2 Processing

"Computers in the future may weigh no more than 1.5 tons."
—Popular Mechanics, 1949

"Take me to your leader."
—Zoog, 2008

In this chapter:
- Downloading and installing *Processing*.
- Menu options.
- A *Processing* "sketchbook."
- Writing code.
- Errors.
- The *Processing* reference.
- The "Play" button.
- Your first sketch.
- Publishing your sketch to the web.

2.1 *Processing* to the Rescue

Now that we conquered the world of primitive shapes and RGB color, we are ready to implement this knowledge in a real world programming scenario. Happily for us, the environment we are going to use is *Processing*, free and open source software developed by Ben Fry and Casey Reas at the MIT Media Lab in 2001. (See this book's introduction for more about *Processing*'s history.)

Processing's core library of functions for drawing graphics to the screen will provide for immediate visual feedback and clues as to what the code is doing. And since its programming language employs all the same principles, structures, and concepts of other languages (specifically Java), everything you learn with *Processing* is *real* programming. It is not some pretend language to help you get started; it has all the fundamentals and core concepts that all languages have.

After reading this book and learning to program, you might continue to use *Processing* in your academic or professional life as a prototyping or production tool. You might also take the knowledge acquired here and apply it to learning other languages and authoring environments. You may, in fact, discover that programming is not your cup of tea; nonetheless, learning the basics will help you become a better-informed technology citizen as you work on collaborative projects with other designers and programmers.

It may seem like overkill to emphasize the *why* with respect to *Processing*. After all, the focus of this book is primarily on learning the fundamentals of computer programming in the context of computer graphics and design. It is, however, important to take some time to ponder the reasons behind selecting a programming language for a book, a class, a homework assignment, a web application, a software suite, and so forth. After all, now that you are going to start calling yourself a computer programmer at cocktail parties, this question will come up over and over again. I need programming in order to accomplish project *X*, what language and environment should I use?

I say, without a shadow of doubt, that for you, the beginner, the answer is *Processing*. Its simplicity is ideal for a beginner. At the end of this chapter, you will be up and running with your first computational design and ready to learn the fundamental concepts of programming. But simplicity is not where *Processing*

ends. A trip through the *Processing* online exhibition (*http://processing.org/exhibition/*) will uncover a wide variety of beautiful and innovative projects developed entirely with *Processing*. By the end of this book, you will have all the tools and knowledge you need to take your ideas and turn them into real world software projects like those found in the exhibition. *Processing* is great both for learning and for producing, there are very few other environments and languages you can say that about.

2.2 How do I get *Processing*?

For the most part, this book will assume that you have a basic working knowledge of how to operate your personal computer. The good news, of course, is that *Processing* is available for free download. Head to *http://www.processing.org/* and visit the download page. If you are a Windows user, you will see two options: "Windows (standard)" and "Windows (expert)." Since you are reading this book, it is quite likely you are a beginner, in which case you will want the standard version. The expert version is for those who have already installed Java themselves. For Mac OS X, there is only one download option. There is also a Linux version available. Operating systems and programs change, of course, so if this paragraph is obsolete or out of date, visit the download page on the site for information regarding what you need.

The *Processing* software will arrive as a compressed file. Choose a nice directory to store the application (usually "c:\Program Files\" on Windows and in "Applications" on Mac), extract the files there, locate the "Processing" executable, and run it.

Exercise 2-1: Download and install Processing.

2.3 The *Processing* Application

The *Processing* development environment is a simplified environment for writing computer code, and is just about as straightforward to use as simple text editing software (such as TextEdit or Notepad) combined with a media player. Each sketch (*Processing* programs are referred to as "sketches") has a filename, a place where you can type code, and some buttons for saving, opening, and running sketches. See Figure 2.1.

