10 Algorithms

Lather. Rinse. Repeat.
—Unknown

10-1 Where have we been? Where are we going?

Our friend Zoog had a nice run. Zoog taught you the basics of the shape drawing libraries in *Processing*. From there, Zoog advanced to interacting with the mouse, to moving autonomously via variables, changing direction with conditionals, expanding its body with a loop, organizing its code with functions, encapsulating its data and functionality into an object, and finally duplicating itself with an array. It's a good story, and one that treated us well. Nonetheless, it's highly unlikely that all of the programming projects you intend to do after reading this book will involve a collection of alien creatures jiggling around the screen (if they do, you are one lucky programmer!). What I'd like to do now is pause for a moment and consider what you have learned and how it can apply to what *you want to do*. What is your idea and how can variables, conditionals, loops, functions, objects, and arrays help you?

In earlier chapters I focused on straightforward "one feature" programming examples. Zoog would jiggle and only jiggle. Zoog didn't suddenly start hopping. And Zoog was usually all alone, never interacting with other alien creatures along the way. Certainly, I could have taken these early examples further, but it was important at the time to stick with basic functionality so that you could really learn the fundamentals.

In the real world, software projects usually involve many moving parts. This chapter aims to demonstrate how a larger project is created out of many smaller "one feature" programs like the ones you are starting to feel comfortable making. You, the programmer, will start with an overall vision, but must learn how to break it down into individual parts to successfully execute that vision.

I will start with an idea. Ideally, I would pick a sample "idea" that could set the basis for any project you want to create after reading this book. Sadly, there is no such thing. Programming your own software is terrifically exciting because of the immeasurable array of possibilities for creation. Ultimately, you will have to make your own way. However, just as I picked a simple creature for learning the fundamentals, knowing you will not really be programming creatures all of your life, I can attempt to make a generic choice, one that will hopefully serve for learning about the process of developing larger projects.

My choice will be a simple game with interactivity, multiple objects, and a goal. The focus will not be on good game design, but rather on good *software design*. How do you go from thought to code? How do you implement your own algorithm to realize your ideas? I will show how this larger project can be divided into four mini-projects and attack them one by one, ultimately bringing all parts together to execute the original idea.

I will continue to emphasize object-oriented programming, and each one of these parts will be developed using a class. The payoff will be seeing how easy it then is to create the final program by bringing the self-contained, fully functional classes together. Before I get to the idea and its parts, let's review the concept of an *algorithm* which you'll need for steps 2a and 2b.

Process

1. **Idea** — Start with an idea.
2. **Parts** — Break the idea down into smaller parts.
    a. **Algorithm Pseudocode** — For each part, work out the algorithm for that part in pseudocode.
    b. **Algorithm Code** — Implement that algorithm with code.
    c. **Objects** — Take the data and functionality associated with that algorithm and build it into a class.
3. **Integration** — Take all the classes from Step 2 and integrate them into one larger algorithm.

10-2 Algorithms: Dance to the beat of your own drum

An algorithm is a procedure or formula for solving a problem. In computer programming, an algorithm is the sequence of steps required to perform a task. Every single example I have created so far in this book involved an algorithm.

An algorithm is not too far off from a recipe.

1. Preheat the oven to 400°F.
2. Whisk together balsamic vineger, olive oil, and mustard.
3. Bake four portabello mushrooms for 12-15 minutes.
4. Arrange mushrooms on a serving platter and top with dressing.

The above is a nice algorithm for cooking portabello mushrooms. Clearly I am not going to write a Processing program to cook mushrooms. Nevertheless, if I did, the above pseudocode might turn into the following code.

```
preheatOven(400);
placeMushrooms(4, "baking dish");
bake(400, 15);
whisk("balsamic", "olive oil", "mustard");
combine("mushrooms", dressing");
```

An example that uses an algorithm to solve a math problem is more relevant to your pursuits. Let's describe an algorithm to evaluate the sum of a sequence of numbers 1 through *N*.

