Processing for Visual Artists

SIGGRAPH 2010 Course Notes

Andrew Glassner
Coyote Wind Studios
andrew@coyote-wind.com

The following course notes are adapted from the book, *Processing for Visual Artists: How To Create Expressive Images and Interactive Art*, by Andrew Glassner, published by AK Peters (2010).

You can download Processing for free! Versions for Mac, Windows, and Linux, can be downloaded from

http://www.processing.org/download/

If you're on Windows, I suggest choosing "Windows" (not "Windows (Without Java)").

You can find documentation on every Processing language element, along with other features like tutorials, examples, and a user forum at the main Processing website:

http://www.processing.org

For errata and notes on my book, and more resources for learning Processing, visit:

http://www.coyote-wind.com/Processing

Chapter 2

Setting Up And Getting Started

The first step in getting started with Processing is to download and install it. That's easy, because Processing is free, and it runs beautifully on Macs, and PCs that use Windows, and even PCs that use Linux. You can download and install it without any worries about bugs, viruses, or other undesired surprises. Point your web browser at

`http://www.processing.org/`

Choose "Download" (if you're on a Windows-based PC, I suggest you choose the link labeled "Windows", and ignore the one labeled "Windows (Without Java)").

You may be offered your choice of versions. Most of the examples in this book were produced with version 1.0.7. Generally, you should pick the most recent version available. If your version is newer than the ones I used, everything in this book should still run perfectly well. You may even find that some of the bugs I encountered (and I point out in the text) have been fixed.

Once you've downloaded the package, install it by following the usual routine for installing programs on your system. The result is a program called Processing installed on your hard drive. Start it up!

When you run Processing, it will pop up a window for you. This is where you do all your communicating with Processing, so let's go over the pieces briefly so you'll know your way around. This window will soon feel like a second home to you.

Processing's window will look something like Figure 2.1.

At the top is the *menu bar*, which has the traditional sorts of menu items like *File*, *Edit*, and so on. You may be used to seeing these at the top of the screen (on Macs, for instance), but in Processing they're all there in the Processing window. Below the menu bar is the *toolbar*, which has just a half-dozen buttons. The round buttons let you *run* your program and *stop* it. The square buttons are shortcuts for opening and saving your files. Beneath the toolbar is a strip of *tabs*. There's one tab for each file in your project; just click on a tab to activate it. When you're starting a new project, there's just one tab with an automatically-generated default name. That name is usually derived from the current date; in the figure, you can see I took that screenshot on May 1. At the far

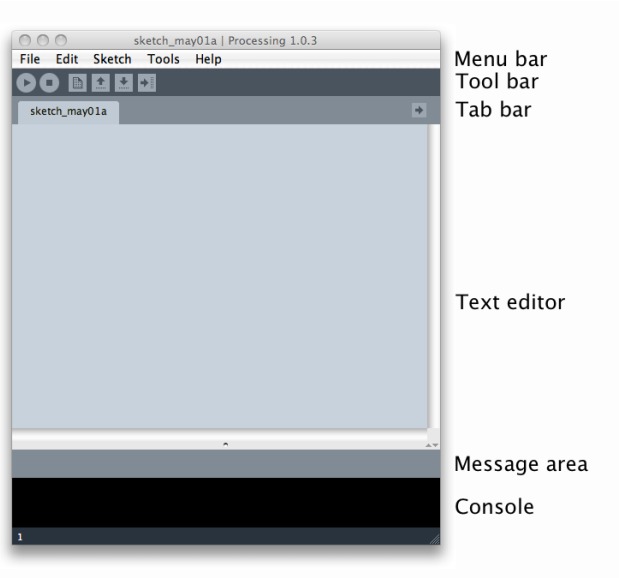

Figure 2.1: A Processing window. [ProcWindowElements.png]

right there's a button marked by an arrow in a box, which will open a new menu for tab-related operations.

Below the tab bar is the *text editor*, which is where you'll enter and edit your Processing programs. Below the text editor is the *message area*, where Processing will display any short, important messages, like error statements. Finally, at the bottom of the window is the *console*. This area of the screen does double duty. When you're getting your program to run, this is where the computer will print detailed information about any problems it discovers. When your program is running, if you tell Processing to print out information for you, it shows up here.

You can resize the window by dragging the bottom-right corner. Under the text editor and above the message area there's a little dot. If you drag that up and down you can change how much of the message area and console get displayed. In the bottom-right of the text editor are two tiny arrows. Clicking the downward-pointing arrow gets rid of the message area and console, so the text editor fills the window. Clicking the upward-pointing arrow restores the bottom two areas. Until you're a pro, I strongly suggest leaving the message area and console visible.

2.1 Hello, World!

Let's get Processing to do a little something right away so we know all is well.

Click in the text editor, and type the following exactly as it appears here:

```
println("Hello World!");
```

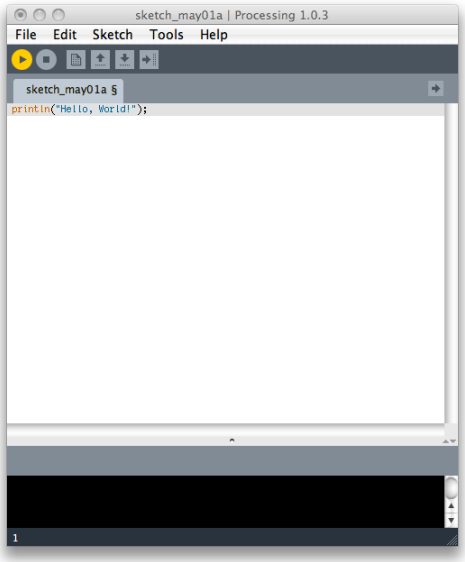

Figure 2.2: A sketch to print "Hello, World!" [HelloWorld.png]

(Full program in settingup/sketches/HelloWorld/HelloWorld.pde)

It should look like Figure 2.2.

Like most text editors, this window contains a *cursor* which indicates where your characters will go when you start to type. You can put the cursor anywhere you want in the text you've written using the mouse; just left-click where you want to start typing, and the cursor will go there. In most word processors the cursor is indicated with a visual symbol like a blinking box. In Processing, your cursor is a vertical blinking bar. Because this bar can be a little hard to spot sometimes, Processing helps you by highlighting in light gray the entire horizontal row of text around the cursor. The line number where the cursor is located is displayed in the bottom-left corner of the window, just under the console.

In the line I just asked you to type, the first word PRINTLN, but in lower case (in Processing, upper and lower case letters are considered different letters, so case matters). Note that the letter after the t is an l (as in lightbulb), and not an i (as in incandescent). You must enter the word `println` in just this way, with no spaces, and entirely lower-case; if you type it in any other way you'll get an error.

The name may make a bit more sense once you know that `println` is shorthand for "print line".

You might wonder, then, why it isn't just `printline`. After all, what was really saved by leaving off that last i and e? Not much, that's for sure. But programmers do this kind of thing all the time, leaving off vowels and extra letters to save a little bit of typing. I do it too, and you'll see me do it lots of times in the examples yet to come. For example, to keep track of the number of points in a drawing, I might name my counter `NumPoints`. Why not `NumberPoints`, or better yet, `NumberOfPoints`? There's

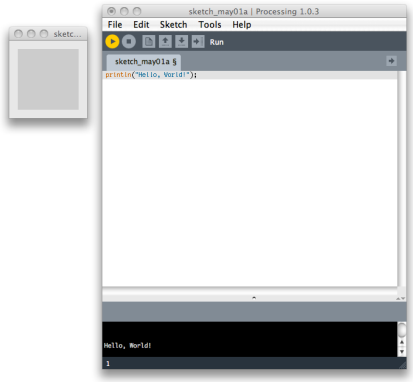

Figure 2.3: Running "Hello, World!" [HelloWorldRun.png]

no single reason for this. The more you have to type, the more you can make a typo, so short names have a little advantage there. They're also faster to type, and can be easier to read. I don't know, those might all just be rationalizations for being lazy. The bottom line is that this is the nearly-universal style, shared by programmers the world over. Some people use fully written-out and descriptive names, and you're welcome to join their elite ranks, but most people abbreviate this way throughout their code.

For example, in later chapters you'll see me build colors out of three values, one each for red, green, and blue. And when I make objects to hold those values, I often call the three objects some variant of `red`, `grn`, and `blu`. On the one hand, that makes them look good to my eye when they're on separate lines: they all have 3 letters and when they appear one under the other it reinforces the idea, at a glance, that they're basically a connected group. On the other hand, it's kind of silly. But as I said, this is a very common practice, so I'm not going to try to hide it from you. The designers of Processing were clearly part of this tradition, using many of these kinds of abbreviations in the language definition itself.

So the "Print Line" command is written `println`, and after a while, that'll seem perfectly natural to you, too.

OK, with all of that half-apology, half-explanation out of the way, let's let Processing loose.

Give it a shot! Click the *Run* button; it's the leftmost button in the tool bar, shown by a circle with a right-pointing triangle. Alternatively, you can run your program by clicking on the *Sketch* entry in the menu bar, and choosing *Run*. Or you can use your keyboard to type in the shortcut for *Run* (it will be listed next to the command on the menu). Figure 2.3 shows what happened on my system.

When you clicked the *Run* button, Processing probably did two things: it created a little square window filled with gray somewhere on the screen, and it printed "Hello World!" in the console at the bottom of the Processing window. If so, congratulations! You have a running copy of Processing, ready for playing and messing around with. If something went wrong, make sure you typed in that line exactly as I gave it. If you're

sure it's right (remember the parentheses, the quotes, and the semicolon at the end) but Processing is still not giving you this result, you can download the file *HelloWorld.pde* from the website. You can either open that from within Processing, or copy and paste it into your own text window. If that doesn't work either, try talking to a knowledgeable friend in order to determine what may have gone wrong, and how to put things right again. Finally, if things are still not working, try looking on the Processing website and user forums for help.

When you pressed the *Run* button, Processing ran through a two-step procedure. The first was to take the text in the text editor and translate it into another language, which made it more convenient for the computer. The second step was then to run that translated program. So your program first gets turned into something else, and then that something else takes over to create your pictures and animation.

I'm mentioning this because it will help you understand the two kinds of messages that Processing offers you when there's a problem with your program.

The first kind of message comes up when Processing does the translation step. What's happening is that Processing reads what you typed it and translates it into another language called *Java*. Sometimes we also say that the program is *compiled* into Java.

The relationship between Processing and Java is something like that between a general contractor and sub-contractors. If you're remodeling your kitchen, you probably spend most of your time talking to the general contractor. In some jobs you'll never talk to the subcontractors at all. But they're the ones actually doing the work. The general contractor might do some designing, and will choose the subcontractors and guide them, but the contractor is rarely pounding nails or gluing cabinets together. In that sense, most of what seems to be happening when you run a Processing program is actually being done by the Java system, running Java code. But in practice you rarely see the Java underpinnings.

So in this book I'll discuss everything as though Processing was the whole enchilada, even though we know that Java and your computer's operating system are major contributors to everything that goes on.

2.2 Debugging

During the first step of translation, Processing might find that it can't complete the job. Maybe you typed something incorrectly, or forgot a letter or a character. As I mentioned before, the *syntax* (or text) of the program at this point has to be perfect or the compilation into Java won't complete, which means you won't have a program to run.

To see this, let's deliberately break the program by introducing a bug. We'll put in a typographic error that will prevent the translation step from running to completion. Click in the text window and delete the closing parenthesis so the line looks like this:

```
println("Hello World!";
```

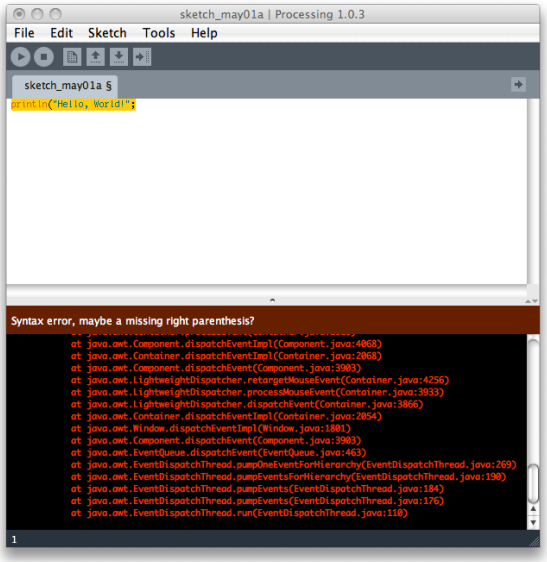

Figure 2.4: Error messages galore! [HelloWorldErrors.png]

Press the *Run* button.

Trouble! Errors! You might be seeing tons of red text in the console window, as in Figure 2.4 (I made the console bigger and wider here so we could see more of those lovely red messages).

Relax, everything is just fine. No need to be worried at all. We have some errors. They happen all the time. Every program you write will have errors. You'll slip up and miss a letter or a parenthesis. You'll spell `println` with an `i` and an `e`. There are a million possible syntax errors, but they're usually innocuous typos. The error reports come in a million flavors, and despite the scary red text, you should expect and welcome them. They're your friends. These error messages are there to help you make your code work as well as it can, and they exist to guide you to places where things are not as expected.

There are two big things to keep in mind when your program produces errors. First, it's no big deal. The most experienced programmer in the world expects to get some errors when they first try to run their code. You dig in and fix them, and that's just part of the process.

The second thing is to remember that you can't break the computer. As I mentioned earlier, you can't make it explode in a shower of sparks like the bridge on *Star Trek* when they get hit with a photon torpedo. You can't scramble your disk, erase your bank statements, or trigger a flood of outgoing spam email. Processing is a safe environment. Despite all that red type, everything is fine.

In fact, everything is better than fine. Processing, in the act of compiling your program into Java, has found something that doesn't look right. Rather than just plow on and finish making the Java program and then running it, only to have it crash or

fail in some confusing and inexplicable way, Processing has held up its hand very early and let you know something went astray.

Look at the message area (the little horizontal bar just under the text editor). That's where Processing tries to give you a succinct summary of the problem. It probably says something like

```
Syntax error, maybe a missing right parenthesis?
```

Well, that's awfully nice of it. Yup, that's just what's wrong (as we know).

Given such a clear report in the message area, why is there all that crazy red text in the console below it?

The console messages in red are the output of the internal processes that Processing is running for you behind the scenes to turn your program into a working Java program. Feel free to skim that text if you're curious, but unless you were born with the Java-language gene, it'll probably mean little to you. And that's just fine; that stuff is there primarily for experts. When I get an error from Processing, I always first look at the one-line summary in the message area just under the text editor. Usually that does a pretty good job of describing the problem and getting me close to where it happened. If I don't quite follow the problem, I might skim the console messages, but they're usually not helpful. So I keep the console window shrunk down to usually just one or two lines, so I can see if anything appears there that I might want to glance at; if so, I enlarge it, scan what's there, and shrink it back down again.

In general, the message area is the first place to look for help on fixing your bugs. Keep in mind that the report in the message area should be taken with a grain of salt: Processing is doing its best to identify what went wrong, but it can sometimes misdiagnose the problem, or get confused about where the error occurred. It usually does a pretty good job on both counts, but as you get more comfortable with Processing you'll occasionally find yourself doing a little freelance detective work to identify what actually went wrong, and where, using Processing's error message as a starting clue.

Notice up in the text editor that Processing has highlighted the problematic line in yellow. That's another nice touch. To get our program to run properly, just go back to that window, click just before the semicolon, and pop the right parenthesis back in there. If you run your repaired program, it should perform as before, printing "Hello, World!" in the console area.

The kind of problem we've just been looking at, resulting from the missing right parenthesis, is called a *syntax error* because it's a problem with the *syntax*, or the written text. It's also called a *compile-time error*, because it gets caught at the time when Processing is *compiling*, or translating, our program into Java. These kinds of errors are usually easy to fix because they're just the result of typing mistakes, rather than something conceptually wrong with the program. They're usually pretty easy to fix.

The other kind of problem is called a *run-time error*, and we'll be seeing plenty of those as well later on.

2.3 Working With Processing

You can save your source code using the options in the *File* menu, and of course you can retrieve it later. But it's just text. and if you prefer a different editor than the one in Processing's text editor window, you can tell Processing to use that instead (turn on the option "Use external editor" in Processing's *Preferences* menu). Alternatively, you can just cut and paste from a window running your favorite editor.

As I mentioned in Chapter 1, Processing does not automatically save your changes for you when you run your program. If something goes wrong while you're working (maybe your operating system freezes up, or someone trips on your computer's power cord), if you haven't saved your program recently you might lose all the work you've done since the last save. You can save by choosing the *File* entry on the menu bar and then *Save*, or just entering the shortcut from your keyboard. I find it a good habit to always save my program every time I run it. Every time I make a change and then want to run the new program, I use the keyboard to enter the shortcuts for *Save* and then *Run*. That way the file on my hard drive is always up to date.

You may have noticed that Processing has been quietly changing the colors of some of the words you type into the text editor. This is called *color-coding*, and it's another instance of Processing trying to be helpful. I love this feature. It's purely cosmetic, and it doesn't change your code or how it runs in any way, but it helps you see at a glance the different kinds of things you're typing. I predict you'll come to like it as well.

As we'll see, there are a few different kinds of things in a Processing source program. Generally, there's words (made of letters and numbers) and punctuation (like parentheses and semicolons). To do its job, Processing has predefined a bunch of words to have specific meanings. We saw one of them in the program above: `println`. These are called *reserved words* or *keywords*. Generally speaking, you can't use reserved words for your own objects. For example, in every Processing program, the word `setup` should be used in only one particular pre-defined way, and you can't change that. If you want to call something of your own by that name, you have to pick something else, like `mySetup`. Some reserved words are actually common words in English, like `for`, `else`, `if`, and `while`. You'll see them as we go along, and a full list appears in Appendix A. To help you spot the reserved words in your program, Processing highlights them in a special color (on my system, they appear in orange).

Processing comes with tons of documentation, all available on the website. There are tutorials on some of the basics, lots and lots of examples of different features, examples of how to do some cool or unusual things, and descriptions of language features. If you have a question the odds are good that you can answer it from the documents. If not, try searching the web - there are lots of Processing-related discussions out there. There are also a few other books on Processing, listed on the website.

Searching for help on Processing on the web in general can be frustrating, because you'll naturally want to put the word "Processing" in your search, but that's such a common English word that it will match a bazillion web pages that have nothing to do with the Processing language or environment. So I've found it's best to thoroughly

scour Processing's own web site for information first, and only move on to more general web searches if I must.

Let's also talk about *re-use* of code. There's absolutely nothing wrong in finding something that already exists and runs, and then adapting it to your own needs. In fact, it's encouraged. It's one of the best ways to learn, and it's a great way to build on other people's efforts. There are dozens of example programs on the Processing website, and hundreds or thousands more in various personal galleries all over the web. If one of them comes close to what you want to do, or just shows you a way to do one step of what you want to do, then by all means copy and adapt it. Of course, you might want to include a comment in your code to the effect that you're using some programming originally written by someone else, and you could include their name, or a link to the website, or some other reasonable acknowledgement. But if the code isn't protected by copyright or patent (and most of the code that people have posted on the web is not - otherwise, why post it?) you're not legally obliged to do that. It's just a nice thing to do. And if you post your code to the web one day, other people may return the favor. The widespread sharing of code is one of the really nice things about the Processing community.

The first time you save yourself an hour, or a day, by using someone else's code, you'll be a fan of code sharing for life. And the first time you see your code in someone else's project, or you get a thank-you email, you'll have that great feeling of giving back and being part of a friendly, international community.

Of course, re-use is also a great idea for your own code. If you wrote something a year ago that could be useful today, copy it and adapt it. Re-use is great. In fact, much of the philosophy of *object-oriented programming*, which we'll talk about in Chapter 14, is based on the idea that the more you can re-use existing pieces of programming, the better off everyone is. After all, if I've designed something, written it, debugged it, and documented it, and it does what you need to do, why should you repeat my effort? Building on other people's experiences and efforts is one of the pillars upon which we built civilization; let it work for you.

