bubble[table.getRowCount()];

  for (int i = 0; i < table.getRowCount(); i++) {

    TableRow row = table.getRow(i);
```

A Table object and an array of Bubble objects. The data from the table will fill the array.

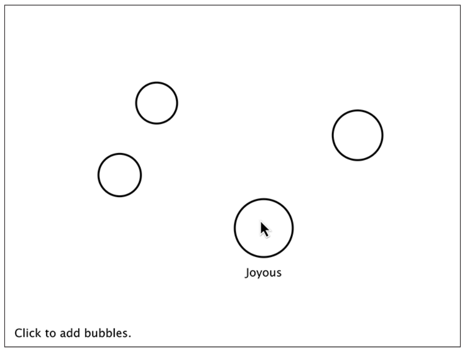


Figure 18-7

Load file into table – "header" indicates file has header row. The size of the array is then determined by the number of rows in the table.

Iterate over all the rows in a table.

```
    float x = row.getFloat("x");
    float y = row.getFloat("y");
    float d = row.getFloat("diameter");
    String n = row.getString("name");
    bubbles[i] = new Bubble(x, y, d, n);
  }
}
```

Access the fields via their column name (or index).

Make a `Bubble` object out of the data from each row.

```
void mousePressed() {
  TableRow row = table.addRow();
  row.setFloat("x", mouseX);
  row.setFloat("y", mouseY);
  row.setFloat("diameter", random(40, 80));
  row.setString("name", "Blah");

  if (table.getRowCount() > 10) {
    table.removeRow(0);
  }

  saveTable(table, "data/data.csv");
  loadData();
}
```

When the mouse is pressed, create a new row and set the values for each column of that row.

If the table has more than 10 rows, delete the oldest row.

This writes the table back to the original CSV file and reload the file so that what's drawn matches.

```
class Bubble {
  float x, y;
  float diameter;
  String name;

  boolean over = false;

  // Create the Bubble
  Bubble(float tempX, float tempY, float tempD, String s) {
    x = tempX;
    y = tempY;
    diameter = tempD;
    name = s;
  }

  // Checking if mouse is over the bubble
  void rollover(float px, float py) {
    float d = dist(px, py, x, y);
    if (d < diameter/2) {
      over = true;
    } else {
      over = false;
    }
  }

  // Display the Bubble
  void display() {
    stroke(0);
    strokeWeight(2);
    noFill();
    ellipse(x, y, diameter, diameter);
    if (over) {
      fill(0);
      textAlign(CENTER);
```

This simple `Bubble` class will be used for several data examples in this chapter. It draws a circle to the window and display a text label when the mouse hovers.

```
        text(name, x, y+diameter/2+20);
      }
    }
  }
```

While unrelated to the main topic of this chapter data, Example 18-3 does include a solution to the second part of Exercise 5-5 on page 77, a rollover with a circle. Here, the distance between a given point and a circle's center is compared to that circle's radius as depicted in Figure 18-8.

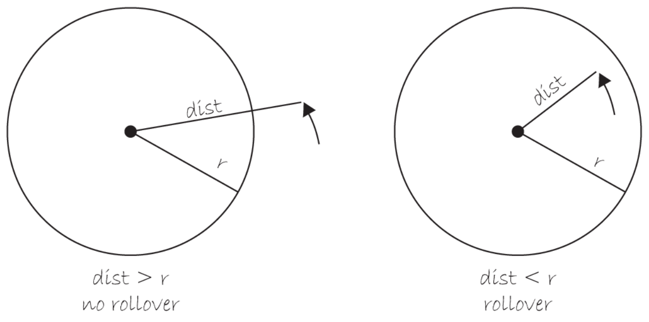


Figure 18-8

```
boolean rollover(int mx, int my) {
  if (dist(mx, my, x, y) < diameter/2) {
    return true;
  } else {
    return false;
  }
}
```

This function returns a boolean value (true or false) depending on whether the point (mx,my) is inside the circle. Notice how radius is equal to half the diameter.

18-6 Data that is not in a standardized format

What if your data is not in a standard format like a table, how do you deal with it then? One of the nice features about `loadStrings()` is that in addition to pulling text from a file, you can also grab a URL. For example:

```
String[] lines = loadStrings("http://www.yahoo.com");
```

When you send a URL path into `loadStrings()`, you get back the raw HTML (Hypertext Markup Language) source of the requested web page. It's the same stuff that appears upon selecting "View Source" from a browser's menu options. You don't need to be an HTML expert to follow this section, but if you are not familiar at all with HTML, you might want to read http://en.wikipedia.org/wiki/HTML.

Unlike with the comma-delimited data from a text file that was specially formatted for use in a Processing sketch, it's not practical to have the resulting raw HTML stored in an array of strings (each element representing one line from the source). Converting the array into one long string can make things a bit simpler. As you saw earlier in the chapter, this can be achieved using `join()`.

```
String onelongstring = join(lines, " ");
```

When pulling raw HTML from a web page, it's likely you do not want all of the source, but just a small piece of it. Perhaps you're looking for weather information, a stock quote, or a news headline. You can take advantage of the text manipulation functions you learned — `indexOf()`, `substring()`, and `length()` — to find pieces of data within a large block of text. You saw an early example of this in Exercise 18-2 on page 385. Take, for example, the following `String` object:

```
String stuff = "Number of apples:62. Boy, do I like apples or what!";
```

Let's say I want to pull out the number of apples from the above text. My algorithm would be as follows:

1. Find the *end of the substring* "apples:" Call it start.
2. Find the *first period* after "apples:" Call it end.
3. Make a *substring* of the characters between start and end.
4. Convert the string to a number (if I want to use it as such).

In code, this looks like:

```
int start      = stuff.indexOf("apples:" ) + 8;  // STEP 1
int end        = stuff.indexOf(".", start);      // STEP 2
String apples  = stuff.substring(start, end);    // STEP 3
int apple_no   = int(apples);                    // STEP 4
```

The index where a string ends can be found by searching for that string and adding its length (here, 8).

The above code will do the trick, but I should be a bit more careful to make sure I don't run into any errors if I do not find the string I am searching for. I can add some error checking and generalize the code into a function:

```
// A function that returns a substring between two substrings
String giveMeTextBetween(String s, String startTag, String endTag) {
  // Find the index of the beginning tag
  int startIndex = s.indexOf(startTag);
  // If I don't find anything
  if (startIndex == -1) {
    return "";
  }
  // Move to the end of the beginning tag
  startIndex += startTag.length();