$SUM(N) = 1 + 2 + 3 + \ldots + N$

where *N* is any given whole number greater than zero.

```
1. Set SUM = 0 and a counter $i = 1$
2. Repeat the following steps while $i$ is less than or equal to $N$.
        a. Calculate SUM + $i$ and save the result in SUM.
        b. Increase the value of $i$ by 1.
3. The solution is now the number saved in SUM.
```

Translating the preceding algorithm into code, I have:

```
int sum = 0;
int n = 10;
int i = 0;

while (i <= n) {

  sum = sum + i;

  i++ ;
}

println(sum);
```

Step 1: Set `i` equal to 0 and counter `i=0`.

Step 2: Repeat while `i <= n`.

Step 2a: Increment `sum`.

Step 2b: Increment `i`.

Step 3: The solution is in `sum`. Print `sum`!

Traditionally, programming is thought of as the process of (1) developing an idea, (2) working out an algorithm to implement that idea, and (3) writing out the code to implement that algorithm. This is what I have accomplished in both the mushroom and summation examples. Some ideas, however, are too large to be finished in one fell swoop. And so I am going to revise these three steps and say that programming is the process of (1) developing an idea, (2) breaking that idea into smaller manageable parts, (3) working out the algorithm for each part, (4) writing the code for each part, (5) working out the algorithm for all the parts together, and (6) integrating the code for all of the parts together.

This does not mean to say you shouldn't experiment along the way, even altering the original idea completely. And certainly, once the code is finished, there will almost always remain work to do in terms of cleaning up code, bug fixes, and additional features. It is this thinking process, however, that should guide you from idea to code. If you practice developing your projects with this strategy, creating code that implements your ideas will hopefully feel less daunting.

10-3 From idea to parts

To practice this development strategy, I will begin with the idea, a very simple game. Before I can get anywhere, I should describe the game in paragraph form.

Rain game

The object of this game is to catch raindrops before they hit the ground. Every so often (depending on the level of difficulty), a new drop falls from the top of the screen at a random horizontal location with a random vertical speed. The player must catch the raindrops with the mouse with the goal of not letting any raindrops reach the bottom of the screen.

Exercise 10-1: Write out an idea for a project you want to create. Keep it simple, just not too simple. A few elements, a few behaviors will do.

Now let's see if I can take the "Rain Game" idea and break it down into smaller parts. How do I do this? For one, I can start by thinking of the elements in the game: the raindrops and the catcher. Secondly, I should think about these elements' behaviors. For example, I will need a timing mechanism so that the drops fall "every so often". I will also need to determine when a raindrop is "caught." Let's organize these parts more formally.

1. Develop a program with a circle controlled by the mouse. This circle will be the user-controlled "rain catcher."
2. Write a program to test if two circles intersect. This will be used to determine if the rain catcher has caught a raindrop.
3. Write a timer program that executes a function every *N* seconds.
4. Write a program with circles falling from the top of the screen to the bottom. These will be the raindrops.

Parts 1 through 3 are simple and each can be completed in one fell swoop. However, with Part 4, even though it represents one piece of the larger project, it's complex enough that I will need to complete this exact exercise by breaking it down into smaller steps and building it back up.

Exercise 10-2: Take your idea from Exercise 10-1 on page 192 and write out the individual parts. Try to make the parts as simple as possible (almost to the point that it seems absurd). If the parts are too complex, break them down even further.

Section 10-4 on page 193 to Section 10-7 on page 204 will follow the process of Steps 2a, 2b, and 2c (see "Process" on page 190) for each individual part. For each part, I will first work out the algorithm in pseudocode, then in actual code, and finish with an object-oriented version. If I do my job correctly, all of the functionality needed will be built into a class which can then be easily copied into the final project itself when I get to Step 3 (integration of all parts).

10-4 Part 1: The catcher

This is the simplest part to construct and requires little beyond what I covered in Chapter 3. Having pseudocode that is only two lines long is a good sign, indicating that this step is small enough to handle and does not need to be made into even smaller parts.

Pseudocode:

- Erase background.
- Draw an ellipse at the mouse location.