Chapter 3

Basic Ideas: Variables

Before we get into programming, I'd like to give you a little piece of Processing to fool around with. Here's a program you can type into the text window and run. As I said earlier, I encourage you to get used to typing these things. Don't worry for now what any of it means. Just type it in. Remember that upper and lower case are different, spelling counts, and include all the punctuation, including the semicolons at the end of most of the lines. The easiest way to indent lines is to use the tab key, but you can use spaces if you prefer (these indentations are just to make the code easier to read, so you can use any number of tabs or spaces that look good to you). When you're done, press the *Run* button (or use the *Run* menu item or keyboard shortcut). You should see something like Figure 3.1. When it's stopped drawing, either close the window or press the *Stop* button.

```
void setup() {
  size(1000, 400);
  background(255);
  smooth();
}

void draw() {
  translate(frameCount*2, 200);
  rotate(radians(frameCount*3));
  float sclSize = sin(radians(frameCount * 3.5));
  scale(map(sclSize, -1, 1, .5, 1));
  drawFigure();
}

void drawFigure() {
  noFill();
  stroke(0, 0, 0, 128);
  rect(-60, -40, 120, 80);
}
```

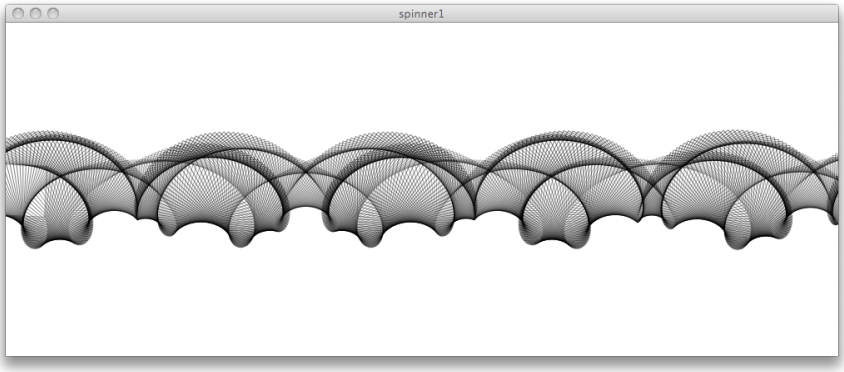

Figure 3.1: A spinning rectangle (*variables/sketches/spinner1/spinner1.pde*) [spinner1.png]

(see output in Figure 3.1. Full program in variables/sketches/spinner1/spinner1.pde)

The point here is just to have some fun. Monkey around with the numbers. Just follow your intuition. You can't hurt anything - the worst that can happen is that nothing gets drawn, or you get an error message. If that happens, you can just undo your changes and try something else. Within a few chapters everything here will be entirely familiar to you, even if it's a meaningless jumble now. Just treat this as a little toy you can fool around with.

Now let's look at how a program like this comes about.

3.1 Naming Things

In the process of writing your program, you're going to create lots of objects, and every one of them will get a name. Processing shares a bunch of weird typographic rules and conventions with virtually every other programming language on the planet, and those rules and conventions apply to every name you invent for every object your program. If you accidentally break the rules you can get some very confusing errors when you try to run your program. So let's get the naming rules clear first, before we dig into writing a program.

The first rule is perhaps the strangest-seeming one: when you create something and give it a name, that name can't contain spaces. If your object's name only needs a single word to describe it, then you have no problem: just call it "kettle" or "sandwich" and you're set. But you can't name it "my favorite kettle" or "reuben sandwich", because those contain spaces, and spaces are always used to separate pieces of text in your code. Spaces are just not allowed in names. But sometimes you really want to name something with a multi-word description, like "number of copies". What to do?

There are two popular answers. The first is to separate words with an underscore

character. So you could call your object `number_of_copies` and all is well. Don't use a minus sign (or dash), though: that means subtraction, and can't be used as part of object names. So never try to use a name like `number-of-copies` because your program either won't compile or else it won't run properly; always use the underscore instead.

The alternative is to run all of the words together, but use capitals to indicate the start of a new word, as in `numberOfCopies`. This style is sometimes called *camel case*, because those tall capitals in the middle of a bunch of shorter, lower-case letters might remind you of a camel's humps.

Some people love vanilla and hate chocolate; some adore the accordion and others loathe it. Feelings can also run high when people discuss underscore versus camel case, with people arguing passionately for the pros and cons of each technique. In practice they're both common, and when you look at other people's code you'll see them both frequently. I suggest you ignore the debate and use whatever feels best to you.

There is also an important rule regarding how names begin. You may not start a name with anything except a letter. A common mistake is to try to use a name like `3musketeers`, but that's not a legal name in Processing. Stick to only letters at the start of your object names. You can use any mix of upper and lower case letters, numbers, and underscores in the rest of the name. So you can name your object for your favorite movie by calling it `threeMusketeers` or `three_musketeers` or even `musketeers3`, just as long as you always start with a letter. Remember that case matters, so `milkBottlesize`, `MilkBottlesize` and `milkBottleSize` are all separate names and are considered to refer to entirely separate objects. Sometimes you'll see a name like `tEaKeTtLe`, which has a mix of upper and lower case that is technically legal, but way too cute unless it's the name of a pet bunny.

For now, I suggest you start every object's name with a lower-case letter (I'll discuss the reason for this in Chapter 4 when I talk about local and global variables).

By the way, one result of the rule that disallows spaces within object names is that generally anywhere you *are* allowed to have a space, you can have as many as you want. So in any Processing program (including those in this book) where you see one space, there could be 3 or even 100 spaces and the computer wouldn't even notice the difference. Newlines (the character that comes from pressing the *return* or *enter* key on your keyboard) and tabs (from pressing the *tab* key) count as spaces, too. Together, spaces, tabs, and newlines are called *white space*, and generally speaking you can use as much white space as you like in your Processing programs to make them look good to you.

Coming up with good, short names for your objects is something of an art. Some people focus on this and come up with great, compact, descriptive names for their objects. Most people just pick something reasonable. A few people are lazy and name everything with single letters, like `a` and `b`. Don't do this! Everyone who looks at your code will be driven crazy, including yourself a year after you wrote it and you come back to add something and you can't figure out what the heck any of those objects are for or what they're supposed to represent. Even a generic name like `counter` tells you

something about why the object exists and how it's used.

Of course, every rule has exceptions. For example, when we want to create objects to hold the X and Y coordinates of a point on the screen, variables named `x` and `y` are perfectly appropriate. And we'll see many examples of using a single letter (like `i`) for objects that control how chunks of code get repeated. But the general rule still applies: pick names that are descriptive and you'll be glad you did.

Finally, there are a few variable names that aren't illegal, but you should really avoid. Don't create any objects named simply `I` (that's a capital I, as in Istanbul) or `O` (that's a capital O, as in Oslo); they look too much like the numbers 1 and 0. And definitely don't combine just these, like `OI01I`. If you want to drive everyone who looks at your code insane (including yourself), pick variable names that are random mixtures of `I`, `O`, `0`, and `1`. You will be long remembered by anyone who has to look at a line like this:

```
OO1IOOIO = IOOIO1I + OOOOI1I;
```

This is legal, but deranged.

The lower-case letter `o` is also best avoided, again because of possible confusion with the digit 0. But oddly enough, the lower-case letter `i` probably the single most popular variable name in all of programming (we'll see why in Chapter 12).

As I mentioned in Chapter 2, the Processing language and its core utilities are defined using about 300 words (and a few dozen symbols). These words are called *reserved words* or *keywords*, and you shouldn't (and often can't) use them in your own programs. These words appear in Processing's text editor in a special color, and if you try to use them in your own way you'll usually get a syntax error message. Appendix A provides a list of all keywords, with short descriptions. Some keywords are common English words (like `for`) and some are words that would be incredibly useful if they weren't reserved (like `red`). As we go through the book, we will eventually see and discuss every one of these keywords.

3.2 Types

Suppose you've just moved across the country, but the movers lost all your stuff. But they apologized and gave you a lot of money to replace it all, so now you're busy buying new versions of what you've lost. And that includes shoes. You're going to need several pairs of shoes, including sneakers for your tennis game next week, hiking boots for climbing mountain trails, and formal shoes for a friend's upcoming wedding. So one day you're at the store and you purchase a new pair of shoes.

You get on the phone with your butler (did I mention the insurance company paid for a butler? They did), and you tell him, "Charles, I've just bought a new pair of shoes." Since he's good at his job, Charles says, "I'll prepare a special box in which to store them upon your return home. What size are these shoes?"

That's a reasonable question. After all, a pair of slippers would only need a small box, while thigh-high river waders would need something large. So you tell him they're formal dress shoes, and need only a medium box.

"Excellent," Charles says, "And what shall I write on the box so that I can retrieve this particular pair of shoes when they're needed?"

Again, even though Charles needs to loosen up a little bit, this is a reasonable question. Since you might end up with a whole lot of boxes of shoes, writing a descriptive label on each box will help you find the one you want quickly. So you tell him to label the box, "Black formal shoes," and the conversation's done.

We've just seen the basics of a fundamental idea in programming called *variables*. Variables are like shoeboxes: they're named containers for things. Some variables contain a number (like 3 or -6.2), some contain a string of characters (like "Appalachian Spring"), some contain a color, and so on. You give every variable in your program a unique name so that you can refer to it unambiguously. You can put stuff into the box (which we call *assigning* a value to the variable) and you can look inside the box to get its contents (which we call *retrieving* the value from the variable).

To create a variable, you follow exactly the same recipe as we just did in the phone call. First, you tell the computer the *type* of the variable; this lets the computer know if it's going to hold a number, a color, a string, and so on. Then you give it a name, using the naming rules I discussed above. From then on you can use that variable to store information and then retrieve it later.

These storage containers are called variables because their value can change over time. If a variable is a number type, you might put the number 6 into the box early in your program, and then replace that with the number 18 later. You don't have to change the value if you don't want to, but even if you never choose to change it, you could, so we still call it a variable. Variables are the most general way to store objects in a program, and every program contains lots of variables of different types.

The process of identifying the variable's type and name is called *declaring* the variable.

Declaring the type of each variable has a few advantages. The most important one for you is that it helps you catch errors. If you know that a variable is supposed to hold text (like someone's name), and you try to put a number into that variable, then you probably have made a mistake, like trying to put a sewing machine into a box sized for a paperback book. Generally speaking, if you try to put something of one type into a variable that has been declared with an incompatible type, the computer will flag that as an error. So using types helps both you and the computer make sure you've always got the right kind of information in your variables. It's not a foolproof way of making sure your program is accurate, but over the years people have discovered that this mechanism is incredibly helpful for finding errors, which is why it's part of most languages.

For some variables, like a number variable holding the current temperature, you'll probably change the value quite often. Other variables, like a number holding the mass of the sun or the year you were born, will probably never change at all. In some

languages, an object that never changes is called a *constant*, and you can in fact tell the computer which objects are variables (meaning they can change) and which are constants (meaning that they cannot change). In Processing, everything's a variable, so if you want to keep a certain number constant, just don't change it (technically, you can force a variable to be a constant if you precede it with the keyword `final` , but that's rarely used in practice. Typically, if you want something to be a constant, you just don't change it).

Let's look at three of Processing's built-in types very quickly. I'll show you how to use them, and then we'll return to consider how to choose among them for different jobs.

The first basic type is the *integer*, written `int` (with a lower-case `i`). This holds a number with no fractional part, but it can be positive, negative, or zero. Some legal values for an `int` are -3, 572, and 0, but not 8.4.

The second basic type is the *floating-point number*, written `float` (with a lower-case `f`). This is the catch-all type for numbers that can (but don't have to) have a fractional part. A `float` can hold integers, too, since they're just floating-point numbers with a 0 after the decimal point. Legal `float` values include -3.6, 572.1308, and 0.0.

A third basic type is for holding text, and it's written `String` (note the upper-case `S`). A `String` can hold any sequence of characters that you can type, including white space like tabs, spaces, and newlines.

To declare a variable, you need to tell the computer only two things: the type of the variable, and its name. You put those two pieces of information on a line by themselves, and end it with a semicolon.

Here are a few legal variable declarations:

```
int frogs;
float appleSize;
String my_name;
```

Note that these lines aren't a complete program. I'll typically call a few isolated lines like this a *snippet* or *fragment*. Sometimes when I show isolated lines of code like this I'll be giving you new versions of lines in a program we've already entered, and sometimes they'll be new lines to add to that program. And sometimes they'll just be little examples of things that you might use in a program, as they are here. If you type in this particular fragment and hit the *Run* button Processing will actually proceed without errors, since these three lines could be technically considered a program (though a completely useless one). More often, little fragments like this won't run all by themselves; they're only meant as examples of a specific point.

In this fragment I'm declaring an integer to tell me how many frogs are in my backyard pond, a floating-point number to tell me the weight of the apple I'm going to have for lunch, and a string of characters to hold my name.

The lines above are perfectly legal and you'll find things like them in almost every Processing program. But of course a variable has not just a type but also a value,

or it wouldn't be of much use. You assign a value to a variable using an *assignment statement*. That has four parts: the name of the variable, an equals sign (=) (in this use, called the *assignment operator*), the value to be assigned, and a semicolon. Here are some legal assignments, using the types declared just above:

```
frogs = 3;
appleSize = 7.1;
my_name = "Captain Stroganoff";
```

Here are some illegal assignment statements, using the variables (and their types) we just made. If you try to make these assignments (following the declarations above), Processing will report them as errors:

```
frogs = "Prince Bob";  // Illegal: assigning a String to an int
frogs = 3.5;           // Illegal: assigning a float to an int
my_name = 5;           // Illegal: assigning an int to a String
```

You'll see that I've put comments on each line; a *comment* in Processing starts with two slashes (//), and continues to the end of the line. Processing ignores everything between those two slashes and the end of line, so you can put anything you want in there: text descriptions, reminders to yourself, very short poems, or even snippets of code you want to keep around but not actually use yet. Here I've used my comments to make, well, comments.

Because it's so common to want to give variables a starting value, you can combine the declaration and assignment statements:

```
int frogs = 3;
float appleSize = -43.9;
String my_name = "Captain Stroganoff";
```

Here I've assigned a negative number to `appleSize`. This doesn't make much sense if we're literally thinking of apples, but the computer doesn't know that. As far as Processing is concerned, `appleSize` refers to a variable of type `float`, and -43.9 is a floating-point number, so this assignment is perfectly legal. Whether it's sensible or not is up to our interpretation.

After you've declared a variable (and perhaps assigned it a starting value), any subsequent assignments must leave off the type declaration. You can only declare the type once. Writing the following two lines one after the other in a real program would be an error:

```
int frogs = 3;
int frogs = 7;  // Illegal: we've already declared frogs
```

And since you can't declare something twice, you certainly can't change its type:

```
int frogs = 3;
float frogs = 7;  // Illegal: we've already declared frogs
```

In other words, you declare a variable's type once, and it has that type as long as it exists.

Here's a legal assignment statement worth noting:

```
float weight_of_apple = 5;
```

So I've put an integer value (that is, a whole number with no decimal part) into a variable of type `float`. And that's perfectly reasonable; the number `5` could be written `5.0`, so the fractional part is simply `0`.

So if a `float` can hold an integer, why have `int` variables at all? There are two main reasons: error-detection and precision.

Error-detection is kind of like defensive programming. Suppose you know that some particular variable should never have a fractional part; that is, it will never need a floating-point type to hold it. For example, a variable might contain the number of chairs in your apartment. And you've decided that there are no such things as "fractional" chairs, like chairs that are broken or too fragile to be used; instead, a thing is either a chair or it is not. There can only be whole numbers of chairs. So you declare your variable this way:

```
int numberOfChairs;
```

Suppose that later in your program on you do this:

```
numberOfChairs = 5.7;
```

The computer will report an error during the compilation step, before the code even starts to run. When you look at this line, you'll realize the problem. You might decide it was just a typo, and `5.7` should have been `5`, so you use your editor to delete the `.7`. Or you might decide that this is really what you meant, and that fractional chairs make sense to you after all, and so you go back and change the declaration to make `numberOfChairs` a `float`, rather than an `int`.

Again, people have found over the years that distinguishing integer-only variables from floating-point variables has helped them find and fix a whole lot of unintentional mistakes, so the distinction has remained in most languages.

You might be tempted to take the easy way out and make everything a `float`. There are a few reasons not to do this. One is that you're defeating the error-catching abilities of the computer, which is after all a good thing designed to help you write bug-free programs. The second reason not to make everything a `float` is precision, or more precisely, the lack of it.

It's a sad fact, but true: when the computer stores a floating-point number, it often gets it wrong. Ever-so-slightly wrong, but wrong. The problem is due to *finite* or *limited precision*. Suppose that you have a garden that's going to be shared by 3 gardeners, and each one gets an equal share of the available area. The field covers 1 acre, so everyone gets 1/3 acre. So you write this program:

```
float total_acres = 1.0;
float number_of_people = 3.0;
float acres_per_person = total_acres/number_of_people;
```

(If you guessed that the last line calculates the value of `total_acres` divided by `number_of_people`, you'd be right!). If you print out the value of `acres_per_person`, it will look something like this: `0.333333`. Depending on your computer and installation and other details, you might see more or fewer digits, but the thing to note is that mathematically, the digit `3` should repeat *forever*. If you write down `1/3` then you've written down a perfectly exact and perfectly correct value for "one-third". But when the computer actually calculates that number, it has to store the result, and it can't store the infinite train of repeating 3's. It has to stop somewhere. So it stores lots of digits, but no more. Whatever value that does gets stored is going to be close to 1/3, but not exact.

Happily, in this case "close" can be very close indeed. Close enough to get to the moon and back, for instance. Of course, whether it's close enough for your purposes will depend on what you're doing, but for everyday image-making purposes, the value that can be stored in a floating-point number is usually good enough (though it will never be exact, and in a moment we'll see a way to make it better).

So 0.33333 is only an approximation of 1/3, and if we start to use it in calculations our results will start to drift from the mathematical ideal. For example, suppose that the computer can only store two digits worth of the number, so `acres_per_person` has the value .33. If we multiply that by three, we get .99. But if the value was stored perfectly, it would be 3/3, or 1.0. The real value is a little bit bigger than the one we calculated (in this case, the difference is 1.0-0.99 = 0.01. If we use more digits we can make the error smaller, but it won't ever get to zero). And as we continue to calculate, the errors will accumulate. Again, these errors are usually so small that we don't have to worry about them in everyday graphics, but they're still there.

By contrast, an `int` value is precise and has no error. A value of 3 is exactly 3. And -78 is exactly -78. Multiply or add two integers and the result is an integer. Add 1 to an integer and the result is exactly 1 more than you had before; no more and no less. This is really useful, particularly when counting things, and when using a variable to control how many times we repeat something. If I want to draw 5 green squares, then I want exactly 5 of them, and not 5.001 (I don't even know what .001 of a green square ought to look like). So for lots of book-keeping and counting tasks, the `int` data type is the perfect choice. It's efficient, and if we ever try to store a floating-point value in there we know it was probably a mistake.

You can't store *any* integer into an `int` variable; the computer does have limits on the largest and smallest values it can store. In Processing, an `int` can hold a value from about -2 billion up to 2 billion (precisely, the range is -2,147,483,648 to 2,147,483,647). If you ever need more than that (which seems unlikely), you can use the `long` data type, which stores integers from about -9 quintillion to 9 quintillion. Wow. But none of the built-in Processing routines accept `long` values, so you'll only be able to use them for

your own internal book-keeping. And a `long` is slower to process than an `int`. Because of these drawbacks, you rarely see people using `long` variables in practice (I haven't used a single one in this book).

So integers are exact, but floating-point numbers are only approximate. But when you do need floating-point numbers, then by all means use the `float` data type. I don't want to give you the wrong impression. For the kinds of programs we usually write with Processing, `float` variables are more than accurate enough. You can create `float` variables and do what you want with them and never give a moment's thought to precision, and you'll probably never have a problem.

But if there does comes a time when a `float` just isn't accurate enough, there's another data type out there called `double`. This is short for *double-precision*, and it's just that: a much more precise floating-point number. You define and use a `double` just like a `float`:

```
double temperature;
temperature = 83.71;
```

So why not use a `double` all the time and forget about `float`? Speed. It takes the computer longer to compute with a `double` than with a `float`. The difference isn't huge if you're just adding two numbers, but if you start adding numbers together thousands of times per image (which is entirely reasonable), then the time difference can start to add up to something you can really see. If you're making moving images, the time you save by using variables of type `float` rather than `double` can make the difference between a smoothly-flowing animation, and something that stutters and jumps. If you don't need the extra precision of a `double`, there's no need to use it.

Which raises the question of how you'll know when you need to switch to `double`? Happily, you can work with Processing for a long time and never need that extra precision. I won't use a `double` for any program in this book. Someday, if you write a program that depends on a lot of math (unlike the ones we'll do here), you may find that your graphics aren't lining up perfectly with each other, or your objects are drifting around on the screen. That's when you can try switching to `double` types and see if it helps. Until then, I suggest avoiding them, though you will see them in other people's code from time to time.

Another downside of `double` is that, like `long`, none of the built-in Processing routines accept numbers in `double` format. If you want to hand a `double` to Processing, you have to turn it into a `float` first by preceding it with the word `float` between parentheses:

```
double doubleNumber = 999999999;
float floatNumber = (float)doubleNumber;
```

Of course, just as you can't put 2 gallons of milk in a 1 gallon container, you can't put a giant `double`-sized number into a smaller `float`-sized variable. Processing will do its best to save a floating-point value that's as close as possible to your double-precision

value, but if your `double` is too big to fit into a `float`, then Processing will just save the biggest floating-point value it can.

So the general rule I'll use for numbers in this book is to use an `int` when I'm counting, and a `float` for just about everything else. I suggest you do the same in your own projects.

Generally speaking, you can assign variables to those of other types as long as the new object can hold the type it's getting. So a `double` can take on the value held in any other kind of number variable: another `double`, or a `float`, or an `int`. A `float` can't directly take the value of a `double`, because it isn't capable of that precision, but it can take on any other `float` or `int`. And finally, an `int` can only take on the value of another `int`.

There are built-in functions that will do their best to convert any type of number to any other type. Each function has the name of the type you want the number to become. So `int(3.78)` will turn the floating-point value 3.78 into an integer.

To do this conversion, `int()` simply throws away the fractional part, leaving you with just 3. Converting a float to an integer is probably the most common example of this kind of *type conversion*, (also called *casting*), but you can coerce variables into a `float` if you want to, using `float()`. Normally you don't need to do that explicitly because such assignments are perfectly legal. If you put an `int` into a `float`, that's no problem. It's only when you're risking losing information (like turning the floating-point 3.78 into the integer 3) that you have to explicitly use one of these conversion functions. Figure 22.9 summarizes all of the type conversion functions in one place.