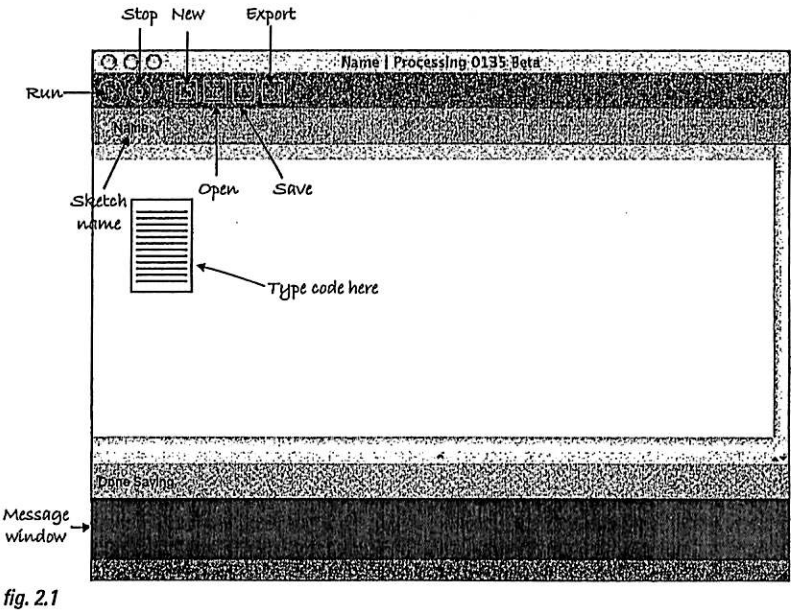


fig. 2.1

To make sure everything is working, it is a good idea to try running one of the *Processing* examples. Go to FILE → EXAMPLES → (pick an example, suggested: Topics → Drawing → ContinuousLines) as shown in Figure 2.2.

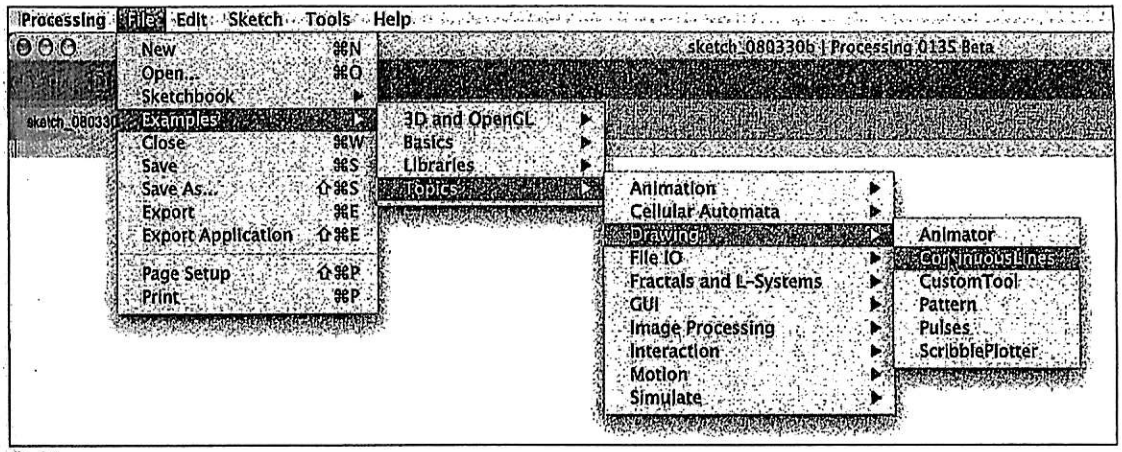


fig. 2.2

Once you have opened the example, click the "run" button as indicated in Figure 2.3. If a new window pops open running the example, you are all set! If this does not occur, visit the online FAQ "Processing won't start!" for possible solutions. The page can be found at this direct link: *http://www.processing.org/faq/bugs.html#wontstart.*

Exercise 2-2: Open a sketch from the Processing *examples and run it.*

fig. 2.3

Processing programs can also be viewed full-screen (known as "present mode" in *Processing*). This is available through the menu option: Sketch → Present (or by shift-clicking the run button). Present will not resize your screen resolution. If you want the sketch to cover your entire screen, you must use your screen dimensions in ***size()***.

2.4 The Sketchbook

Processing programs are informally referred to as *sketches*, in the spirit of quick graphics prototyping, and we will employ this term throughout the course of this book. The folder where you store your sketches is called your "sketchbook." Technically speaking, when you run a sketch in *processing*, it runs as a local application on your computer. As we will see both in this Chapter and in Chapter 18, *Processing* also allows you to export your sketches as web applets (mini-programs that run embedded in a browser) or as platform-specific stand-alone applications (that could, for example, be made available for download).

Once you have confirmed that the *Processing* examples work, you are ready to start creating your own sketches. Clicking the "new" button will generate a blank new sketch named by date. It is a good idea to "Save as" and create your own sketch name. (Note: *Processing* does not allow spaces or hyphens, and your sketch name cannot start with a number.)

When you first ran *Processing*, a default "Processing" directory was created to store all sketches in the "My Documents" folder on Windows and in "Documents" on OS X. Although you can select any directory on your hard drive, this folder is the default. It is a pretty good folder to use, but it can be changed by opening the *Processing* preferences (which are available under the FILE menu).

Each *Processing* sketch consists of a folder (with the same name as your sketch) and a file with the extension "pde." If your *Processing* sketch is named *MyFirstProgram*, then you will have a folder named *MyFirstProgram* with a file *MyFirstProgram.pde* inside. The "pde" file is a plain text file that contains the source code. (Later we will see that *Processing* sketches can have multiple pde's, but for now one will do.) Some sketches will also contain a folder called "data" where media elements used in the program, such as image files, sound clips, and so on, are stored.

Exercise 2-3: Type some instructions from Chapter 1 into a blank sketch. Note how certain words are colored. Run the sketch. Does it do what you thought it would?