  // Find the index of the end tag
  int endIndex = s.indexOf(endTag, startIndex);
  // If I don't find the end tag,
  if (endIndex == -1) {
    return "";
  }
  // Return the text in between
  return s.substring(startIndex, endIndex);
}
```

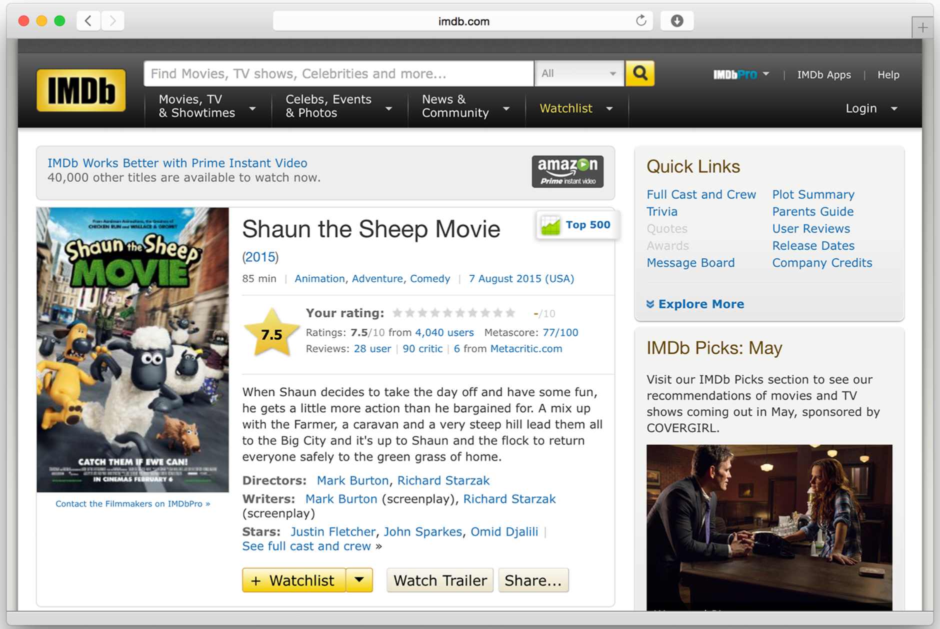
A function to return a substring found between two strings. If beginning or end "tag" is not found, the function returns an empty string.

`indexOf()` can also take a second argument, an integer. That second argument means: Find the first occurrence of the search string after this specified index. I use it here to ensure that `endIndex` follows `startIndex`.

With this technique, you are ready to connect to a website from within Processing and grab data to use in your sketches. For example, you could read the HTML source from nytimes.com and look for today's headlines, search finance.yahoo.com for stock quotes, count how many times the word "flower" appears on your favorite blog, and so on. However, HTML is an ugly, scary place with inconsistently formatted pages that are difficult to reverse engineer and parse effectively. Not to mention the fact that companies change the source code of web pages rather often, so any example that I might make while I am writing this paragraph might break by the time you read this paragraph.

For grabbing data from the web, an XML (Extensible Markup Language) or JSON (JavaScript Object Notation) feed will prove to be more reliable and easier to parse. Unlike HTML (which is designed to make content viewable by a human's eyes) XML and JSON are designed to make content viewable by a computer and facilitate the sharing of data across different systems. Most data (news, weather, and more) is available this way, and I will look at examples in Section 18-8 on page 406 and Section 18-10 on page 414. Though much less desirable, manual HTML parsing is still useful for a couple reasons. First, it never hurts to practice text manipulation techniques that reinforce key programming concepts. But more importantly, sometimes there is data you really want that is not available in an API format, and the only way to get it is with such a technique. (I should also mention that regular expressions, an incredibly powerful techinque in text pattern matching, could also be employed here. As much as I love regex, it's unfortunately beyond the scope of this book.)

An example of data only available as HTML is the Internet Movie Database. IMDb contains information about movies sorted by year, genre, ratings, etc. For each movie, you can find the cast and crew list, a plot summary, running time, a movie poster image, the list goes on. However, IMDb has no API and does not provide its data as XML or JSON. Pulling the data into Processing therefore requires a bit of detective work. Let's look at the page for the *Shaun the Sheep Movie*.



imdb.com

Find Movies, TV shows, Celebrities and more... All

IMDbPro | IMDb Apps | Help

Movies, TV & Showtimes | Celebs, Events & Photos | News & Community | Watchlist | Login

IMDb Works Better with Prime Instant Video
40,000 other titles are available to watch now.

Shaun the Sheep Movie
(2015)
Top 500
85 min | Animation, Adventure, Comedy | 7 August 2015 (USA)

Your rating: -/10
7.5
Ratings: 7.5/10 from 4,040 users Metascore: 77/100
Reviews: 28 user | 90 critic | 6 from Metacritic.com

When Shaun decides to take the day off and have some fun, he gets a little more action than he bargained for. A mix up with the Farmer, a caravan and a very steep hill lead them all to the Big City and it's up to Shaun and the flock to return everyone safely to the green grass of home.

Directors: Mark Burton, Richard Starzak
Writers: Mark Burton (screenplay), Richard Starzak (screenplay)
Stars: Justin Fletcher, John Sparkes, Omid Djalili | See full cast and crew »

Contact the Filmmakers on IMDbPro »

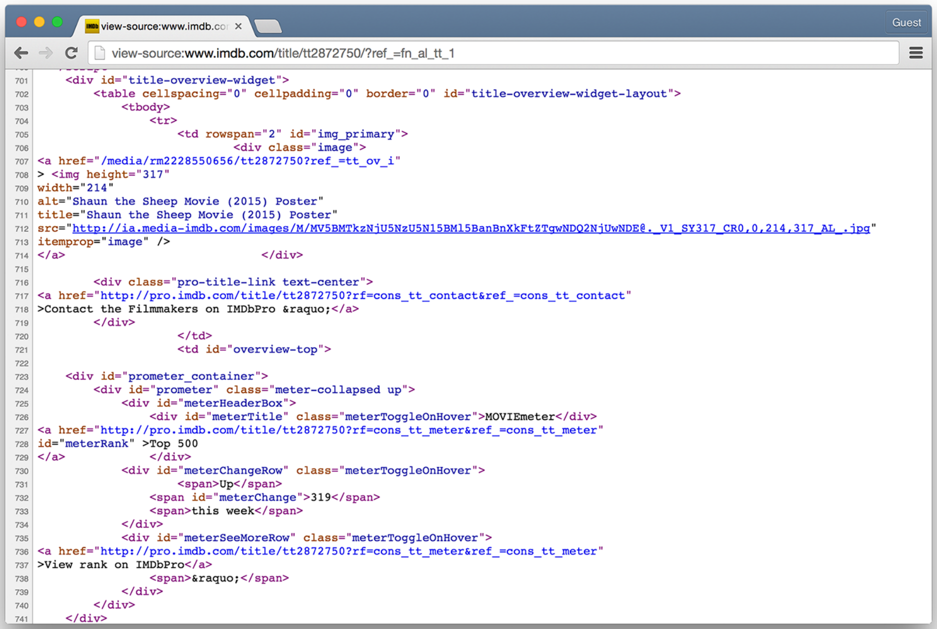
+ Watchlist | Watch Trailer | Share...

Quick Links
Full Cast and Crew | Plot Summary
Trivia | Parents Guide
Quotes | User Reviews
Awards | Release Dates
Message Board | Company Credits
Explore More

IMDb Picks: May
Visit our IMDb Picks section to see our recommendations of movies and TV shows coming out in May, sponsored by COVERGIRL.

Figure 18-9

Looking in the HTML source from the above URL, I find a giant mess of markup.



view-source:www.imdb.com/title/tt2872750/?ref_=fn_al_tt_1

```
    <div id="title-overview-widget">
        <table cellspacing="0" cellpadding="0" border="0" id="title-overview-widget-layout">
            <tbody>
                <tr>
                    <td rowspan="2" id="img_primary">
                            <div class="image">
<a href="/media/rm2228550656/tt2872750?ref_=tt_ov_i"
> <img height="317"
width="214"
alt="Shaun the Sheep Movie (2015) Poster"
title="Shaun the Sheep Movie (2015) Poster"
src="http://ia.media-imdb.com/images/M/MV5BMTkzNjU5NzU5N15BMl5BanBnXkFtZTgwNDQ2NjUwNDE@._V1_SY317_CR0,0,214,317_AL_.jpg"
itemprop="image" />
</a>                            </div>

        <div class="pro-title-link text-center">
<a href="http://pro.imdb.com/title/tt2872750?rf=cons_tt_contact&ref_=cons_tt_contact"
>Contact the Filmmakers on IMDbPro &raquo;</a>
        </div>
                    </td>
                    <td id="overview-top">

    <div id="prometer_container">
        <div id="prometer" class="meter-collapsed up">
            <div id="meterHeaderBox">
                <div id="meterTitle" class="meterToggleOnHover">MOVIEmeter</div>
<a href="http://pro.imdb.com/title/tt2872750?rf=cons_tt_meter&ref_=cons_tt_meter"
id="meterRank" >Top 500
</a>            </div>
            <div id="meterChangeRow" class="meterToggleOnHover">
                    <span>Up</span>
                <span id="meterChange">319</span>
                <span>this week</span>
            </div>
            <div id="meterSeeMoreRow" class="meterToggleOnHover">
<a href="http://pro.imdb.com/title/tt2872750?rf=cons_tt_meter&ref_=cons_tt_meter"
>View rank on IMDbPro</a>
                <span>&raquo;</span>
            </div>
        </div>
    </div>
```

Figure 18-10

It's up to me to pore through the raw source and find the data I am looking for. Let's say I want to know the running time of the movie and grab the movie poster image. After some digging, I find that the movie is 139 minutes long as listed in the following HTML.