If you try to assign a floating-point number to an integer variable without running it through `int()` first, Processing will try to protect you from accidentally losing information by flagging it as an error.

We'll talk more about `String` types in Chapter 20. where we'll see that they're basically for holding and manipulating text, like someone's name, or the title of a book. We'll see that you can also convert strings to numbers, and vice-versa.

There are only a couple of basic types left to cover (there are a few other special-purpose types that we won't get into here, but of course they're described in detail in the documentation). The `color` type is used to hold (surprise!) a color, and we'll discuss that type in some detail in Chapter 4.

The other useful type is called a `boolean` (pronounced bool'-ee-in), named for the mathematician George Boole. A `boolean` is a special kind of data type because it can only hold two values: `true` and `false`. These two values aren't numbers, and they're not strings; they're just keywords with a special meaning to Processing as the values that a `boolean` variable can take on.

You'll frequently see people using an `int` to do a job better filled by a `boolean`. For example, suppose someone wrote a program to display current weather conditions, and in the program there's a variable called `it_is_raining`. If it's dry right now that variable has the value 0, otherwise it's 1. Here's a declaration of the variable with the value 1, to mean "true".

```
int it_is_raining = 1;
```

So why not use a `boolean`? History. Many early programming languages didn't have a `boolean` data type, and people got into the habit of using integers. By universal convention, a value of 0 in an integer means false. Usually a 1 means true, but often anything other than a 0 can mean true as well. Because many people learn to program by reading other people's code (which is a great way to learn), this tradition persists. It's not so bad, but I think if you're trying to represent something that really does seem to have a true/false nature, it's better to use a `boolean`. After all, they were designed for exactly that purpose, and it's a little easier to understand than an integer's value. Here's the same variable, but initialized as a `boolean`:

```
boolean it_is_raining = true;
```

To me, this is clearer. Later on, we'll see that we can test a `boolean` to see if it is `true` or `false`, and take different actions depending on the result.

3.3 Using = For Assignment

I said earlier that variables can have new values put into them at any time. Just assign them a new value and it's done!

Suppose that it's raining when our program begins. So the `boolean` variable `it_is_raining` might be declared this way:

```
boolean it_is_raining = true;
```

Later on, it stops raining. We'd simply say this:

```
it_is_raining = false;
```

That's all there is to it. The new value gets written into the variable and the old value is discarded. It's true: variables vary!

A variable's new value can come from all kinds of operations. A very common operation is to count some number of things in your program. Each time you find another one of those things, you add one to the value of some variable. If your variable was named `counter`, you'd write it like this:

```
counter = counter+1;
```

The computer takes the current value of `counter`, adds 1 to is, and saves that into `counter`, over-writing the value that was there before.

If you're a mathematician, this probably looks like insanity. Clearly, if we read this line of Processing like an equation, it can't possibly be true. There's no value of `counter` that could ever equal its own value plus 1. But of course, as we've seen, the

equals sign in Processing is not a test of equality, but an assignment operator. In this statement, the right-hand side of the equals sign is evaluated first: take the value of `counter` and add 1 to it. When that's done, the result is put into `counter` (over-writing what it used to have).

Often we pronounce the equals sign as "gets" or "becomes", so we'd read the above line out loud as "counter gets counter plus 1". Some people pronounce the single equals sign as "equals", but I think that's potentially confusing.

This use of the equals sign is weird. It sure isn't mathematically correct, but this is how almost every language writes assignment statements. Not everyone is happy about that. In some languages the equals sign is replaced by something else. One choice is a makeshift arrow, `<-`, so that the line above would be written

```
counter <- counter+1;
```

Though it's not perfect, in a lot of ways I think that this is better. But the equals sign has a lot of history behind it and it seems to refuse to go away. This use of the equals sign for assignment comes to us from the very first programming languages, and it's just too well embedded now to be easily dethroned.

3.4 Semicolons and Errors

We've seen semicolons at the end of many lines of Processing code above. Going forward, we'll use semicolons used about a million times in every program. Let's see what that's all about.

Almost every sentence in English ends with a period (or occasionally different punctuation, like a question mark or exclamation mark). Even if it's obvious when a sentence is over, that period is still required. There, that sentence just now ended with a period. And so does this one.

In Processing, every *statement* ends with a semicolon. So far, we've seen three kinds of statements: variable declarations, assignment statements, and combined declaration-assignment statements. And just like every English sentence ends with a period, every one of these Processing statements ends with a semicolon. Every single one. And that's the rule: every statement ends with a semicolon. The only tricky thing about this rule is knowing what a statement is!

Until you get the hang of it, you might find yourself including semicolons where they're not required, or leaving one out where it should have gone. Not to worry. I'll identify various statements as we go along, and you'll pick up where the semicolons belong and where they don't. After a while, it'll be second nature to you. Until then, the computer will help you along by reporting a syntax when a necessary semicolon is missing.

The general principle is that every time you complete an action, you end with a semicolon. If you want to take more than one action at a time then you wrap the multiple statements in *curly braces*: `{` and `}`. The curly braces themselves don't end

with a semicolon because they're not actually a statement, they're just a grouping device. The curly braces generally tell Processing "treat all of the statements inside these braces like one chunk of statements."

For example, if you were instructing the computer to bake cookies you might have a step involving adding butter. That would end with a semicolon (don't worry about the parentheses for now):

```
add_butter();
```

If you wanted to add a bunch of things at once, you might wrap them in curly braces:

```
{
   add_chocolate_chips();
   add_walnuts();
   add_mint();
}
```

We'll see later that this lets us package up that group and refer to it conveniently with a single name, like `my_winter_cookie_recipe()`.

Curly braces are used for a few other grouping tasks as well, probably because there are only three kinds of paired grouping symbols on the keyboard. Processing is pretty consistent about using the square brackets `[` and `]`, though it's a little more flexible with the curly brackets `{` and `}`, and the round parentheses `(` and `)` are drafted into a variety of uses. The angle brackets `<` and `>` are used only for math, and not for grouping things. Don't worry about keeping all these character pairs straight, because there are really just a few patterns and you'll pick those up without even noticing it. But in the beginning, keep an eye out for whether you should be using curly braces or parentheses, and try to get a feel for when a "statement" is over and you need to provide a semicolon.

If you mess up and include an extra semicolon here and there, or leave out a required one, the computer will catch it and tell you.

Which raises an interesting question: if the computer "knows" when you've made a mistake, why doesn't it just fix it for you? For example, when it sees that a semicolon is required, but it's not there, why doesn't it just put one there for you and carry on?

This sounds like a reasonable thing to do, and people have tried it. And over time, they've abandoned it. The problem is that although most of the time it's perfectly reasonable for the computer to fix your simple errors for you, every now and then that can backfire when the computer guesses wrongly. And those backfires can be maddening to track down and fix. So maddening that you turn off the auto-fixing entirely.

This is the same reason that spell-checkers and grammar-checkers these days suggest corrections to you, rather than automatically fix them. When you wrote "The frog ate the man," maybe that's what really happened, and you don't want the computer to assume otherwise. Or maybe "Hen3ry" really is your character's name (the "3" is

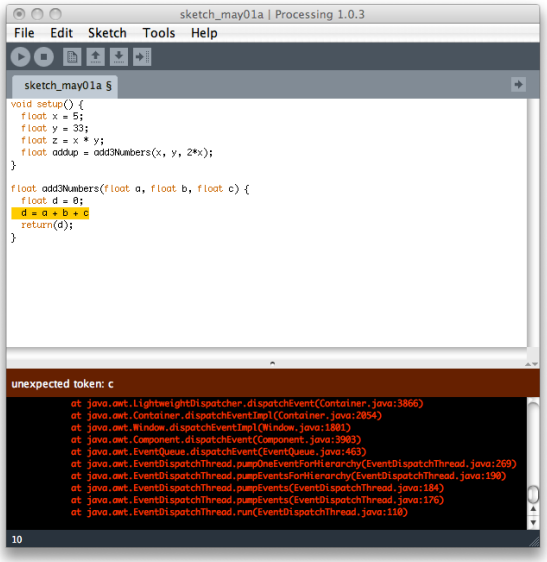

Figure 3.2: A syntax error. [varerror1.png]

silent). The bottom line is that the computer is good at finding things that seem wrong, but because it can't read your mind, it's far less skilled at making them right.

So people have largely given up on having the computer rewrite your program for you. When there's an error, and the computer catches it, it tells you that there's a problem, and it tries to identify the problem as clearly and usefully as possible, but then it leaves it up to you to do something about it. And although it can be annoying to spend time fixing a bunch of minor typos, you'll be spared the horrible process of tracking down and fixing computer-generated "repairs".

3.5 Comments and Printing

We're going to start writing code in the next chapter, so I want to make you aware of a couple of really important tools: the *comment* and the *print statement*.

As we've seen, when you first hit the *Run* button, Processing first translates your text into the Java language. If something goes wrong during this translation stage, you'll get error messages in the console and a shorter (and usually more helpful) error message in the message area. The line on which Processing got hung up will also be highlighted in yellow in the text editor.

Figure 3.2 shows an example.

Don't worry about the program or the messages here; I just want to illustrate the general way they show up. These kinds of errors are usually typos, like mis-spelling a word or forgetting a semicolon. Because they are caught as a result of an error in *syntax*, or the text of the program, these kinds of problems are often referred to in

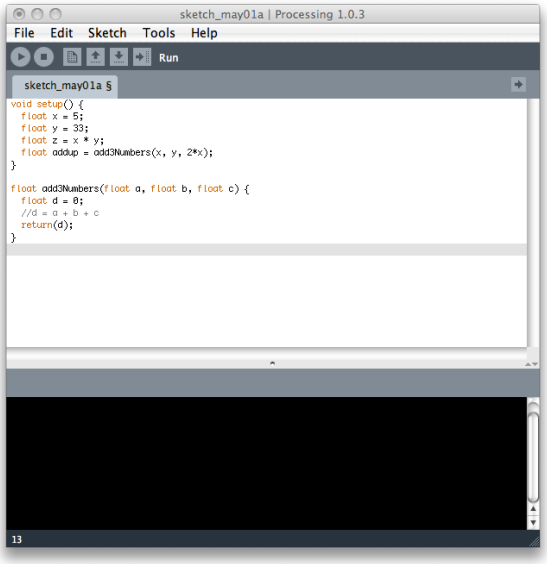

Figure 3.3: Commenting out a line. [varcomment.png]

general as *syntax errors*. Everyone gets lots of them when they first try to run a new piece of code; it's a rare day for me when a new program of more than a half-dozen lines doesn't have at least one syntax error in there somewhere.

If you have enough information to diagnose and fix the problem, then just make your corrections in the text editor and hit *Run* again. Processing will start over again, translating your program all over again from a fresh start. If you repaired this problem, then Processing will keep going. It it hits another problem later on, it will again stop, with the new offending line in yellow and a message about the error.

If the problem isn't obvious to you and you're having trouble isolating it, you can start to *comment out* lines. As we saw earlier, by placing two slashes at the start of a line (`//`), you're telling Processing to ignore everything to the end of the line. It shows up in the text editor, but Processing doesn't try to read it or translate it. You can put anything you want after those characters: a note, a smiley face, anything, and it will be ignored.

The good thing is that this means your error, if there is one, will be ignored as well. Figure 3.2 shows a program with an error. If I was having trouble finding the problem. I could just comment out the highlighted line. The commented version is shown in Figure 3.3; notice that the commented-out text is automatically shown in light gray. If I run this, there are no problems.

This lets me remove the line from Processing's attention. If the program gets successfully translated (and starts to run), then I'll know that this line is my problem, and I have to dig in a little deeper to fix it.

Sometimes you want to comment out a lot of lines. You can certainly put two slashes at the start of every line, but that can become a hassle. To comment out a whole bunch

of code, use the *multi-line comment* characters. Write `/*` to start a multi-line comment, and `*/` to end it. Anything between those marks is a comment.

Warning: Don't put one multi-line comment inside another! The first `*/` ends all comments. You'll be able to see this in the text editor because comments are in gray. After your first `*/` your code will be colorful again.

Commenting is a great way to isolate the errors that crop up during the translation stage, which are collectively called *syntax errors* or *compile-time errors*.

The other kind of error is the one that occurs when your program is actually running, and that's called a *run-time error*. There is no one best way to handle run-time errors; they're the tricky ones that will occupy most of your debugging attention.

A time-honored way to debug run-time problems is with the *print statement*. In Processing, there are two forms of print statements: `print()` and `println()`. They are identical except that `println()` adds a newline character to the end of what's printed out (which is like pressing the return key to start a new line of text).

Much of the time you'll use these statements to print out the values of variables. I find it's very useful to name the variable in the print statement, so if I have a few print statements in my program in different places, I know what I'm looking at:

```
println("a has the value " + a);
```

You can see that `println` lets you print both text and variables. When it sees text between quotes, it just prints that directly. A variable is printed with its value. To mix both kinds of things on one line, you use the plus sign (`+`) as glue. Note that I've included a space in my string, so the value of the variable doesn't come immediately after the last letter. Here's a two-variable example:

```
float x = 3.0;
float y = 8.7;
println("x = " + x+"  y = " + y);
```

Note that I put spaces around the equals sign, because I find that makes the output easier to read. Both print statements send their messages to Processing's console. On my system, errors in the console show up in red type, but the messages I generate with print statements show up in white. If I run the program above, I get the results shown in Figure 3.4.

Of course, you can use comments to help you fix run-time errors as well, by effectively eliminating chunks of your program (if a line is commented out, it isn't translated, and if it isn't translated, it isn't run). Here I'll break the print statement above into two lines:

```
float x = 3.0;
float y = 8.7;
println("x = " + x);
println("y = " + y);
```

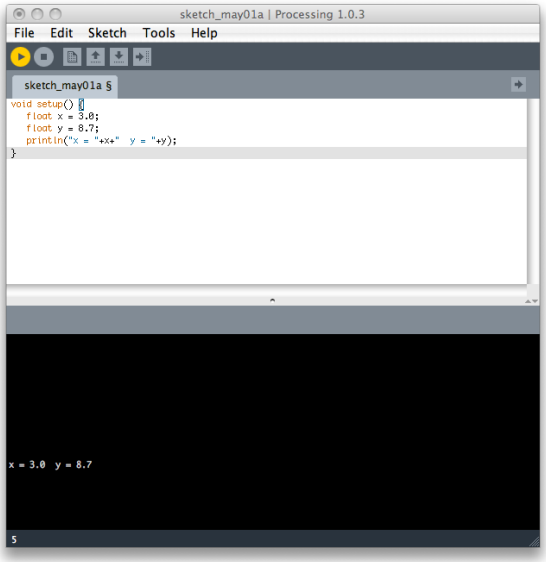

Figure 3.4: Printing to the console. [varprint.png]

The output of this program is:

```
x = 3.0
y = 8.7
```

Now I'll comment out the first print statement:

```
float x = 3.0;
float y = 8.7;
//println("x = " + x);
println("y = " + y);
```

The output of this new program is:

```
y = 8.7
```

If my program had been misbehaving, and commenting out the line made it work correctly, that would be strong evidence that there was something wrong with that line.

Another Spinner

If you liked playing with the program at the start of this chapter, here's a variation on it that produces a different kind of image. Instead of moving a rectangle around, I'll move an ellipse. Again, kick around the numbers for yourself and have fun with them.

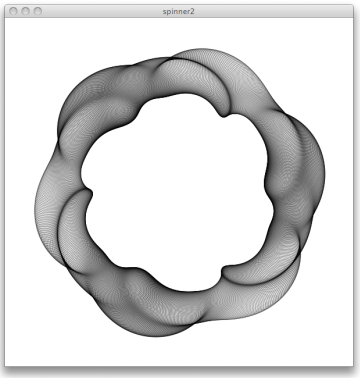

Figure 3.5: A spinning ellipse (*variables/sketches/spinner2/spinner2.pde*) [spinner2.png]

```
void setup() {
  size(600, 600);
  background(255);
  smooth();
  translate(300, 300);
  for (float i=0; i<360; i += 0.5) {
    pushMatrix();
       rotate(radians(i));
       translate(0, 200);
       rotate(radians(i*3));
       scale(map(sin(radians(i*6)), -1, 1, .5, 1), map(sin(radians(i*3)), -1, 1, .5
       drawEllipse();
    popMatrix();
  }
}
void drawEllipse() {
  noFill();
  stroke(0, 0, 0, 128);
  ellipse(0, 0, 120, 80);
}
```

(see output in Figure 3.5. Full program in variables/sketches/spinner2/spinner2.pde)

Chapter 4

Functions and Tests

Before we get started, here's another toy project for you to fool around with. As before, you can fiddle with the numbers freely to see what kinds of pictures you can make. This one uses random numbers, so each time you run it you'll get a different picture.

```
float NoiseScale = 0.005;
float NoiseOffsetX, NoiseOffsetY;

void setup() {
   size(800, 600, P2D);
   background(255);
   smooth();
   noFill();
   stroke(0, 0, 0, 32);
   for (int i=0; i<300; i++) {
      NoiseOffsetX += 5;
      NoiseOffsetY += 7.1;
      drawOneStream();
   }
}

void drawOneStream() {
   float px = 0;
   float py = height/2.0;
   float vx = 1;
   float vy = 0;
   int pcnt = 0;
   while ((px>=0) && (px<width) && (py<height) && (py>=0)) {
      point(px, py);
      float xNoise = noise((pcnt+NoiseOffsetX) * NoiseScale);
      float yNoise = noise((pcnt+NoiseOffsetY) * NoiseScale);
      vx = ((2*vx) + 1 + map(xNoise, 0, 1, -1, 1))/4.0;
```

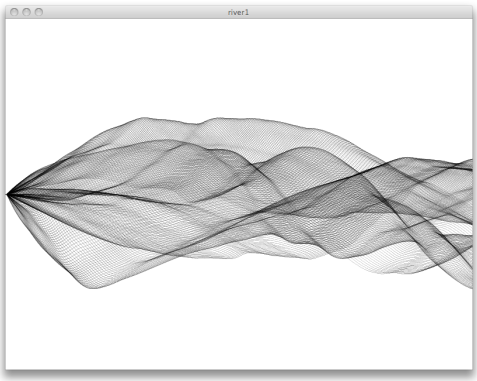

Figure 4.1: A program to draw fabric flowing like a river (*functions/sketches/river1/river1.pde*) [river1.png]

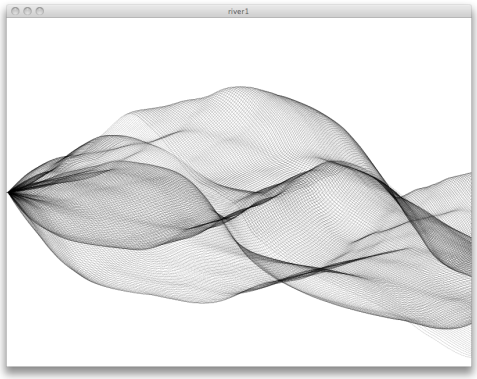

Figure 4.2: Another river. [river1b.png]

```
      vy = ((3*vy) + map(yNoise, 0, 1, -1, 1))/4.0;
      px += vx;
      py += vy;
      pcnt++;
   }
}
```

(see output in Figure 4.1. Full program in functions/sketches/river1/river1.pde)

Figures 4.2 and 4.3 show a couple of other images from this program. I just kept running it over and over and saved three that I liked.

Processing lets you package up collections of actions into chunks that you can invoke all at once. The program above does just that.

Why is this useful? Cooking gives us a perfect analogy here. If you want to tell someone how to make a pasta dinner, but they've never done it before, you might have to walk them through every little step:

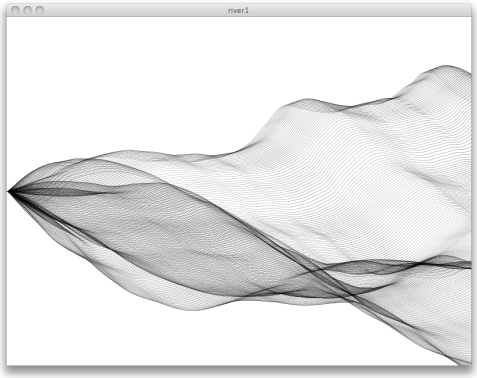

Figure 4.3: Another river. [river1c.png]

1. Fill a pot with water
2. Put the pot on the stove
3. Turn on the stove
4. Wait for the water to boil
5. Open the package of pasta
6. Pour the pasta into the water
7. Many more steps!

If you were both careful, they could follow your instructions precisely and prepare a fine meal with pasta, tomato sauce, fresh vegetables, and so on.

You could package up all those steps, and then refer to them as just a single instruction:

1. Cook pasta

Then you could have other pre-packaged collections of steps to choose a wine, prepare a salad, and so on.

Packaging up multiple steps into a single operation is an efficient way to refer to all of those steps at once. Of course, once you create this packaged version, you have to put it where someone else can find it.

There are lots of words in popular use for describing this kind of packaged-up collection of steps. You'll often see it called a a *function* , a *procedure* , a *subroutine*, or simply a *routine* . Although some computer languages distinguish these words, in Processing they all mean the same thing, so I'll use them interchangeably in this book.

You'll also sometimes see a function sometimes referred to as a *method*, but in this book I'll use that word only for functions that are part of your own custom objects, as discussed in Chapter 14.