2.5 Coding in *Processing*

It is finally time to start writing some code, using the elements discussed in Chapter 1. Let's go over some basic syntax rules. There are three kinds of statements we can write:

- Function calls
- Assignment operations
- Control structures

For now, every line of code will be a function call. See Figure 2.4. We will explore the other two categories in future chapters. Functions have a name, followed by a set of arguments enclosed in parentheses. Recalling Chapter 1, we used functions to describe how to draw shapes (we just called them "commands" or "instructions"). Thinking of a function call as a natural language sentence, the function name is the verb ("draw") and the arguments are the objects ("point 0,0") of the sentence. Each function call must always end with a semicolon. See Figure 2.5.

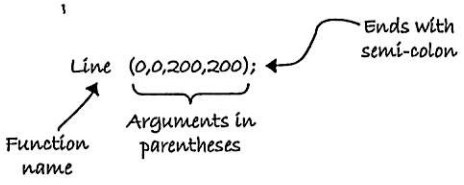


fig. 2.4

We have learned several functions already, including ***background(), stroke(), fill(), noFill (), noStroke(), point(), line(), rect(), ellipse(), rectMode(),*** and ***ellipseMode()***. *Processing* will execute a sequence of functions one by one and finish by displaying the drawn result in a window. We forgot to learn one very important function in Chapter 1, however—***size()***. ***size()*** specifies the dimensions of the window you want to create and takes two arguments, width and height. The ***size()*** function should always be first.

```
size(320,240);
```

Opens a window of width 320 and height 240.

Let's write a first example (see Figure 2.5).

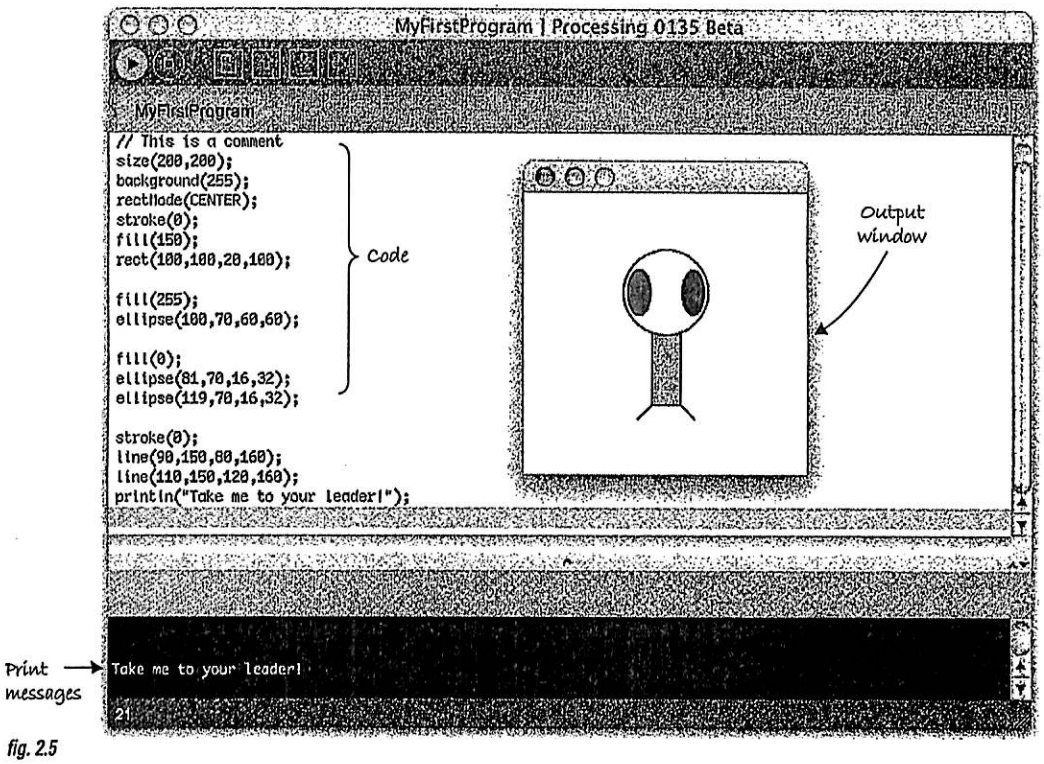


fig. 2.5

There are a few additional items to note.

