Chapter 4. Methods

We saw in Chapter 1 that programs can simulate real-world objects as they move or change over time by rapidly drawing a sequence of carefully designed static images. This is known as *animation*. Implementing animation requires that a program display objects, which can be done using the geometric drawing methods discussed in Chapter 2; remember key information about the objects over time, which is done using the typed variables discussed in Chapter 3; and group blocks of code for setting up and drawing the state of the objects in each static frame, which is done using methods.

We have used several of Processing's pre-defined methods in earlier chapters, but in this chapter we will see how to define our own methods and how they facilitate code reuse and procedural abstraction. We will also look at a special method in Processing that makes it easy to write programs that implement animation.

4.1. Example: Animation

Computer animation simulates how real objects move and change and is commonly used in video games and in digital movie effects. Some current movies have been completely produced using digital animation. We will begin with a simple example of animating a blue ball so that it moves around on a two-dimensional white background. As illustrated in Figure 4-1, we'd like the ball to start in the upper left corner and move to the lower right corner. This simple illustration will be our initial vision in this chapter and we will build a program that simulates the movement of this ball. Later in the chapter we will also experiment with some other animation effects.

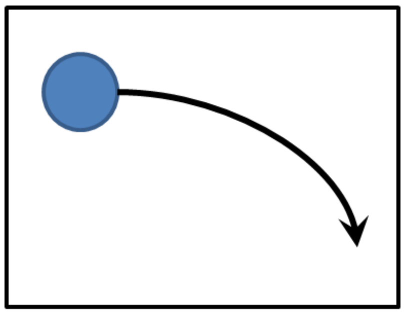
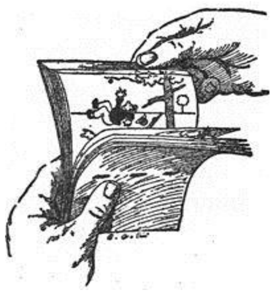


Figure 4-1. A moving ball

One of the earliest forms of animation to be developed is a flip book that contains pictures that vary gradually from one page to the next, so that when the pages are turned rapidly, the pictures appear to move or change.[1] As technology improved, photos taken by cameras replaced the hand-drawn images to create sequences of still frames that could be shown as movies. Regardless of the technology, however, *animation is always based on the same principle –displaying carefully designed sets of still images in rapid succession.*



[1] According to Wikipedia (http://en.wikipedia.org/wiki/Flip_book), the first flip book appeared in 1868 when it was patented by John Barnes Linnett who called it a *kineograph* ("moving picture").

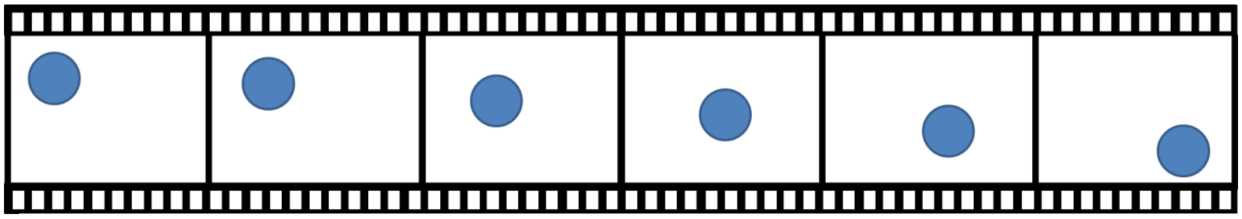


Figure 4-2. Six animation frames for a moving ball

For our ball-moving example, to animate the ball one could create a sequence of still images such as the frames shown in Figure 4-2. When shown in rapid succession, the human vision system perceives the six individual balls shown in these six frames as a single ball moving through space. To achieve this, a program must be able to do the following:

- Set up the static frame on which the animation takes place — in this case, a white background.
- Draw a sequence of individual frames each of which shows a ball in a location slightly changed from the previous frame.

In this chapter we will use methods provided by Processing to perform these two tasks and will also discuss how to make our own programmer-defined methods.

4.2. Animation Methods

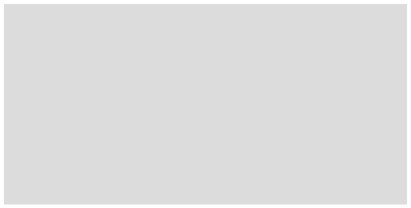
The first task in implementing our vision of an animated ball is to separate the instructions required to set up the static frame on which the animation takes place, which are to be executed only once, from the instructions that draw the individual frames of the animation, which are executed repeatedly, once for each frame. To achieve this separation, a program must create two "blocks" of code, one for set up and the other for drawing. Processing provides built-in methods for this purpose, which it calls automatically.

4.2.1. Defining Methods

Consider the task of setting up a light grey background as the static frame for our animation. In previous chapters, we would have written a sequence of statements like the following:

```
size(200, 100);
background(220);
```

In this section we will see how to block these statements together so that they are executed together, just once, at the beginning of the program execution. In Processing, this is done by writing a definition for the built-in **`setup()` method**. The following code illustrates how this is done.



```
void setup() {
  size(200, 100);
  background(220);
}
```

There are several things to be noted about this definition of the `setup()` method:

- The most familiar part is the sequence of two statements, a call to the `size()` method followed by a call to `background()`. This definition of the `setup()` method bundles these statements into a **block** using a matched pair of curly braces (`{ }`). This indicates that Processing should execute the statements together, separately from other blocks of statements.
- The pair of parentheses that follow the name tells Processing that `setup` is the name of a method rather than a variable or constant. In this text, as a reminder of this distinction, we are attaching the parentheses when using the name of a method — e.g., `setup()`.
- The word `void` indicates that this method returns nothing when it is called. Rather, such void methods perform some action — in this case, setting up a 200 × 100 output frame that has a gray background — but don't return a value that can be used in some expression.
  We have already used void methods in several of our programs. For example, in Chapter 2, we used the void method `smooth()` to render graphics in a particular way. But we have also used methods that do return values. For example, the `random()` method returns a random value of type `float` that can be used in an expression.
- The empty parameter list, `()`, indicates that this method does not expect any arguments when it is called. The `smooth()` method is another such method that has no arguments. The `random()` method, on the other hand, expects a numeric argument when it is called — e.g., `random(255)`. Methods that expect multiple arguments must use a comma-separated list of arguments — e.g., `size(100, 100)`.
- *Processing already knows about the* `setup()` *method and will automatically call it once at the beginning of program execution.* For example, if we run the program shown above, it produces the gray output frame as shown. *Note, however, that we may not include our own explicit calls to* `setup()` *in our programs.*

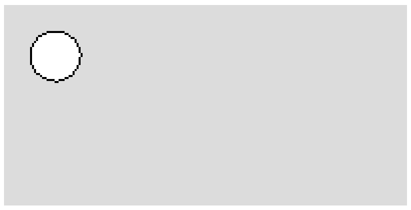
Our definition of the `setup()` method in this example is one example of the following general pattern for method definitions:

Method Definition

```
returnType methodName(parameterDeclarations) {
   statements
}
```

- *`returnType`* is the type of value returned by the method or `void` if the method does not return a value;
- *`methodName`* is the identifier that names the method;
- *`parameterDeclarations`* is a comma-separated list of parameter declarations of the form *`type`* *`identifier`* that is omitted if there are no parameters;
- *`statements`* is a set of statements that define the behavior of the method, implemented as a block of statements delineated by braces.