There are lots of different pasta shapes, but preparing each of them follows roughly the same process. Rather than have one recipe for macaroni, one for spaghetti, one for wagon wheels, and so on, you might have just one recipe where you can specify the kind of pasta that should be used.

When a function takes one or more values that control or specify what it does, each of those values is called a *parameter*, or (borrowing a term from mathematics) *argument*. Generally speaking, these words both mean the same thing: a value that gets given to a function, that it can then use to do its work.

So the *cook-pasta* function could take an argument that tells it what kind of pasta to include in the description. I'll write this by naming the function, and the putting the argument in parentheses:

1. cook-pasta (wagon-wheels)

2. cook-pasta (ziti)

Then the instructions inside *cook-pasta* could conceivably use this information to customize their process. For example, the function might have you boil the wagon wheels for a minute longer than the ziti.

Routines and functions are incredibly useful; almost every program you ever write will have multiple functions in it. So let's dig into them.

4.1 Writing Functions

Let's say that you're writing a program in which you need to constantly add up three different numbers. These might be the last three screen positions of an object, or three shades of red, or any other three things that can be described by a number. Of course, it's easy enough to add three numbers, but let's pretend that was a mind-bendingly difficult task (maybe instead of just adding the numbers together, we do all kinds of complicated calculations on them instead). Then we'd want to write it once and get it perfect, then save it in a function so we wouldn't have to repeat it over and over.

We start out by *declaring* a function. This has some elements in common with declaring a variable, but some differences, too. In general, you give the type of the function, its name, the values it takes as inputs (that is, its parameters or arguments), and then the statements you want it to execute.

First off, every function has to have a name. The name must obey the rules we saw earlier for names (it must start with a letter, it must contain no spaces, and it can use only letters and numbers). Let's call this one `add3Numbers`. The declaration so far is just the name of the function:

```
add3Numbers
```

Now I'll define the arguments that it takes as input. These are just variables like any other, so each argument has an associated name and type.

The list of arguments is naturally enough called an *argument list*. Here's how to type it correctly. The argument list always appears between a pair of parentheses that come after the function name. If there's more than one argument, they're each separated by commas.

These parentheses are *always* required. If you don't have any arguments to go in them, then you just type the parentheses with nothing between them (though you can put in some spaces if you want). This is your way to tell Processing "this function has no arguments." You might think that an easier way to do this would be to leave the parentheses out altogether, and that's a reasonable idea, but that's not how Processing does it.

So let's start with our empty argument list:

```
add3Numbers()
```

Lots of functions don't take any arguments, and we'll see plenty of them as we go on. But in this example we do have arguments, so let's put them in:

```
add3Numbers(number1, number2, number3)
```

Remember that every variable has a name and a type. The same thing holds for the arguments in a function; Processing has to know the type of each argument. Let's suppose that these are all floating-point numbers, so I'll explicitly give them that type:

```
add3Numbers(float number1, float number2, float number3)
```

We're getting closer.

From here on in the book, any time I refer to a function I'll indicate that by using a pair of parentheses after the function name. So if I'm talking about a variable named `numberOfApples` I'll write it just that way. But let's suppose that this was the name of a function. Then I'll write `numberOfApples()` so that we can tell right away that it's a function. If it takes arguments I won't list them, because it would take up a lot of room and not add much meaning. So when I talk about this function in the text, I'll refer to it as `add3Numbers()`, even though we know it takes three floating-point numbers as inputs.

Remember that the purpose of this function is to add up the numbers we give it. That means it returns a value when it's done, but what's the type of that value?

In this case, it's going to be a floating-point number (because otherwise this would be a pretty lousy function). In Processing, there's a neat conceptual shorthand that lets us conveniently describe the type of a variable that is returned by a function: the function itself is given a type.

In other words, we know that `add3Numbers()` is the name of a function, and when we're done with it, somehow it's going to return a floating-point number as its result. The way we say that `add3Numbers()` returns a `float` is to say that the function

`add3Numbers()` itself has a type, and that type is `float`. This is really just a shorthand to save on typing, but it's a good one that makes sense.

To see why, let's look at a line of code that might *invoke*, or *call* this function:

```
float mySum = add3Numbers(1.0, 2.2, -14.1);
```

When Processing sees the call to `add3Numbers()` it calls the function with the numbers listed above as arguments (or inputs or parameters). Then when `add3Numbers()` is done, it gives back a `float`, and that gets put into `mySum`. So the value going into `mySum` is a floating-point number. It does make sense to say that the function `add3Numbers()` itself really does have the type `float`, because that's the type of the object that results from calling it.

So in the definition of a function, we provide its type just like we provide the type of a variable, by simply providing the type before the name:

```
float add3Numbers(float number1, float number2, float number3)
```

Finally, the code that implements this function (called the *body*) follows the definition between a pair of curly braces:

```
float add3Numbers(float number1, float number2, float number3)
{ }
```

We're so close to having a real function I can taste it. But if we tried to compile a program with these two lines in it, Processing would highlight the declaration line in yellow and give us an error message in the message area:

```
This method must return a result of type float
```

That's a great error message! Processing knows that `add3Numbers()` must return a `float` (because that's how we declared it), but right now the function doesn't return anything. In fact, it doesn't do anything at all, because there are no statements between the curly braces. There must be at least one statement, to return a value.

Naturally enough, that's called a *return statement*. It consists of the word `return` followed by the value to be returned and a semicolon (I like to put parentheses around the returned value because I think that makes it easier to read the code, but you don't have to surround the return value with parentheses if you don't want to):

```
float add3Numbers(float number1, float number2, float number3)
{
   return (number1+number2+number3);
}
```

Note the semicolon at the end: a *return statement* is a statement like any other, and so must end with a semicolon.

We now have written a complete function in Processing!

The *return statement* is very flexible. You can return all sorts of things from functions: variables of type `int` and `float`, of course, but also variables containing objects of your own design (we'll see these in Chapter 14). Be careful to make sure that the type of object you're returning is the same type as the function itself, or you'll get an error like the one above.

When the computer executes your `return` statement, that's the end of your function; the computer leaves the function and returns to the calling program that invoked it in the first place.

To use our function, we can write a little test program to call it, and then print out the result.

4.2 setup() and draw()

To write a little test program in Processing, put your code inside a function named `setup()` of type `void`; this function takes no arguments (I'll talk about `void` in just a moment).

```
void setup() {
   // testing stuff goes here
}
```

The `setup()` function is a special function in Processing, and effectively tells the system "start here" when it runs your program. It has to have the name `setup()` with a type of `void`, and take no arguments. You put your test code between the curly braces that define the `setup()` procedure.

This is the first of two functions you see in almost every Processing program. The other, which we'll in a moment, is `draw()`.

When you run a program, the first thing Processing does is look for a function named `setup()`. If it can find that, it runs the code that's defined in that function. If there's no `setup()` (like in some of our earlier examples), Processing will just run the commands you've given it in the text window. But except for the very smallest little programs, you're always going to have a `setup()`, and that's always where Processing begins.

Of course, Processing is all about graphics, so your program will probably want to draw stuff. And since Processing is also all about animation, that stuff will probably move over time. Your computer is already redrawing the entire screen for you many times a second. The exact number varies from one computer to the next, depending on its speeds and the particular choices you've used for your monitor or display. The number is typically around 60 refreshes per second. So if you're looking at your computer monitor right now, odds are pretty good that it's actually being redrawn for you,

over and over, 60 times or more every second. Because it's happening so fast, and the electronics are well designed, you don't see any negative effects of this redrawing (if it ran too slowly, you'd see the image *flicker* , like an old-time movie).

Each time the screen is redrawn we say the computer has drawn a new *frame* . So you're probably getting something like 60 frames per second from your computer. For each frame, Processing looks for a function you've written called `draw()`, also with type `void`. If you haven't got such a function, then Processing just doesn't do anything. But if there is a `draw()` procedure, then Processing will call it at the start of every frame. As the name suggests, typically your `draw()` function will draw to the graphics window. Frequently, one of the first things we do when writing a new project is to tell `draw()` to start out by erasing the whole window. This covers the window completely with some background color, so that everything that follows is drawn on top of that. When `draw()` is done with all its work, then your computer takes the picture that it created and *bam* it puts it up on the screen in the graphics window. Then 1/60 of a second later it happens again. And again. And again, over and over, as long as your program runs.

So to recap, Processing executes `setup()` once at the start, and then `draw()` for every single new frame as long as your program runs.

Which raises an interesting question: when and how does your program ever stop? There are two ways to stop: when your program itself decides to quit, or the user decides to stop it. If you want to stop a sketch that Processing has started up, you can just click the exit button in the window (this varies from one platform to another, but it's the little button that's usually in the top-left or top-right of the window that makes the window go away).

Sometimes that doesn't work because your program has hit a strange error. If you can't stop a program (that is, dismiss the window) by clicking the graphics window's dismiss button, press the *Stop* button on Processing's menu bar (that's the second round button over; the one with a square in it).

You might want your program to run for a limited time and then stop itself. For example, it might be part of a museum display. When a museum visitor pushes a button, your program runs, shows them some animation, and stops. You don't want your visitor to have to close the window or press a button to end the program, but instead you want your program itself to stop when it's done.

You can do that with the function `exit()`. If you call `exit()`, that hits the brakes immediately: your program stops right there and the window goes away, just as if the user had pressed the *Stop* button. Most of the programs in this book will be *free-running*, so they will just go and go and go until we stop them manually.

So let's get our new `setup()` function into our example. Where should the definition of `add3Numbers()` go? Not inside of `setup()`! Each function in your program is a citizen of the world. No function appears within any other function. If you have three functions, they get listed one after the other. In fact, they can come in any order, and they can even be spread out among multiple source files if that helps you organize your program. Part of the translation step that Processing goes through when you press the

Run button is to locate each procedure and remember where it is. I'll keep everything in one file for quite a while, and certainly it's the easiest way to write short programs like this.

So I'll first create `setup()`, and then put my short test program inside of that procedure. Then after the closing curly brace of `setup()` I'll provide the definition of `add3Numbers()`.

Here's our program:

```
void setup() {
   float v1 = 3.0;
   float v2 = 8.7;
   float v3 = -3.0;
   float sum = add3Numbers(v1, v2, v3);
   println("The final sum is "+sum);
}

float add3Numbers(float number1, float number2, float number3)
{
   return (number1+number2+number3);
}
```

(Full program in functions/sketches/addup1/addup1.pde)

If you hit the *Run* button (in the upper-left of the window, at the left side of the tool bar, it's the circular button with a little triangle in it), you'll get this result in the message area:

```
The final sum is 8.7
```

It's a running Processing program! And though it looks simple, don't be fooled. This program is doing a ton of cool stuff. We're creating variables, invoking a function, passing those variables as arguments, using them in a function to compute a result, passing that result back to the calling procedure, storing it in a variable, and printing it out. That's very cool.

Many functions return nothing. For example, you might call a function to draw a square on the screen. Once it's done the job, the function returns, but it doesn't send back any information. How do you tell Processing "this function doesn't return anything"? One approach would be to simply leave off a type on the function. But that's a little risky, because the computer wouldn't be able to help you distinguish between when you meant to return nothing, and when you did mean to return a value but you simply forgot to declare its type. So instead, you explicitly say "this function returns nothing" by saying that it returns the type `void`.

The type `void` is unusual, because it's not really a type the way `int` and `float` are types. Its only use in Processing is to give a type to a function that doesn't return anything. In other words, declaring a function with the return "type" of `void` means "this function does not return anything."

4.3 Curly Braces

Before I march on, I'll make a comment about style. As much as people like to argue over politics and text editors, they can get *really* worked up over the placement of curly braces.

The happy truth is that the computer doesn't care where you put your curly braces, so you can format your code in any way that pleases you. But there are people who feel strongly that one style or another makes it easier to share, change, and debug their programs. Here are the three most popular styles, in no particular order. I've seen them all in practice, and each one works exactly as well as the others. One thing almost everyone does have in common is that the statements inside the block are indented by one tab. It's nice to see everyone agreeing on *something*.

I'm going to compute the sum on one line and return it on another, so that we have at least two lines in each function's body.

1. The opening brace is on its own line, and not indented. The closing brace is not indented.

```
float add2Numbers(float number1, float number2)
{
   float sum = number1 + number2;
   return (sum);
}
```

2. The opening brace is on the same line as the definition. The closing brace is indented.

```
float add2Numbers(float number1, float number2) {
   float sum = number1 + number2;
   return (sum);
   }
```

3. The opening brace is on the same line as the definition. The closing brace is not indented.

```
float add2Numbers(float number1, float number2) {
   float sum = number1 + number2;
   return (sum);
}
```

I'm sure you can see why so much blood has been spilled over these choices. It's hard to imagine anything that could be more important.

If each of these styles look pretty much the same and equally useful to you, then select the one you like most and try to stick with it. If you feel strongly that one of

these is vastly superior to the others, you now have a ready-made source for endless hours of entertaining arguments.

I think one reason people get so worked up about this is that when you format your own code the same way for a long time, it does indeed make the code easy to read at a glance. That's a very good thing. Unfortunately, it can make other styles of formatting seem weird, and thus harder to read. By the same token, people who are used to a style other than yours will think your code looks weird (but of course they're wrong).

I like the last style in the list, where the opening brace is on the same line as the function declaration, and the closing brace is indented one tab stop less than the body. There are several places where curly braces are used in pairs like this, and I use this same convention for all those uses.

By the way, once in a while you can pack a whole function onto one line (if it's short enough):

```
float add2(float n1, float n2) { return (n1+n2); }
```

With occasional exceptions, I find that this usually isn't worth it. Writing your code in a consistent style really does make it easier to write, debug, and change later. The computer doesn't care about your formatting; the one-liner runs no faster than a three-line version. So I usually opt for consistency over a little bit of compactness. On the other hand, sometimes writing something really small on one line is a nice way of visually emphasizing that the function really is doing a tiny little job.

Sometimes very small functions whose principal jobs is just to save you some typing are called *convenience functions* (or *convenience routines*). This isn't a formal term, but a way we sometimes describe something that we think of as more of a little helper function, rather than something that does significant work.

4.4 Integer Division

In `add3Numbers()` above I added up three numbers using the plus sign, and I hope that seemed reasonable to you. To write mathematical expressions, you can add, subtract, multiply, and divide using the standard symbols `+`, `-`, `*`, and `/`. You can use parentheses to control what gets done first:

```
int result1 = 2 * (3 + 3);  // this has the value 2*6 = 12
int result2 = (2 * 3) + 3;  // this has the value 6+3 = 9
```

There are some rules about what happens first if you don't provide parentheses: all the multiplications and divisions in your expression happen before any of the additions and subtractions (we say that multiplication has a higher *precedence* than addition). So if you just wrote `3+2*3`, Processing would do the multiply first, resulting in a value of 9.

Even though I know the precedence rules, I almost always include the parentheses anyway. That way I don't have to think about precedence, and the intended calculation is instantly clear to me and anyone else who looks at it. The extra parentheses don't slow down the computer even the slightest bit, or change how the program runs. Their only cost is two characters in my source file. For the increased clarity, I think that's usually worth it.

There's an odd but very important quirk to keep in mind when you divide numbers: if you divide two integers, the result is always an integer. I don't like this rule very much, and it messes me up all the time, but that's how things are.

For example, if you write `3/2` the value of that expression is `1`, not `1.5`. That's because both values are integers. Even if you store the result in a `float`, that variable will still have the value of `1.0`, because the variables involved in the expression were all integers. In essence, Processing computes the floating-point result of your division statement, and then immediately throws away the fractional value. So an expression like `20*(3/2)` has the value `20`, not `30`.

If you want your division operations to include fractional parts in the result, you need to make sure at least one of the values involved is a `float`. You can do this with numbers simply by putting a decimal point and a zero after them. So `3/2.0` or `3.0/2` both evaluate to `1.5`. If your expression involves variables that are all integers, you can multiply one of them by `1.0` to make it a `float`.

Note that if this "promotion" to floating-point is within parentheses, then the promotion ends at those parentheses.

Suppose I have the expression `(1/2)*(1.0*3/2)`. I've made sure that the expression on the right is floating-point by multiplying by 1.0. That makes the rightmost value 1.5. But I'm multiplying that by `(1/2)`, and since that's not inside the parentheses of the other expression, that division has a value of 0. The result of this expression is 0.

Instead, I could write `(1.0/2)*(1.0*3/2)`. Now both values end up as floating-point numbers, and so the final result is 0.75.

There are two little techniques that can save you a character here and there when you want to make a number into a floating-point value. First, you don't have to include both the decimal point and a zero after a number to make it floating-point. Just the decimal itself will do. So `3/2.` and `3./2` both return 1.5. Alternatively, you can put the letter `f` after a number to make it floating-point. So `3.0/2` will result in 1.5, as will `3f/2`. My style is usually to include the decimal point and a zero (that is, I write `3.0` when I want to turn 3 into a floating-point value), but `3.` and `3f` work just as well.

So I could also write our expressions above as `(1./2)*(1.*3/2)` or even `(1f/2)*(3f/2)`.

All of these extra decimal points and letters strike me as a bit messy. But you have to use one of these types of expressions if you want fractional parts in division operations that would otherwise return integers.

Remember, within each set of parentheses you'll need to have at least one floating-point number involved or the result will be an integer.

This issue only applies to division, because it's the only operator that can create a floating-point value from two integers. You don't have to think about this issue for

addition, subtraction, or multiplication.

The thing that triggers Processing to throw away the fraction is the type of the numbers involved, not their values. So as long as one of the variables in an expression is a `float`, then the fractional part is retained.

This takes some getting used to. It can seem strange, but don't feel shy about including a multiplication by 1.0 somewhere in your expression when you're dividing integers. I do it all the time.

4.5 Combined Operators

Processing offers you some very useful notational shorthands that are shared by many modern languages. They make it easy to change the values of variables before or after they get used in an expression.

For example, it's very common to want to add or subtract 1 from a variable, such as when you're counting things:

```
counter = counter + 1;
numItems = numItems - 1;
```

I think I've written lines like this about a billion times (I might be underestimating). To save you typing time, you can use the shorthands `++` and `--`:

```
counter++;
numItems--;
```

These do exactly the same thing as the lines just above; they're just a shorthand way of writing them. Think of `counter++` as a three-step operation: 1. Retrieve the value of `counter`. 2. Add 1 to that value. 3. Save the result back into `counter`.

This shorthand has another trick up its sleeve, though. Notice that the `++` and `--` appear *after* the variable. This means that if the variable appears in an expression, the add-1 or subtract-1 operation also happens *after* the variable is used.

To see this in action, consider four different types of games, and what happens when a player joins or quits. The number of people in each game is given by the variable `numPlayers`.

Suppose you're playing a game where you have some number of tokens, or markers, that move around a board. Each player needs a token. Somewhere during the game, another player shows up. So you need to increment the count of the number of players, and the number of tokens, so that they stay the same. You can do that in one line:

```
numTokens = ++numPlayers;
```

So the computer retrieves the value of `numPlayers`, and because the `++` operator appears before the variable, it is first incremented by 1, and then assigned to `numTokens`.

Suppose later that someone leaves. We want to decrease the number of players, and assign that new, smaller value to the number of tokens. We could write this:

```
numTokens = --numPlayers;
```

Following the same pattern as last time, the computer gets the value of `numPlayers`, first reduces it by 1, and then assigns the result to `numTokens`.

Now suppose you're playing musical chairs, where there is always one less chair than there are players. So if, for example, `numPlayers` is 8, then `numChairs` will be 7. Now someone joins the game. The number of players has to go up by one, and we need to make sure we have one fewer chairs than the new number of players. This would be a wrong way to do it:

```
numChairs = ++numPlayers;  // not correct for musical chairs
```

If `numPlayers` started out as 8, then this could would first take `numPlayers`, add 1 to it to make it 9, and then assign the result `numChairs`. Both variables would have the value 9, which isn't what we want.

What we want is to get the current value of `numPlayers` and assign *that* to `numChairs`, and only then increment the number of players. We could write

```
numChairs = numPlayers++;  // correct for musical chairs
```

This statement takes the current number of players, first assigns that the number of chairs, and *then* adds one to `numPlayers`. The result is that `numPlayers` has the value 9, and `numChairs` has the value 8 (the previous value of `numPlayers`). When the `++` operator comes *after* the variable, it tells the computer to use the value, and *then* increment it.

Finally, suppose we're playing a game that's the opposite of musical chairs, where there's always one extra, empty chair (it's not a great game, I admit), and one of our players leaves. We could write

```
numChairs = numPlayers--;
```

So if before this line was executed `numPlayers` was 8, then the computer will take that value, assign it to `numChairs`, and then decrement `numPlayers` to make it 7. That's just what we want: one more chair than there are players.

You'll see this shorthand all over the place; it's incredibly useful. And though I used a lot of words above, you'll soon come to see something like `baseballs++` and you'll know without even thinking about it that the computer will retrieve the value of `baseballs`, use it in the context of the expression, and then increment it by `1` when it's done.

If the `++` and `--` come after the variable, we call that a *post-increment* and *post-decrement* operation, meaning that the increment or decrement come after the rest of the statement. If they come ahead of the variable, it's called *pre-increment* and *pre-decrement*.

