Chapter 3
In this chapter:
– The flow of a computer program
– The meaning behind setup() and draw()
– Mouse interaction
– Your first dynamic Processing sketch
– Handling events, such as mouse clicks and key presses

3-1 Go with the flow
If you have ever played a computer game, interacted with a digital art installation, or watched a screensaver at three in the morning, you have probably given very little thought to the fact that the software that runs these experiences happens over a period of time. The game starts, you find the secret treasure hidden in magical rainbow land, defeat the the scary monster who-zee-ma-whats-it, achieve a high score, and the game ends.

What I want to focus on in this chapter is that flow over time. A game begins with a set of initial conditions: you name your character, you start with a score of zero, and you start on level one. Let’s think of this part as the program’s SETUP. After these conditions are initialized, you begin to play the game. At every instant, the computer checks what you are doing with the mouse, calculates all the appropriate behaviors for the game characters, and updates the screen to render all the game graphics. This cycle of calculating and drawing happens over and over again, ideally 30 or more times per second for a smooth animation. Let’s think of this part as the program’s DRAW.

This concept is crucial to your ability to move beyond static designs (as in Chapter 2) with Processing.
1. Set starting conditions for the program one time.
2. Do something over and over and over and over (and over…) again until the program quits.

Consider how you might go about running a race.
1. Put on your sneakers and stretch. Just do this once, OK?
2. Put your right foot forward, then your left foot. Repeat this over and over as fast as you can.
3. After 26 miles, quit.

Exercise 3-1
In English, write out the flow for a simple computer game, such as Pong. If you're not familiar with Pong, visit: http://en.wikipedia.org/wiki/Pong.

3-2 Our good friends, setup() and draw()

Now that you are good and exhausted from running marathons in order to better learn programming, you can take this newfound knowledge and apply it to your first dynamic Processing sketch. Unlike Chapter 2’s static examples, this program will draw to the screen continuously (i.e., until the user quits). This is accomplished by writing two “blocks of code”: setup() and draw(). Technically speaking, setup() and draw() are functions. I will get into a longer discussion of writing your own functions in a later chapter; for now, you can understand them to be two sections where you write code.

What is a block of code?

A block of code is any code enclosed within curly brackets.

```
{  
  A block of code  
}
```

Blocks of code can be nested within each other, too.

```
{  
  A block of code  
  {  
    A block inside a block of code  
  }  
}
```
This is an important construct as it allows you to separate and manage code as individual pieces of a larger puzzle. A programming convention is to indent the lines of code within each block to make the code more readable. Processing will do this for you via the menu option Edit → Auto-Format. Getting comfortable with organizing your code into blocks while more complex logic will prove crucial in future chapters. For now, you only need to look at two simple blocks: `setup()` and `draw()`.

Let’s look at what will surely be strange-looking syntax for `setup()` and `draw()`. See Figure 3-1.

```
void setup() {
  // Initialization code goes here.
}

void draw() {
  // Code that runs forever goes here.
}
```

Admittedly, there is a lot of stuff in Figure 3-1 that you are not entirely ready to learn about. I have covered that the curly brackets indicate the beginning and end of a block of code, but why are there parentheses after “setup” and “draw”? Oh, and, my goodness, what is this “void” all about? It makes me feel sad inside! For now, you have to decide to feel comfortable with not knowing everything all at once, and that these important pieces of syntax will start to make sense in future chapters as more concepts are revealed.

For now, the key is to focus on how Figure 3-1’s structures control the flow of a program. This is shown in Figure 3-2.
How does it work? When you run the program, it will follow the instructions precisely, executing the steps in `setup()` first, and then move on to the steps in `draw()`. The order ends up being something like: 1a, 1b, 1c, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b…

Now, I can rewrite the Zoog example as a dynamic sketch. See Example 3-1.

**Example 3-1**

**Zoog as dynamic sketch**
Take the code from Example 3-1 and run it in Processing. Strange, right? You will notice that nothing in the window changes. This looks identical to a static sketch! What is going on? All this discussion for nothing?
Well, if you examine the code, you will notice that nothing in the draw() function varies. Each time through the loop, the program cycles through the code and executes the identical instructions. So, yes, the program is running over time redrawing the window, but it looks static since it draws the same thing each time!

Exercise 3-2: Redo the drawing you created at the end of Chapter 2 as a dynamic program. Even though it will look the same, feel good about your accomplishment!