Translating it into code is easy:

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

void draw() {
  background(255);                      Erase background.
  stroke(0);
  fill(175);
  ellipse(mouseX, mouseY, 64, 64);      Draw an ellipse at the
}                                       mouse location.
```

This is a good step, but I'm not done. As stated, my goal is to develop the rain catcher program in an object-oriented manner. When I take this code and incorporate it into the final program, I will want to have it separated out into a class so that I can make a `Catcher` object. My pseudocode is therefore revised to look like the following.

Setup:

- Initialize catcher object.

Draw:

- Erase background.
- Set catcher location to mouse location.
- Display catcher.

Example 10-1 shows the code rewritten assuming a `Catcher` object.

Example 10-1. Catcher

```
Catcher catcher;

void setup() {
  size(400, 400);
  catcher = new Catcher(32);
}

void draw() {
  background(255);

  catcher.setLocation(mouseX, mouseY);

  catcher.display();
}
```

Erase background.

Set catcher location to mouse location.

Display catcher.

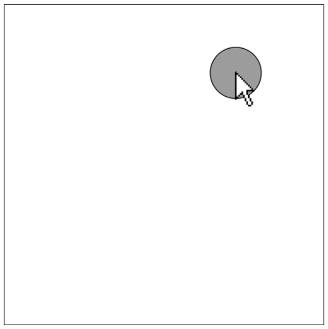


Figure 10-1

The `Catcher` class itself is rather simple, with variables for location and size, and two functions, one to set the location and one to display.

```
class Catcher {
  float r;   // radius
  float x;   // location
  float y;

  Catcher(float tempR) {
    r = tempR;
    x = 0;
    y = 0;
  }

  void setLocation(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }

  void display() {
    stroke(0);
    fill(175);
    ellipse(x, y, r*2, r*2);
  }
}
```

10-5 Part 2: Intersection

Part 2 of the list requires me to determine when the catcher and a raindrop intersect. Intersection *functionality* is what I want to focus on developing in this step. I will start with a simple bouncing ball class (which you saw in Example 5-6) and work out how to determine when two bouncing circles intersect. During the "integration" process, this `intersect()` function will be incorporated into the `Catcher` class to catch raindrops.

Here is an algorithm for this intersection part.

Setup:

- Create two ball objects.

Draw:

- Move balls.
- If ball #1 intersects ball #2, change color of both balls to white. Otherwise, leave color gray.
- Display balls.

Certainly the hard work here is the intersection test, which I will get to in a moment. First, here is what I need for a simple bouncing `Ball` class without an intersection test.

Data:

- x and y location.
- Radius.
- Speed in x and y directions.

Functions:

- Constructor.
  — Set radius based on argument.
  — Pick random location.
  — Pick random speed.
- Move.
  — Increment x by speed in x direction.
  — Increment y by speed in y direction.
  — If ball hits any edge, reverse direction.
- Display.
  — Draw a circle at x and y location.

I am now ready to translate this into code.

Example 10-2. Bouncing ball class

```
class Ball {
  float r;              // radius
  float x, y;           // location
  float xspeed, yspeed; // speed

  Ball(float tempR) {

    r = tempR;


    x = random(width);
    y = random(height);

    xspeed = random(-5, 5);
    yspeed = random(-5, 5);
  }

  void move() {

    x += xspeed;
    y += yspeed;

    // Check horizontal edges
    if (x > width || x < 0) {
      xspeed *= -1;
    }

    // Check vertical edges
    if (y > height || y < 0) {
      yspeed *= -1;
    }
  }

  // Draw the ball
  void display() {
    stroke(0);
    fill(0, 50);
    ellipse(x, y, r*2, r*2);
  }
}
```

Set radius based on argument.

Pick random location.

Pick random speed.

Increment x and y speed.

If ball hits any edge, reverse direction.

Draw a circle at location.

From here, it's pretty easy to create a sketch with two ball objects. Ultimately, in the final sketch, I'll need an array for many raindrops, but for now, two ball variables will be simpler.

Example 10-3. Two ball objects