- The *Processing* text editor will color *known* words (sometimes referred to as "reserved" words or "keywords"). These words, for example, are the drawing functions available in the *Processing* library, "built-in" variables (we will look closely at the concept of *variables* in Chapter 3) and constants, as well as certain words that are inherited from the Java programming language.
- Sometimes, it is useful to display text information in the *Processing* message window (located at the bottom). This is accomplished using the ***println()*** function. ***println()*** takes one argument, a *String* of characters enclosed in quotes (more about *Strings* in Chapter 14). When the program runs, *Processing* displays that *String* in the message window (as in Figure 2.5) and in this case the *String* is "Take me to your leader!" This ability to print to the message window comes in handy when attempting to *debug* the values of variables (see Chapter 12, Debugging).
- The number in the bottom left corner indicates what line number in the code is selected.
- You can write "comments" in your code. Comments are lines of text that *Processing* ignores when the program runs. You should use them as reminders of what the code means, a bug you intend to fix, or a to do list of items to be inserted, and so on. Comments on a single line are created with two forward slashes, //. Comments over multiple lines are marked by /* followed by the comments and ending with */.

```
// This is a comment on one line

/* This is a comment that
spans several lines
of code */
```

A quick word about comments. You should get in the habit right now of writing comments in your code. Even though our sketches will be very simple and short at first, you should put comments in for everything. Code is very hard to read and understand without comments. You do not need to have a comment for every line of code, but the more you include, the easier a time you will have revising and reusing your code later. Comments also force you to understand how code works as you are programming. If you do not know what you are doing, how can you write a comment about it?

Comments will not always be included in the text here. This is because I find that, unlike in an actual program, code comments are hard to read in a book. Instead, this book will often use code "hints" for additional insight and explanations. If you look at the book's examples on the web site, though, comments will always be included. So, I can't emphasize it enough, write comments!

```
//A comment about this code
line(0,0,100,100);
```

A hint about this code!

Exercise 2-4: Create a blank sketch. Take your code from the end of Chapter 1 and type it in the Processing *window. Add comments to describe what the code is doing. Add a* ***println()*** *statement to display text in the message window. Save the sketch. Press the "run" button. Does it work or do you get an error?*

2.6 Errors

The previous example only works because we did not make any errors or typos. Over the course of a programmer's life, this is quite a rare occurrence. Most of the time, our first push of the play button will not be met with success. Let's examine what happens when we make a mistake in our code in Figure 2.6.

Figure 2.6 shows what happens when you have a typo—"elipse" instead of "ellipse" on line 9. If there is an error in the code when the play button is pressed, *Processing* will not open the sketch window, and will instead display the error message. This particular message is fairly friendly, telling us that we probably meant to type "ellipse." Not all *Processing* error messages are so easy to understand, and we will continue to look at other errors throughout the course of this book. An Appendix on common errors in Processing is also included at the end of the book.