3-3 Variation with the mouse
Consider this: What if, instead of typing a number into one of the drawing functions, you could type “the mouse’s x location” or “the mouse’s y location.”

```java
line(the mouse’s x location, the mouse’s y location, 100, 100);
```

In fact, you can, only instead of the more descriptive language, you must use the keywords `mouseX` and `mouseY`, indicating the horizontal or vertical position of the mouse cursor.

**Example 3-2**

`mouseX and mouseY`
void setup() {
    size(200, 200);
}

void draw() {
    background(255);
    // Body
    stroke(0);
    fill(175);
    rectMode(CENTER);
    rect(mouseX, mouseY, 50, 50);
}

Try moving `background()` to `setup()` and see the difference! (Exercise 3-3)

`mouseX` is a keyword that the sketch replaces with the horizontal position of the mouse. `mouseY` is a keyword that the sketch replaces with the vertical position of the mouse.

Exercise 3-3

Explain why you see a trail of rectangles if you move `background()` to `setup()`, leaving it out of `draw()`.
If you are following the logic of `setup()` and `draw()` closely, you might arrive at an interesting question: *When does Processing actually display the shapes in the window? When do the new pixels appear?*

On first glance, one might assume the display is updated for every line of code that includes a drawing function. If this were the case, however, you would see the shapes appear onscreen one at a time. This would happen so fast that you would hardly notice each shape appearing individually.

However, when the window is erased every time `background()` is called, a somewhat unfortunate and unpleasant result would occur: flicker. Processing solves this problem by updating the window only at the end of every cycle through `draw()`. It’s as if there were an invisible line of code that renders the window at the end of the `draw()` function.

```java
void draw() {
```
// All of your code
// Update Display Window - invisible line of code you
don’t see
}

This process is known as *double-buffering* and, in a lower-level environment, you may find that you have to implement it yourself. Again, I’d like to take a moment to thank Processing for making our introduction to programming friendlier and simpler by taking care of this for you and me. It should also be noted that any colors you have set with `stroke()` or `fill()` carry over from one cycle through `draw()` to the next.

I could push this idea a bit further and create an example where a more complex pattern (multiple shapes and colors) is controlled by `mouseX` and `mouseY` position. For example, I can rewrite Zoog to follow the mouse. Note that the center of Zoog’s body is located at the exact location of the mouse (`mouseX, mouseY`), however, other parts of Zoog’s body are drawn relative to the mouse. Zoog’s head, for example, is located at `(mouseX, mouseY-30)`. The following example only moves Zoog’s body and head, as shown in Figure 3-5.
Example 3-3

Zoog as dynamic sketch with variation
void setup() {
  size(200, 200); // Set the size of the window
}

void draw() {
  background(255); // Draw a white background

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

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(mouseX, mouseY, 20, 160);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(mouseX, mouseY-30, 60, 60);

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

  // Draw Zoog's legs
  stroke(0);
  line(90, 150, 80, 160);
  line(116, 150, 120, 160);
}

---

**Exercise 3-4**

**Complete Zoog so that the rest of its body moves with the mouse.**

// Draw Zoog's eyes
fill(0);
ellipse(______,______, 16, 32);
ellipse(______,______, 16, 32);
// Draw Zoog's legs
stroke(0);
line(________,________,________,________);
line(________,________,________,________);

Exercise 3-5: Recode your design so that shapes respond to the mouse (by varying color and location).
In addition to mouseX and mouseY, you can also use pmouseX and pmouseY. These two keywords stand for the previous mouseX and mouseY locations, that is, where the mouse was the last time the sketch cycled through draw(). This allows for some interesting interaction possibilities. For example, let’s consider what happens if you draw a line from the previous mouse location to the current mouse location, as illustrated in the diagram in Figure 3-6.

Exercise 3-6: Fill in the blank in Figure 3-6.
By connecting the previous mouse location to the current mouse location with a line each time through `draw()`, I am able to render a continuous line that follows the mouse. See Figure 3-7.
Exercise 3-7

Update Exercise 3-4 on page 40 so that the faster the user moves the mouse, the wider the drawn line. Hint: look up `strokeWeight()` in the Processing reference ([https://processing.org/reference/strokeWeight_html](https://processing.org/reference/strokeWeight_html)).