The second task in implementing our vision of an animated ball is to draw the individual frames of the animation in rapid sequence. For this, we can use Processing's built-in **`draw()`** method. The following program shows an example of how it is used in a program and the output produced. When this program is executed, Processing first calls `setup()`, which sets the size and color of the static background frame. *Processing does this only once*. It then begins to *call* `draw()` *repeatedly*. This results in an ellipse in the upper-left of the static background being *drawn over and over again* unless we stop it — for example, by clicking on Processing's stop button (■) or selecting Stop on the Sketch menu.



```
void setup() {
  size(200, 100);
  background(220);
}

void draw() {
  stroke(0);
  fill(255);
  ellipse(25, 25, 25, 25);
}
```

While this example does not produce a very interesting animation — the ball never moves — it does illustrate a valuable feature of methods. *The* `setup()` *and* `draw()` *methods separate the program into distinct blocks that each have their own task.* The setup block prepares the static background frame; the draw block draws each individual frame. This separation, the bundling and naming of distinct tasks, prove useful as we write programs of increasing complexity because it allows us to write separate methods for the various tasks to be performed and then combine them into a complete program.

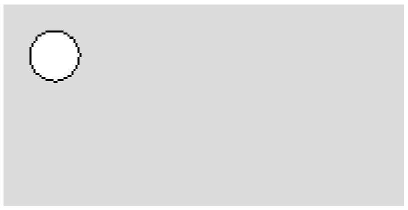
Note that when we introduce methods to our programs, we must always write our statements inside the defining block of statements for one of program's methods, either **`setup()`**, **`draw()`** or some other method. We can still declare constants and variables outside of the methods, but all other statements must be in some method. For example, in program just given, we must put the **`stroke(0)`** call in either **`setup()`** or **`draw()`**; it cannot be on a line by itself outside those methods. When Processing sees a **`setup()`** method, it calls that method to start the program and generates a syntax error if it finds any statements outside of methods.

The reason that this example program doesn't produce a very interesting animation is because it repeatedly draws an ellipse of the same size and same color in the same place. To produce the illusion of movement, the `draw()` method must vary what it draws in each successive frame. This requires that `draw()` "remember" what is has drawn before and make small changes in each successive frame.

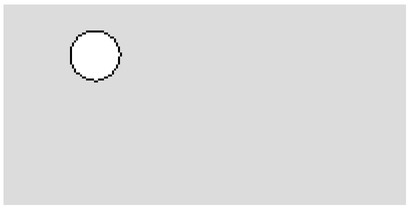
The following program remembers the location of the ball and modifies this location on each successive output frame. The output shown includes snapshots of every tenth animation frame, starting with the first.

When Processing executes the program, the animation takes somewhat less than 100 frames for a ball to appear to move completely across the output frame.[2]

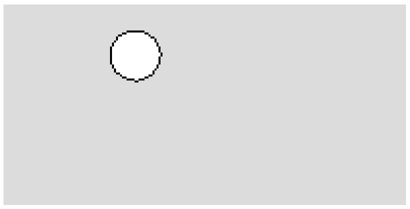
Frame 1.



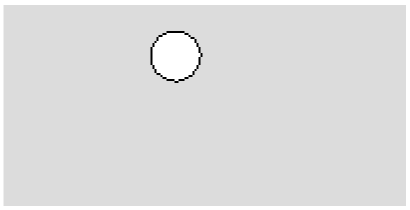
Frame 11.



Frame 21.



Frame 31.



```
int ballX, ballY;   // the location of the ball

void setup() {
  size(200, 100);

  // Start the ball in the upper left.
  ballX = 25;
  ballY = 25;
}

void draw() {
  // Erase the ball at the previous location.
  background(220);

  // Draw the ball at the current location.
  stroke(0);
  fill(255);
  ellipse(ballX, ballY, 25, 25);

  // Move the ball to the right.
  ballX += 2;
 }
```

There are two key features of this program that make the animation work:

- The program introduces two new variables, **`ballX`** and **`ballY`**, to represent the current location of the ball at each step of the animation. It declares these variables at the beginning of the program, outside of both **`setup()`** and **`draw()`**, so that both methods can use them: **`setup()`** initializes their values to 25; **`draw()`** uses them to specify the position of the ellipse representing the ball and then adds 2 to its *x* coordinate so that next time through, the ellipse will be 2 pixels to the right. Had we declared these variables inside of either of the methods, the variables would not have been available to the other method. This illustrates the concept of *scope*, which dictates the context in which an identifier names its constant or variable. This concept will be discussed in more detail below. For now, follow the practice of declaring variables and constants that must be shared by both **`setup()`** and **`draw()`** at the top of the program.

[2] The only way to appreciate the effect of these animations is to run them yourself. Source code for all of the programs presented in this text is available on the book's website (http://cs.calvin.edu/processing).

- The **draw()** method redraws the gray background each time it is called, which erases any balls that it may have drawn before. This is implemented by moving the call to **background()** from **setup()**, where it would only be called once, to **draw()**, where it will be called for each frame.

Taken together, these two features of the program create an animation that the human visual system will perceive as a single ball moving across a gray background.

By default, Processing calls the `draw()` *method at the rate of 60 frames per second.* A program can control this rate using the **frameRate()** method, which requires an integer argument that specifies the desired frame rate. Processing will achieve the specified rate provided that the computer is fast enough to support it.

4.2.2. Revisiting the Example

Our first animation program, shown in the previous section, uses the `setup()` and `draw()` methods to animate a ball flying across the output pane. In this section we extend this program by modifying both the *x* and *y* coordinates of the ball so that it moves to the right and downward in a way that imitates the effect of gravity. An algorithm that specifies this behavior is shown here. This algorithm specifies the behavior of both the `setup()` and the `draw()` methods.

```
Given:
    • The constant BALL_SIZE represents the size of the ball.
    • The constant GRAVITY represents the desired vertical acceleration.
    • width and height represent the width and height of the display window.
Algorithm (for setup()):
    1. Create a display window that is width x height screen pixels.
    2. Set the initial position of the ball to the upper left part of the display window by setting
       ballX = 15 and ballY = 15.
    3. Set deltaX = 5 and deltaY = 0.
Algorithm (for draw()):
    1. Erase the display window.
    2. Draw the ball at its current location.
    3. Change the ball's location by setting ballX = ballX + deltaX and ballY = ballY +
       deltaY.
    4. Change the ball's location by setting deltaY = deltaY + GRAVITY.
```

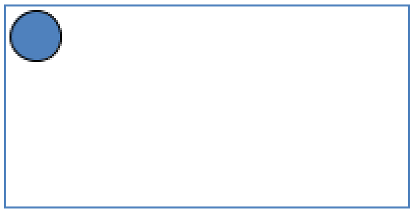
This design specifies algorithm very similar to the one used to move the ball from left to right by a constant amount for each animation frame. They key difference is that it adds steps to modify the ball's y coordinate (`ballY`) as well. `setup()` step 3 initializes a y acceleration factor (`deltaY`) and `draw()` step 3 modifies the y coordinate (`ballY`). These changes have the effect of moving the ball from top to bottom. In addition, `draw()` step 4 increases the y acceleration by a contant (`GRAVITY`). This causes the ball to drop more quickly as the animation goes on, which models the effect of gravity.

Note that this algorithm includes separate sequences for each of the key methods (`setup()` and `draw()`). We do this because each method is to some degree separate from the others and thus has its own operations and control flow.