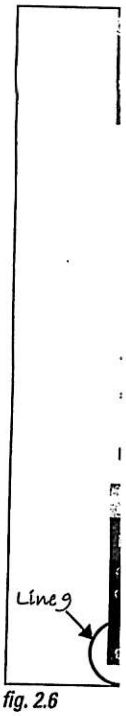


fig. 2.6

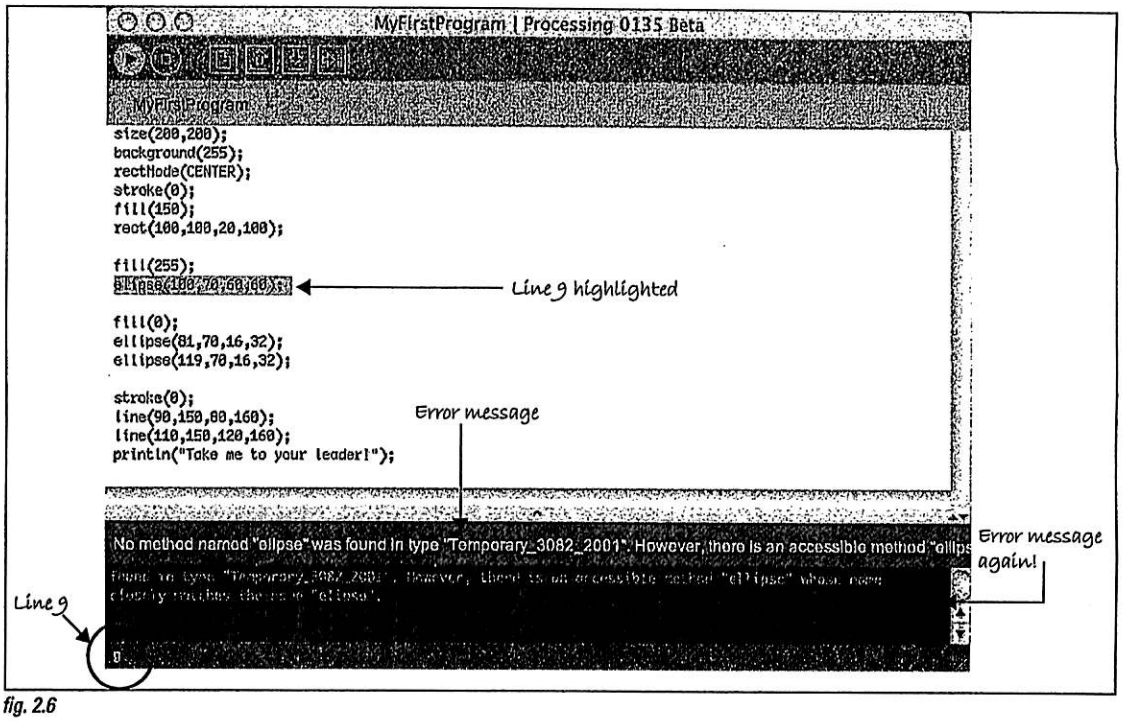


fig. 2.6

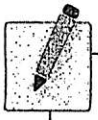
Processing *is case sensitive!*

If you type *Ellipse* instead of *ellipse*, that will also be considered an error.

In this instance, there was only one error. If multiple errors occur, *Processing* will only alert you to the first one it finds (and presumably, once that error is corrected, the next error will be displayed at run time). This is somewhat of an unfortunate limitation, as it is often useful to have access to an entire list of errors when fixing a program. This is simply one of the trade-offs we get in a simplified environment such as *Processing*. Our life is made simpler by only having to look at one error at a time, nevertheless we do not have access to a complete list.

This fact only further emphasizes the importance of incremental development discussed in the book's introduction. If we only implement one feature at a time, we can only make one mistake at a time.

Exercise 2-5: Try to make some errors happen on purpose. Are the error messages what you expect?



Exercise 2-6: Fix the errors in the following code.

```
size(200,200;
background();
stroke 255;
fill(150)
rectMode(center);
rect(100,100,50);
```

2.7 The *Processing* Reference

The functions we have demonstrated—***ellipse(), line(), stroke(),*** and so on—are all part of *Processing*'s library. How do we know that "ellipse" isn't spelled "elipse", or that ***rect()*** takes four arguments (an "*x* coordinate," a "*y* coordinate," a "width," and a "height")? A lot of these details are intuitive, and this speaks to the strength of *Processing* as a beginner's programming language. Nevertheless, the only way to know for sure is by reading the online reference. While we will cover many of the elements from the reference throughout this book, it is by no means a substitute for the reference and both will be required for you to learn *Processing*.

The reference for *Processing* can be found online at the official web site (*http://www.processing.org*) under the "reference" link. There, you can browse all of the available functions by category or alphabetically. If you were to visit the page for ***rect()***, for example, you would find the explanation shown in Figure 2.7.

As you can see, the reference page offers full documentation for the function ***rect()***, including:

- **Name**—The name of the function.
- **Examples**—Example code (and visual result, if applicable).
- **Description**—A friendly description of what the function does.
- **Syntax**—Exact syntax of how to write the function.
- **Parameters**—These are the elements that go inside the parentheses. It tells you what kind of data you put in (a number, character, etc.) and what that element stands for. (This will become clearer as we explore more in future chapters.) These are also sometimes referred to as "arguments."
- **Returns**—Sometimes a function sends something back to you when you call it (e.g., instead of asking a function to perform a task such as draw a circle, you could ask a function to add two numbers and *return* the answer to you). Again, this will become more clear later.
- **Usage**—Certain functions will be available for *Processing* applets that you publish online ("Web") and some will only be available as you run *Processing* locally on your machine ("Application").
- **Related Methods**—A list of functions often called in connection with the current function. Note that "functions" in Java are often referred to as "methods." More on this in Chapter 6.