The formula for calculating the speed of the mouse’s horizontal motion is the absolute value of the difference between `mouseX` and `pmouseX`. The absolute value of a number is defined as that number without its sign:
• The absolute value of -2 is 2.
• The absolute value of 2 is 2.

In Processing, you can get the absolute value of the number by placing it inside the `abs()` function, that is `abs(-5)` equals 5. The speed at which the mouse is moving is therefore:

```java
float mouseSpeed = abs(mouseX - pmouseX);
```

Fill in the blank below and then try it out in Processing!

```java
stroke(0);
_____________________________(______________);
line(pmouseX, pmouseY, mouseX, mouseY);
```

### 3-4 Mouse clicks and key presses

You are well on your way to creating dynamic, interactive Processing sketches through the use of the `setup()` and `draw()` framework and
the `mouseX` and `mouseY` keywords. A crucial form of interaction, however, is missing — clicking the mouse!

In order to learn how to have something happen when the mouse is clicked, you need to return to the flow of the program. You know `setup()` happens once and `draw()` loops forever. When does a mouse click occur? Mouse presses (and key presses) are considered *events* in Processing. If you want something to happen (such as “the background color changes to red”) when the mouse is clicked, you need to add a third block of code to handle this event. This event “function” will tell the program what code to execute when an event occurs. As with `setup()`, the code will occur once and only once. That is, once and only once for each occurrence of the event. An event, such as a mouse click, can happen multiple times of course!

These are the two new functions you need:

- `mousePressed()` — Handles mouse clicks.
- `keyPressed()` — Handles key presses.

The following example uses both event functions, adding squares whenever the mouse is pressed and clearing the background whenever a key is pressed.

**Example 3-5**

```java
mousePressed() and keyPressed()
```
In Exercise 3-5 on page 40, I have four functions that describe the program’s flow. The program starts in `setup()` where the size and background are initialized. It continues into `draw()`, looping endlessly. Since `draw()` contains no code, the window will remain blank. However, I have added two new functions: `mousePressed()` and `keyPressed()`. The code inside these functions sits and waits. When the user clicks the mouse (or presses a key), it springs into action, executing the enclosed block of instructions once and only once.

**Exercise 3-8:**

Add `background(255);` to the `draw()` function. Why does the program stop working?

I am now ready to bring all of these elements together for Zoog.
• Zoog’s entire body will follow the mouse.
• Zoog’s eye color will be determined by mouse location.
• Zoog’s legs will be drawn from the previous mouse location to the current mouse location.
• When the mouse is clicked, a message will be displayed in the message window: “Take me to your leader!"

Note the addition in Exercise 3-6 on page 41 of the function `frameRate()`, which requires a value of at least one, enforces the speed at which Processing will cycle through `draw()`. `frameRate(30)`, for example, means 30 frames per second, a traditional speed for computer animation. If you do not include `frameRate()`, Processing will attempt to run the sketch at 60 frames per second. Since computers run at different speeds, `frameRate()` is used to make sure that your sketch is consistent across multiple computers. This frame rate is just a maximum, however. If your sketch has to draw one million rectangles, it may take a long time to finish the draw cycle and run at a slower speed.

**Example 3-6**

*Interactive Zoog*
Lesson One Project

(You may have completed much of this project already via the exercises in Chapter 1–Chapter 3. This project brings all of the elements together. You could either start from scratch with a new design or use elements from the exercises.)

1. Design a static screen drawing using RGB color and primitive shapes.
2. Make the static screen drawing dynamic by having it interact with the mouse. This might include shapes following the mouse, changing their size according to the mouse, changing their color according to the mouse, and so on.

```java
void setup() {
// Set the size of the window
size(200, 200);
frameRate(30);
}

void draw() {
// Draw a white background
background(255);

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

// Draw Zoog's body
stroke(0);
fill(175);
rect(mouseX, mouseY, 20, 100);

// Draw Zoog's head
stroke(0);
fill(255);
ellipse(mouseX, mouseY-30, 60, 60);

// Draw Zoog's eyes
fill(mouseX, 0, mouseY);
ellipse(mouseX-19, mouseY-30, 16, 32);
ellipse(mouseX+19, mouseY-30, 16, 32);

// Draw Zoog's legs
stroke(0);
line(mouseX-10, mouseY+50, pmouseX-10, pmouseY + 60);
line(mouseX+10, mouseY+50, pmouseX+10, pmouseY + 60);
}

void mousePressed() {
println("Take me to your leader!");
}
```
Chapter 4
In this chapter:
- Variables: What are they?
- Declaring and initializing variables
- Common uses for variables
- Variables you get “for free” in Processing (a.k.a. built-in variables)
- Using random values for variables