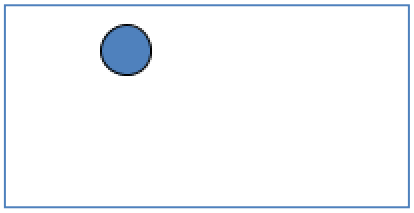
The following code implements this algorithm and includes a sequence of four frames of output that it produces. To see the gravitational effect, observe that the ball moves to the right at a constant rate from one frame to the next, but the rate at which it is falling increases from frame to frame. Once again, you need to actually execute the program yourself to appreciate the animation.

```
final int BALL_SIZE = 25;
final float GRAVITY = 0.20; // fall acceleration

int width = 200, height = width/2;
float ballX, ballY;          // ball location
float deltaX, deltaY;        // ball velocity

void setup() {
  size(width, height);
  smooth();

  // Start the ball in the upper left.
  ballX = 15;
  ballY = 15;

  // Set the rate of change for x and y.
  deltaX = 5;
  deltaY = 0;
}

void draw() {
  // Erase the ball at the previous location.
  background(255);

  // Draw the ball at the current location.
  fill(80, 130, 190);
  ellipse(ballX, ballY, BALL_SIZE, BALL_SIZE);

  // Move the ball to the right and down.
  ballX += deltaX;
  ballY += deltaY;
  deltaY += GRAVITY;
}
```

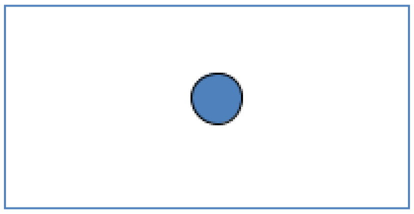
Frame 1.



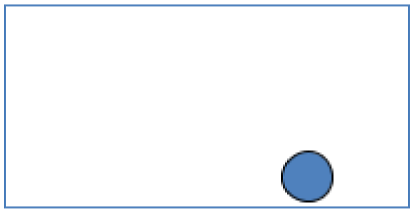
Frame 10.



Frame 19.



Frame 28.



Note that when Processing runs this program, it keeps running, regardless of whether the ellipse it is drawing can be seen or not. In this example, only the first 30 frames include a visible ellipse; the rest are blank because the ellipse's position is outside the output frame. As noted earlier, *Processing programs that contain the* `draw()` *method continue to run until the user presses the stop button (■) or selects Stop on the Sketch menu*. Execution also stops when an `exit();` or `stop();` statement is encountered. The `noLoop()` method can also be used to keep `draw()` from executing repeatedly. The `exit()` and `stop()` methods are explained in the next section and the `noLoop()` method in the next chapter.

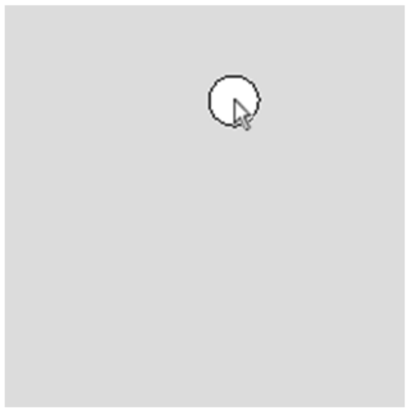
4.3. Interaction Methods

In the previous section we looked at how the `setup()` and `draw()` methods are used to support drawing and animation. Processing also provides built-in methods for detecting **events** such as the user moving the mouse, clicking the mouse, or pressing a key on the keyboard. A program can provide definitions for these methods to specify how to respond to these user interactions. This approach to programming is called **event-driven** programming.

4.3.1. Responding to Mouse and Keyboard Events

One common form of user interaction is based on mouse events. Whenever the user clicks the mouse on the output pane, Processing automatically calls the **`mouseClicked()`** method. In addition, it keeps track of where the mouse pointer is by storing the *x* coordinate of its location in the variable **`mouseX`** and the *y* coordinate of its location in the variable **`mouseY`**.

The following program illustrates how a program can use these features to create an interactive program. This program responds to a mouse click by drawing an ellipse at the location of the mouse pointer when the click occurred. The output shown was produced by the user clicking the mouse when its pointer was in the upper middle of the output panel. Processing calls `mouseClicked()` when the user clicks the mouse when the pointer is positioned in the output frame and it stores the location of the pointer in the pre-defined variables `mouseX` and `mouseY`. These values are then used as the *x* and *y* coordinates of the center when the ellipse is drawn. Note that the empty `draw()` method must still be included in order for the program to run interactively, even though its body is empty.



```
void setup() {
  size(200, 200);
  background(220);
}

void draw() {
}

void mouseClicked() {
  ellipse(mouseX, mouseY, 25, 25);
}
```

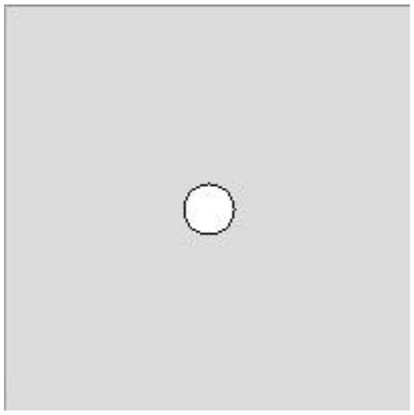
Processing provides the following built-in mouse event methods and calls them automatically when the specified event occurs:

- `mouseClicked()` — pressing and releasing the mouse button
- `mousePressed()` — pressing the mouse button without releasing it
- `mouseDragged()` — moving the mouse around while it is pressed
- `mouseReleased()` — releasing the mouse after it has been pressed
- `mouseMoved()` — moving the mouse when it is not pressed

Processing also provides the following variables whose values are set by these methods:

- `mouseX`, `mouseY` — the *x* and *y* coordinates of the current position of the mouse pointer
- `pmouseX`, `pmouseY` — the *x* and *y* coordinates of the previous position of the mouse pointer (i.e., in the previous output frame)
- `mousePressed` — a `boolean` variable that is set to true if the mouse was pressed and false if not

Processing also provides built-in methods for **keyboard events**. For example, suppose we replace the definition of `mouseClick()` in the previous example with a definition of the `keyPressed()` method as shown in following example. When executed, this modified program opens an output frame and waits for the user to press a key on the keyboard. When this occurs, Processing calls the `keyPressed()` method, which draws an ellipse in the center of the output..



```
void setup() {
  size(200, 200);
  background(220);
}

void draw() {
}

void keyPressed() {
  ellipse(100, 100, 25, 25);
}
```

Processing provides the following built-in keyboard event methods and calls them automatically when the specified event occurs:

- `keyPressed()` — pressing some key on the keyboard
- `keyReleased()` — releasing a key after it has been pressed

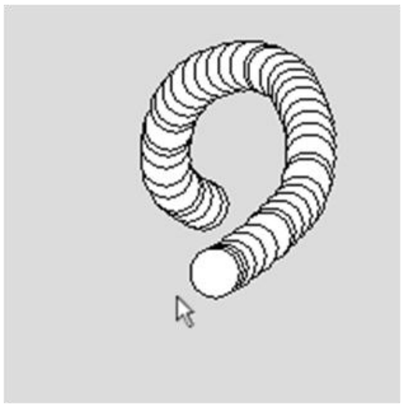
It also provides the following variables whose values are set by these methods:

- `key` — the character of the key just pressed or released (a `char` value)
- `keyPressed` — a `boolean` variable that is set to true if a key was pressed and false if not

4.3.2. Integrating Animation and Interaction Events

We have looked at animation and interaction events and now we consider how we might integrate these. Our programs will use `draw()` to drive the animation and interaction methods to handle user events. The program shown in this section illustrates how to do this.