fig. 2.7

Processing
go to to H

2.8 The

One of th
It is a nic

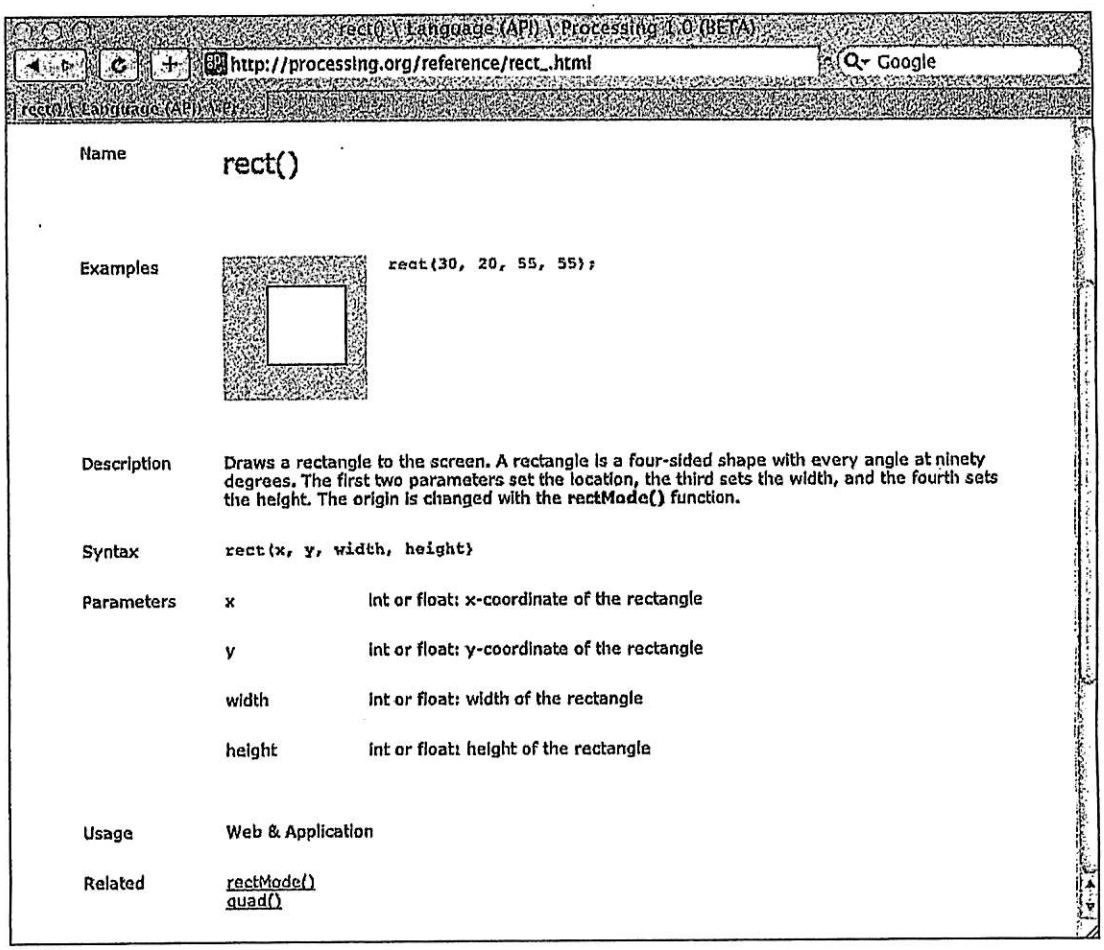


fig. 2.7

Processing also has a very handy "find in reference" option. Double-click on any keyword to select it and go to to HELP → FIND IN REFERENCE (or select the keyword and hit SHIFT+CNTRL+F).

Exercise 2-7: Using the Processing *reference, try implementing two functions that we have not yet covered in this book. Stay within the "Shape" and "Color (setting)" categories.*

Exercise 2-8: Using the reference, find a function that allows you to alter the thickness of a line. What arguments does the function take? Write example code that draws a line one pixel wide, then five pixels wide, then 10 pixels wide.

2.8 The "Play" Button

One of the nice qualities of *Processing* is that all one has to do to run a program is press the "play" button. It is a nice metaphor and the assumption is that we are comfortable with the idea of *playing* animations,

movies, music, and other forms of media. *Processing* programs output media in the form of real-time computer graphics, so why not just *play* them too?

Nevertheless, it is important to take a moment and consider the fact that what we are doing here is not the same as what happens on an iPod or TiVo. *Processing* programs start out as text, they are translated into machine code, and then executed to run. All of these steps happen in sequence when the play button is pressed. Let's examine these steps one by one, relaxed in the knowledge that *Processing* handles the hard work for us.

Step 1. Translate to Java. *Processing* is really Java (this will become more evident in a detailed discussion in Chapter 23). In order for your code to run on your machine, it must first be translated to Java code.

Step 2. Compile into Java byte code. The Java code created in Step 1 is just another text file (with the .java extension instead of .pde). In order for the computer to understand it, it needs to be translated into machine language. This translation process is known as compilation. If you were programming in a different language, such as C, the code would compile directly into machine language specific to your operating system. In the case of Java, the code is compiled into a special machine language known as Java byte code. It can run on different platforms (Mac, Windows, cellphones, PDAs, etc.) as long as the machine is running a "Java Virtual Machine." Although this extra layer can sometimes cause programs to run a bit slower than they might otherwise, being cross-platform is a great feature of Java. For more on how this works, visit http://java.sun.com or consider picking up a book on Java programming (after you have finished with this one).

Step 3. Execution. The compiled program ends up in a JAR file. A JAR is a Java archive file that contains compiled Java programs ("classes"), images, fonts, and other data files. The JAR file is executed by the Java Virtual Machine and is what causes the display window to appear.

2.9 Your First Sketch

Now that we have downloaded and installed *Processing*, understand the basic menu and interface elements, and have gotten familiar with the online reference, we are ready to start coding. As I briefly mentioned in Chapter 1, the first half of this book will follow one example that illustrates the foundational elements of programming: *variables, arrays, conditionals, loops, functions,* and *objects*. Other examples will be included along the way, but following just one will reveal how the basic elements behind computer programming build on each other.