4-1 What is a variable?
I admit it. When I teach programming, I launch into a diatribe of analogies in an attempt to explain the concept of a variable in an intuitive manner. On any given day, I might say, “A variable is like a bucket.” You put something in the bucket, carry it around with you, and retrieve it whenever you feel inspired. “A variable is like a storage locker.” Deposit some information in the locker where it can live safely, readily available at a moment’s notice. “A variable is a lovely, yellow post-it note, on which is written the message: I am a variable. Write your information on me.”

I could go on. But I won’t. I think you get the idea. And I’m not entirely sure you really need an analogy since the concept itself is rather simple. Here’s the deal. The computer has memory. Why is it called memory? Because it’s what the computer uses to remember stuff it needs.
Technically speaking, a *variable* is a named pointer to a location in the computer’s memory (“memory address”) where data is stored. Since computers only process information one instruction at a time, a variable allows a programmer to save information from one point in the program and refer back to it at a later time. For a Processing programmer, this is incredibly useful; variables can keep track of information related to shapes: color, size, location. Variables are exactly what you need to make a triangle change from blue to purple, a circle fly across the screen, and a rectangle shrink into oblivion.

Out of all the available analogies, I tend to prefer the *piece of paper* approach: *graph paper*.

Imagine that the computer’s memory is a sheet of graph paper and each cell on the graph paper has an address. With pixels, I discussed how to refer to those cells by column and row numbers. Wouldn’t it be nice if you could name those cells in memory? With variables, you can.

Let’s name one “Jane’s Score” (you’ll see why I am calling it that in the next section) and give it the value 100. That way, whenever you want to use Jane’s score in a program, you don’t have to remember the value 100. It’s there in memory and you can ask for it by name. See Figure 4-2.

![Figure 4-2](image-url)
The power of a variable does not simply rest with the ability to remember a value. The whole point of a variable is that those values vary, and more interesting situations arise as you periodically alter that value.

Consider a game of Scrabble between Sasha and Malia. To keep track of the score, Sasha takes out paper and pencil, and scrawls down two column names: “Sasha’s Score” and “Malia’s Score.” As the two play, a running tally is kept of each player’s points below the headings. If you imagine this game to be virtual Scrabble programmed on a computer, you suddenly can see the concept emerge of a variable that varies. That piece of paper is the computer’s memory and on that paper, information is written — “Sasha’s Score” and “Malia’s Score” are variables, locations in memory where each player’s total points are stored and that change over time. See Figure 4-3.

<table>
<thead>
<tr>
<th>Sasha’s score</th>
<th>Malia’s score</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>53</td>
<td>44</td>
</tr>
<tr>
<td>66</td>
<td>68</td>
</tr>
<tr>
<td>87</td>
<td>91</td>
</tr>
<tr>
<td>101</td>
<td>98</td>
</tr>
</tbody>
</table>

FIGURE 4-3

In the Scrabble example, the variable has two elements — a name (e.g., “Sasha’s Score”) and a value (e.g., 101). In Processing, variables can hold different kinds of
values and you are required to explicitly define the type of value before you can use a given variable.

**Exercise 4-1:** Consider the game Pong. What variables would you need to program the game? (If you are not familiar with Pong, see [http://en.wikipedia.org/wiki/Pong](http://en.wikipedia.org/wiki/Pong)).

### 4-2 Variable declaration and initialization

Variables can hold primitive values or references to objects and arrays. For now, I’m just going to worry about primitives — I’ll get to objects and arrays in a later chapter. Primitive values are the building blocks of data on the computer and typically involve a singular piece of information, like a number or character. Variables are declared by first stating the type, followed by the name. Variable names must be one word (no spaces) and must start with a letter (they can include numbers, but cannot start with a number). They cannot include any punctuation or special characters, with the exception of the underscore: “_”

A type is the kind of data stored in that variable. This could be a whole number, a decimal number, or a character. Here are data types you will commonly use:

- **Whole numbers**, such as 0, 1, 2, 3, −1, −2, and so on are stored as integers and the type keyword for integer is int.
- **Decimal numbers**, such as 3.14159, 2.5, and −9.95 are typically stored as floating point values and the type keyword for floating point is float.
- **Characters**, such as the letters “a,” “b,” “c,” and so on are stored in variables of type char and are declared as a letter enclosed in single quotes, that is, ‘a’.