```
<div class="txt-block">
  <h4 class="inline">Runtime:</h4>
    <time itemprop="duration" datetime="PT139M">139 min</time>
</div>
```

For any given movie, the running time itself will be variable, but the HTML structure of the page will stay the same. I can therefore deduce that running time will always appear in between:

```
<time itemprop="duration" datetime="PT139M">
```

and:

```
</time>
```

Knowing where the data starts and ends, I can use `giveMeTextBetween()` to pull out the running time.

```
String url = "http://www.imdb.com/title/tt0058331";
String[] lines = loadStrings(url);
// Get rid of the array in order to search the whole page
String html = join(lines, " ");



// Searching for running time
String start = "<time itemprop=\"duration\" datetime=\"PT139M\">";
```

A quote in Java marks the beginning or end of a string. So how do you include an actual quote in a `String` object? The answer is via an "escape" sequence. (You encountered this in Exercise 17-8 on page 375.) A quote can be included using a backward slash, followed by a quote. For example: `String q = "This String has a quote \"in it";`

```
String end = "</time>";
String runningtime = giveMeTextBetween(html, start, end);
println(runningtime);
```

Example 18-4 retrieves both the running time and movie poster image from IMDb and displays it onscreen.

Example 18-4. Parsing IMDb manually

```
String runningtime;
PImage poster;

void setup() {
  size(300, 350);
  loadData();
}

void draw() {
  // Display all the stuff I want to display
  background(255);
  image(poster, 10, 10, 164, 250);
  fill(0);
  text("Shaun the Sheep", 10, 300);
  text(runningtime, 10, 320);
}

void loadData() {
  String url = "http://www.imdb.com/title/tt2872750/";

  String[] lines = loadStrings(url);
  String html = join(lines, "");

  String start = "<time itemprop=\"duration\" datetime=\"PT139M\">";
  String end = "</time>";
  runningtime = giveMeTextBetween(html, start, end);

  start = "<link rel='image_src' href=\"";
  end = "\">";
  String imgUrl = giveMeTextBetween(html, start, end);
  poster = loadImage(imgUrl);
}

String giveMeTextBetween(String s, String before, String after) {

  String found = "";

  // Find the index of before
  int start = s.indexOf(before);
  if (start == -1) {
    return "";
  }

  // Move to the end of the beginning tag
  // and find the index of the "after" String
  start += before.length();
  int end = s.indexOf(after, start);
  if (end == -1) {
    return "";
  }

  // Return the text in between
  return s.substring(start, end);
}
```

Get the raw HTML source into an array of strings (each line is one element in the array). The next step is to turn array into one long string with `join()`.

Searching for running time.

Searching for the URL of the poster image.

Now, load that image!

This function returns a substring between two substrings (before and after). If it can't find anything it returns an empty string.

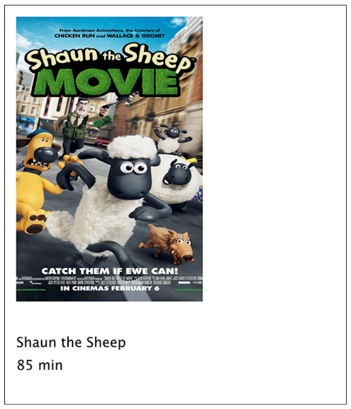


Figure 18-11

Exercise 18-8: Expand Example 18-4 to also search for the movie's rating on IMDb.

Exercise 18-9: Expand Example 18-4 to data related to more than one movie. Can you retrieve the list of all movies released in a given year? Consider creating a `Movie` *class that has a function for retrieving data related to itself.*

Exercise 18-10: Wikipedia is another site with lots of data not available via an API. Create a sketch that grabs information from a Wikipedia page.

18-7 Text analysis

Loading text from a URL need not only be an exercise in parsing out small bits of information. It's possible with Processing to analyze large amounts of text found on the web from news feeds, articles, and speeches, to entire books. A nice source is *Project Gutenberg* which makes available thousands of public domain texts. Algorithms for analyzing text merits an entire book itself, but let's look at some basic techniques.

A text concordance is an alphabetical list of words that appear in a book or body of text along with contextual information. A sophisticated concordance might keep a list of where each word appears (like an index) as well as which words appear next to which other words. In this case, I'm going to create a simple concordance, one that simply stores a list of words and their corresponding counts, i.e., how many times they appeared in the text. Concordances can be used for text analysis applications such as spam filtering or sentiment analysis. To accomplish this task, I am going to use the Processing built-in class `IntDict`.

As you learned in Chapter 9, an array is an ordered list of variables. Each element of the array is numbered and be accessed by its numeric index.

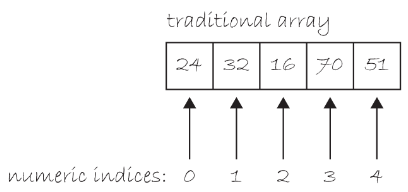


Figure 18-12

However, what if instead of numbering the elements of an array you could name them? This element is named "Sue," this one "Bob," this one "Jane," and so on and so forth. In programming, this kind of data structure is often referred to as an *associative array*, *map*, or *dictionary*. It's a collection of (key, value) pairs. Imagine you had a dictionary of people's ages. When you look up "Sue" (the key), the definition, or value, is her age, 24.

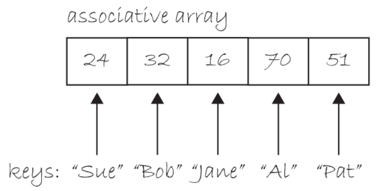


Figure 18-13

Associative arrays can be incredibly convenient for various applications. For example, you could keep a list of student IDs (student name, id) or a list of prices (product name, price) in a dictionary. Here a dictionary is the perfect data structure to hold the concordance. Each element of the dictionary is a word paired with its count.

While there are many classes in Java for dealing with advanced data structures like maps, Processing provides you with a set of three built-in dictionary classes that are easy to use: `IntDict`, `FloatDict`, and `StringDict`. In all of these classes, the key is always a string while the value is variable (an integer, floating point number, or String). For the concordance, I'll use an `IntDict`.

Creating an `IntDict` is as easy as calling an empty constructor. Let's say you want a dictionary to keep track of an inventory of supplies.

```
IntDict inventory = new IntDict();
```

Values can be paired with their keys using the `set()` method.

```
inventory.set("pencils", 10);
inventory.set("paper clips", 128);
inventory.set("pens, 16");
```

`set()` assigns an integer to a String.

There are a variety of other methods that can be called to change the value associated with a particular key. For example, if you wanted to add five pencils, you can use `add()`.

```
inventory.add("pencils", 5);
```

The value of "pencils" is now 15.

A particularly convenient method for the concordance example is `increment()` which adds one to a key's value.

```
inventory.increment("pens");
```

The value of "pencils" is now 16.

To retrieve a value associated with a particular key, the `get()` method is used.