The example will follow the story of our new friend Zoog, beginning with a static rendering with simple shapes. Zoog's development will include mouse interaction, motion, and cloning into a population of many Zoogs. While you are by no means required to complete every exercise of this book with your own alien form, I do suggest that you start with a design and after each chapter, expand the functionality of that design with the programming concepts that are explored. If you are at a loss for an idea, then just draw your own little alien, name it Gooz, and get programming! See Figure 2.8.

Example 2-

```
size(20
backgro
smooth(

// Set
ellipse
rectMod

// Draw
stroke(
fill(15
rect(1(

// Draw
fill(25
ellipse

// Dra
fill(0
ellips
ellips

// Dra
stroke
line(9
line(1
```

Let's prete
just cannot
significant
choices:

- Retyp
- Visit
  copy/

Certainly
resource fc
learning, tl
as you typ
running th
works.

You will k
running a
typing.

Example 2-1: Zoog again

```
size(200,200); // Set the size of the window
background(255); // Draw a white background
smooth();

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

// Draw Zoog's body
stroke(0);
fill(150);
rect(100,100,20,100);

// Draw Zoog's head
fill(255);
ellipse(100,70,60,60);

// Draw Zoog's eyes
fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);

// Draw Zoog's legs
stroke(0);
line(90,150,80,160);
line(110,150,120,160);
```

The function ***smooth()*** enables "anti-aliasing" which smooths the edges of the shapes. ***no smooth()*** disables anti-aliasing.

Zoog's body.

Zoog's head.

Zoog's eyes.

Zoog's legs.

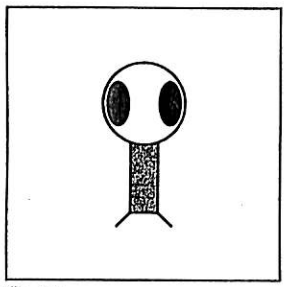


fig. 2.8

Let's pretend, just for a moment, that you find this Zoog design to be so astonishingly gorgeous that you just cannot wait to see it displayed on your computer screen. (Yes, I am aware this may require a fairly significant suspension of disbelief.) To run any and all code examples found in this book, you have two choices:

- Retype the code manually.
- Visit the book's web site (*http://www.learningprocessing.com*), find the example by its number, and copy/paste (or download) the code.

Certainly option #2 is the easier and less time-consuming one and I recommend you use the site as a resource for seeing sketches running in real-time and grabbing code examples. Nonetheless, as you start learning, there is real value in typing the code yourself. Your brain will sponge up the syntax and logic as you type and you will learn a great deal by making mistakes along the way. Not to mention simply running the sketch after entering each new line of code will eliminate any mystery as to how the sketch works.

You will know best when you are ready for copy/paste. Keep track of your progress and if you start running a lot of examples without feeling comfortable with how they work, try going back to manual typing.

*Exercise 2-9: Using what you designed in Chapter 1, implement your own screen drawing, using only 2D primitive shapes—**arc()**, **curve()**, **ellipse()**, **line()**, **point()**, **quad()**, **rect()**, **triangle()**—and basic color functions—**background()**, **colorMode()**, **fill()**, **noFill()**, **noStroke()**, and **stroke()**. Remember to use **size()** to specify the dimensions of your window. Suggestion: Play the sketch after typing each new line of code. Correct any errors or typos along the way.*

2.10 Publishing Your Program

After you have completed a *Processing* sketch, you can publish it to the web as a Java applet. This will become more exciting once we are making interactive, animated applets, but it is good to practice with a simple example. Once you have finished Exercise 2-9 and determined that your sketch works, select FILE → EXPORT.

Note that if you have errors in your program, it will not export properly, so always test by running first!

A new directory named "applet" will be created in the sketch folder and displayed, as shown in Figure 2.9.