Characters are useful when determining what letter on the keyboard has been pressed, and for other uses involving strings of text (see Chapter 17).

In Figure 4-4, I have a variable named count of type int, which stands for integer. Other possible data types are listed below.
Don’t forget

- Variables must have a type. Why? This is how the computer knows exactly how much memory should be allocated to store that variable’s data.
- Variables must have a name.

All primitive types

- boolean: true or false
- char: a character, “a,” “b,” “c,” etc.
- byte: a small number, −128 to 127
- short: a larger number, −32,768 to 32,767
- int: a big number, −2,147,483,648 to 2,147,483,647
- long: a ridiculously huge number
- float: a decimal number, such as 3.14159
- double: a decimal number with a lot more decimal places (only necessary for advanced programs requiring mathematical precision).

Once a variable is declared, I can then assign it a value by setting it equal to something. In most cases, if you forget to initialize a variable, Processing will give it a default value, such as 0 for integers, 0.0 for floating points, and so on. However, it’s good to get into the habit of always initializing variables in order to avoid confusion.

```java
int count;
count = 50;
```

Declare and initialize a variable in two lines of code.

To be more concise, I can combine the above two statements into one.
What’s in a name?

Tips for choosing good variable names

• Avoid using words that appear elsewhere in the Processing language. In other words, do not call your variable `mouseX`, since there already is one!

• Use names that mean something. This may seem obvious, but it’s an important point. For example, if you are using a variable to keep track of score, call it “score” and not, say, “cat.”

• Start your variable with a lowercase letter and join together words with capitals. Words that start with capitals are reserved for classes (Chapter 8). For example: “frogColor” is good, “Frogcolor” is not. this canTake some gettingUsedTo but it will comeNaturally soonEnough.

A variable can also be initialized by another variable (x equals y), or by evaluating a mathematical expression (x equals y plus z, etc.). Here are some examples:

Example 4-1

Variable declaration and initialization examples

```java
int count = 0; // Declare an int named count, assigned the value 0
char letter = 'a'; // Declare a char named letter, assigned the value 'a'
double d = 132.32; // Declare a double named d, assigned the value 132.32
boolean happy = false; // Declare a boolean named happy, assigned the value false
float x = 4.0; // Declare a float named x, assigned the value 4.0
float y; // Declare a float named y (no assignment)
y = x + 5.2; // Assign the value of x plus 5.2 to the // previously declared y
float z = x * y + 15.0; // Declare a variable named z, assign it the value which // is x times y plus 15.0.
```
Exercise 4-2

Write out variable declaration and initialization for the game Pong.

4-3 Using a variable

Though it may initially seem more complicated to have words standing in for numbers, variables make our lives easier and more interesting. Let’s take a simple example of a program that draws a circle onscreen.

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

void draw() {
  background(255);
  stroke(0);
  fill(175);
  ellipse(100, 100, 50, 50);
}
```

In Chapter 3, you learned how to take this simple example one step further, changing the location of a shape to `mouseX, mouseY` in order to assign its location according to the mouse.
ellipse(mouseX, mouseY, 50, 50);

Can you see where this is going? `mouseX` and `mouseY` are named references to the horizontal and vertical location of the mouse. They are variables! However, because they are built into the Processing environment (note how they are colored red when you type them in your code), they can be used without being declared. Built-in variables (a.k.a. *system variables*) are discussed further in the next section.

What I want to do now is create my own variables by following the syntax for declaring and initializing outlined above, placing the variables at the top of my code. You can declare variables elsewhere in your code and I will get into this later. For now to avoid any confusion, all variables should be at the top.

**Rule of thumb: when to use a variable**

There are no hard and fast rules in terms of when to use a variable. However, if you find yourself hard-coding in a bunch of numbers as you program, take a few minutes, review your code, and change these values to variables.

Some programmers say that if a number appears three or more times, it should be a variable. Personally, I would say if a number appears once, use a variable. Always use variables!

**Example 4-2**

**Using variables**

```java
int circleX = 100;
int circleY = 100;