To stay safe, don't use a variable more than once in an expression if you're using one of these operators. For example, don't do this, where I'm using `grapes` twice but one of them has a post-increment applied to it:

```
int cost_of_fruit_bowl = (2 * grapes++) * (3 + grapes);  // don't do this!
```

And never, ever apply more than one of these to the same variable in a single expression:

```
int cost_of_fruit_bowl = (2 * grapes++) * (3 + --grapes);  // don't do this!
```

As King Lear said, "That way madness lies."

Use an extra line or two to do the calculation one step at a time. It's worth it.

You can, of course, freely mix these operators in an expression as long as they apply to different variables:

```
int cost_of_fruit_bowl = (2 * plums++) * (3 + --oranges);
```

By the way, people pronounce these things in different ways when they read them out loud. I usually just say "plus plus" and "minus minus", which sounds a little funny but communicates just fine.

This shorthand comes in a second equally-important flavor: `+=`. This variant handles the case where you want to add two numbers and store the result back into one of them. For example, if a mouse eats some cheese, his weight goes up by the cheese's weight:

```
mouseWeight = mouseWeight + cheeseWeight;
```

Again, this kind of thing happens all the time when programming. If you were just adding `1` to `mouseWeight`, you could write `mouseWeight++` (or `++mouseWeight`). But here we want to add something other than `1`, and the construct `+=` is used to handle this. You can simply write:

```
mouseWeight += cheeseWeight;
```

In other words, we have the 4-step process: 1. Retrieve the value of `mouseWeight`. 2. Retrieve the value of `cheeseWeight`. 3. Add the values together. 4. Save the result back into `mouseWeight`.

This shorthand creates the same result as the previous example, but it's just a little more concise. It has the benefit that you're not typing `mouseWeight` twice. That's nice from a labor-saving standpoint, but even better from a programming point of view. Suppose that the mouse in question is playing hide-and-seek with the household cat. The mouse might find a hiding place and wait, leading you to have a variable in your program called `mouseWait` that tells you how many seconds the mouse waits before moving. If you were working with someone else and you told them what to do on the phone, they might write this code:

```
mouseWeight = mouseWait + cheeseWeight;
```

This could cause very strange results, and you'd have to track down and fix this error. Of course, I picked homonyms here for fun, but when you repeat a variable name a few times on a line you're asking for trouble if you get one of them wrong. Eliminating one of those instances is a good thing. In general, the less your code repeats itself (in any way), the better.

There are four of these types of operators, one for each of the four basic functions: `+=` to add and assign, `-=` to subtract and assign, `/=` to divide and assign, and `*=` to multiply and assign.

The `+=` form (including `-=`, `*=`, and `/=`) is a terrific tool, and you should get in the habit of using it whenever possible. Consider an expression like this:

```
numberOfApples += 5;
```

Just a glance tells you that you now have 5 more apples than before. Consider this:

```
numberOfApples *= 2;
```

You can instantly see that you're doubling the number of apples.

These forms also have the advantage of reducing bugs by helping you not to repeat yourself.

Suppose you're creating a neighborhood scene. There are a bunch of cone-shaped objects in your world: ice-cream cones, traffic cones, simple teepees, and so on, in addition to a generic "cone" that you might use for a child's toy. Each kind of cone has its own variables that define its color, weight, height, and so on. Suppose that near some intersection you want to make the traffic cones twice as tall as usual. You might write

```
traffic_cone_height = cone_height * 2;
```

Later on, looking at your pictures, something doesn't look right. You eventually decide it's the cones, and after a period of hunting through your code you find the line above. What you wanted to so was to double the height of `traffic_cone_height`, but when you typed the code you accidentally named the "generic" cone height stored in `cone_height`. What you meant to write was

```
traffic_cone_height = traffic_cone_height * 2;
```

You couldn't have slipped this way if you'd written it with the `*=` shorthand:

```
traffic_cone_height *= 2;
```

So by getting rid of one of the repetitions of the variable name, we also got rid of a possible source of problems.

This is a general theme that we're going to see over and over: *minimize repetition.*

That's so important, I'll say it again: *minimize repetition.*

If you can type `traffic_cone_height` just once on a line rather than twice, that's one less thing that can go wrong.

The `*=` shorthand works for all four of the basic operations (`+`, `-`, `*`, and `/`). For example, consider

```
total_apples = total_apples - order_size;
```

This works, but it's cleaner to write

```
total_apples -= order_size;
```

For another example, consider

```
total_bill = total_bill * taxPercentage;
```

This also works, but it's shorter and clearer to write

```
total_bill *= taxPercentage;
```

It's not a big difference, but it can help you avoid making errors, and it's a easier to take in the meaning of the line at a glance.

4.6 Starting A Program

In Processing, as in many other languages, there are a bunch of things you do at the start of almost every program. Particularly while learning Processing, it's usually a good idea to simply type (or copy and paste) these lines into every new program right off the bat, before you even start to create your own program. This kind of recurring language is called *boilerplate* (the term is historical, but was adopted by journalists to refer to language that's used over and over again with little to no change; today lawyers also refer to standard legal language in contracts as boilerplate).

It's reasonable to ask why you have to bother: if these things are so common, shouldn't the system just assume them for you? You have to do them yourself because eventually you will want to write a program that doesn't use the boilerplate. If the system always put it in, you might forget it was there. There's not much of it, and it's much clearer to have it in front of you where you can't miss it.

The boilerplate I'll provide here does three things:

1. Puts up a graphics window on the screen.
2. Fills it with a reddish-gray background.
3. Instructs Processing to update the graphics in that window many times a second.

The updating step won't do anything in the boilerplate, but the placeholder will be there for us to modify.

Here's my boilerplate; almost every Processing program I write starts with this skeleton (though I change the specific numbers depending on the size of the graphics window and the color I want to put in it):

```
void setup() {
   size(600, 400);
   background(192, 64, 0);
}

void draw() {
   // drawing goes here
}
```

(Full program in functions/sketches/skeleton1/skeleton1.pde)

This is actually a complete program in Processing. Try it out: type it in and hit the *Run* button. You'll see a window pop up, 600 pixels wide by 400 pixels high, filled with a reddish-gray color. That's all this program does, so once you see it you can close the window.

If you look at the above code and then take a guess about how this program works (in a general way), you'll probably be right. The first function, `setup()`, takes no arguments, and because it returns no results, it has type `void`. It calls two other functions that are built into the Processing language. The first, `size()`, creates a graphics window of the given width and height. I've chosen to make my boilerplate window a little wider than it is tall because I like how that looks.

Here's something that's important to remember: if you call `size()`, then it should be the *first line* in `setup()`. And you will almost always want to call `size()`, because that's what makes the graphics window for you.

Inside of `setup()` I call `background()`, which fills the graphics window with a specific color. The three arguments to `background()` define the red, green, and blue components of the color, each on a scale from 0 (meaning dark) to 255 (meaning bright). I'll return to color later on, but for now, if these 3 numbers don't create a color that tickles your fancy, feel free to change each of them to anything in the range 0 to 255 until you like what you see. You'll find that every graphics object in this book, and every window on which they get drawn, has a color. More times than not, this color is given by three numbers in the range 0-255 that appear in a listing, just as the numbers `192, 64, 0` appeared above. Where did these three numbers come from?

I used an online color-picking program and noodled around until I found a color I liked. At the bottom of the window it listed the red, green, and blue values of that color (almost every color-picking program out there will give you these numbers for the colors you pick). I just wrote them down and typed them in.

I used a jazzed-up color picker, but there's a basic one built right into Processing; you can bring it up by selecting *Tools* from the menu bar at the top of the Processing

window, and then choosing *Color Selector*. So my colors aren't the result of any magic, just lots of playing and eyeballing and looking for nice combinations. Except for the leaf project in Chapter 15, where I referenced photographs of fall colors, I found all the colors in this book that way.

Returning to our boilerplate, the second function, also of type `void`, is called `draw()`. Right now our version of `draw()` is empty except for a placeholder comment.

We don't have to tell Processing to call `setup()`, because that's the first thing the system does when it runs the program (after checking for errors). When Processing starts, it looks for a function called `void setup()` and runs it automatically.

The other thing Processing does automatically is to call a function named `void draw()` many times a second. Each time the computer is ready to redraw the picture, which is typically around 60 times per second, Processing will call the function named `draw()` if you've provided one.

If you want to have your own functions named `setup()` and `draw()` you're out of luck, because those names are special, and reserved by Processing for its own use. The same goes for `size()` and `background()`. As I mentioned earlier, Processing has dozens of such reserved words that you can't use for your own purposes. This is a common affair in programming languages, and not such a bad thing. It means when you look at someone else's code, or even your own, you can quickly identify the function that starts things up by simply looking for `setup()`, and you can find the function that actually draws the picture by looking for `draw()`. It's not uncommon for people to use names like `mySetup()` or `myDraw()` when they really want to use those names for their functions, but with a little thought you can probably find more descriptive names (like `setupAllFurniture()` or `draw_my_spinning_wheel_of_fate()`).

Both `setup()` and `draw()` must be declared as shown: they take no arguments and have a type of `void`. Of course, this boilerplate doesn't really do anything interesting yet, but it takes care of basic housekeeping: a window of a certain size is made, it's colored in with a background color, and then the `draw()` function gets called once for every frame, over and over (even though it does nothing), until you manually stop the program.

This is the boilerplate for a graphics program. If you're not doing any graphics, you can leave out `draw()`, and if you're not writing any functions of your own, you can even leave out `setup()`. For example, our earlier program that printed out "Hello World!" was just a one-line print statement. I'll continue to use those kinds of bare-bones examples when I want to do something simple, like just printing out a value or two. But since Processing is all about graphics, you'll probably want to use this boilerplate (or something like it) when you start your own projects.

4.7 Animation and Global Variables

The boilerplate above creates the "bones" of a working program, so sometimes I call it a "skeleton" project. It compiles and runs, but it doesn't do too much.

So let's make something happen! Right now the only thing we can see is the color of the background in the graphics window, so let's animate that.

To do this, I'll take the three color numbers out of the call to `background()`, and replace them with three variables that I'll define at the top of the file:

```
int redval = 192;
int grnval = 64;
int bluval = 0;

void setup() {
   size(600, 400);
   background(redval, grnval, bluval);
}

void draw() {
   // drawing goes here
}
```

This program does exactly what the last one did. But you can see something new here: three variables that are declared outside of the two procedures. The three variables hold the color's red, green, and blue values. As you can see from the listing above, it's kind of visually nice to see these three variables lined up neatly one under the other; I find that the visual appearance helps reinforce the idea that these three variables are closely related in a kind of "clump". So I used three letters for each of the color names. If I'd written them out in full, it might have looked like this:

```
int redval = 192;
int greenval = 64;
int blueval = 0;
```

I could still get them to line up a bit by making use of the fact that Processing lets us use any number of spaces anywhere that single space would do:

```
int redval   = 192;
int greenval = 64;
int blueval  = 0;
```

I like the first way best, but all of these choices do the same thing; it's just a question of taste.

By the way, another common spelling trick is to write `lo` and `hi` for *low* and *high*. Since each of these has two letters, they stack up nicely. For example, if you wanted to have a range for a variable called `v`, you might save the low and high ends of the range in variables named something like this:

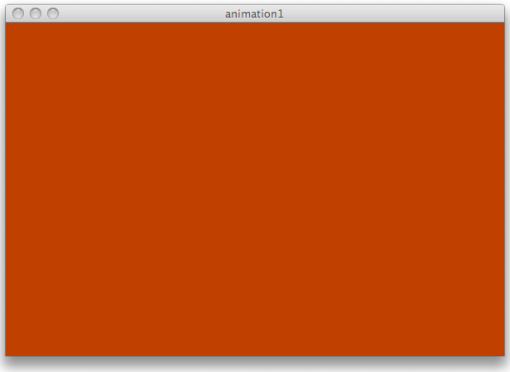

Figure 4.4: A changing background (*functions/sketches/animation1/animation1.pde*) [animation1.png]

```
float vlo = 2.0;
float vhi = 8.5;
```

You might be wondering why didn't I just call the color variables `red`, `green`, and `blue`? You guessed it: those are reserved words in Processing (we'll see what they're used for later in this chapter). So I tacked on the kind of redundant `val` to the end of each name to avoid that conflict.

Returning to the main discussion, I declared these three variables outside the body of any of the functions. Why did I do that? Suppose that I declared the three variables inside of `setup()` - that is, between the curly braces that mark the beginning and end of that function:

```
void setup() {
   size(600, 400);
   int redval = 192;
   int grnval = 64;
   int bluval = 0;
   background(redval, grnval, bluval);
}
```

(see output in Figure 4.4. Full program in functions/sketches/animation1/animation1.pde)

That's perfectly fine and it will work flawlessly. But in Processing, like most other modern languages, variables that are declared inside of a function are only available inside that function. If we want `draw()` to be able to use these variables, it can't do it. They are "inside" of `setup()`, and not available to lines of Processing code outside of `setup()`. Lines inside of `setup()` that come after the declarations can use these variables freely, but to any line outside of that function, they don't exist at all. Trying

to refer to them in any way, even just to read their values, will be an error, because outside of `setup()` they have no existence.

By analogy, any variable declared inside a function is like a book that exists inside your house. That book is available to anyone in your house, but nobody outside your house can read the book; in fact, they don't even know it exists. If we try to use these variables from within `draw()`, we'll get an error.

Let's try it. Run this program:

```
void setup() {
   size(600, 400);
   int redval = 192;
   int grnval = 64;
   int bluval = 0;
   background(redval, grnval, bluval);
}

void draw() {
   background(redval, grnval, bluval);
}
```

(Full program in functions/sketches/animation2/animation2.pde)

When you press *Run*, the single line of `draw()` will be highlighted in yellow, and in the message area you'll get this short report:

```
Cannot find anything named "redval"
```

That's telling us that when we're inside of `draw()`, there's nothing with the name `redval`. And that's exactly right, because that variable can only be referred to by lines of code inside the curly braces of `setup()`, and after the variable was declared.

To make variables available to all functions, we declare them outside of all functions. Since they don't belong to any function, they're public property, and they can be accessed from anywhere.

Remember the phrase "Think globally, act locally"? Those are two common terms used to identify these two kinds of variables. The ones defined outside of all functions are called *global variables*, or simply *globals*. Those that are defined inside a particular function are called *local variables*, or *locals*.

You can tell what the type of a variable is by looking at its declaration, but you can only tell whether a variable is local or global by finding that declaration and then looking around to see if it's inside a function or not. Suppose you're looking at someone else's code and you want to know if a variable is local or global; it could be a hassle to go searching around to figure out which it is.

There are many different conventions that people use to distinguish between local and global variables. My convention in this book will be to start global variables with a capital letter, and start local variables with a lower-case letter.

So the local-variable version of `setup()` looks like this:

```
void setup() {
   size(600, 400);
   int redval = 192;
   int grnval = 64;
   int bluval = 0;
   background(redval, grnval, bluval);
}
```

All the local variables start with a lowercase letter. The global-variable version looks like this:

```
int Redval = 192;
int Grnval = 64;
int Bluval = 0;

void setup() {
   size(600, 400);
   background(Redval, Grnval, Bluval);
}
```

(Full program in functions/sketches/animation3/animation3.pde)

Now we're getting somewhere. The next step will be to change `draw()`. Each time the function is called, it'll redraw the window with the current background color:

```
void draw() {
   background(Redval, Grnval, Bluval);
}
```

Now we can see why these are globals: I want to use them in more than one function. To make things interesting, I'll change the value of `Redval` on each call:

```
void draw() {
   Redval = Redval + 1;
   background(Redval, Grnval, Bluval);
}
```

I could have used our shorthand to increment `Redval`, say by writing `Redval++` or `++Redval`, but I'll do it this way for now.

Try this out! Type in the whole program as it stands now (because we're going to make the red value grow brighter over time, I'm going to start it here at 12, so it has some room to grow):

```
int Redval = 12;
int Grnval = 64;
```

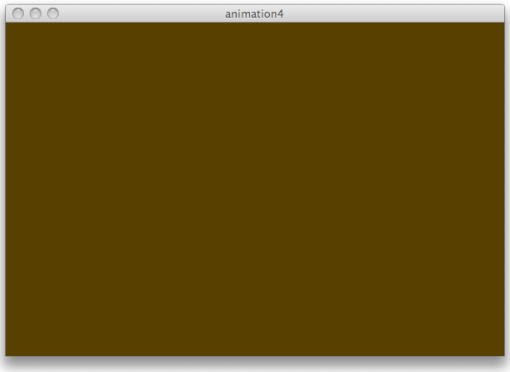

Figure 4.5: The background color animation (frame from animation) (*functions/sketches/animation4/animation4.pde*) [animation4.png]

```
int Bluval = 0;

void setup() {
   size(600, 400);
   background(Redval, Grnval, Bluval);
}

void draw() {
   Redval = Redval + 1;
   background(Redval, Grnval, Bluval);
}
```

(see output in Figure 4.5. Full program in functions/sketches/animation4/animation4.pde)

To run this, press the *Run* button. You'll see the window appears and gradually becomes redder. Congratulations! Your Processing code is animating!

This program works just fine, but you'll notice that the screen gets redder and redder and then stops changing. That's because the value of `Redval` reaches 255 after a while, and that means "maximum red" (I'll talk more about colors later, and we'll see why this strange number is the maximum value, but for now please just roll with it: colors range from 0 to 255). If we hand `background()` a color value greater than 255, it silently assumes you meant 255.

4.8 If Statements

I'd like the colors in our window to keep changing as time goes on. There are lots of ways to do this. One way is to include another statement in `draw()`, which will say,

"If the value of `Redval` is larger than 255, then set the value to 0". If we do this, then the next time `draw()` gets called `Redval` will be 0, and then 1, and then 2, and so on, until it reaches 256 again, when it goes back to 0, and the cycle goes on until we stop the program.

Processing (like most languages) gives us a way to do this: the *if statement* . An *if statement* is really just like the expression I wrote above. Conceptually, the *if statement* has a *test* and an *action*; if the test is true, the action is taken. Let's take a first stab at it by just writing down the English version I gave above:

```
If the value of Redval is larger than 255, then set its value to 0
```

Now let's turn this into a bit of Processing. The first thing is to replace the capital I in If with a lower-case letter:

```
if the value of Redval is larger than 255, then set its value to 0
```

Not much of a change, but an important one. Now we'll fix up the test part. In a Processing *if statement* , the test is placed within a pair of parentheses that come after the word `if`:

```
if (the value of Redval is larger than 255), then set its value to 0
```

Now we'll replace the English text within the parentheses with a little mathematical expression that means the same thing:

```
if (Redval > 255), then set its value to 0
```

Notice here that the greater-than sign `>` is used in its usual way, unlike the equals sign. We're not asserting that `Redval` really *is* greater than 255, because this is merely the test in the *if statement* . If `Redval` is indeed greater than 255, we say that the result of the test is true, otherwise it is false. If the test is true, Processing executes the last part of the *if statement* , which tells it what action to take. If the test is false, the action statement is ignored; Processing simply skips right over it.

The action statement here is an assignment statement, which we're used to from before:

```
if (Redval > 255), then Redval=0
```

The word "then" is good English, but it's not part of Processing. It's not even a keyword. In an *if statement* , the word "then" is just assumed. If you do include it, Processing will complain. So let's remove it:

```
if (Redval > 255), Redval=0
```

We have two things left to do to make this a real Processing statement. First, we get rid of the comma, which (like "then") is appropriate in English but not for Processing:

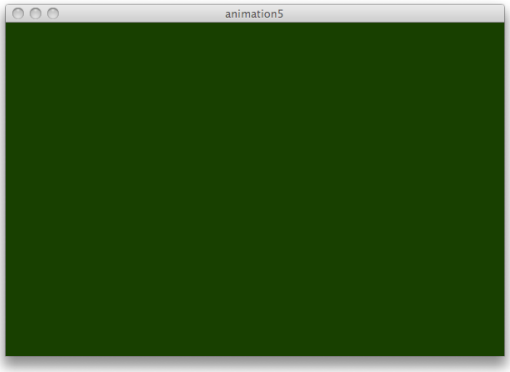

Figure 4.6: The improved background changer (frame from animation) (*functions/sketches/animation5/animation5.pde*) [animation5.png]

```
if (Redval > 255) Redval=0
```

Finally, we add a semicolon to the end, to indicate the end of the *if statement* :

```
if (Redval > 255) Redval=0;
```

That's it! Here's the new program:

```
int Redval = 192;
int Grnval = 64;
int Bluval = 0;

void setup() {
   size(600, 400);
   background(Redval, Grnval, Bluval);
}

void draw() {
   Redval = Redval+1;
   if (Redval > 255) Redval=0;
   background(Redval, Grnval, Bluval);
}
```

(see output in Figure 4.6. Full program in functions/sketches/animation5/animation5.pde)

Give it a try. You'll find that the red part of the color gets brighter and brighter, and then suddenly goes to zero, and then it starts to climb again, over and over until you stop the program.

Try messing around with this program. Add a few more lines to get all the colors changing. See what happens when they change by different amounts on each frame. You might even try having one color component get smaller over time, rather than larger (hint: you'll need to change the *if statement* to make sure it stays in the range 0-255).