The following program starts by drawing an ellipse in the center of the output panel and then allows the user to drag the ellipse around by pressing and dragging the mouse. The position of the ellipse is based upon where it was previously located and the current location of the mouse pointer. Each time the user drags the mouse, the ellipse appears to move 1/10th of the way from its current location to where the mouse pointer is located.[3] When the mouse button is released, the `stop()` method stops the program.[4] Note that this program draws the background only once — in `setup()` — so that we can see where the ellipses have been drawn over the course of the animation.



```
final float FOLLOW_FACTOR = 0.1;

int ballX, ballY;

void setup() {
  size(200, 200);
  background(220);
  ballX = 100;
  ballY = 100;
}

void draw() {
  ellipse(ballX, ballY, 25, 25);
}

void mouseDragged() {
  ballX += (mouseX - ballX) * FOLLOW_FACTOR;
  ballY += (mouseY - ballY) * FOLLOW_FACTOR;
}

void mouseReleased() {
  stop();
}
```

4.3.3. Revisiting the Example

Using interaction methods, we can modify the example program shown in Section 4.2.2, which animated a falling ball, to remove the movement and, instead, allow the user to drag the ball around using the mouse. An algorithm that produces this modified behavior is shown here.

Given:
- The constant `BALL_SIZE` represents the size of the ball.
- The constant `FOLLOW_FACTOR` represents the desired rate at which the ball should follow the mouse.
- `WIDTH` and `HEIGHT` represent the width and height of the display window.
- `mouseX` and `mouseY` represent the current coordinates of the mouse.

Algorithm (for `setup()`)**:**
1. Create a display window that is `WIDTH` x `HEIGHT` screen pixels.
2. Set the initial position of the ball to the middle of the display window.

Algorithm (for `draw()`)**:**

[3] Of course, the ellipse doesn't actually move — a new ellipse is drawn in the next frame. But it is easier and more intuitive to think of it as moving.

[4] Another useful method is `save()` which can be used to save a copy of the output canvas. For example, adding the statement `save("MovingEllipse.jpg");` in the `mouseReleased()` method would save an image of the output frame in a JPEG file named "MovingEllipse.jpg" in the current program folder.

1. Erase the display window.
2. Draw the ball at its current location.

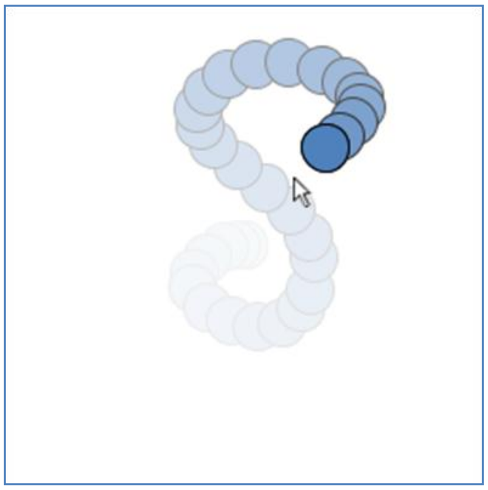
Algorithm (for `mouseDragged()`)**:**

1. Move the ball's location toward the position of the mouse by setting `ballX` = `ballX` + ((`mouseX` – `ballX`) * `FOLLOW_FACTOR`).
2. Do a similar modification for the ball's y coordinate.

This algorithm is similar to the one shown in Section 4.2.2, but it replaces the movement and acceleration features with the mouse-following features shown in the `mouseDragged()` algorithm steps 1 and 2. The addition of the following rate factor (`FOLLOW_FACTOR`) keeps the mouse a little bit behind the cursor.

The following program implements this algorithm. Note that the `draw()` algorithm step 1 is implemented in a way that produces a kind of disintegrating vapor trail of the previous positions of the ellipse. This effect is achieved by drawing a semi-transparent white rectangle over the entire output frame with each mouse drag. This rectangle covers the previously drawn balls but allows them to show through. These successive frames, with each mouse drag "whitewashing" the previous layers, have the effect of gradually erasing the balls over time.

```
final int SIZE = 25;
final float FOLLOW_FACTOR = 0.1;
final int WIDTH = 250, HEIGHT = WIDTH;

float ballX, ballY;

void setup() {
  size(WIDTH, HEIGHT);
  background(255);
  smooth();

  // Start the ball in the middle.
  ballX = WIDTH / 2;
  ballY = HEIGHT / 2;

  // Set the animation to 25 frames/second.
  frameRate(25);
}
```



```
void draw() {
  // Slowly whitewash the previous frames.
  noStroke();
  fill(255, 25);
  rect(0, 0, WIDTH, HEIGHT);

  // Draw the ball at the current location.
  stroke(0);
  fill(80, 130, 190);
  ellipse(ballX, ballY, SIZE, SIZE);
}

void mouseDragged() {
  // Move the ball part-way toward the
  // mouse location
  ballX += (mouseX - ballX) * FOLLOW_FACTOR;
  ballY += (mouseY - ballY) * FOLLOW_FACTOR;
}

void mouseReleased() {
  stop();
}
```

4.4. New Method Definitions

In the previous sections, we defined the **`setup()`**, **`draw()`** and other methods to support animation and interactivity. The approach was to provide definitions for any of a set of methods that Processing has already declared and will undertake to call at the appropriate times during the execution of a program. Methods are, in fact, a considerably more general mechanism that allows programmers to declare and define their own methods, call them when needed and potentially to reuse them as needed. This reuse brings many benefits, including the elimination of redundant code and the creation of more understandable code.

This section reconsiders the chapter example, extending it somewhat and then details the definition and use of methods.

4.4.1. Reconsidering the Example

Our most recent version of the ball-animation example implements a ball that follows the mouse pointer around the output window. This brings to mind the video game *Pac-Man* that hit the video arcades in the 1990s and became one of the most famous arcade games of all time.[5] The traditional Pac-Man is a small yellow circle with a hungry, chomping mouth and this raises the intriguing possibility of redesigning our ball as a more complex object that resembles a Pac-Man.

The program we develop in this section draws a Pac-Man with a closed mouth as a yellow circle:

and a Pac-Man with an open mouth as a yellow circle with black-filled section of an ellipse drawn on top:

[5] See http://en.wikipedia.org/wiki/Pac-Man.

For the ellipse section in this last picture, we can use Processing's **`arc()` method** which is called with a statement of the form[6]

```
arc(x, y, width, height, start, stop);
```

```
where
```

x, and *y*, are the coordinates of the center of the arc's ellipse (cf. the `ellipse()` method);
`width` and *`height`* are the width and height of the arc's ellipse (cf. the `ellipse()` method);
`start` is the angle at which to start the arc, measured in radians; and
`stop` is the angle at which to stop the arc, measured in radians.

For a closed mouth, we can use 0 for both `start` and `stop`. For an open mouth, we can use `-PI/8` for `start` and `PI/8` for `stop`, where **`PI`** is a predefined constant provided in Processing whose value is an approximation to π (3.14159 …).[7] The integer variable `biteFactor` is used to switch back and forth between a closed and an open mouth: a value of 0 represents a closed mouth and a value of 1 represents an open. Switching between the values 0 and 1 value of `biteFactor` is switched between 0 and 1 by adding one and reducing the result modulo 2:

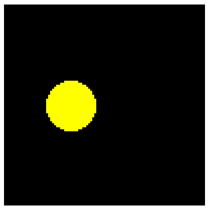
```
biteFactor = (biteFactor + 1) % 2;
```

Given this technique, we could specify the code for an animated, chomping Pac-Man as follows:

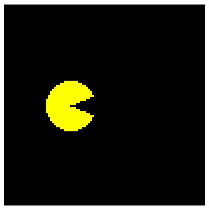
[6] See http://www.processing.org/reference/arc_.html.