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

void draw() {
  background(255);
  stroke(0);
  fill(175);
  ellipse(circleX, circleY, 50, 50);
}
```

Running this code, you’ll see the same result as in the first example: a circle appears in the middle of the screen. Nevertheless, you should open your heart and remind yourself that a variable is not simply a placeholder for one constant value. It’s called a variable because it *varies*. To change its value, you write an *assignment operation*, which assigns a new value.

Up until now, every single line of code I wrote called a function: `line()`, `ellipse()`, `stroke()`, etc. Variables introduce assignment operations to the mix. Here is what one looks like (it’s the same as how you initialize a variable, only the variable does not need to be declared).
A common example is incrementation. In the above code, `circleX` starts with a value of 100. If you want to increment `circleX` by one, you say `circleX` equals itself plus 1. In code, this amounts to:

```
circleX = circleX + 1;
```

Let’s try adding that to the sketch (and let’s start `circleX` with the value of 0).

### Example 4-3

#### Varying variables

```java
int circleX = 0;
int circleY = 100;

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

void draw() {
  background(255);
  stroke(0);
  fill(175);
  ellipse(circleX, circleY, 50, 50);
  circleX = circleX + 1;
}
```

What happens? If you run Example 4-3 in Processing, you will notice that the circle moves from left to right. Remember, `draw()` loops over and over again, all the while retaining the value of `circleX` in memory. Let’s pretend I am the computer for a moment. (This may seem overly simple and obvious, but it’s key to your understanding of the principles of programming motion.)

### Exercise 4-3

Change Example 4-3 so that instead of the circle moving from left to right, the circle grows in size. What would you change to have the circle follow the mouse as it grows? How could you vary the speed at which the circle grows?
int circleSize = 0;
int circleX = 100;
int circleY = 100;
void setup() {
  size(200, 200);
}
void draw() {
  background(0);
  stroke(255);
  fill(175);
}

1. Remember circleX = 0 and circleY = 100
2. Run setup(). Open a window 200 x 200
3. Run draw()
   • Draw circle at (circleX, circleY) → (0, 100)
   • Add one to circleX = 0 + 1 = 1

4. Run draw()
   • Draw circle at (1, 100)
   • Add one to circleX = 1 + 1 = 2

5. Run draw()
   • Draw circle at (2, 100)
   • Add one to circleX = 2 + 1 = 3

6. And so on and so forth!

FIGURE 4-5
Practicing how to follow the code step-by-step will lead you to the questions you need to ask before writing your own sketches. *Be one with the computer.*

- What data I need to remember for the sketch?
- How do I use that data to draw shapes on the screen?
- How do I alter that data to make my sketch interactive and animated?

**4-4 Many variables**

Let’s take the example one step further and use variables for every piece of information I can think of. I will also use floating point values to demonstrate greater precision in adjusting variable values.

**Example 4-4**

*Many variables*
float circleX = 0;
float circleY = 0;
float circleW = 50;
float circleH = 100;
float circleStroke = 255;
float circleFill = 0;
float backgroundColor = 255;
float change = 0.5;

// Your basic setup
void setup() {
  size(200, 200);
}

void draw() {
  // Draw the background and the ellipse
  background(backgroundColor);
  stroke(circleStroke);
  fill(circleFill);
  ellipse(circleX, circleY, circleW, circleH);

  // Change the values of all variables
  circleX = circleX + change;
  circleY = circleY + change;
  circleW = circleW + change;
  circleH = circleH - change;
  circleStroke = circleStroke - change;
  circleFill = circleFill + change;
  
  // We've got eight variables now! All of type float.
  // Variables are used for everything: background, stroke, fill, location, and size.
  // The variable change is used to increment and decrement the other variables.

Exercise 4-4

Recreate the images below.

• Step 1: Write code that draws the following screenshots with hard-coded values. (Feel free to use colors instead of grayscale.)
• Step 2: Replace all of the hard-coded numbers with variables.
• Step 3: Write assignment operations in draw() that change the value of the variables. For example variable1 = variable1 + 2;. Try different expressions and see what happens!
4-5 System variables

As you saw with `mouseX` and `mouseY`, Processing includes built-in variables for commonly needed pieces of data associated with all sketches (such as the `width` of the window, the `key` pressed on the keyboard, and more). When naming your own variables, it’s best to avoid these built-in variable names, however, if you inadvertently use one, your variable will become primary and override the system one. Here is a list of commonly used built-in variables (there are more, which you can find in the Processing reference).

- **width** — Width (in pixels) of sketch window.
- **height** — Height (in pixels) of sketch window.
- **frameCount** — Number of frames processed.
- **frameRate** — Rate that frames are processed (per second).
- **displayWidth** — Width (in pixels) of entire screen.
- **displayHeight** — Height (in pixels) of entire screen.
- **key** — Most recent key pressed on the keyboard.
- **keyCode** — Numeric code for key pressed on keyboard.
- **keyPressed** — True or false? Is a key pressed?
- **mousePressed** — True or false? Is the mouse pressed?
- **mouseButton** — Which button is pressed? Left, right, or center?

Following is an example that makes use of several of the above variables; I’m not ready to use them all yet, as I’ll need to cover some more concepts first.

**Example 4-5**

*Using system variables*
Using **width** and **height**, recreate the following screenshot. Here’s the catch: the shapes must resize themselves relative to the window size. (In other words, no matter what you specify for `size()`, the result should look identical.)

![Shape Diagram]

4-6 Random: variety is the spice of life

So, you may have noticed that the examples in this book so far are a bit, say, humdrum. A circle here. A square here. A grayish color. Another grayish color.
There is a method to the madness (or lack of madness in this case). It all goes back to the driving principle behind this book: *incremental development*. It’s much easier to learn the fundamentals by looking at the individual pieces, programs that do one and only one thing. You can then begin to add functionality on top, step by step.

Nevertheless, you have waited patiently through four chapters and I have arrived at the time where you can begin to have a bit of fun. And this fun will be demonstrated via the use of the function `random()`. Consider, for a moment, Example 4-6, whose output is shown in Figure 4-7.

**Example 4-6**

**Ellipse with variables**