The *if statement* I used above shows another programmer's stylistic choice. Remember that I'm happy to let `Redval` grow until it reaches 256, at which time I want to reset it to zero. I might have written this instead:

```
if (Redval == 256) Redval=0;
```

What the heck is `==`? That, unfortunately, is how you test for equality. The equals sign, of course, would be the most sensible choice here, but that's already used to mean assignment, as we've seen. If we forget that and use just a single equals sign in the test, things are really weird:

```
if (Redval = 256) Redval=0;  // this is wrong
```

This says that the test in the *if statement* is actually an assignment statement (remember, think of the equals sign like an arrow). That's no test! The "test" now reads "set the value of `Redval` to 256". So is that true? Is it false? It's neither; it just doesn't make any sense to ask. If you write this, Processing will tell you that you've made an error, flagging the line in yellow and giving you the message

```
cannot convert from int to boolean
```

That's because it's expecting the value of the test to be `true` or `false`, the values that can be taken by a `boolean`. The result of an assignment statement is the value that's being assigned. That's actually very convenient, because it lets us *chain* together assignment statements like this:

```
float weight1;
float weight2;
weight1 = weight2 = 103.5;
```

Reading from the right, first we assign the value 103.5 to the variable `weight2`. The result of that is the value that was assigned, or 103.5. So then we assign that value to `weight1`. Very nice.

So the value we get by using the single equals sign in our test `(Redval = 256)` is 256, which is neither `true` or `false`. Hence, Processing reports an error.

Since a single equals sign is already used for assigning values to variables, Processing uses two equals signs to test for equality (a choice shared by most modern languages). So to test if `Redval` is equal to 256, and set it to zero if it is, we'd write this:

```
if (Redval == 256) Redval=0;
```

Using two equals signs to test for equality is a kludge, no doubt about it. Duct tape and string. We're filling a hole with a piece of gum. But that's how it is. The greater-than and less-than signs work as usual, but if you want to test for equality, you use two equal signs. Don't worry, it'll soon become second nature, and in the meantime, Processing will warn you if you get it wrong.

So why didn't I write the test as `(Redval==256)` to begin with? I could have. But I program defensively. What I was thinking here was, "I want to make sure that `Redval` never gets larger than 255. If it does, I want it to go back to zero." Now in a simple program like this, I know exactly what's happening to `Redval`, and I know that once it's 255, it will next be 256, and then 257, and so on. But what if the program was more complicated? Suppose `Redval` is computed by some complex process in another part of the program, and for all I know it could jump by 2 or even 10 in a single step?

There's a more subtle possibility. Remember that I said that floating-point numbers are often ever-so-slightly wrong? Here I'm using an `int` for `Redval`. But suppose I used a `float`. I might rewrite the incrementing statement like this

```
Redval = Redval + 1.0;
```

This makes it clear that `Redval` is a `float`, but even if I left the increment at 1 as before it would work. But after `Redval`'s been used for a while, and computed with, and maybe is the result of some other computation, it might be 255.0001 before the assignment statement, and then 256.0001 after. If that's the case, then the test for equality will fail, because 256 is not the same as 256.0001. As far as this little program is concerned they should be pretty much the same thing, but the computer doesn't know that. So `Redval` won't get set to 0, and the next time it'll be 257.0001, and then 258.0001, and so on, and the color on the screen will stop changing.

So my habit is to program defensively. I want `Redval` to be reset to 0 whenever it gets beyond 255. By using the greater-than test, it will be true any time `Redval` is more than 255, whether it's by exactly 1 or 7 or some tiny fraction.

Programming is often about making these little tradeoffs as you go. If you program for the more general case when you're first writing your code, it gives you a little more flexibility later on if and when you make changes. That's something you'll probably grow into over time, as you get more familiar with programming and develop your own style. For now, it's worth trying to understand why other people write code the way they do; then you can adopt the practices that you like, and consciously set aside the ones that don't appeal to your sense of style.

Let's add one more twist to our animation. Right now, the red component gets brighter until it hits 255, and then it starts again at 0, and the other colors do nothing. Let's say that when the red component gets to 255, the green component should increase by 10. Here's a first stab at a solution:

```
if (Redval > 255) Redval=0;
if (Redval > 255) Grnval = Grnval + 10;
```

This isn't going to work right. The problem is that the first test resets `Redval` to 0 when it exceeds 255, so by the time we reach the second test `Redval` will never be larger than 255. We could reverse the order:

```
if (Redval > 255) Grnval = Grnval + 10;
if (Redval > 255) Redval=0;
```

That's better. But as I've said before, there's a strategy I urge you to always keep in mind: never repeat yourself. If you find yourself ever writing the same code twice, or even nearly the same code, you should ask yourself if there's a better way. The most important problem with repeating is that it makes your programs more susceptible to bugs. If you mean to repeat something exactly and you make a small error, you might spend a lot of time trying to track it down. But a bigger problem arises when you later come back to your code and make a change. If for any reason you only adjust one of the repeated sections and not the other, the program can start to behave very strangely, and again, you can spend a long time trying to figure out why some parts seem to work the new way and some work the old way.

We can remove the repetition with one *if statement* that does both of these assignment statements one after the other. To replace any statement with a list of statements, all you have to do is enclose them in curly braces.

So here's how we can combine the two lines above into one statement:

```
if (Redval > 255) {
   Grnval = Grnval + 10;
   Redval = 0;
}
```

Here I've again used the indenting style with curly braces that I prefer, but you can format the code any way you like. I grant you that this doesn't look like a big efficiency step, since I've traded two lines of program for four. But the braces don't slow down the program at all. And this actually is a better program, because we're not repeating the *if statement* , so it's cleaner. By not repeating the *if statement* twice, the program is in fact slightly faster! And we can easily add more statements to this list (often called a *compound statement*, or *block statement*, or sometimes simply a *block*).

Of course, now that the green value is moving up as well it will eventually pass 255 itself, and we'll want to reset it to zero (or do something else interesting). If you're feeling adventurous, try writing code that will catch when the green value goes over 255 and use that to bump up the blue value in a similar way.

By the way, in addition to testing for greater-than (`>`), less-than (`<`), and equal-to (`==`), there's also greater-than-or-equal-to (`>=`), less-than-or-equal-to (`<=`), and not-equal-to (`!=`).

For example, if you want to see if `myValue` is not equal to 3, you might write

```
if (myValue != 3) {
   // myValue is something other than 3
}
```

Sometimes it's conceptually easier to write a test in a way that tests if it's false, rather than true. Suppose that you have a function out there somewhere called `isReady()`. It goes off and checks for a piece of hardware connected to your computer, and returns a `boolean`. The result is `true` if the device is connected and ready to communicate with. You'd like to test this and print a message if it's not ready yet.

You could write this:

```
if (isReady() == false) {
   println("Quickly, plug in the flux capacitor!");
}
```

Another way to write the very same test is to see if it isn't `true`:

```
if (isReady() != true) { ... }
```

It turns out that you can use the exclamation point `!` to *negate* any `boolean`. So if you put a `!` in front of something that's `true`, you get back `false`, and vice-versa. So we could also write the test this way:

```
if (!(isReady() == true)) { ... }
```

The real beauty comes from noticing that `isReady()` returns a `boolean` already, so there's no need for the additional testing. We can write the test this easily:

```
if (!isReady()) { ... }
```

In other words, if `isReady()` returns `false`, then the `!` makes it `true`, the test succeeds, and we print the message. We often read the `!` out loud as "not", so `!isReady()` is pronounced "not is-ready". Another common pronunciation of the `!` is "bang", so so `!isReady()` is pronounced "bang is-ready".

If statements have another trick up their sleeve: an optional extra clause called `else`. After the action statement (or block) you can include the word `else`, and then follow it with another statement (or block). The code associated with the `else` is executed if the first code is not; in other words, if the test is true, the first block is executed, otherwise (or else) the second one is:

```
boolean ready_for_time_travel;
if (ready_for_time_travel) {
   println("We're ready, hang on to your hat!");
} else {
   println("We're not ready.  I need the flux capacitor!");
}
```

Here the blocks are only one statement long, so technically I didn't need the curly braces. But they don't hurt or slow down the code at all. I almost always use the curly braces anyway when I use `else` with `if`; that makes it easy to later add lines to either clause. We can get the same result the other way by using the not-equals test:

```
boolean ready_for_time_travel;
if (!ready_for_time_travel) {
   println("We're not ready.  I need the flux capacitor!");
} else {
   println("We're ready, hang on to your hat!");
}
```

These two snippets produce the same result, but the thinking behind them is different. Sometimes it makes more conceptual sense to think of taking action when something has a given property (for example, if it is a leap year) and sometimes it seems better to act when it doesn't have a given property (for example, if it is *not* a leap year).

If statements can also be used to build tests with multiple testing criteria. Let's suppose you want to do some particular action if you have 10 apples on hand and a customer ordered 5. You could find out if both were true by *nesting* one *if statement* inside another:

```
if (apples_on_hand == 10) {
   if (customer_order == 5) {
      // we have 10 apples, and customer asked for 5
   }
}
```

There's an easier way to do this. Suppose that we had some piece of glue that let us represent the idea of *and*, so we could build a single test that said "10 apples are on-hand *and* the customer ordered 5". There is such a piece of glue, and it's written `&&` (this is another odd bit of syntax, but it's common among most modern languages). We could write the test this way:

```
if ((apples_on_hand == 10) && (customer_order == 5)) {
   // we have 10 apples, and customer asked for 5
}
```

When read aloud, some people say just "and" for `&&`, but I find that can be confused with a single ampersand. Some people say "logical and", which is clear and is in fact the formal name for this thing. I usually pronounce it as "and-and".

So the code in the curly braces (currently just a comment) only gets executed if *both* of these tests are true. You can see that my style is to put parentheses around

each clause. You don't have to do that. I do it because I find that code is easier to read with parentheses around each test.

This is a useful way to write tests of the *and* variety, but as we saw above it's not strictly necessary because we could just put one *if statement* inside another to get the same result. But suppose we wanted to make an *or* type of test. That is, we want to execute some code if we have 10 apples on-hand *or* the customer asks for 5. We might write something like this:

```
if (apples_on_hand == 10) {
   // do a thing
}
if (customer_order == 5) {
   // do the same thing
}
```

This has some problems. First, we're repeating code in two places, and that's usually a bad idea. But worse, if both of these conditions are true, we'd end up repeating the code twice. At the very best that's a waste of time, and at worst it could cause our code to malfunction. We could write a whole bunch of code to work around this problem, or we could just write `||` (that's two vertical bars, one right after the other) to glue the tests together into a compound *or* statement:

```
if ((apples_on_hand == 10) || (customer_order == 5)) {
   // do the thing
}
```

In this case, the code between the braces (now just a comment) will get executed if either (or both) of these clauses are true.

As with the logical-and `&&`, when read aloud some people say just "or" for `||`, but I find that can be confused with a single vertical bar, which is also used for a kind of "or" function. Some people say "logical or", but I usually pronounce `||` as "or-or". It can sound like a seal barking, I know, but it works for me.

You can have as many clauses as you want in a test, and you can make sure they get grouped correctly using parentheses. Here's an example:

```
if (((month==3) || (month==5)) && ((day==1) || (day==15)) {
   // do something
}
```

Now the action (so far, just a comment) will be executed *if* the month is March or May, *and* the date is the 1st *or* 15th. In other words, four days out of the year will satisfy this test: March 1, March 15, May 1, and May 15.

Finally, you can negate the result of any test with `!` (that's just a single exclamation point). In the last example, we wanted to execute code on the 4 days I identified. But suppose we want to execute the code if it's March or May, and the date is *not* the 1st or 15th? Here's one way to get there:

```
if (((month==3) || (month==5)) && !((day==1) || (day==15)) {
   // do something
}
```

All I did was put a ! before the second chunk. So now the test succeeds if it's March or May, *and* the day is *not* either the 1st *or* the 15th.

These kinds of compound tests can usually be written in lots of different ways, all of which are logically equivalent. Here's another test that achieves the exact same result:

```
if (((month==3) || (month==5)) && (day != 1) && (day != 15)) {
   // do something
}
```

This is probably about as complicated as you want one of these tests to get. If you have too many of these clauses it can get very hard to read and understand the code, even if you wrote it! Simpler is usually better, even in programming.

To that end, you might want to build your test in pieces. It takes a little more text, and might run imperceptibly slower, but it's easier to read at a glance:

```
boolean marchOrMay = (month == 3) || (month == 5);
boolean neither1or15 = (day != 1) && (day != 15);
if (marchOrMay && neither1or15) {
   // do something
}
```

Breaking things up into pieces like this is usually a much better approach than writing a big complicated *if statement* . You could of course explain a complex *if statement* with comment or two, but writing it out clearly makes it easier to write, understand, and debug.

4.9 Conditional

There are a couple of variants on the *if statement* that can be handy.

The first is called the *conditional*, and it's really nothing more than a text shorthand. Instead of writing

```
if (test) statement1; else statement2;
```

you write

```
test ? statement1 : statement2;
```

This saves you a few characters. The place you see this the most often is in assignment statements:

```
float carSize = ( passengers > 3 ) ? sedan_size : sports_car_size;
```

You don't have to include the parentheses around the test in this kind of conditional statement, but I almost always do, anyway. I think they make the test a little easier to read, and easier to see at a glance that it is indeed a test.

4.10 Switch

Another if-like tool is the `switch` statement. This helps you write a complicated *if statement* in a clean way. For example, suppose you've written a game where someone is shooting an arrow at a target. When the arrow hits, you want to draw a glowing ball based on the score value of the location where they hit. Each zone has its own color of ball with a different size; the higher your score, the bigger and brighter the ball. Here's one way to do that sort of thing (rather than implement the whole game, I'll just focus here on drawing the glowing ball).

The function `drawBall()` contains three Processing functions that we haven't discussed yet: `color()` defines a color, `fill()` tells Processing to use that color when drawing, and `ellipse()` draws an ellipse (or circle). We'll discuss `color()` next in Chapter 5, and we'll cover the other two procedures in Chapter 6.

```
void setup() {
   size(600, 400);
   background(87, 66, 8);
   noStroke();
   int score = 40;   // the score of this arrow

   // draw the ball for the current value of score
   if (score == 40) {
      drawBall(160, 212, 20, 20);
   } else if (score == 30) {
      drawBall(80, 231, 48, 3);
   } else if (score == 20) {
      drawBall(40, 255, 93, 8);
   } else if (score == 10) {
      drawBall(20, 255, 140, 5);
   } else {
      drawBall(180, 32, 32, 32);
   }
}

void drawBall(int radius, int redval, int grnval, int bluval) {
   fill(color(redval, grnval, bluval));
```

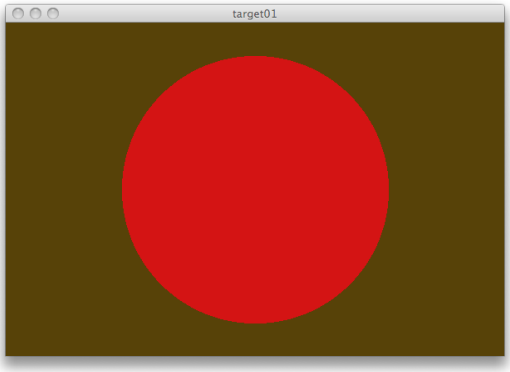

Figure 4.7: The bull's-eye target (*functions/sketches/target01/target01.pde*) [target01.png]

```
   ellipse(300, 200, 2*radius, 2*radius);
}
```

(see output in Figure 4.7. Full program in functions/sketches/target01/target01.pde)

That certainly works, but it's kind of ugly and can be hard to read. The `switch` statement is a different way to write the same thing. You state the word `switch` followed by a pair of parentheses and a pair of curly braces. You put a variable between the parentheses, and this guides Processing to choose one option among many that you provide between the curly braces.

Each of these potential choices begins with the word `case`, followed by a space, and then a value. If the variable that came after `switch` has the value given in that `case` clause, the following code is executed until you reach a `break` statement. When Processing hits the word `break` it jumps immediately to the closing curly brace, and continues on from there. On the other hand, if the variable doesn't have the value given after that appearance of `case`, Processing looks for another `case` statement and repeats the test. It keeps this up until one of the `case` statements matches.

If none of them match, then it executes the code after a final clause labeled `default` (if there is one).

Here's the *if statement* above written in switch form:

```
switch (score) {
   case 40:
      drawBall(160, 212, 20, 20);
      break;
   case 30:
      drawBall(80, 231, 48, 3);
      break;
```

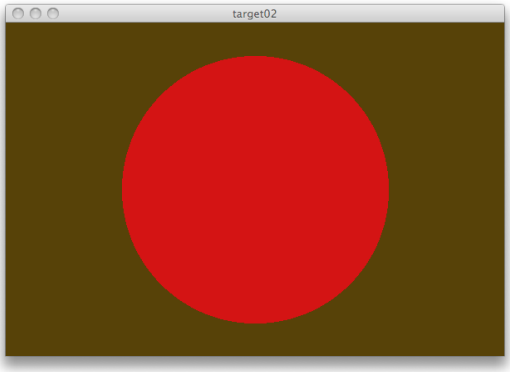

Figure 4.8: Target with the switch statement (*functions/sketches/target02/target02.pde*) [target02.png]

```
   case 20:
      drawBall(40, 255, 93, 8);
      break;
   case 10:
      drawBall(20, 255, 140, 5);
      break;
   default:
      drawBall(180, 32, 32, 32);
      break;
}
```

(see output in Figure 4.8. Full program in functions/sketches/target02/target02.pde)

This does the same thing, and it's a lot cleaner to read. It's a little longer, granted, but I think it's worth the tradeoff. Each of the `case` clauses starts a short block of code (in this example, only one line long) that gets executed if the `switch` variable matches its value. For example, if `score` has the value 30, then we execute the line

```
drawBall(80, 231, 48, 3);
```

Since the next line reads `break`, the program jumps to the closing curly brace and proceeds.

Notice the last test, labeled `default`. This allows you to specify some actions to be taken if none of the earlier conditions are met. It's kind of the equivalent of `else` in an *if statement* . The `default` clause isn't mandatory, but I recommend you always use it. Even if you've covered all the possibilities you expect, you can use the `default` clause to catch any errors. I often use my `default` clause to contain something like this:

```
switch (score) {
   case 1:
      ...
      break;
   case 2:
      ...
      break;
   default:
      println("Error! I don't know how to handle score = "+score);
      break;
}
```

You might think this looks a little odd. Given what we've seen before, you might expect the lines after each `case` statement to be in a block - that is, within curly braces. When you hit the closing curly brace, you're done with the block and you exit the switch. Why isn't it done that way, and instead there are these `break` statements?

There are two reasons for this. The first is that this form lets you list multiple case statements one after the other, creating an *or*-type test. Let's suppose that you added a 25-point zone to your target, but if the player strikes that, you still want to draw the 20-point ball. Then you could write this clause:

```
case 20:
case 25:
   drawBall(40, 255, 93, 8);
   break;
```

In this fragment, if `score` had the value 20, the first `case` statement would match, so Processing would start looking for code to execute. It would skip the next `case` statement, since there's nothing there to do, and continue on to the call to `drawBall()`. If `score` had the value 25, the second `case` would match, and the same call to `drawBall()` would be executed.

In other words, the same ball would be drawn for either score. Stacking `case` statements like this is way of building a kind of *or* test.

The other advantage of this form is that if you leave out a `break` statement, Processing ignores the next `case` statement and keeps on executing each line of code. This continues until you hit a `break` or exit the switch by reaching the closing curly brace.

This lets you pile up choices. For example, let's say that rather than providing just one glow per score, each score gets all the lower-scoring glows as well, for a super-vibrant crazy colorful reward. Here's one way to get that result:

```
switch (score) {
   case 40:
      drawBall(160, 212, 20, 20);
   case 30:
```

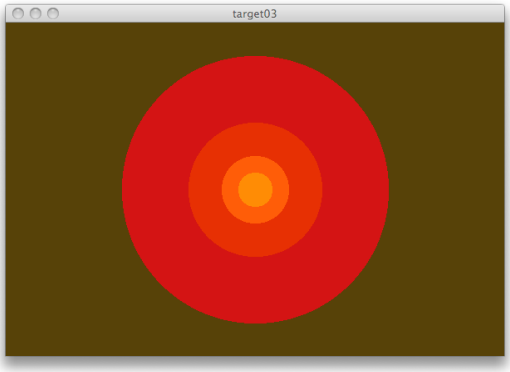

Figure 4.9: The switch statement without breaks (*functions/sketches/target03/target03.pde*) [target03.png]

```
      drawBall(80, 231, 48, 3);
   case 20:
      drawBall(40, 255, 93, 8);
   case 10:
      drawBall(20, 255, 140, 5);
      break;
   default:
      drawBall(180, 32, 32, 32);
      break;
}
```

(see output in Figure 4.9. Full program in functions/sketches/target03/target03.pde)

When score is 10, you just get the smallest ball. When score is 20, you get both the score-10 ball *and* the score-20 ball. And so on, all the way up to the big red score-40 ball.

Be careful with the `switch` statement, though, because if you forget the `break` statement where you meant it to appear, you'll get strange results that can sometimes be a challenge to debug. In the example above, I have a `break` after the case for when `score` is 10. Try taking this `break` out. Then every score results in a big dark-gray ball! That's because I'm drawing that ball for the `default` case, and without a `break` statement, that case gets executed last, and so the circle it draws covers up all the previous ones.

By the way, you don't really need a `break` statement at the end of the last clause in a switch statement, because there's nothing after it. It doesn't hurt in any way, but it doesn't help, either; it's kind of a nothing. But I include it anyway because it's good to be in the habit of always including a `break` statement at the end of every case in a switch.