[7] Processing also provides constants `TWO_PI`, `HALF_PI`, and `QUARTER_PI`, representing 2π, π/2, and π/4, respectively. It also provides `degrees()` and `radians()` to convert between radians and degrees. See Processing's web reference (http://www.processing.org/reference) for more information.

Even-numbered frames



Odd-numbered frames.



```
int biteFactor;

void setup() {
  size(100, 100);
  frameRate(5);
  biteFactor = 0;
}

void draw() {
  background(0);

  // Draw the Pac-Man head.
  noStroke();
  fill(255, 255, 0);
  ellipse(33, 50, 25, 25);

  // Draw the mouth.
  fill(0);
  arc( 33, 50, 25, 25,
       biteFactor*(-PI/8.0), biteFactor*(PI/8.0) );

  // Switch the bite factor (values: 0 or 1).
  biteFactor = (biteFactor + 1) % 2;
}
```

Note that this new block of code designed to draw the Pac-Man head and mouth is simply coded directly into the `draw()` method definition. To draw two Pac-Man figures side by side, we could simply copy and paste this block of code two times in the `draw()` method. This approach would work, but, unfortunately, does not scale well. Just imagine how long the `draw()` method would be if we wanted to draw a dozen or two dozen Pac-Man figures all in the same sketch; we would have to copy and paste the Pac-Man-drawing instructions over and over, one for each Pac-Man.

This would be so much easier if we had a Pac-Man-drawing method that we could call, but unfortunately Processing does not provide one. Processing's pre-defined drawing methods are restricted to more commonly used figures — e.g., points, lines, ellipses, etc. — rather than at more complex or arbitrary figures — e.g., Pac-Man figures. This is a good, general approach for Processing to take. Thousands of programmers work productively at this level of abstraction and one wonders how many programmers would ever want a Pac-Man-drawing method anyway, but all programmers inevitably find themselves in situations where they would like to create their own unique, higher-level methods.

4.4.2. User-Defined Methods

In the previous section, we discussed the need for new, higher-level methods that aren't already implemented in Processing. *Processing supports this by allowing programmers to define their own methods.* As an example, consider the task of computing the sum of an arithmetic sequence S. The most famous instance of this computation is the problem of adding the numbers from 1 to 100. One could do this by toiling through the long arithmetic summation: $1 + 2 + \cdots + 100$, but the problem is much easier if one notices the pattern $1 + 2 + \cdots + 100 = (1 + 100) + (2 + 99) + \cdots + (50 + 51) = 101 \times 50 = 5050$.

The general form of this problem can be specified using the number of elements (n), the starting point (a) and the common difference (d). In the previous example, n=100, a=1 and d=1, but one could consider other sequences as well, including: 7, 10, 13, 16, 19, 22, in which *n*=6, *a*=7 and *d*=3. To compute the sum of the general form of an arithmetic sequence, we can use Gauss's Formula:

$$na + \frac{nd(n-1)}{2}$$

Given the general usefulness of such a formula, a programmer might be well-served defining a method to perform the computation, given values for *a*, *n* and *d*. The method can be specified using the following algorithm:

Algorithm (for `sequenceSum()`)**:**

1. **Receive** `n`, `start`, and `difference`, representing the number of elements (an integer), starting number for the sequence (a floating point value), and common difference between sequence elements (another floating point value), **from the calling program**.
2. **Return** `n*start + n*difference*(n-1)/2`.

We can implement this algorithm as a method using the following code:

```
float computeSequenceSum(int n, float start, float difference) {
   return n*start + (n*difference*(n-1))/2.0;
}
```

This code follows the pattern for new method definitions:

Method Definition

```
returnType methodName(parameterDeclarations) {
   statements
}
```

- *returnType* is the type of value returned by the method or `void` if the method does not return a value;
- *methodName* is the identifier that names the method;
- *parameterDeclarations* is a comma-separated list of parameter declarations of the form *type identifier* that is omitted if there are no parameters;
- *statements* is a set of statements that define the behavior of the method.

Note:

- Its return type is **`float`**, which indicates that the method computes and returns a floating point value;
- The name of the method, **`computeSequenceSum`**, gives an indication of what the method does in order to make its use more understandable. This is the name used to call, or invoke, the method. In this text, we attach the parentheses, **`()`**, when using the name of a method — i.e., **`computeSequenceSum()`**.

- The parameter declarations specify that method must receive an integer and two floating point values from its calling program: the number of elements (**n**), the starting point (**start**) and the increment (**difference**);
- A method bundles statements together into a **block** using a matched pair of curly braces ({ }). As with the **setup()** and **draw()** methods discussed above, this indicates that Processing should execute the statements together, separately from other blocks of statements;
- The **return** statement computes the value of the given expression and returns it to the calling program.

Now that we've defined the **computeSequenceSum()** method, we can use it throughout our program as needed, without having to re-implement its computation over and over again. In addition, we can provide the method to other programmers who can use it without completely understanding how it was implemented. This is an example of **procedural abstraction**. Programmers who know *what* the procedure (aka., method) named **computeSequenceSum()** does can use it without knowing *how* it does it. This is a powerful concept. Throughout this text, we've been using methods whose complex operations were carefully designed and implemented by other programmers for use by all programmers. For example, the **random()** method generates pseudo-random numbers using a complex algorithm that we, as the users of this method, do not need to completely understand. This allows programmers to address continuously new problems without having to repeatedly "reinvent the wheel" for each new program.

The following sections discuss the key aspects of this method definition and of how it is used.

4.4.3. Parameters

The parameter list in our method definition, specified within the parentheses, **()**, is shown here:

```
float computeSequenceSum(int n, float start, float difference) {
   return n*start + (n*difference*(n-1))/2.0;
}
```

Parameters are variables that store the values passed to the method. These values can then be used in the method body. In our example, the parameter list indicates that the **computeSequenceSum()** method expects one integer and two floating point values from its calling program. The integer specifies the number of elements and the two floating point values specify the starting point and the sequence increment respectively. Note that the parameter list specifies the type of each of these parameters. It is therefore a *declaration* of new variables. When the method is called, Processing will allocate variables for each of the three parameters and initialize their values to whatever the calling program provides.

This parameter mechanism generalizes the method definition. This method can now be used to compute the sum of any sequence based on its starting point, number of terms and its sequence increment. This allows programmers to use this method to compute the famous summation of the numbers 1 through 100, by specifying the three values 100, 1, 1, or any other sequence, by specifying the appropriate number of elements, starting point and sequence increment.

For some method definitions, the parameter list may be empty, in which case the parentheses are still included, **`()`**, but there are no parameters specified within them. Before this section, our method definitions have had empty parameter lists, e.g., **`setup()`** and **`draw()`**.

When writing algorithms for methods that receive values, we use a special term to denote the parameter-passing operation. As an example, step 1 in the **`computeSequenceSum()`** algorithm above, indicates that the method *receives* a list of values *from the calling program*. This text follows this convention when specifying algorithms for methods with parameters.