```
int num = inventory.get("pencils");
```

The value of `num` is 16.

Finally, dictionaries can be sorted by their keys (alphabetical) or values (smallest to largest or the reverse) with the methods `sortKeys()`, `sortKeysReverse()`, `sortValues()`, and `sortValuesReverse()`.

The concordance now becomes a rather simple program to write. All I need to do is load in a text file, chop it into words with `splitTokens()` and call `increment()` on an `IntDict` for every single word found in the text. The following example does precisely this with the entire text of Shakespeare's play, *A Midsummer Night's Dream*, displaying a simple graph of the most used words.

Example 18-5. Text concordance using IntDict

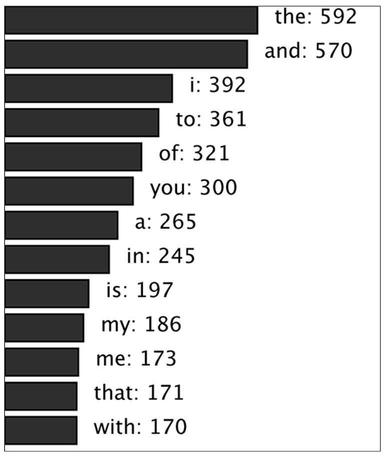


Figure 18-14

```
String[] allwords;

String delimiters = " ,.?!;:[]";

IntDict concordance;

void setup() {
  size(360, 640);

  // Load A Midsummer Night's Dream into an array of
strings
  String url = "http://www.gutenberg.org/cache/epub/
1514/pg1514.txt";
  String[] rawtext = loadStrings(url);

  // Join the big array together as one long string
  String everything = join(rawtext, "" );

  allwords = splitTokens(everything, delimiters);


  // Make a new empty dictionary
  concordance = new IntDict();


  for (int i = 0; i < allwords.length; i++) {
    String s = allwords[i].toLowerCase();
    concordance.increment(s);
  }


  concordance.sortValuesReverse();
}

void draw() {
  background(255);

  // Display the text and total times the word appears
  int h = 20;
  String[] keys = concordance.keyArray();

  for (int i = 0; i < height/h; i++) {
    String word = keys[i];
    int count = concordance.get(word);

    fill(51);
    rect(0, i*20, count/4, h-4);
    fill(0);
    text(word + ": " + count, 10+count/4, i*h+h/2);
    stroke(0);
  }
}
```

Any punctuation is used as a delimiter.

All the lines in *A Midsummer Night's Dream* are first joined as one big string and then split up into an array of individual words. Note the use of `splitTokens()` since I am using spaces and punctuation marks as delimiters.

It's useful to convert each word to lower case so that, for example, "The" and "the" are both counted as the same word.

For every single word, increase its count in the dictionary.

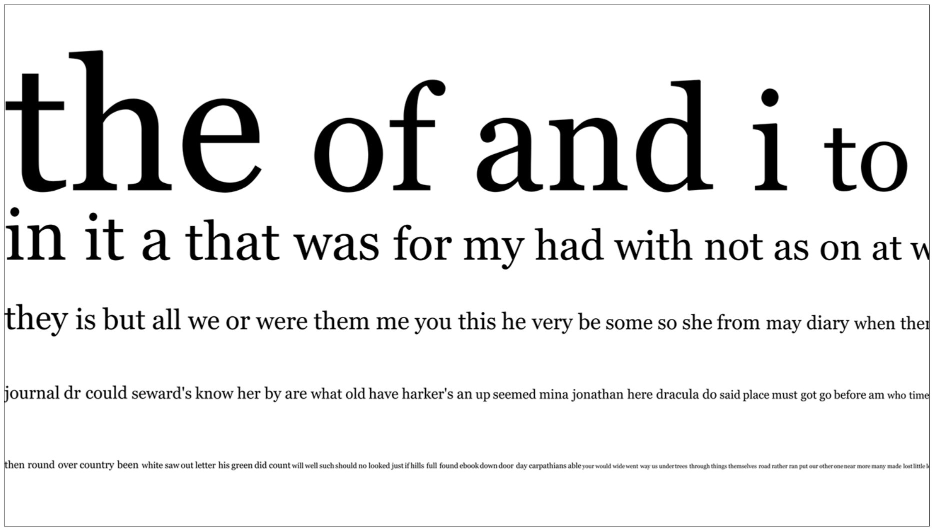
Sort the dictionary so that words that appear most often are first.

In order to iterate over every word in the dictionary, first ask for an array of all of the keys.

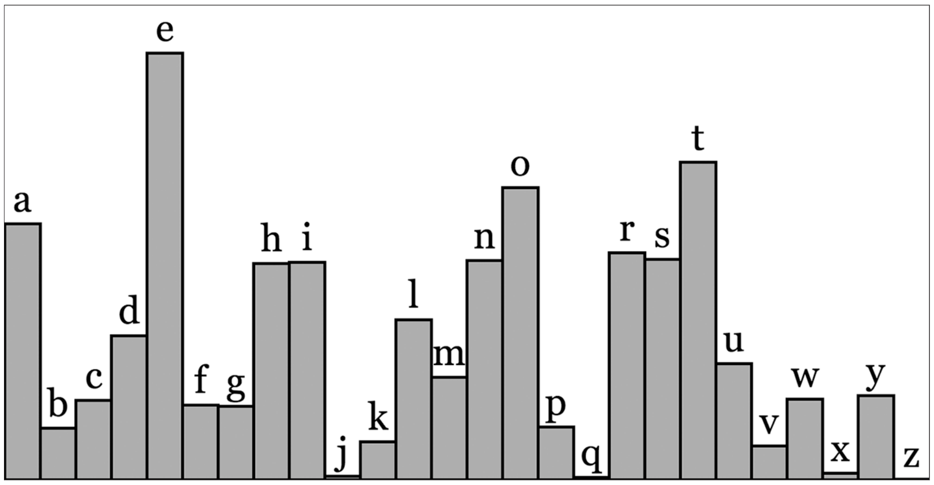
Look at each key one at a time and retrieve its count.

Displaying a rectangle along with the count as a simple graph.

Exercise 18-11: Create a sketch that visualizes the process of generating the concordance. One example is an animation where each word is read one at a time. When a new word is found, it's added to the sketch window; when it's a word already found, the font size increases.


the of and i to
in it a that was for my had with not as on at w
they is but all we or were them me you this he very be some so she from may diary when ther
journal dr could seward's know her by are what old have harker's an up seemed mina jonathan here dracula do said place must got go before am who time
then round over country been white saw out letter his green did count will well such should no looked just if hills full found ebook down door day carpathians able your would wide went way us under trees through things themselves road rather ran put our other one near more many made lost little

Exercise 18-12: Count the number of times each letter of the alphabet appears in a text. Here is one possibility (but you should be more creative). Note this sketch will require the use of the `charAt()` *function. How would you do this with an array or with an* `IntDict`*?*



Exercise 18-13: In James W. Pennebaker's book The Secret Life of Pronouns, *Pennebaker describes his research into how the frequency of words that have little to no meaning on their own (I, you, they, a, an, the, etc.) are a window into the emotional state or personality of an author or speaker. For example, heavy use of the pronoun "I" is an indicator of "depression, stress, or insecurity". Create a Processing sketch that analyzes your use of pronouns across a body of text. For more, visit http://www.analyzewords.com/.*

Before moving on, I'll briefly mention that Processing also includes three classes for lists of numbers and strings: `IntList`, `FloatList`, and `StringList`. In other words, if you just want a list of words (without their counts) you could use a `StringList` rather than an `IntDict`. These lists will come up again in Section 23-4 on page 509.

18-8 XML

The examples in Section 18-6 on page 396 demonstrated the process of manually searching through text for individual pieces of data. If your data is available via a standardized format such as XML or JSON, however, these manual techniques are no longer required. XML is designed to facilitate the sharing of data across different systems, and you can retrieve that data using the built-in Processing XML class.

XML organizes information in a tree structure. Let's imagine a list of students. Each student has an ID number, name, address, email, and telephone number. Each student's address has a city, state, and zip code. An XML tree for this dataset might look like Figure 18-15.

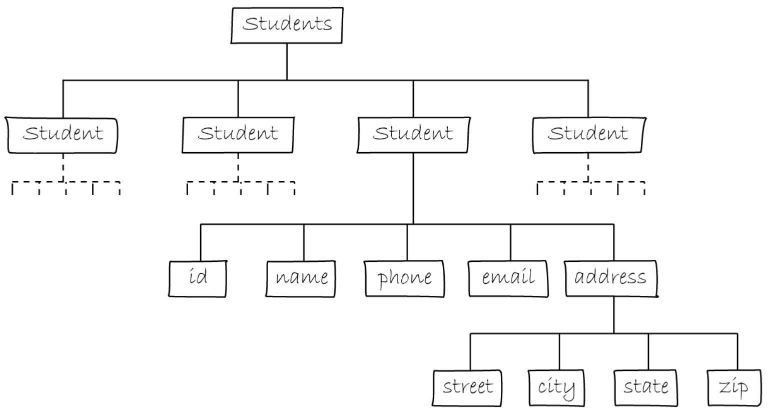


Figure 18-15

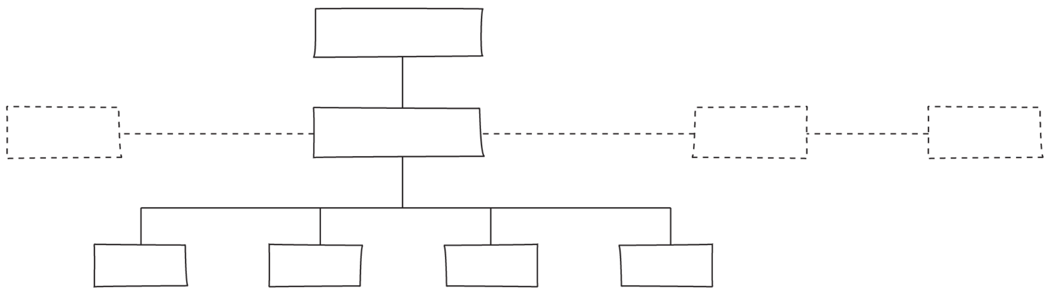
The XML source itself (with two students listed) is:

```
<?xml version = "1.0" encoding = "UTF-8 "?>
<students>
  <student>
    <id>001</id>
    <name>Daniel Shiffman</name>
    <phone>555-555-5555</phone>
    <email>daniel@shiffman.net</email>
    <address>
```

```
      <street>123 Processing Way</street>
      <city>Loops</city>
      <state>New York</state>
      <zip>01234</zip>
    </address>
  </student>
  <student>
    <id>002</id>
    <name>Zoog</name>
    <phone>555-555-5555</phone>
    <email>zoog@planetzoron.uni</email>
    <address>
      <street>45.3 Nebula 5</street>
      <city>Boolean City</city>
      <state>Booles</state>
      <zip>12358</zip>
    </address>
  </student>
</students>
```

Note the similarities to object-oriented programming. You can think of the XML tree in the following terms. The XML document represents an array of student objects. Each student object has multiple pieces of information, an ID, a name, a phone number, an email address, and a mailing address. The mailing address is also an object that has multiple pieces of data, such as street, city, state, and zip.

Exercise 18-14: Revisit the `Bubble` *class from Example 18-3. Design an XML tree structure for these Bubble objects. Diagram the tree and write out the XML source. (Use the empty diagram and fill in the blanks below.)*



```
<?xml version = "1.0"?>
<________>
 <bubble>
   <________>160</________>
   <________>103</________>
   <________>43.19838</diameter>
   <________>Happy</________>
  <________>
</bubbles >
```

Let's look at some data made available from a web service such as Yahoo Weather. Here is the raw XML source. (Note I have edited it slightly for simplification purposes.)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0">
  <channel>
    <item>
      <title>Conditions for New York, NY at 12:49 pm EDT</title>
      <geo:lat>40.67</geo:lat>
      <geo:long>-73.94</geo:long>
      <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY//link>
      <pubDate>Thu, 24 Jul 2014 12:49 pm EDT</pubDate>
      <yweather:condition text="Partly Cloudy" code="30" temp="76"/>
      <yweather:forecast day="Thu" low="65" high="82" text="Partly Cloudy"/>
    </item>
  </channel>
</rss>
```

The data is mapped in the tree stucture shown in Figure 18-16.

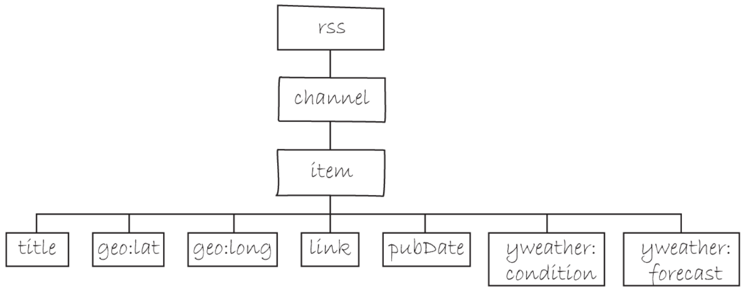


Figure 18-16

You may be wondering what the top level "RSS" is all about. Yahoo's XML weather data is provided in RSS format. RSS stands for "Really Simple Syndication" and is a standardized XML format for syndicating web content (such as news articles, etc.). You can read more about RSS on Wikipedia.

Now that you have a handle on the tree structure, let's look at the specifics inside that structure. With the exception of the first line (which simply indicates that this page is XML formatted), this XML document contains a nested list of *elements*, each with a start tag, that is, `<channel>`, and an end tag, that is, `</channel>`. Some of these elements have content between the tags:

```
<title>Conditions for New York, NY at 12:49 pm EDT</title>
```

and some have attributes (formatted by *Attribute Name* equals *Attribute Value in quotes*):

```
<yweather:forecast day="Thu" low="65" high="82" text="Partly Cloudy"/>
```

18-9 Using the Processing XML class

Since the syntax of XML is standardized, I could certainly use `split()`, `indexof()`, and `substring()` to find the pieces I want in the XML source. The point here, however, is that because XML is a standard format, I don't have to do this. Rather, I can use an XML parser. In Processing, XML can be parsed using the built-in Processing class `XML`.

```
XML xml = loadXML("http://xml.weather.yahoo.com/forecastrss?p=10003");
```

Here, instead of `loadStrings()` or `loadTable()`, I'm now calling `loadXML()` and passing in the address (URL or local file) of the XML document. An `XML` object represents one element of an XML tree. When a document is first loaded, that `XML` object is always the root element. Referring back to XML tree diagram (Figure 18-16), I find the current temperature via the following path:

1. The root of the tree is **RSS**.
2. **RSS** has a child named **channel**.
3. **Channel** has a child named **item**.
4. **Item** has a child named **yweather:condition**.
5. The temperature is stored in **yweather:condition** as the attribute **temp**.

The children of an element are accessed via the `getChild()` function.

```
XML channel = xml.getChild("channel");
```

Accessing the "channel" child element of the root element

The content of an element itself is retrieved with one of the following methods: `getContent()`, `getIntContent()`, or `getFloatcContent()`. `getContent()` is for generic use and will always give you the content as a String. If you intend to use the content as a number, Processing will convert it for you with either `getIntContent()` or `getFloatcContent()`. Attributes can also be read as either numbers — `getInt()`, `getFloat()` — or text — `getString()`.

Following steps one through five outlined above through the XML tree, I have:

```
XML root = loadXML("http://xml.weather.yahoo.com/forecastrss?p=10003");
```

Step 1

```
XML channel     = root.getChild("channel");
```

Step 2

```
XML item        = channel.getChild("item");
```

Step 3

```
XML yweather    = item.getChild("yweather:condition");
```

Step 4

```
int temperature = yweather.getInt("temp");
```

Step 5

However, this is a bit long-winded and so can be compressed down into one (or three, as below) lines of code.

```
XML root = loadXML(http://xml.weather.yahoo.com/forecastrss?p=10003);

XML forecast =
  root.getChild("channel").getChild("item").getChild("yweather:condition");
int temperature = forecast.getInt("temp");
```

Steps 2-4

Finally, the second line of code above can be further condensed to:

```
XML forecast = xml.getChild("channel/item/yweather:condition");
```

Steps 2-4

Following is the above code put together in an example that retrieves weather data for multiple zip codes by parsing Yahoo's XML feed.

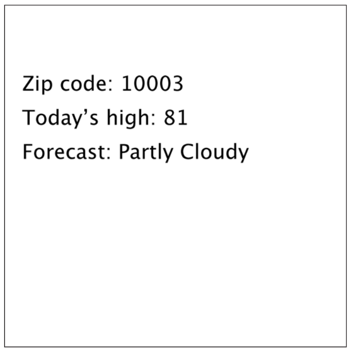
Example 18-6. Parsing Yahoo weather XML

```
int temperature = 0;
String weather = "";


// The zip code
String zip = "10003";

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

  // The URL for the XML document
  String url = "http://xml.weather.yahoo.com/
forecastrss?p=" + zip;

  // Load the XML document
  XML xml = loadXML(url);

  XML forecast = xml.getChild("channel/item/yweather:forecast");


  temperature = forecast.getInt("high");
  weather = forecast.getString("text");
}

void draw() {
  background(255);
  fill(0);

  // Display all the stuff I want to display
  text("Zip code: " + zip, 10, 50);
  text("Today's high: " + temperature, 10, 70);
  text("Forecast: " + weather, 10, 90);
}
```

Temperature is stored as a number and the description of the weather as a string.