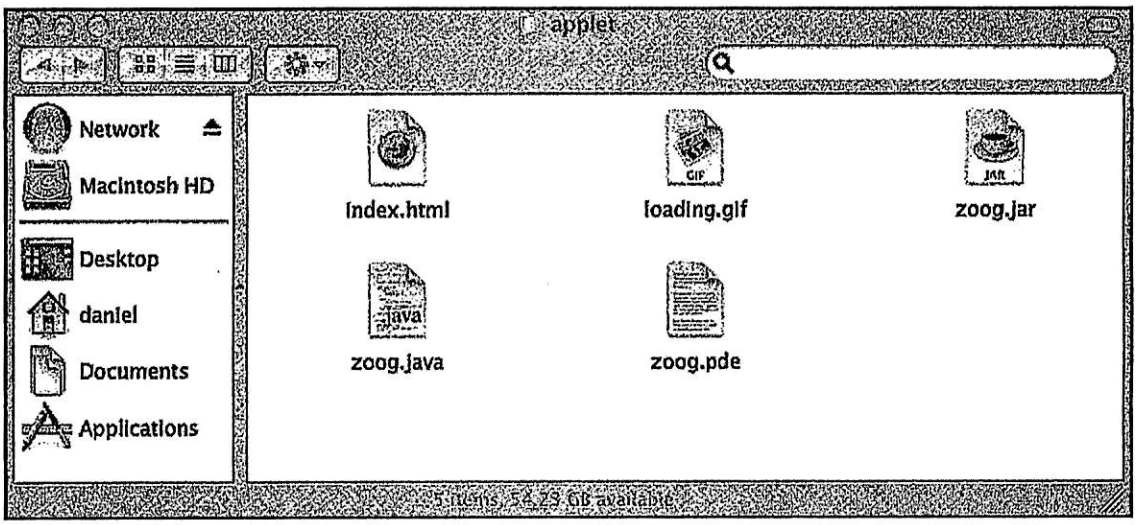


fig. 2.9

You now have the necessary files for publishing your applet to the web.

- **index.html**—The HTML source for a page that displays the applet.
- **loading.gif**—An image to be displayed while the user loads the applet (*Processing* will supply a default one, but you can create your own).
- **zoog.jar**—The compiled applet itself.
- **zoog.java**—The translated Java source code (looks like your *Processing* code, but has a few extra things that Java requires. See Chapter 20 for details.)
- **zoog.pde**—Your *Processing* source.

To see the a
web browse
(or you can
some tips fo

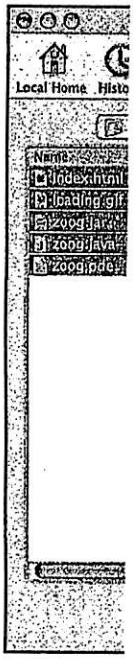


fig. 2.10

To see the applet working, double-click the "index.html" file which should launch a page in your default web browser. See Figure 2.10. To get the applet online, you will need web server space and FTP software (or you can also use a *Processing* sketch sharing site such as http://www.openprocessing.org). You can find some tips for getting started at this book's web site.

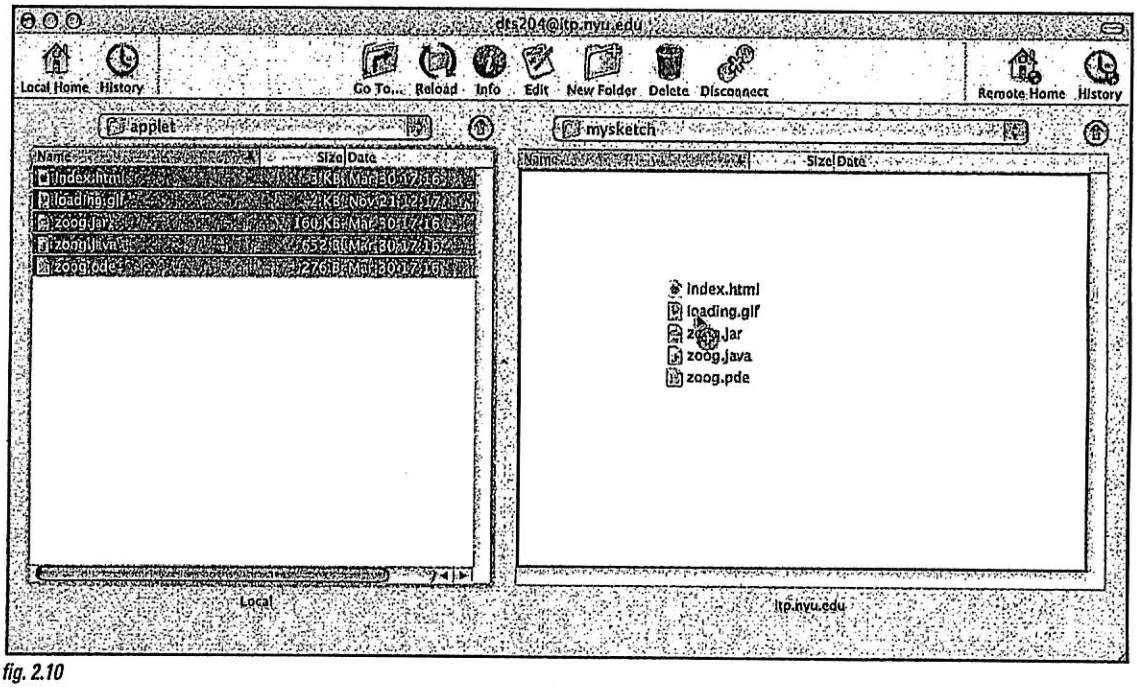


fig. 2.10

Exercise 2–10: Export your sketch as an applet. View the sketch in the browser (either locally or by uploading).