```cpp
float r = 100;
float g = 150;
float b = 200;
float a = 200;

float diam = 20;
float x = 100;
float y = 100;

void setup() {
    size(200, 200);
    background(255);
}

void draw() {
    // Use those variables to draw an ellipse
    noStroke();
    fill(r, g, b, a);
    ellipse(x, y, diam, diam);
}
```

*Declare and initialize your variables*

*Use those variables! (Remember, the fourth argument for a color is transparency).*
There it is, my dreary circle. Sure, I could adjust variable values and move the circle, grow its size, change its color, and so on. However, what if every time through `draw()`, I made a new circle, one with a random size, color, and position? The `random()` function allows me to do exactly that. `random()` is a special kind of function; it’s a function that returns a value. You have encountered this before. In Exercise 3-7 on page 41 I used the function `abs()` to calculate the absolute value of a number. The idea of a function that calculates a value and returns it will be explored fully in Chapter 7, but I am going to take some time to introduce the idea now and let it sink in a bit. Unlike most of the functions you are comfortable with (e.g., `line()`, `ellipse()`, and `rect()`), `random()` does not draw or color a shape on the screen. Instead, `random()` answers a question; it returns that answer to you. Here is a bit of dialogue. Feel free to rehearse it with your friends.
Me: Hey random, what’s going on? Hope you’re well. Listen, I was wondering, could you give me a random number between 1 and 100?
Random: Like, no problem. How about the number 63?
Me: That’s awesome, really great, thank you. OK, I’m off. Gotta draw a rectangle 63 pixels wide, OK?

Now, how would this sequence look in the slightly more formal, Processing environment? The code below the part of “me” is played by the variable w.

```java
float w = random(1, 100); // A random float between 1 and 100.
rect(100, 100, w, 50);
```

The `random()` function requires two arguments and returns a random floating point number ranging from the first argument to the second. The second argument must be larger than the first for it to work properly. The function `random()` also works with one argument by assuming a range between zero and that argument. In addition, `random()` only returns floating point numbers. This is why I declared w above as a float. However, if you want a random integer, you can convert the result of the random function to an int.