Not only is a `switch` statement often easier to make sense of when reading your code, compared to some big chain of `if` statements, it often runs faster, because the computer can jump directly to the right clause without testing everything along the way.

4.11 Speed and Control

One of the wonderful things about Processing is that it can run on almost any computer. It can even run inside a web browser! But that power brings with it a problem: making things run at the right speed.

Every computer has one or more processor chips inside, and they're responsible for everything that's going on: spinning the fans to keep the insides cool, drawing the screen, responding to the mouse, checking your email, updating the clock in the corner of your screen, playing the music you're listening to on your headphones, and oh yeah, running your program. Who knows how much time it has left to devote to your program? Even if we stripped away every one of these other tasks, almost every model of every computer runs at a different internal speed.

This is relevant to us because we want our animations to run at the speed we want. For example, movies are typically shot and shown at 24 frames per second. Imagine instead if every movie camera shot film at its own random speed, and every movie theater in the world showed its movies at its own, also arbitrary speed. It would be terrible: some movies would be slightly too fast, others slightly too slow, and some would be so insanely fast or slow that they'd be unwatchable.

But that's the situation we have now with computing. You design your gorgeous animation so it runs perfectly on your computer that runs at its own speed (like the film camera), and now other people view it on their computers running at their speeds (like the film projectors). Is there any way to be sure that people see on their machine something like what you designed on yours?

This problem is huge, and not limited to animation. Color, for instance, is a giant problem, as you know if you've ever ordered clothing online and you were surprised by the color of the actual product when it arrived. A lot of people have worked very hard on the color problem, but we're not quite there yet.

Solving the animation problem is even harder, because it's just so hard to figure out how fast a computer is running, and how much time it has to draw pictures for you. So there are two essential questions: how many frames per second can a particular computer draw, and how much time can it spend on each frame, given the other things it has to do?

You can tell Processing that you'd like it to aim for a certain *framerate*, or number of frames per second. If you call `frameRate()` with an integer value, Processing will aim for that number of frames per second. The default is 60 frames per second.

If your computer can't do all the work you're asking of it and still draw your frames at this speed, you'll get a slower *update rate*. You can ask Processing how many frames

per second you're actually getting (as opposed to the number you asked for) by checking the variable `frameRate`. Note that this has the same name as the function, but is lacking the parentheses. Processing keeps this variable up-to-date as time goes by (because sometimes your computer might be busier than others, or your graphics might take longer to draw sometimes, which could slow down your frame rate). The value in `frameRate` is only an estimate, but it's usually pretty close.

A related idea to the frame rate is the *frame count*. This is just a number that tells you how many times your `draw()` function has completed. So when you start, it's 0. After the first frame is drawn, it's 1, and so on. You can find out the current frame count just by using the system variable `frameCount`. It's surprisingly handy for all kinds of things that you want to change from frame to frame. I'll use `frameCount` often in this book.

If you don't include a `draw()` function, then of course it never gets called. But there are times, particularly when debugging your programs, when you only want to call `draw()` once. As we've seen, if you have a `draw()` function defined then normally Processing will start producing your graphics as soon as the program begins.

But sometimes you might not want your program to start drawing immediately. For example, your animation might be part of a museum kiosk. You've designed it so the program normally sits there silently until one of the museum's visitors pushes a button. Then it starts and runs through a full performance of the animation, and then stops again.

As we saw before, you can call `exit()` to quit the program, but then it would be hard for the user to start it again. What you want is a way to tell Processing, "start calling `draw()` now" and later, "stop calling `draw()`". When it's calling `draw()`, your animation is updating at the current frame rate. When it's not calling `draw()`, nothing's getting drawn.

If you want Processing to stop calling `draw()`, you call the function `noLoop()`.

As we'll discuss in Chapter 7, you can still respond to user events like keyboard presses and mouse clicks even when you're not animating. So if your visitor pushes a button on the display, that might correspond to hitting (say) the space key on a keyboard. If your program used that as the trigger to start calling `draw()` again, it can respond to this key press by calling `loop()`.

If you want `draw()` to run just once, you can call `redraw()`.

So this tells us something about how to regulate the speed of our program, but just asking for a huge frame rate doesn't guarantee we're going to get it. If we want our program to run at a certain speed, and the computer we're running it on just can't deliver that much horsepower, then we won't get the performance we want. To make sure our programs run as fast as possible, we have to write them so that they are as efficient as possible.

Efficiency is a broad and deep topic. It's usually not as important in Processing as in other languages, but it can be an issue, so I'll mention efficiency considerations from time to time as we work through our projects. Generally you'll want to keep it in the back of your mind while working, but don't let it be your main focus. The goal of any

program is to make it work. Making it fast is secondary.

But from time to time I'll talk about some operation or process as *expensive*. This usually means either that it's slow, or that it consumes a lot of the computer's memory (which often makes it slow). Sometimes you just have to do "expensive" things. If you're showing a flying bird and you really need to show each feather moving individually, you're going to have to draw each feather, and that's that. But you can still try to draw each one as quickly as possible.

And some expensive things can be made cheap, either by re-organizing them, or by *cheating*.

When we talk about "cheating" in programming, we almost never mean some nefarious, morally objectionable process. We mean faking it. Finding a cheap way to do something that looks expensive. Suppose I'm showing a parade at the zoo, and there are dozens of animals walking across the screen. Lions, tigers, elephants, every one of them is a complicated object with tons of moving pieces and fur and muscles and textures. Drawing those animals is going to take a lot of time.

Then I notice that the flamingos are far away the entire time. And even then, from where I stand, they are almost always obscured. In fact, I really only see a little flash of bright pink every now and then between the legs of the other animals.

I might get rid of the 30 individual flamingos altogether and replace them all with a single big pink rectangle that I draw behind all the other animals. The result, from the viewer's point of view, is identical. But I've saved myself an enormous amount of work.

I might then go on to notice that I only see the near side of each animal. There's no reason to draw the far-away parts that I can't see. So I'll cut my animals in half, and only draw the near part. As long as everything is positioned and moves in just the right way that the cheat isn't given away, I've managed to draw my graphics faster than before but they'll look just as good.

That's the essence of "cheating" in this context: you find a way to reduce your workload, typically in a way that is specialized to exactly the situation at hand.

So in the projects we'll see throughout the book, I'll keep an eye open to places where something that is obviously expensive can be replaced with something that's cheaper and not much more complicated. I'll rarely sweat the tiny details because all of our programs will be relatively bite-sized. But the issue of efficiency can be important when we start to do a lot of work for every frame, so I'll come back to efficiency in Chapter 18.

Chapter 5

Color

In this chapter I'll talk about colors. But before we get into it, here's another program for you to play around with. This program creates images that move over time, so the real fun comes from watching it run.

```
float StartAngle = 0;
float AngleBump = 0;
color Color1 = color(180, 95, 10);
color Color2 = color(0, 80, 110);

void setup() {
   size(400, 400);
   smooth();
}

void draw() {
   background(Color2);
   noStroke();
   float radius = 400;
   int circleCount = 0;
   float angle = StartAngle;
   while (radius > 0) {
      fill(Color1);
      ellipse(200, 200, radius, radius);
      fill(Color2);
      arc(200, 200, radius, radius, angle, angle+PI);
      radius -= 30;
      angle += AngleBump;
   }
   StartAngle += .01;
   AngleBump += .005;
}
```

Figure 5.1: Spinning wheels (frame from animation) (*color/sketches/wheels01/wheels01.pde*) [wheels01.png]

(see output in Figure 5.1. Full program in color/sketches/wheels01/wheels01.pde)

Figure 5.1, Figure 5.2, and Figure 5.3 show three screenshots of the this running program, taken at different moments as it ran.

Colors are everywhere in a Processing sketch. Everything you draw has a color, and sketches often cook up new colors and even random colors as they run.

We've already seen the two key elements that usually go into a color description: colors are usually constructed with three numbers (one each for red, green, and blue), and each number is in the range 0-255. Let's look more closely at this.

The three-number, red-green-blue (or RGB) format is a very common one in computer graphics. It started before computer graphics even existed. When engineers were inventing color television, they had to find a way to display the broadest possible range of colors on a screen. They were guided by an important feature of our biology: human eyes contain three types of cells that respond to color. One type responds largely to reddish colors, another to greenish, and the last to blue-ish. They reasoned that they could probably make people see any color they wanted if they were able to stimulate these three types of cells the right way. So they coated the inside of the display screen with chemicals called *phosphors* that glowed when hit with a beam of electrons. Happily, you can go digging in the Earth and find rocks that contain phosphors that glow with different colors. So they found some red, green, and blue-glowing phosphors, and it all worked out very well: you can create a huge variety of colors just by mixing different amounts of red, green, and blue light.

Figure 5.4 shows the *RGB Color Cube*. In one corner, where all three values are zero, you find black. In the opposite diagonal, where all three values are 255, you find white. At the ends of the red, green, and blue edges you'll find the strongest versions of each of those colors. The other corners hold mixed pairs of these strongest colors: red+green=yellow, red+blue=magenta, green+blue=cyan. In part a of the figure I've shown the entire color cube. The green axis is hidden from us, so in part b I've removed a chunk from the near corner, revealing the cube's insides. In part c I've sliced the cube

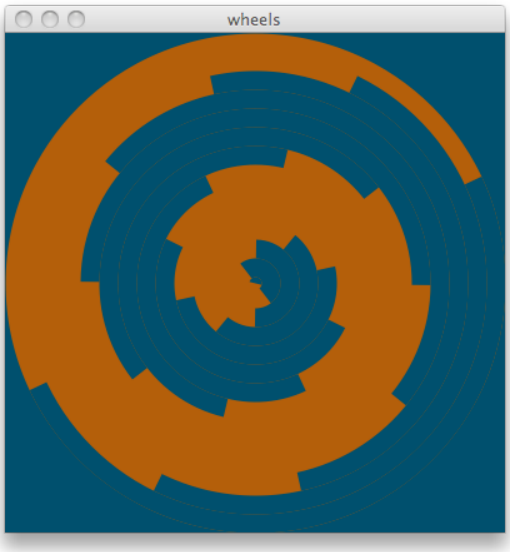

Figure 5.2: Spinning wheels a bit later [wheels02.png]

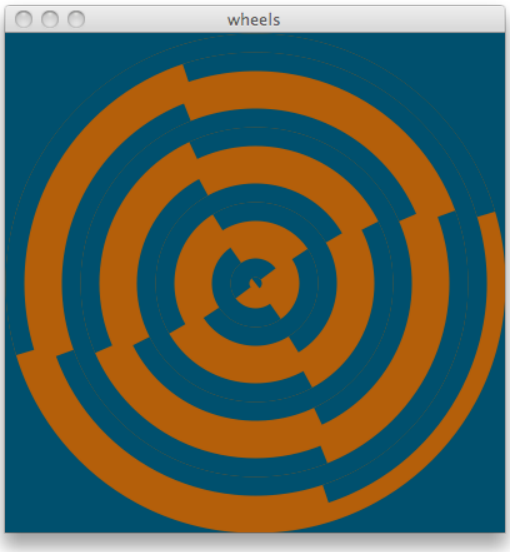

Figure 5.3: Spinning wheels yet later [wheels03.png]

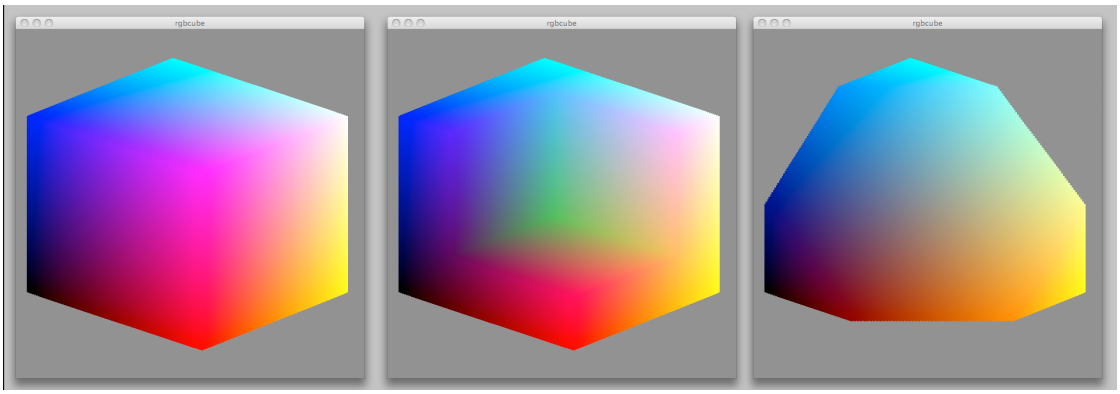

Figure 5.4: The RGB Color Cube. a) The RGB Cube. b) A box taken out of the near corner. c) The cube sliced along the diagonal, revealing the middle. [rgbcube.png]

in half along a plane through the center of the cube, perpendicular to the line from the black corner to the white corner.

If you're familiar with mixing pigments, you probably think something is wrong here, because the primary colors for pigments are red, yellow, and blue. It turns out that when you're mixing light you use three different primary colors than when you're mixing pigments, like paints or polymer clay. The reasons for this are fascinating, but it would take us too far afield to go into it here. If your curiosity is aroused, there is a ton of information on this all over the web. You can search on the term *subtractive mixing* for discussions of pigment mixing (for example, using the primaries cyan, magenta, and yellow) and *additive mixing* for discussions of light mixing (with the primaries red, green, and blue).

If you'd like to see how red, green, and blue light is mixed to make colors on the computer, go to the Processing window and from the *Tools* menu choose *Color Selector*. There you'll be able to select colors interactively, and watch the live feedback as the tool shows you how much of each primary hue is in each color. Figure 5.5 shows the Color Selector in action.

Processing's built-in color selector is useful in a pinch, but it's a pretty bare-bones affair. There are lots of much more powerful programs out there for picking colors, and even picking sets of colors that need to work together. You can find them embedded in other packages (like the Color Selector in Processing), as well as in standalone commercial packages, and even free web apps. In that last category, I like Kuler (http://kuler.adobe.com/).

So red, green, and blue are the primaries for mixing light, but why represent their strengths with numbers from 0 to 255? The answer comes from the early days of computer graphics.

When computer graphics was just getting started, color TV monitors were the best way to show computer-generated pictures, particularly those that moved. So naturally people stored their pictures in the three-color format, so they could directly drive the

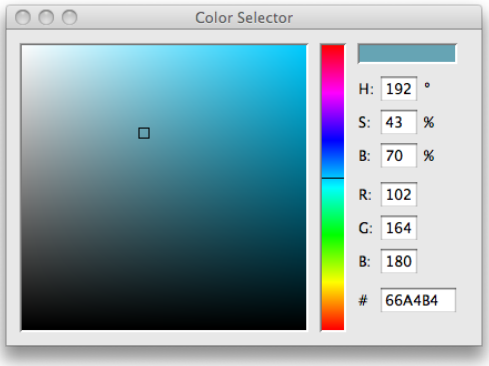

Figure 5.5: Processing's Color Selector. [colorselector.png]

TV electronics. Thus every dot in the picture corresponded to one or more dots on the screen, and it was described with three numbers, one each for the red, green, and blue brightness of that dot.

A value of 0 seems perfectly reasonable to mean a given color component should be "fully off", but how about "fully on"? You want some range of values between off and on, and generally speaking, more is better. If we had only, say, 10 values, you would see all kinds of artifacts in the images. One of the most obvious is called *posterization*, which looks like a poster printed with only a few colors: you get big flat regions of color, with very visible boundaries between regions of different colors. To get smooth color transitions, you need lots of intermediate colors. So how many?

You might have a USB thumb drive in your pocket. It might hold something like 4 or 8 gigabytes of memory. That's 4 or 8 *billion* bytes. A byte is 8 bits, and a bit is a single 0 or 1. It's hard to believe now, but not too long ago every byte was precious. For example, the entire computer system onboard the Apollo spacecraft that went to the moon had only 60 kilobytes of memory. That's 60,000 bytes. For everything: regulating the systems, planning the descent to the moon, running life support, firing the jets, getting into the right position for re-entry, getting to the moon, getting home again, everything. Sixty thousand bytes. Every single byte was precious, and programmers had strict limits on how many bytes they could use. The entire Apollo spacecraft computer system wouldn't have been big enough to hold even a single typical song in mp3 format. Not one. Amazing.

This was the situation in the early days of computer graphics. So when people needed to represent color values, they tried economizing on storage by using a single byte per color. A byte can hold numbers from 0 (which is all 0's) up to 255 (which is all 1's). That's it - there aren't any more beads on the abacus: a byte can hold a number from 0 to 255, end of story. So colors were represented with three bytes, one each for red, green, and blue. If we count the numbers from 0 to 255, there are 256 values total (remember to include the zero!).

The choice of a byte was driven largely by the need to save memory, but it turned out to be a pretty good choice. By and large, pictures created with 256 levels per color looked good and were mostly free of artifacts, so that way to describe colors became very popular.

That's not the end of the story, of course. There are other ways to describe colors than combinations of red, green, and blue (and even then, to be careful you have to specify just exactly what you mean by "red, green, and blue"). And 256 levels per component isn't enough for applications like feature films or high-end photography; people working in those fields use more levels per color.

But the one-byte-per-component model, with 256 levels for each of red, green, and blue, has become a standard in much of computer graphics. It's like having four wheels on a car; maybe three or five would be better, but four is the norm. The move to have more levels per color component has picked up steam in recent years, with previously professional-grade rendering and photo-editing software coming down in price, but the 8-bit approach is still popular, and that's what Processing uses. For the kinds of things we do with Processing, 256 levels per component is usually just fine.

To create a color in Processing, you declare an object to be of type `color` (with a lower-case `c`):

```
color myColor;
```

To actually fill that in with a color value, you call the built-in function `color()`, and hand it three numbers, one each for red, green, and blue. Here's a list of some common colors:

```
color pure_red = color(255, 0,   0);
color pure_grn = color(0,   255, 0);
color pure_blu = color(0,   0,   255);
color yellow   = color(255, 255, 0);
color cyan     = color(0,   255, 255);
color magenta  = color(255, 0,   255);
color white    = color(255, 255, 255);
color black    = color(0,   0,   0);
```

Happily, `color()` can take either three `int` or `float` variables as arguments (so you could use, say, 153.6 as one of the color values).

Um, wait a second. What was that I just said, that the function `color()` can take either three `int` or `float` arguments. That doesn't seem to fit the rules we've seen so far, but it sounds useful. Can we write our own procedures that do that?

We sure can. Let's see how.

5.1 Function Overloading

Suppose that we write two version of a procedure, but give them both the same name. They'll differ only in the argument list. The first time the argument list will have one

parameter. The second time, the argument list will have two parameters. Processing considers those two different procedures. When you call one of them, it automatically goes to the version of the function that matches the number and types of arguments you're calling it with.

For example, let's suppose that we want to find the size of some things lying around the house. Some things have only length (like a yardstick), some have area (like a sheet of paper), and some have volume (like a microwave oven). I'll write three versions of a function I'll name `findSize()` that will return to us a floating-point number with the proper measurement. In `setup()`, I'll call each of these:

```
void setup() {
   float length = findSize(3);
   println("length = "+length);
   float area = findSize(4, 6);
   println("area = "+area);
   float volume = findSize(2, 3, 5);
   println("volume = "+volume);
}

float findSize(float length) {  // 1D size - just returns the input length
   return (length);
}

float findSize(float width, float height) { // 2D size - return area
   return (width*height);
}

float findSize(float width, float height, float depth) { // 3D size - return volume
   return (width*height*depth);
}
```

(Full program in color/sketches/overload1/overload1.pde)

The output of this program appears in the console window:

```
len = 3.0
area = 24.0
volume = 30.0
```

There's nothing wrong in defining a version you never call, but don't try to call a version you haven't created. That's like calling any other function that doesn't exist. If I try to call

```
float f = findSize(1, 2, 3, 4);
```

I'll get an error.

Writing multiple versions of a procedure is a useful technique, but it can be a dangerous one. If the different versions of your functions do very different things, you can accidentally find yourself calling the wrong one. And which one you do get can be subtle. Let's rewrite `findSize()` in two versions: they both return a `float`, but one of them takes a `float` as input, the other an `int`:

```
void setup() {
   float len = findSize(3);
   println("len 3 = "+len);
   float flen = findSize(3.0);
   println("len 3.0 = "+flen);

}

float findSize(float len) {  // return the input
   return (len);
}

float findSize(int len) {  // return the input * 5
   return (len*5);
}
```

(Full program in color/sketches/overload2/overload2.pde)

Here's the output:

```
len 3 = 15.0
len 3.0 = 3.0
```

The technique of using multiple versions of a function that are selected based on their definitions goes by a few names, the most common of which is probably *overloading*. Processing uses overloading all over the place internally; that's why `color()` can take any mixture of `int` and `float` values as inputs. But this technique can make debugging tricky, since in complex situations it can be hard to be sure which function is being called.

In this example, if we pass `findSize()` a `float` we get back the number we sent it, but if we pass it an `int` we get back that value multiplied by five. Suppose that while developing this program you originally use both versions of `findSize()`, but over time you remove the `int` based calls one at a time until none happen to remain. Then you put the project aside.