Here, I grab the XML element that I want.

And then I pull the attributes from that XML element.



```
Zip code: 10003
Today's high: 81
Forecast: Partly Cloudy
```

Figure 18-17

Other useful XML functions are:

- `hasChildren()` — checks whether or not an element has any children
- `getChildren()` — returns an array containing all child elements
- `getAttributeCount()` — counts the specified element's number of attributes
- `hasAttribute()` — checks whether or not an element has the specified attribute

In this example I'm accessing the child nodes by their name (i.e., "channel," "item,", etc.), however they can also be accessed numerically via an index (starting at zero, same as an array). This is convenient when looping over a list of children, much like I did with tabular data when I iterated over rows in a table.

In Example 18-3, I used a `Table` to store information related to `Bubble` objects. An XML document can also be used in the same manner. Here is a possible solution to Exercise 18-14 on page 407, an XML tree of `Bubble` objects. (Note that this solution uses element *attributes* for x and y coordinates; this was not the format provided in Exercise 18-14 on page 407 since I had not yet covered attributes.)

```
<?xml version="1.0" encoding="UTF-8"?>
<bubbles>
  <bubble>
    <position x="160" y="103"/>
    <diameter>43.19838</diameter>
    <label>Happy</label>
  </bubble>
  <bubble>
    <position x="372" y="137"/>
    <diameter>52.42526</diameter>
    <label>Sad</label>
  </bubble>
  <bubble>
    <position x="273" y="235"/>
    <diameter>61.14072</diameter>
    <label>Joyous</label>
  </bubble>
  <bubble>
    <position x="121" y="179"/>
    <diameter>44.758068</diameter>
    <label>Melancholy</label>
  </bubble>
</bubbles>
```

I can use `getChildren()` to retrieve the array of `<bubble>` elements and make a `Bubble` object from each one. Here is the example which uses the identical `Bubble` class from earlier (not included below). The new code is in bold.

Example 18-7. Using Processing's XML class

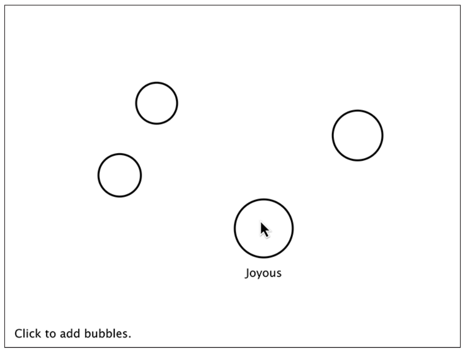


Figure 18-18

```
// An Array of Bubble objects
Bubble[] bubbles;
// An XML object
XML xml;

void setup() {
  size(480, 360);
  loadData();
}

void loadData() {
  // Load XML file
  xml = loadXML("data.xml");
  // Get all the child nodes named "bubble"
  XML[] children = xml.getChildren("bubble");

  bubbles = new Bubble[children.length];

  for (int i = 0; i < bubbles.length; i++) {

    XML positionElement = children[i].getChild("position");
    float x = positionElement.getInt("x");
    float y = positionElement.getInt("y");
```

The size of the `Bubble` array is determined by the total XML elements named "bubble."

The position element has two attributes: "x" and "y".Attributes can be accessed as an integer or float via `getInt()` and `getFloat()`.

```
    // The diameter is the content of the child named "diameter"
    XML diameterElement = children[i].getChild("diameter");
    float diameter = diameterElement.getFloatContent();
```

Notice, however, with the content of an XML node, I retrieve via `getIntContent()` and `getFloatContent()`.

```
    // The label is the content of the child named "label"
    XML labelElement = children[i].getChild("label");
    String label = labelElement.getContent();

    // Make a Bubble object out of the data read
    bubbles[i] = new Bubble(x, y, diameter, label);
  }
}

void draw() {
  background(255);
  // Display all bubbles
  for (int i = 0; i < bubbles.length; i++) {
    bubbles[i].display();
    bubbles[i].rollover(mouseX, mouseY);
  }
}
```

Exercise 18-15: Use the following XML document to initialize an array of objects. Design the objects to use all of the values in each XML element. (Feel free to rewrite the XML document to include more or less data.) If you do not want to retype the XML, it's available at this book's website.

```
<?xml version = "1.0"?>
<particles>
  <particle>
    <location x = "99" y = "192"/>
    <speed x = "-0.88238335 " y = "2.2704291"/>
    <size w = "38" h = "10"/>
  </particle>
  <particle>
    <location x = "97" y = "14"/>
    <speed x = "2.8775783" y = "2.9483867"/>
    <size w = "81" h = "43"/>
  </particle>
  <particle>
    <location x = "159" y = "193"/>
    <speed x = "-1.2341062" y = "0.44016743"/>
    <size w = "19" h = "95"/>
  </particle>
  <particle>
    <location x = "102" y = "53"/>
    <speed x = "0.8000488" y = "-2.2791147"/>
    <size w = "25" h = "95"/>
  </particle>
  <particle>
    <location x = "152" y = "181"/>
    <speed x = "1.9928784" y = "-2.9540048"/>
    <size w = "74" h = "19"/>
  </particle>
</particles>
```

In addition to `loadXML()`, Processing also includes a `saveXML()` function for writing XML files to your sketch folder. You can modify the XML tree itself by adding or removing elements with `addChild()` or `removeChild()`, as well as modify the content of elements or attributes with `setContent()`, `setIntContent()`, `setFloatContent()`, `setString()`, `setInt()`, and `setFloat()`.

Exercise 18-16: Add the functionality of creating new bubbles on mouse clicks to Example 18-7. Here is some code to get you started.

```
void mousePressed() {

  // Create a new XML bubble element

  XML bubble = xml.addChild(___________);

  XML position = bubble.addChild(__________);

  position.setInt("x", __________);

  position.setInt(________, __________);

  XML diameter = bubble.addChild(__________));

  diameter.__________(random(40, 80));
  XML label = bubble.addChild("label");

  label.__________("new label");

  saveXML(xml, "data/data.xml");
  loadData();
}
```

18-10 JSON

Another increasingly popular data exchange format is JSON (pronounced like the name Jason), which stands for JavaScript Object Notation. Its design was based on the syntax for objects in the JavaScript programming language (and is most commonly used to pass data between web applications) but has become rather ubiquitous and language-agnostic. While you don't need to know anything about JavaScript to work in Processing, it won't hurt to get a sense of some basic JavaScript syntax while learning it.

JSON is an alternative to XML and the data can be looked at in a similarly tree-like manner. All JSON data comes in the following two ways: an object or an array. Luckily, you already know about these two concepts and only need to learn a new syntax for encoding them.

Let's take a look at a JSON object first. A JSON object is like a Processing object only with no functions. It's simply a collection of variables with a name and a value (or "name/value pair"). For example, following is JSON data describing a person:

```
{
  "name":"Olympia",
  "age":3,
  "height":96.5,
  "state":"giggling"
}
```

Each name/value pair is separated by a comma.

Notice how this maps closely to classes in Processing.

```
class Person {
  String name;
  int age;
  float height;
  String state;
}
```

There are no classes in JSON, only the object literals themselves. Also an object can contain, as part of itself, another object.

```
{
  "name":"Olympia",
  "age":3,
  "height":96.5,
  "state":"giggling",
  "brother":{
    "name":"Elias",
    "age":6
  }
}
```

The value of "brother" is an object containing two name/value pairs.