```java
int w = int(random(1, 100)); // A random integer between 1 and 100.
rect(100, 100, w, 50);
```

Notice the use of nested parentheses. This is a nice concept to get used to as it can sometimes be quite convenient to call functions inside of functions.

The `random()` function returns a float, which is then passed to the `int()` function that converts it to an integer. If you wanted to go nuts nesting functions, you could even condense the above code into one line:

```java
rect(100, 100, int(random(1, 100)), 50);
```

Incidentally, the process of converting one data type to another is referred to as casting. In Java (which Processing is based on) casting a float to an int can also be written this way:

```java
int w = (int) random(1, 100); // The result of random(1, 100) is a floating point. It can be converted to an integer by “casting.”
```

OK, I am now ready to experiment with `random()`. Example 4-7 shows what happens if you take every variable associated with drawing the ellipse (fill, location, size) and assign it to a random number each cycle through `draw()`. The output is shown in Figure 4-8.

**Example 4-7**

*Filling variables with random values*
float r;
float g;
float b;
float a;
float diam;
float x;
float y;

void setup() {
    size(200, 200);
    background(255);
}

void draw() {
    r = random(255);
    g = random(255);
    b = random(255);
    a = random(255);
    diam = random(20);
    x = random(width);
    y = random(height);

    // Use values to draw an ellipse
    noStroke();
    fill(r, g, b, a);
    ellipse(x, y, diam, diam);

    Each time through draw(), random values are picked for color, size, and position of a new ellipse.
I am now ready to revisit Zoog, our alien friend, who was happily following the mouse around the screen when we last checked in. Here, I’ll add two pieces of functionality to Zoog.

- **New feature #1** — Zoog will rise from below the screen and fly off into space (above the screen).
- **New feature #2** — Zoog’s eyes will be colored randomly as Zoog moves.

Feature #1 is solved by simply taking the previous program that used `mouseX` and `mouseY` and substituting my variables in their place. Feature #2 is implemented by creating three additional variables `eyeRed`, `eyeGreen`, and `eyeBlue` that will be used for the `fill()` function before displaying the eye ellipses.
Example 4-8

Variable Zoog
Exercise 4-6

Revise Example 4-8 so that Zoog shakes left and right as Zoog moves upward. Hint: this requires the use of random() in combination with zoogX.
Examine Example 3-6 more closely, you might notice that all shapes are drawn relative to the point \((\text{zoog}X, \text{zoog}Y\)). Zoog’s body is drawn directly at \((\text{zoog}X, \text{zoog}Y\)), Zoog’s head is drawn a bit higher up at \((\text{zoog}X, \text{zoog}Y-30)\), and the eyes a little bit to the right and left of Zoog’s center.

If \text{zoog}X and \text{zoog}Y were equal to zero, where would Zoog appear? The top left of the window! You can see an example of this by removing \text{zoog}X and \text{zoog}Y from the sketch and draw Zoog relative to \((0,0)\).

(Color functions like \text{stroke}() and \text{fill}() have been removed for simplicity.)

```cpp
// Draw Zoog’s body
rect(0, 0, 20, 100);

// Draw Zoog’s head
ellipse(0, -30, 60, 60);

// Draw Zoog’s eyes
ellipse(-19, -30, 16, 32);
ellipse(19, -30, 16, 32);

// Draw Zoog’s legs
stroke(0);
line(-10, 50, -20, 60);
line(10, 50, 20, 60);
```

If you run the above code you’d see a partial Zoog in the top left as depicted in Figure 4-10. Another technique for moving Zoog (instead of adding \text{zoog}X and \text{zoog}Y to each drawing function) is to use the Processing function \text{translate}(). \text{translate}() specifies a horizontal and vertical offset for shapes in the display window. In cases such as this, it can be more convenient to set an offset via \text{translate}() rather implement the math in each
subsequent line of code. Here is an example implementation that moves Zoog relative to \texttt{mouseX} and \texttt{mouseY}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{zoog_center}
\caption{Zoog’s center is at (0,0).}
\end{figure}

\section*{Example 4-9}

\textbf{Translated Zoog}
There is a lot more to translation than what I’ve very briefly shown you here. So much so that an entire chapter later in this book (Chapter 14) is dedicated to `translate()` and other related functions known as *transformations*. Translation, for example, is required to rotate shapes in Processing as well as a key to unlocking how to draw in virtual three-dimensional space. However, I’m going to stop here since the first half of this book’s focus is the fundamentals of programming. I’ll leave more advanced topics in computer graphics for later. However, if you feel that exploring transformations now might add to your own work, it would not be so unreasonable to briefly skip to Chapter 14 and read up until Example 14-15 before returning to read Chapter 5. There the discussion starts to include topics I have not yet covered up until
Exercise 4-7: Using variables and the `random()` function, revise your design from the Lesson One Project to move around the screen, change color, size, location, and so on. Consider using `translate()` to set Zoog’s position in the window.