A month later you have a great idea for improving your project. You pop it open and find yourself working on a section that has a call to `findSize()`, and you want to add a few more calls to that function as part of your new code. But you've forgotten that there are multiple versions of `findSize()` out there (perhaps the function is in a

different source file, so it's not even on your screen at all). So you add some new lines of code, including some new calls to `findSize()`... and you know where this is going. One or more of your new calls passes an `int` rather than a `float`. Your code compiles perfectly cleanly (because, of course, there *is* an `int`-based version of `findSize()` out there to be called), but your new program goes haywire. You scratch your head and you read the code over and over, but it looks perfect. In order for you to find this bug, you're going to have to re-discover those multiple versions of `findSize()` that are sitting out there, innocently helpful, but also crazy dangerous.

Overloading can be a great time-saving trick and a handy convenience, but with great power comes great responsibility. Definitely use overloading when appropriate, but be very, very careful.

A good rule of thumb for overloading in Processing is to restrict your overloads to different versions of the input parameters that all do identical operations. For example, one version of `color()` takes 3 `int` variables, and the other takes 3 `float` variables, but they both create a color from those numbers. That's pretty safe.

If you want to do different things based on the number of arguments, or their types, I strongly recommend that you write several different procedures, each with its own unique name. It's only a tiny amount of additional work, but it can save you from some very frustrating debugging sessions.

5.2 Defining Colors

When used with care, overloading can be very useful. For example, `color()` has yet another overloaded version that's frequently handy: if you give `color()` just one number, it uses it for all three values, creating a shade of gray:

```
color black      = color(0);
color dark_gray  = color(64);
color mid_gray   = color(128);
color light_gray = color(192);
color white      = color(255);
```

Note that the word `color` is used in these examples in two different but related ways: to declare the type of object (on the left of the equals sign) and to invoke a built-in function to create one of those objects (on the right).

You'll get used to talking about colors in this fashion very quickly. If you set all three values to the same number, you get grays from black to white. So here are the same colors as above, only more explicitly described:

```
color black      = color(0,   0,   0);
color dark_gray  = color(64,  64,  64);
color mid_gray   = color(128, 128, 128);
color light_gray = color(192, 192, 192);
color white      = color(255, 255, 255);
```

Once you've made a color this way, you can hand it off to Processing to use when drawing things. For example, instead of giving `background()` the three red, green, and blue values directly, as we have been, you can just hand it a color variable instead. Here's a rewrite of our background-changing program:

```
int Redval = 192;
int Grnval = 64;
int Bluval = 0;
color MyColor;

void setup() {
   size(600, 400);
   MyColor = color(Redval, Grnval, Bluval);
   background(MyColor);
}

void draw() {
   Redval = Redval+1;
   if (Redval > 255) Redval=0;
   MyColor = color(Redval, Grnval, Bluval);
   background(MyColor);
}
```

(Full program in color/sketches/color1/color1.pde)

Of course, this isn't much of a savings in this example, but later on the ability to save colors this way will be very convenient.

If you want to disassemble a color variable, Processing offers you three built-in functions, quite sensibly called `red()`, `green()`, and `blue()`. They each take a `color` as input, but they each return a `float`, even though the value is always an integer. Remember that to convert a `float` to an `int`, hand it to the built-in function `int()`:

```
color MyColor  = color(Redval, Grnval, Bluval);  // create a color
float newRed   = red(MyColor);        // get the red component
float newGrn   = green(MyColor);      // get the green component
float newBlu   = blue(MyColor);       // get the blue component
newRed = newRed * 2;                  // make the red double-bright
color NewColor = color(newRed, newGrn, newBlu);   // save the new color
```

For this little example I didn't check the value of `newRed`, but in general you should always make sure your color values are in the range 0-255. Processing won't complain if they're not, but you probably won't see the results you were hoping for.

You'll see this quite a lot in some Processing programs: people compute a color in one function, and then they take it apart in another function, change it, and save

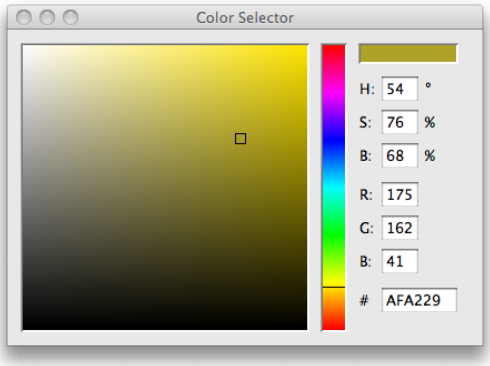

Figure 5.6: Processing's Color Selector. [colorselector2.png]

it again. There's a tradeoff here: the packing/unpacking business takes some time, but it's very convenient to have the color packaged up in a single variable, rather than having to deal with the three red, green, and blue components all the time. If program speed is a significant concern, you might want to keep them separate, otherwise it's probably conceptually easier to keep them together as a `color`, and pull it apart when needed.

5.3 HSB Colors

The red-green-blue color specification is used throughout computer graphics, but it's notoriously difficult to get a good intuitive feeling for. If you look at Figure 5.4, it might not be obvious how to maneuver through this cube to find a particular color. If you want to take a shot at mixing up some colors in RGB, go to the Processing window and under the *Tools* menu choose *Color Selector*. You can mix RGB values there using the color design tool. But if you have a particular color in mind, say a dark brown or a light lime-green, it can be frustrating to try to find the right RGB values to get there.

Computer graphics researchers and vision scientists have invented a variety of more useful ways to describe colors. Processing offers you one of the more popular variations, called the *hue-saturation-brightness* (or simply HSB) model. A similar (but different) way of describing colors that's available in some other graphics program is called the *hue-saturation-lightness*, or HSL model; be sure that you don't accidentally use HSL values in your Processing program or things won't look right! The Color Selector also shows you the HSB values for your colors, as in Figure 5.6.

Colors described with HSB use three values that normally run from 0 to 255, just like RGB, but of course the numbers mean different things. The built-in color selector also shows the HSB values of each color you point to.

The idea behind HSB is that all the colors are arranged in a cylinder, as in Figure 5.7. Black is at the middle of the bottom face, and white is directly above it. Between them,

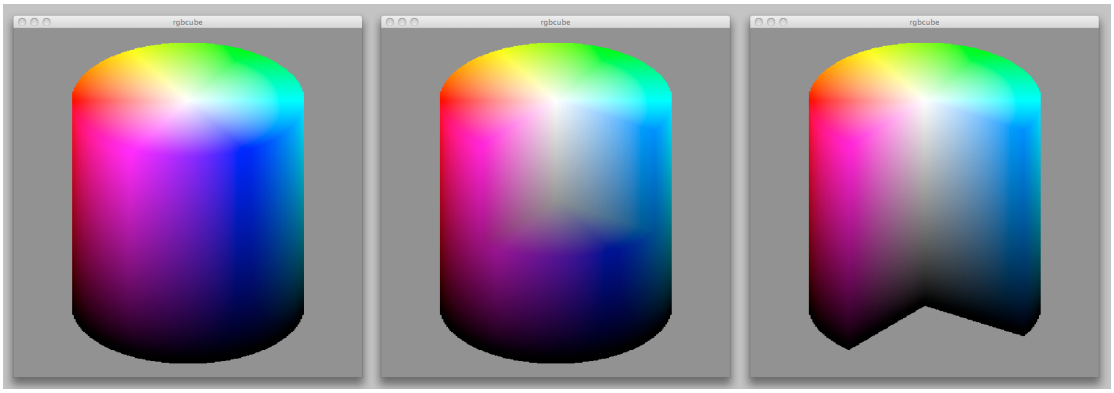

Figure 5.7: The HSB Color Cylinder. [hsbcylinder.png]

along the axis of the cylinder, are the different values of gray. We say that *brightness* has a value of 0 at the bottom of the cylinder, and 255 at the top.

In part a of Figure 5.7 I've shown the whole cylinder. Notice that all the colors at the bottom, where lightness is 0, are black. In part b I've removed a chunk from the upper half, revealing the insides. And in part c I've extended that removal so you can see what's going on all the way up the cylinder. The black-to-white axis lies in the middle of the cylinder, and colors get more saturated as we move to the edges.

You can move away from the central core grays with the other two values. Suppose you want to create a red-like color. Then you choose a value for *hue* that directs you towards red (in Processing, that's 0). Increasing values of hue run around the cylinder clockwise in this figure, from the reds into the yellows, the greens, and the blues. The distance you move out from the core towards the edge of the cylinder is given by *saturation*, which is 0 at the core and 255 out at the edge.

This is a much easier way to design a color, particularly when using an interactive tool. If you want a light lime-green, for instance, you'd pick a hue value toward green, a pretty large value of brightness (since you want a light color) and then you'd push outwards with saturation to move from grays to greens.

By the way, you might be wondering why the RGB color space is a cube and the HSB space is a cylinder. Why not the other way around? Or maybe they should both be cones, or spheres? These shapes are partly by convention, but they also make sense. The three RGB axes are independent and equal, so it makes sense that they should all have the same function in the space: three straight lines forming the edges of a cube does that nicely. In HSB, we have the idea that hue "wraps around" the spectrum, as you can see from Figure 5.7. So brightness and value are both straight lines, but it makes sense to give hue a geometric shape that matches its cyclic nature. A circle does that job, so hue gets mapped onto a circle and the result is a cylinder.

If you're intrigued, try recreating these figures with RGB on the cylinder and HSB on a cube, and see if that looks any more convenient to you. Or try some other 3D shapes that occur to you and see if they better match your intuition for organizing

colors.

When you create a color with `color()`, Processing normally assumes the three values you're handing it are in RGB, each in the range 0-255. You can tell it you'd rather it interpreted your colors in HSB by using the built-in function `colorMode()`. In its basic form, you call this function with just one argument: either `RGB` or `HSB` (remember to use all capitals). Here's two ways to create the same blue color:

```
colorMode(RGB);                    // default mode
color c1 = color(102, 164, 180);   // set RGB values
colorMode(HSB);                    // now use HSB
color c2 = color(136, 110, 180);   // set HSB values
```

Internally, both `c1` and `c2` represent the exact same color; Processing doesn't even remember which mode you used to make the color.

Just as you can extract the RGB values of a color with `red()`, `green()`, and `blue()`, you won't be surprised to learn you can get the HSB values with `hue()`, `saturation()`, and `brightness()`. Since Processing doesn't care how you created the color, you can get the RGB or HSB values of any color at any time, no matter how it was originally defined.

We'll see that Processing has a variety of similar mode commands. They all tell the system how to interpret numbers that are given to a function, just as `colorMode()` tells `color()` whether you're giving it RGB or HSB values. My general advice will be to avoid mode commands, but they're a necessary evil when creating colors; if you want to specify a color as HSB there's just no other reasonable way to go about it. My suggestion is that you stick with the default color mode `RGB` as much as possible. If you really want to use HSB, then use `colorMode()` to go into HSB, create your colors, and then immediately use `colorMode()` to go back to RGB. That way you'll know that your program is always in the RGB mode, unless you're explicitly setting colors and just a few lines below a call to `colorMode()`. That's what I'll do from now on.

The HSB color space is a great place to work if you want to adjust your colors in your program. To make a color darker, for instance, you would just reduce its brightness. Here's one way to do that:

```
// assume we have a color myColor
float myBrightness = brightness(myColor);
myBrightness = myBrightness - 10;
colorMode(HSB);
myColor = color(hue(myColor), saturation(myColor), myBrightness);
colorMode(RGB);
```

One very interesting quality of hue is that's defined around a circle. That is, if you increase hue from 0 to 255 you'll start from red and come back to red. So let's write a version of our background-color changing program that just rolls the hue value around and around the circle.

```
int Hueval = 0;
int Satval = 110;
int Brival = 110;
color MyColor;

void setup() {
   size(600, 400);
   colorMode(HSB);
   MyColor = color(Hueval, Satval, Brival);
   colorMode(RGB);
   background(MyColor);
}

void draw() {
   Hueval = Hueval+1;
   colorMode(HSB);
   MyColor = color(Hueval, Satval, Brival);
   colorMode(RGB);
   background(MyColor);
}
```

(Full program in color/sketches/hsb1/hsb1.pde)

Very nice. But if you let it run, you'll find it stops when it gets back to red. That's because `Hueval` grows beyond 255, so Processing simply interprets it as 255, since it knows that's its maximum value. This isn't a general feature of all languages or even all functions in Processing; it's just a special feature of `color()` that values less than 0 or greater than 255 are "clamped" to 0 and 255 respectively (you can change these maximum and minimum values with additional parameters to `colorMode()`, but there's rarely any need to).

In the previous program I stopped `Redval` from getting out of control this way:

```
Redval = Redval+1;
if (Redval > 255) Redval=0;
```

It turns out that there's a built-in shortcut to handle just this kind of situation. It's called the *modulo operator*, and it's written with a percent sign `%`. Like the plus sign, it takes two numbers, one on the left and one on the right. Let's call those `L` and `R`. In words, it does this:

> As long as `L >= R`, subtract `R` from `L`

Here, >= means "greater than or equal to."

So suppose we write `10 % 3` (when said out loud, this is pronounced "ten modulo three"). Since 10 >= 3, we subtract 3 from 10 getting 7, and since 7 >= 3, we subtract

3 again getting 4, and since 4 >= 3 we subtract 3 yet again and get 1. That's now smaller than 3 so we stop, and that's the result of `10 % 3`: the number 1. Let's try the number 9: Since 9 >= 3 we subtract 3 getting 6, and since 6 >= 3 we subtract 3 again getting 3, and (this is the interesting part) since 3 >= 3 (remember the equals sign) we subtract 3 again getting 0, and since 0 is less than 3 we stop. So the result of `9 % 3` is 0.

This is also called the *remainder* of dividing one number by another. For example, 16/3 is 5 with a remainder of 1 (that is, 3 fits into 16 a total of 5 times, with 1 left over), so `16%3` is 1.

Using `L` and `R` again, then `L%R` is what's left when you remove `R` from `L` as many times as you can before `L` becomes smaller than `R`. The modulo operator is the perfect way to make numbers "loop around"; that is, when they get too large for a given range they pop back to the beginning of the range. Suppose `Hueval` has the value 250. Then `Hueval % 256` is simply 250. When `Hueval` makes it to 254, then `Hueval%256` is just 254, and when `Hueval` is 255, then `Hueval%256` is 255. But when `Hueval` hits 256, then `256 % 256` is 0: we've popped back to the start.

Now we have two choices. We can assign 0 back into `Hueval` and start it over again from there. Or we can just keep letting `Hueval` increase forever, getting bigger and bigger, but using the result of the modulo operator when we make the color. For example, the next time around `Hueval` will be 257, so `Hueval % 256` is 1. The next time it'll be 2, and so on. Even when `Hueval` reaches 2560000, the modulo operator will still wrap it around to 0 (internally, the operator uses a different but much more efficient process to compute this result; it doesn't actually do all those subtractions, so feel free to use it with giant numbers).

The first approach could be written this way:

```
Hueval = Hueval+1;
MyColor = color(Hueval % 256, Satval, Brival);
```

Here I'm using the modulo operator when I make the color. The second approach could be written this way:

```
Hueval = (Hueval+1) % 256;
MyColor = color(Hueval, Satval, Brival);
```

They both produce the same results, but I think the second version is much better. It keeps the value of `Hueval` in a reasonable range, so if we examine it (say while debugging) we don't have to figure out the modulo step in our heads.

Here's a full listing of the new program, using the second approach:

```
int Hueval = 0;
int Satval = 110;
int Brival = 110;
color MyColor;
```

```
void setup() {
   size(600, 400);
   colorMode(HSB);
   MyColor = color(Hueval, Satval, Brival);
   colorMode(RGB);
   background(MyColor);
}

void draw() {
   Hueval = (Hueval+1) % 256;
   colorMode(HSB);
   MyColor = color(Hueval, Satval, Brival);
   colorMode(RGB);
   background(MyColor);
}
```

(Full program in color/sketches/hsb2/hsb2.pde)

If you use Processing's built-in color tool (under *Tools* and then *Color Selector*) you'll find that the H, S, and B fields are labeled with a degree marker, and two percent signs. Traditionally, the hue is written as an angle from 0 to 360 degrees, and saturation and brightness are written as percentages from 0 to 100. Unfortunately, even in HSB mode Processing expects the three numbers you give it to be in the range 0 to 255. That means you can't just copy the HSB numbers into your Processing code, the way you can with the RGB numbers.

Instead, you have to convert each one. Multiply the hue by 255.0/360.0, and saturation and brightness by 255.0/100.0 (of course, you can pre-compute these numbers to save a little bit of computation time). If you want to copy the numbers from the Color Selector and paste them right into your code (let's call them simply `H`, `S` and `B`), then plug them into this template:

```
MyColor = color(H * 0.71, S * 25.5, B * 25.5);
```

Although you have to multiply the numbers as above to use them, at least Processing's built-in Color Selector gives you the values in HSB. Many other color tools use one of alternative color definitions, which can be a challenge to convert into HSB. If you want to convert RGB values to HSB, here's a little program that does the job. Enter the RGB values at the top, and the HSB values will be printed to Processing's console window.

```
void setup() {

   // replace these values with your RGB color
   float r = 102;
```

```
   float g = 164;
   float b = 180;

   colorMode(RGB);
   color clr = color(r, g, b);
   float hue = hue(clr);
   float sat = saturation(clr);
   float bri = brightness(clr);

   println("RGB ("+r+", "+g+", "+b+") = HSB ("+hue+", "+sat+", "+bri+")");
}
```

(Full program in color/sketches/hsb3/hsb3.pde)

When I run this with the values given above, I get these results:

```
RGB (102.0, 164.0, 180.0) = HSB (136.21794, 110.5, 180.0)
```

Having all those digits of precision is nice, but it makes for a lot of typing (and possible typos) if you're using these kinds of results to manually enter colors. Except for the most precise kinds of color applications, when you're describing colors by hand you can probably ignore everything to the right of the decimal point; I would call this color simply (136, 110, 180) in HSB.

Note that if you like to use short variable names like `r`, `g`, and `b` you need to be careful when changing color models, because you might be tempted to use `b` for "brightness", and you run the risk of all kinds of confusing bugs because you lose track of whether `b` refers to "blue" or "brightness". I speak from experience here. Don't suffer as I have. Heed these words of cosmic wisdom, Grasshopper, and name your variables well.

In Chapter 7 I'll talk about the idea of *linear interpolation*, or *lerp*. It's basically a means for finding a new value between two others. For example, if you had endpoints 10 and 20, then a lerp halfway between them would be the value 15. A lerp a quarter of the way would be 12.5.

Processing provides a built-in means for doing the same thing with colors. This lets you compute colors in between two other colors, which can be handy for tasks like blending and making smooth gradients. The function `lerpColor()` takes two colors and a number between 0 and 1, and it produces a color at the specified point along the range between the two inputs.

Note that the color space in use at the moment makes a very big difference in how colors get interpolated by `lerpColor()`. Here's a little test program to show the difference:

```
void setup() {
   color c0 = color(199, 172, 115 );
   color c1 = color( 46, 106, 148 );
   size(800, 450);
```

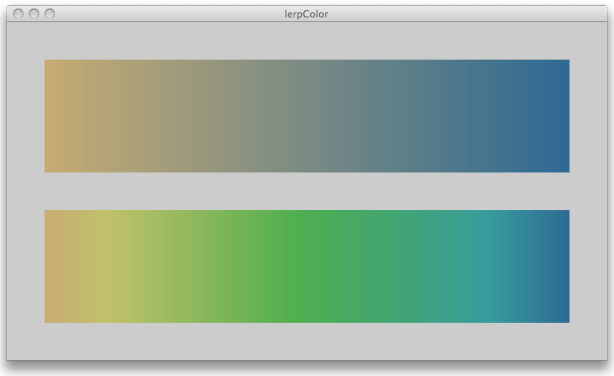

Figure 5.8: Blending the same colors. Above: using the RGB color space. Below: the HSB color space. (*color/sketches/lerpColors/lerpColors.pde*) [lerpColors.png]

```
  noStroke();
  int numSteps = 700;
  for (int i=0; i<numSteps; i++) {
    float a = i/(numSteps-1.0);

    // RGB upper
    colorMode(RGB);
    fill(lerpColor(c0, c1, a));
    rect(50+i, 50, 1, 150);

    // HSB lower
    colorMode(HSB);
    fill(lerpColor(c0, c1, a));
    rect(50+i, 250, 1, 150);
    colorMode(RGB);
  }
}
```

(see output in Figure 5.8. Full program in color/sketches/lerpColors/lerpColors.pde)

The upper row in Figure 5.8 shows the result of blending in RGB space. In essence we draw a straight line in the RGB cube between the two points, and then pull out colors along that line.

The lower row shows the result of blending the exact same colors, only in the HSB space. The model here is that we're making a kind of spiral in the HSB cylinder, curving around in hue while moving up or down in value and in or out in brightness. The yellows and greens in the lower row are a result of all those hues we pass through on our way from the beige to the blue.

It's clear that these are very different results! Neither one is "right" or "wrong"; which one works for you best depends on how you want your work to look.

When you blend two bright colors in RGB it's not uncommon for the intermediate colors to become gray or muddied, since the closer you get to the center of the cube the more gray your colors become. When you blend two bright colors in HSB, they'll typically stay bright throughout, since the brightness is blended separately from the hue and saturation. While you're getting used to working with colors on the computer, it's often useful to see the results you get from blending in both color spaces, and then choosing the one that looks best to you. Just remember that if you do switch the color mode to HSB, switch it back to RGB when you're done!