In XML, the preceding JSON data would look like the following (for simplicity I'm avoiding the use of XML attributes).

```
<xml version="1.0" encoding="UTF-8"?>
<person>
  <name>Olympia</name>
  <age>3</age>
  <height>96.5</height>
  <state>giggling</state>
  <brother>
    <name>Elias</name>
    <age>6</age>
  </brother>
</person>
```

Multiple JSON objects can appear in the data as an array. Just like the arrays you use in Processing, a JSON array is simply a list of values (primitives or objects). The syntax, however, is different with square brackets indicating the use of an array rather than curly ones. Here is a simple JSON array of integers:

```
[1, 7, 8, 9, 10, 13, 15]
```

You might find an array as part of an object.

```
{
  "name":"Olympia",
  "favorite colors":[
    "purple",
    "blue",
    "pink"
  ]
}
```

The value of "favorite colors" is an array of strings.

Or an array of objects themselves. For example, here is what the bubbles would look like in JSON. Notice how this JSON data is organized as a single JSON object "bubbles," which contains a JSON array of JSON objects, the bubbles. Flip back to compare to the CSV and XML versions of the same data.

```
{
  "bubbles":[
    {
      "position":{
        "x":160,
        "y":103
      },
      "diameter":43.19838,
      "label":"Happy"
    },
    {
      "position":{
        "x":372,
        "y":137
      },
      "diameter":52.42526,
      "label":"Sad"
    },
    {
      "position":{
        "x":273,
        "y":235
      },
      "diameter":61.14072,
      "label":"Joyous"
    }
  ]
}
```

Exercise 18-17: Make up the rest of this missing JSON (loosely based on a weather API).

```
{
  "cities": [
    {
      "name":"New York",
      "weather":{

        "high":___________________,


        ___________________
      },
      "wind"___________________


      ___________________


      }
    },
    {
      "name":___________________,
      "weather":{

        "high":___________________,


        ___________________
      },


      ___________________


        ___________________


      ___________________


    ___________________


  ___________________


___________________
}
```

18-11 JSONObject and JSONArray

Now that I've covered the syntax of JSON, I can look at using the data in Processing. One thing that can be a bit tricky about working with JSON in Processing is that I have to treat objects and arrays differently. With XML, I simply had a single `XML` class with all the parsing functionality I needed. With JSON, I have two classes: `JSONObject` and `JSONArray`, and I'll have to be conscientious about picking which class I am using during the parsing process.

The first step is simply loading the data with either `loadJSONObject()` or `loadJSONArray()`. But which one? I have to look and see what is at the root of the JSON file, an object or array. This can be a little tricky. Let's look at these two JSON examples:

Sample 1:

```
[
  {
    "name":"Elias"
  },
  {
    "name":"Olympia"
  }
]
```

Sample 2:

```
{
  "names":[
    {
      "name":"Elias"
    },
    {
      "name":"Olympia"
    }
  ]
}
```

Look how similar the above two samples look. They both contain exactly the same data, two names "Elias" and "Olympia." There is one, very key difference, however, to how the data is formatted: the very first character. Is it a "[" or a "{"? The answer will determine whether you're loading an array ("[") or an object("{").

```
JSONObject json = loadJSONObject("file.json");
```

JSON objects start with a curly bracket: {

```
JSONArray json = JSONArray("file.json");
```

JSON arrays start with a square bracket: [

Typically, even if the data is ultimately organized as an array of objects (such as the array of "bubble" objects), the root element of the JSON data will be an object that contains that array. Let's look at the bubble data one more time.

```
{
  "bubbles":[
    {
      "position":{
        "x":160,
        "y":103
      },
      "diameter":43.19838,
      "label":"Happy"
    },
    {
      "position":{
        "x":372,
        "y":137
      },
      "diameter":52.42526,
      "label":"Sad"
    }
  ]
}
```

With the above, I first load an object and then pull the array out of that object.

```
JSONObject json = loadJSONObject("data.json");
JSONArray bubbleData = json.getJSONArray("bubbles");
```

Load the entire JSON file as an object.

Pull the array of bubbles out of that object.

Just as with XML, the data from an element is accessed via its name, in this case "bubbles." With a `JSONArray`, however, each element of the array is retrieved via its numeric index.

```
for (int i = 0; i < bubbleData.size(); i++) {
  JSONObject bubble = bubbleData.getJSONObject(i);
}
```

Iterating over a `JSONArray`.

And when you're looking for a specific piece of data from a `JSONObject`, such as an integer or string, the functions are identical to those of XML attributes.

```
  JSONObject position = bubble.getJSONObject("position");

  int x = position.getInt("x");
  int y = position.getInt("y");

  float diameter = bubble.getFloat("diameter");
  String label = bubble.getString("label");
```

Get the position object from the bubble object.

Get x and y as integers from the position object.

Diameter and label are available directly from the `Bubble` object.

Putting this all together, I can now make a JSON version of the bubbles example (leaving out the `draw()` function and `Bubble` class which haven't changed.)

Example 18-8. Using Processing's JSON classes

```
// An Array of Bubble objects
Bubble[] bubbles;

void setup() {
  size(480, 360);
  loadData();
}

void loadData() {
  JSONObject json = loadJSONObject("data.json");
  JSONArray bubbleData = json.getJSONArray("bubbles");


  bubbles = new Bubble[bubbleData.size()];


  for (int i = 0; i < bubbleData.size(); i++) {

    JSONObject bubble = bubbleData.getJSONObject(i);

    // Get a position object
    JSONObject position = bubble.getJSONObject("position");
    // Get (x,y) from JSON object "position"
    int x = position.getInt("x");
    int y = position.getInt("y");

    // Get diamter and label
    float diameter = bubble.getFloat("diameter");
    String label = bubble.getString("label");

    bubbles[i] = new Bubble(x, y, diameter, label);
  }
}
```

Load the JSON file and grab the array.

The size of the array of `Bubble` objects is determined by the length of the JSON array

Iterate through the array, grabbing each JSON object one at a time.

Put the `Bubble` objects into an array.

Exercise 18-18: Retrieve the description and current temperature using the following JSON from openweathermap.org.

```
{
  "weather":[
    {
      "id":801,
      "main":"Clouds",
      "description":"few clouds",
      "icon":"02d"
    }
  ],
  "main":{
    "temp":73.45,
    "humidity":83,
    "pressure":999,
    "temp_min":70,
    "temp_max":75.99
  }
}
JSONObject json = loadJSONObject(
   "http://api.openweathermap.org/data/2.5/weather?q=New%20York");

JSONObject main = json.getJSONObject(____________);

int temp = main.____________("temp");

// Grab the description (this is just one way to do it)

____________ weather = json.getJSON___________("weather");

String des = weather.getJSONObject(____)._________(_________);
```

18-12 Threads

As you have seen, the various loading functions — `loadStrings()`, `loadTable()`, `loadXML()`, and `loadJSONObject()` — can be used for retrieving data from URLs. Nonetheless, unless your sketch only needs to load the data once during `setup()`, you may have a problem. For example, consider a sketch that grabs the price of AAPL stock from an XML feed every five minutes. Each time `loadXML()` is called, the sketch will pause while waiting to receive the data. Any animation will stutter. This is because these loading functions are "blocking." In other words, the sketch will sit and wait at that line of code until `loadXML()` completes its task. With a local data file, this is extremely fast. Nonetheless, a request for a URL (known as an "HTTP request") in Processing is *synchronous*, meaning your sketch waits for a response from the server before continuing. Who knows how will that take? No one; you are at the mercy of the server!

The answer to this problem lies in the concept of *threads*. By now you are quite familiar with the idea of writing a program that follows a specific sequence of steps — `setup()` first then `draw()` over and over and over again! A thread is also a series of steps with a beginning, a middle, and an end. A Processing sketch is a single thread, often referred to as the *animation* thread. Other threads' sequences, however, can run independently of the main animation loop. In fact, you can launch any number of threads at one time, and they will all run concurrently.

Processing does this quite often, such as with library functions like `captureEvent()` and `movieEvent()`. These functions are triggered by a different thread running behind the scenes, and they alert Processing whenever they have something to report. This is useful when you need to perform a task that takes too long and would slow down the main animation's frame rate, such as grabbing data from the network. Here, you want to handle the request *asynchronously* in a different thread. If that thread gets stuck or has an error, the entire program won't grind to a halt, since the error only stops that individual thread and not the main animation loop.