4.4.4. Return Values

The return type and return statement in our method definition are shown here:

```
float computeSequenceSum(int n, float start, float difference) {
   return n*start + (n*difference*(n-1))/2.0;
}
```

The initial word `float` indicates that this method returns an floating point value when it is called. The corresponding **`return`** statement computes and returns the value of the given expression where the type of the expression must match this specified return type. In this case, the value of the expression, **`n*start + (n*difference*(n-1))/2`**, is a floating point value as required. At most one return statement is allowed in a method. The pattern for the return statement is as follows.

Return Statement

`return` *`expression`*`;`

- *`expression`* is a valid expression whose type is the same as (or is compatible with) the specified return type of the containing method.

When writing algorithms for methods that return values, we use a special term to denote the value-return operation. As an example, step 2 in the **`computeSequenceSum()`** algorithm above indicates that the method *returns* a value to the calling program. This text follows this convention when specifying algorithms for methods with return values.

Methods whose return type is specified as **`void`** perform actions but do not return a value. For example, in Chapter 2, we used the void method **`smooth()`** to render graphics in a particular way; we didn't expect it to compute and return a value. The animation and interaction methods discussed in this chapter, e.g., **`setup()`** and **`draw()`**, are **`void`** methods as well; they setup and drive animations, but they don't compute or return values. **`void`** methods do not have **`return`** statements.

As an example, consider the problem of printing simple status messages to the output console. A program might need to do this at various points in its execution and it would like the output to be formatted in a consistent manner. We can implement a method that does this using the following code:

```
void printStatusMessage(String message) {
  println("Status message: " + message);
}
```

This method is a **void** method, that is, its return type is **void**. Its purpose is to print the message that it receives from its calling program in a consistent manner. It does not compute or return any value.

4.4.5. Method Invocation

We conceived of and implemented the **computeSequenceSum()** method ourselves, which means that Processing cannot know about it in the way that it knows about its own pre-defined methods, e.g., **setup()** and **draw()**. As a consequence, we cannot expect Processing to automatically call our new, user-defined methods as it does its own methods. This means that it is our responsibility to modify our program to call the new **computeSequenceSum()** method in the appropriate way and in the appropriate place(s). The following code computes the value of the famous sum of 1+2+…+100 using our new **computeSequenceSum()** method and prints the solution (5050.0) in the console output panel:

```
void setup() {
  println(computeSequenceSum(100, 1, 1));
}

float computeSequenceSum(int n, float start, float difference) {
  return n*start + n*difference*(n-1)/2.0;
}
```

This code segment includes the full definition of the **computeSequenceSum()** method, but it also includes a call to **computeSequenceSum()** in the **setup()** method. When this program is executed, Processing automatically calls the **setup()** method as discussed above, but we must call **computeSequenceSum()** explicitly. Note also that even though this program doesn't implement animation, the **setup()** is still required because that's the method Processing calls it to start the program running.

We are then, of course, free to call the method as many times as we would like. For example, the following invocation computes the sum of a sequence of length 6, starting at 7 with an increment of 3:

```
computeSequenceSum(6, 7, 3)
```

This code follows the pattern for invoking new methods:

Method Invocation

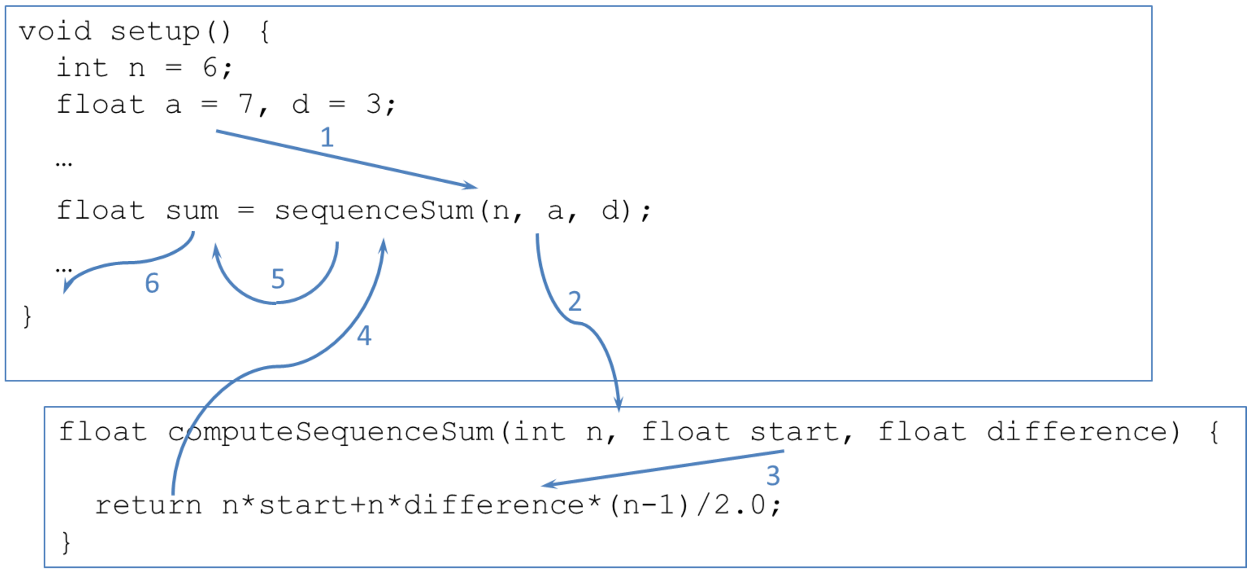
methodName(*argumentList*)

- *methodName* is the identifier that names the method being called;
- *argumentList* is a comma-separated list of arguments, where arguments are expressions that return a value of the type expected by the corresponding parameters in the method parameter list. The argument list is omitted if the method expects no arguments.

The *arguments* in the method call must match the *parameters* in the method definition in number, order and type. Thus, given the call `computeSequenceSum(100, 1, 1)`, the arguments, `100`, `1`, `1`, are passed to the parameters `n`, `start` and `difference` respectively. In this case, the parameter list specifies one integer value followed by two floating point values, so the three arguments must either be an integer followed by two floating point values, or values that can be cast to integer, float and float.

For methods that specify no parameters, the argument list must be empty. The **smooth()** method, for example, specifies no parameters and thus allows no arguments when it is called. The **random()** method, on the other hand, expects a numeric argument when it is called — e.g., **random(255)**. Methods that expect multiple arguments must use a comma-separated list of arguments — e.g., **size(100, 100)**.

The flow of data and control through our example program can be pictured as shown here:



```
void setup() {
  int n = 6;
  float a = 7, d = 3;
  …
  float sum = sequenceSum(n, a, d);
  …
}
```

```
float computeSequenceSum(int n, float start, float difference) {

  return n*start+n*difference*(n-1)/2.0;
}
```

(1) As part of the execution of the program, the declaration statement for `sum` is encountered.

(2) The variables **n**, **a**, and **d**, which are specified as arguments to the **sequenceSum()** method, are evaluated and the resulting values are copied into the method definition's parameters **n**, **start**, and **difference**, respectively, and control transfers from **setup()** to **sequenceSum()**. Note that the names of the argument variables and the parameters are independent of one another. They can, but need not be, the same.

(3) The **sequenceSum()** method's **return** expression is evaluated using the values of **n**, **start**, and **diff**.

(4) The return statement makes the value of this expression the return value of the **sequenceSum()** method and control transfers back to **setup().**

(5) The return value is assigned to the variable **sum**. If the program calls the method but doesn't include this assignment statement capturing the result, then the value computed by the method is discarded.

(6) Control proceeds to the next statement in `setup()`.

4.4.6. Testing

Given that our new method may be used many times, potentially by different programmers, it is important that it be both correct and as efficient as possible. Clearly a programmer must be very careful when designing and implementing a method, but even in the best of circumstances, programmers can make mistakes. Just as programs must be tested, see the discussion of this in Chapter 2, methods should also be tested.

Testing is a significant undertaking in and of itself and, as such, is generally integrated throughout the development lifecycle. During the initial analysis phase, test cases should be identified. In the case of computing the sum of a sequence, we should identify test cases that we can use to exercise the method we design and implement. These test cases should include a variety of examples, including tricky boundary cases that might be easily overlooked, such as the following:

Input (n, a, d)	**Expected output**	**Description**
100, 1, 1	5050.0	We can start with this famous example, for which the correct solution is well-known.
6, 7, 3	87.0	This is another simple example mentioned during the analysis and design phases.
0, 0, 0	0.0	This is a tricky boundary case that could lead to a division-by-zero error.
4, 0, -1	-6.0	There's no reason the increment can't be negative.
2, 2.5, 2.5	7.5	Neither is there any reason the start and increment values must be integers.

For a real, production method, we'd likely want to specify more test cases and work out the appropriate solutions by hand, but this will suffice for our purposes here. We can run these test cases with this simple program, which is often called a **driver** program:

```
void setup() {
  println(computeSequenceSum(100, 1, 1));
  println(computeSequenceSum(6, 7, 3));
  println(computeSequenceSum(0, 0, 0));
  println(computeSequenceSum(4, 0, -1));
  println(computeSequenceSum(2, 2.5, 2.5));
}

float computeSequenceSum(int n, float start, float difference) {
  return n*start + n*difference*(n-1)/2.0;
}
```

When we run this program, we see that the output matches our expectations. This increases our confidence in the correctness of the implementation. However, it's difficult to be completely sure of the correctness of our implementations – not with this simple example and certainly not with more complex examples.

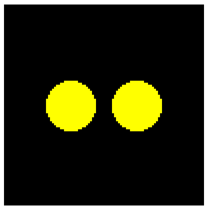
With Processing code, testing can often be done by watching the methods' graphical output, but this approach doesn't help with non-graphical code. Thus, programmers must often write testing programs such as the one shown here that allow them to test the behavior of their code.

4.4.7. Scope

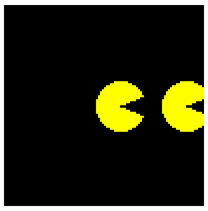
As our programs grow in size and complexity, we will find ourselves using more and more identifiers. Processing programs, for example, often manipulate a variety of geometric objects, all of which have *x* and *y* coordinates that are most appropriately named with the identifiers x and y.

As an illustration, consider the Pac-Man reformulation of the chapter example introduced in Section 4.4.1. That program animated a single Pac-Man with a chomping mouth. The following code extends that program by animating two Pac-Man figures as they march off the output pane. As you read through the code and execute it for yourself, be careful to distinguish what the identifiers x and y refer to at each place they appear in the program.

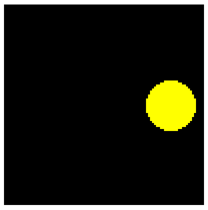
Frame 1.



Frame 6.



Frame 11.



```
float x, y;
int biteFactor;

void setup() {
  size(100, 100);
  frameRate(5);
  x = width/3;
  y = height/2;
  biteFactor = 0;
}

void draw() {
  background(0);

  drawPacman(x, y);
  drawPacman(x + 33, y);

  // Move the Pac-Man figures to the right.
  x += 5;

  // Switch the bite factor (values 0 or 1).
  biteFactor = (biteFactor + 1) % 2;
}

void drawPacman(float x, float y) {
  // Draw the body.
  noStroke();
  fill(255, 255, 0);
  ellipse(x, y, 25, 25);

  // Draw the mouth.
  fill(0);
  arc(x, y, 25, 25,
      biteFactor*(-PI/8.0), biteFactor*(PI/8.0) );
}
```

This code uses the variables x and y to store the position of the left-most Pac-Man from frame to frame and also draws a second Pac-Man exactly 33 pixels to the right of the first one by setting its *x* coordinate to x+33. This keeps the two Pac-Man figures moving in tandem. Note that the identifiers **x** and **y** are used in different places to refer to different things:

- In the **setup()** and **draw()** methods, x identifies the variable holding the *x* coordinate of the left-most Pac-Man figure; this use of this identifier x is declared in the first line of the program and is coded in blue. **y** identifies the variable holding the *y* coordinate of both Pac-men; it is also declared in the first line of the program and coded in blue.
- In the **drawPacman()** method, **x** refers to the *x* coordinate of the Pac-Man figure currently being drawn by the method, which may be either the left-most or the right-most Pac-Man figure; this use of the identifier **x** is declared in the parameter list of the **drawPacman()** method. The **y** parameter is treated in a similar manner.

For this program to work correctly, Processing must distinguish between these two uses of the identifiers **x** and **y.** It does this using **scope**. An identifier's scope is the part of the program where that identifier is associated with a given declaration. When Processing sees an identifier, it searches for a declaration of that identifier in the current block; if it finds on there, it uses it, otherwise it proceeds to the surrounding block, and so forth. In this example, the use of the identifiers **x** and **y** in the calls to **ellipse()** and **arc()** are associated with the parameters **x** and **y** declared by the **drawPacman()** method. These uses are found inside the red bounding box drawn around the **drawPacman()** method. In contrast, the use of **x** and **y** in **draw()** are associated with the declarations of **x** and **y** in the first line of the program. These uses are found inside the blue bounding box that surrounds the entire program.

Variables declared within a method, either as a parameter or in the method's definition block, are called **local variables**. The scope of their identifiers is limited to the small bounding boxes around the method that declares them. In the example, the **x** and **y** parameters declared in the **drawPacman()** method are local to that method; their scope is limited to the red bounding box around the **drawPacman()** method. Variables declared at the top of a program are called **global variables**. The scope of their identifiers ranges throughout the whole program. In the example, the **x** and **y** parameters declared at the top of the program are global to the entire program. If an identifier has more than one declaration, the most local declaration takes precedence.

It is best to declare variables as locally as possible. Methods should declare parameters to receive working values from the calling program so that the method's code can control the type and use of those values. As a general rule, Processing programs should use global variables only when there are values that must be shared between several of Processing's pre-defined methods. For example, the global values of **x** and **y** in this sample Pac-Man program must be shared by both **setup()** and **draw()** for the animation to work properly, so they are declared globally. Everything else is declared locally.

4.4.8. Method Documentation

Methods must be documented. They are significant elements of the programs they support and may be used by many different programmers, so the creator(s) of a method should provide complete documentation on the purpose and behavior of the method, including a precise specification of what each parameter, if any, is used for. For example, the following is an appropriate level and form of documentation for **computeSequenceSum()** method:

```
/**
 * computeSequenceSum() computes the sum of the sequence specified
 * by the arguments using Gauss’s formula.
 *
 * @param n - the number of numbers in the sequence
 * @param start - the first value in the sequence
 * @param difference - the increment between each number
 */
float computeSequenceSum(int n, float start, float difference) {
  return n*start + (n*difference*(n-1))/2.0;
}
```

Yes, the documentation is actually longer that the method itself. This is entirely appropriate. It explains what the program does and the purpose of each parameter. This information is critical for programmers hoping to use the method correctly and productively. While the identifier chosen for each parameter gives a pretty good idea of the purpose of that parameter, a brief explanation is sometimes needed to make things sufficiently clear. This documentation is also likely as long or longer that the documentation for the program itself. This also is entirely appropriate. The method may eventually make its way into a library of useful methods that is used by many other programs and thus, the method itself may be used more frequently than the program for which it was originally designed and implemented.