Writing your own thread can be a complex endeavor that involves extending the Java `Thread` class. However, the `thread()` method is a quick and dirty way to implement a simple thread in Processing. By passing in a string that matches the name of a function declared elsewhere in the sketch, Processing will execute that function in a separate thread. Let's take a look at a skeleton of how this works.

```
void setup() {
  thread("someFunction");
}

void draw() {

}

void someFunction() {
  // This function will run as a thread when called via
  // thread("someFunction") as it was in setup!
}
```

The `thread()` function receives a string as an argument. The string should match the name of the function you want to run as a thread. In the above example it's `"someFunction"`.

Let's look at a more practical example. For an example of data that changes often, I'll use time.jsontest.com which gives you the current time (in milliseconds). While I could retrieve this from the system clock, this works well for demonstrating continuously requesting data that changes over time. Not knowing about threads, my first instinct might be to say:

```
void draw() {
  JSONObject json = loadJSONObject("http://time.jsontest.com/");
  String time = json.getString("time");
  text(time, 40, 100);
}
```

The code will stop here and wait to receive the data before moving on.

This would give me the current time every cycle through `draw()`. If I examine the frame rate, however, I'll notice that the sketch is running incredibly slowly (and all it needs to do is draw a single string!). This is where calling the parsing code as a separate thread will help a lot.

```
String time = "";

void draw() {
  thread("requestData");
  text(time, 40, 100);
}

void requestData() {
  JSONObject json = loadJSONObject("http://time.jsontest.com/");
  time = json.getString("time");
}
```

Now the code will move on to the next line while `requestData()` executes in a separate thread.

The logic is identical, only I am not requesting the data directly in `draw()`, but executing that request as a separate thread. Notice that I am not doing any drawing in `requestData()`. This is key as executing drawing functions in code that runs on a separate thread can cause conflicts with the main animation thread (i.e., `draw()`) resulting in strange behavior and errors.

In the above example, I likely don't want to request the data sixty times per second (the default frame rate). Instead I might make use of the `Timer` class from Section 10-6 on page 201, and ask for the data once per second. Here is a full example that does exactly that with added animation to show that `draw()` never stutters.

Example 18-9. Threads

```
Timer timer = new Timer(1000);
String time = "";

void setup() {
  size(200, 200);
  thread("retrieveData");
  timer.start();
}

void draw() {
  background(255);
  if (timer.isFinished()) {
    retrieveData();

    timer.start();
  }

  fill(0);
  text(time, 40, 100);

  translate(20, 100);
  stroke(0);
  rotate(frameCount*0.04);
  for (int i = 0; i < 10; i++) {
    rotate(radians(36));
    line(5, 0, 10, 0);
  }
}

// get the data
void retrieveData() {
  JSONObject json = loadJSONObject("http://time.jsontest.com/");
  time = json.getString("time");
}
```

Start by requesting the data asynchronously in a thread.

Every one second, make a new request.

And restart the timer.

Here I draw a little animation to demonstrate that the `draw()` loop never pauses.

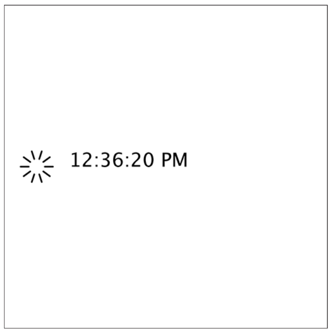


Figure 18-19

Exercise 18-19: Update the weather XML or weather JSON example to request the data in a thread.

18-13 APIs

It's a bit silly for me to call this section "APIs" given that most of this chapter is about data from APIs. Still, it's worth taking a moment to pause and reflect. What makes something an API versus just some data you found, and what are some pitfalls you might run into when using an API?

As I've stated, an API (Application Programming Interface) is an interface through which one application can access the services of another. These can come in many forms. Openweathermap.org, as you saw in Exercise 18-18 on page 421, is an API that offers its data in JSON, XML, and HTML formats. The key element that makes this service an API is exactly that offer; openweathermap.org's sole purpose in life is to offer you its data. And not just offer it, but allow you to query it for specific data in a specific format. Let's look at a short list of sample queries.

http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139
A request for current weather data for a specific latitude and longitude.

http://api.openweathermap.org/data/2.5/forecast/daily?q=London&mode=xml&units=metric&cnt=7&lang=zh_cn
A request for a seven day London forecast in XML format with metric units and in Chinese.

http://api.openweathermap.org/data/2.5/history/station?id=5091&type=day
A request for a historical data for a given weather station.

One thing to note about openweathermap.org is that it does not require that you tell the API any information about yourself. You simply send a request to a URL and get the data back. Other APIs, however, require you to sign up and obtain an access token. *The New York Times* API is one such example. Before you can make a request from Processing, you'll need to visit *The New York Times* Developer site and request an API key. Once you have that key, you can store it in your code as a string.

```
// This is not a real key
String apiKey = "40e2es0b3ca44563f9c62aeded4431dc:12:51913116";
```

You also need to know what the URL is for the API itself. This information is documented for you on the developer site, but here it is for simplicity:

```
String url = "http://api.nytimes.com/svc/search/v2/articlesearch.json";
```

Finally, you have to tell the API what it is you are looking for. This is done with a "query string," a sequence of name value pairs describing the parameters of the query joined with an ampersand. This functions similarly to how you pass arguments to a function in Processing. If you wanted to search for the term "processing" from a `search()` function you might say:

```
search("processing");
```

Here, the API acts as the function call, and you send it the arguments via the query string. Here is a simple example asking for a list of the oldest articles that contain the term "processing" (the oldest of which turns out to be May 12th, 1852).

```
String query = "?q=processing&sort=oldest";
```

The name/value pairs that configure the API query are: (q,processing) and (sort,oldest)

This isn't just guesswork. Figuring out how to put together a query string requires reading through the API's documentation. For *The New York Times*, it's all outlined on the *Times'* developer website. Once you have your query you can join all the pieces together and pass it to `loadJSONObject()`. Here is a tiny example that simply displays the most recent headline.

Example 18-10. NYTimes API query

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

  String apiKey = "40e2ea0b3ca44563f9c62aeded0431dc:18:51513116";
  String url = "http://api.nytimes.com/svc/search/v2/articlesearch.json";
  String query = "?q=processing&sort=newest";

  // Make the API query
  JSONObject json = loadJSONObject(url+query+"&api-key="+apiKey);
```

Here, I format the call to the API by joing the URL with the API key with the query string.

```
  String headline = json.getJSONObject("response").getJSONArray("docs").
    getJSONObject(0).getJSONObject("headline").getString("main");
  background(255);
  fill(0);
  text(headline, 10, 10, 180, 190);
}
```

Grabbing a single headline from the results.

Some APIs require a deeper level of authentication beyond an API access key. Twitter, for example, uses an authentication protocol known as "OAuth" to provide access to its data. Writing an OAuth application requires more than just passing a String into a request and is beyond the scope of this book. However, in these cases, if you're lucky, you can find a Processing library that handles all of the authentication for you. There are several APIs that can be used directly with Processing via a library, and you can find a list of them in the "Data / Protocols" section of the libraires reference page for some ideas. Temboo, for example, offers a Processing library that handles OAuth for you and provides direct access to many APIs (including Twitter) in Processing. With Temboo, you can write code that looks like:

```
TembooSession session = new TembooSession("ACCOUNT_NAME", "APP_NAME", "APP_KEY");
```

Temboo acts as a go-between you and Twitter, so first you just authenticate with Temboo.

```
Tweets tweets = new Tweets(session);
tweets.setCredential("your-twitter-name");
tweetsChoreo.setQuery("arugula");
TweetsResultSet tweetsResults = tweets.run();
```

Then you can configure a query to send to Twitter itself and grab the results.

```
JSONObject searchResults = parseJSONObject(tweetsResults.getResponse());

JSONArray statuses = searchResults.getJSONArray("statuses");

JSONObject tweet = statuses.getJSONObject(0);
String tweetText = tweet.getString("text");
```

Finally, you can search through the results and grab a tweet.