Note that though this method documentation mentions the use of Gauss’s formula, it focusses primarily on the method’s behavior and use. This helps establish the procedural abstraction provided by the method, which is more important for programmers using the method than are the details of how the method works.

4.5. A Pac-Man Example

This section extends the Pac-Man example introduced in the previous section. It uses animation to make the Pac-Man figures move, interaction methods to make multiple Pac-Man figures follow the location of the mouse when the user presses or drags the mouse, and user-defined methods to draw the Pac-Man figures, increment the coordinate values, and compute the facing angle for each Pac-Man. The algorithm that we will implement is as follows.

Given:

- The constant `BALL_SIZE` represents the size of the ball.
- The constant `FOLLOW_FACTOR` represents the desired rate at which the ball should follow the mouse.
- `WIDTH` and `HEIGHT` represent the width and height of the display window.
- `mouseX` and `mouseY` represent the current coordinates of the mouse.
- `x1, y1` and `x2, y2` represent the current coordinates of the two figures.
- `targetX` and `targetY` represent the current coordinates of the target.
- `biteFactor` represents whether the Pac-Man mouth is open or closed.

Algorithm (for `setup()`):

1. Create a display window that is `WIDTH` x `HEIGHT` screen pixels.
2. Set the initial position of the two Pac-Man figures ball to the middle of the display window.
3. Set the initial target of the figure movement to be the center of the display window.
4. Set `biteFactor` = 0.

Algorithm (for `draw()`):

    1. Erase the display window.
    2. Draw the two Pac-Man figures at their respective locations (see `drawPacman()` and `computeFacingAngle()` below).
    3. Toggle the bite-factor between 0 and 1 for the next frame.

Algorithm (for `mouseDragged()` and `mousePressed()`)**:**

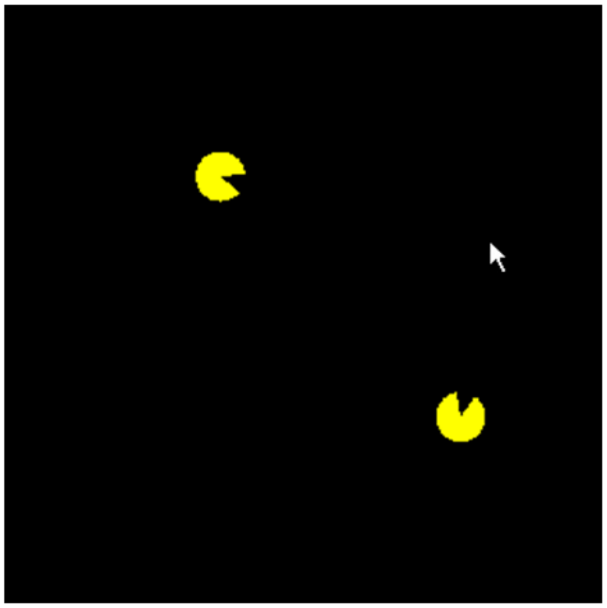
    1. Set the target of the figure movement to be the location of mouse.

Algorithm (for `drawPacman()`)**:**

    1. Receive x and y coordinates and the facing angle from the calling program.
    2. Draw a yellow circle at the given coordinates of diameter `BALL_SIZE`.
    3. Draw a black mouth on the circle that is open or closed based on `biteFactor`.

This algorithm serves as a rather higher level design for the program we are writing. For example, `draw()` step 2 involves some details that aren't completely specified here. The point of this design to give a general structure to the program; we leave some details to the implementation step. As our programs get more complicated, using algorithms in this manner will become more and more valuable. Until this example, we might have been able to skip the algorithm steps altogether because they have been rather simple, but with this example we begin to see the value of carefully designing the algorithm first before moving on to the task of implementation.

The following program implements this algorithm.



```
/**
```

```
 * Pacman implements a pacman figure (see
 * http://en.wikipedia.org/wiki/Pacman) that follows the
 * mouse when it is pressed. Note that it will follow the
 * mouse off the screen.
 *
 * @author nyhl, jnyhoff, kvlinden, snelesen
 * @version Fall, 2011
 */

final int SIZE = 25;
final float FOLLOW_FACTOR = 0.10;
final int WIDTH = 300, HEIGHT = WIDTH;
float x1, y1, x2, y2, targetX, targetY;
int biteFactor = 0;

void setup() {
  size(WIDTH, HEIGHT);
  frameRate(8);
  x1 = y1 = 10;
  x2 = width - 10;
  y2 = height - 10;
  targetX = WIDTH / 2;
  targetY = HEIGHT / 2;
}

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

  // Position and draw the first Pac-Man figure.
  x1 = computeNewCoordinate(x1, targetX);
  y1 = computeNewCoordinate(y1, targetY);
  drawPacman(x1, y1, computeFacingAngle(x1, y1, targetX, targetY));
  // Position and draw the second Pac-Man figure.
  x2 = computeNewCoordinate(x2, targetX);
  y2 = computeNewCoordinate(y2, targetY);
  drawPacman(x2, y2, computeFacingAngle(x2, y2, targetX, targetY));

  // Reset the bite factor for the next frame.
  biteFactor = (biteFactor + 1) % 2;
}

// Treat both mouse pressing and dragging in the same manner.
void mousePressed() {
  changeTarget();
}
void mouseDragged() {
  changeTarget();
}
void changeTarget() {
  targetX = mouseX;
  targetY = mouseY;
}




/**
```

```
     * This method computes a new x or y coordinate for a figure to
     * simulate motion.
     *
     * @param current - the current coordinate value
     * @param target - the destination value of the coordinate
     */
    float computeNewCoordinate(float current, float target) {
      return current + (target - current) * FOLLOW_FACTOR;
    }

    /**
     * This method computes the angle that faces a figure toward
     * the mouse.
     *
     * @param sourceX - the current x coordinate
     * @param sourceY - the current y coordinate
     * @param targetX - the target x coordinate
     * @param targetY - the target y coordinate
     */
    float computeFacingAngle(float sourceX, float sourceY,
                             float targetX, float targetY) {
      return atan2(targetY - sourceY, targetX - sourceX);
    }

    /**
     * This method draws the Pac-Man figure and the given
     * coordinates, facing in the direction of the given angle.
     *
     * @param x - the desired x coordinate
     * @param y - the desired y coordinate
     * @param angle - the desired angle of orientation
     */
    // Draw the Pac-Man figure.
    void drawPacman(float x, float y, float angle) {
      noStroke();
      fill(255, 255, 0);
      ellipse(x, y, SIZE, SIZE);
      fill(0);
      arc(x, y, SIZE, SIZE,
          biteFactor*(-PI/8.0) + angle, biteFactor*(PI/8.0) + angle );
    }
```

In this implementation, Pac-Man #1 starts in the upper left corner of the output canvas, Pac-Man #2 starts in the lower right. The `draw()` method calls `computeNewCoordinate()` four times, once for each coordinate value; the new coordinate values are computed based on the current mouse position.