```
// Two Ball variables
Ball ball1;
Ball ball2;

void setup() {
  size(400, 400);
  // Initialize Ball objects
  ball1 = new Ball(64);
  ball2 = new Ball(32);
}

void draw() {
  background(255);
  // Move and display Ball objects
  ball1.move();
  ball2.move();
  ball1.display();
  ball2.display();
}
```

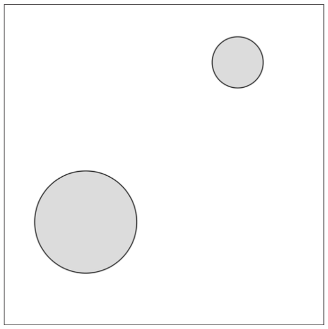


Figure 10-2

Now that I have set up a system for having two circles moving around the screen, I need to develop an algorithm for determining if the circles intersect. In Processing, the `dist()` function calculates the distance between two points (see Section 7-7 on page 132). I also have access to the radius of each circle (the variable `r` inside each object). The diagram in Figure 10-3 shows how I can compare the distance between the circles and the sum of the radii to determine if the circles overlap.

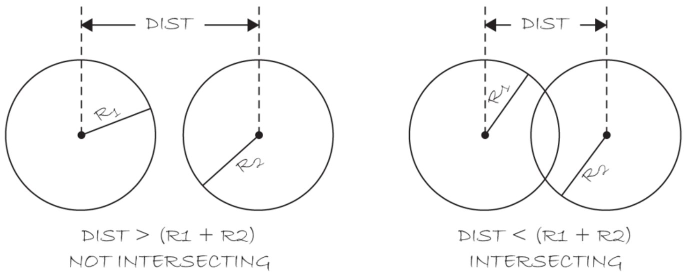


Figure 10-3

OK, so assuming the following:

- x_1,y_1: coordinates of circle one
- x_2,y_2: coordinates of circle two
- r_1: radius of circle one
- r_2: radius of circle two

I have the statement:

> If the distance between (x_1,y_1) and (x_2,y_2) is less than the sum of r_1 and r_2, circle one intersects circle two.

My job now is to write a function that returns true or false based on the above statement.

```
// A function that returns true or false based on whether two circles intersect
boolean intersect(float x1, float y1, float x2, float y2, float r1, float r2) {
  // Calculate distance
  float distance = dist(x1, y2, x2, y2);
  if (distance < r1 + r2) {
    return true;
  } else {
    return false;
  }
}
```

If distance is less than the sum of radii, the circles touch.

Now that the function is complete, I can test it with data from `ball1` and `ball2`.

```
boolean intersecting = intersect(ball1.x,ball1.y,ball2.x,ball2.y,ball1.r,ball2.r);
if (intersecting) {
  println("The circles are intersecting!");
}
```

The above code is awkward due to the number of arguments to `intersect()` and it is useful to go one step further, incorporating the intersection test into the `Ball` class itself. Let's first look at the entire main program as it stands.

```
// Two ball variables
Ball ball1;
Ball ball2;

void setup() {
  size(400, 400);
  // Initialize Ball objects
  ball1 = new Ball(64);
  ball2 = new Ball(32);
}

void draw() {
  background(255);
  // Move and display Ball objects
  ball1.move();
  ball2.move();
  ball1.display();
  ball2.display();
  boolean intersecting = intersect(ball1.x,ball1.y,ball2.x,ball2.y,ball1.r,ball2.r);
  if (intersecting) {
    println("The circles are intersecting!");
  }
}

// A function that returns true or false based on whether two circles intersect
// If distance is less than the sum of radii the circles touch
boolean intersect(float x1, float y1, float x2, float y2, float r1, float r2) {
  float distance = dist(x1, y2, x2, y2);    // Calculate distance
  if (distance < r1 + r2) {                 // Compare distance to r1 + r2
    return true;
  } else {
    return false;
  }
}
```

Since I have programmed the balls in an object-oriented fashion, it's not terribly logical to suddenly have an `intersect()` function that lives outside of the ball class. A ball object should know how to test if it is intersecting another ball object. The code can be improved by incorporating the intersect logic into the class itself, saying `ball1.intersect(ball2);` or, "Does ball one intersect ball two?"

```
void draw() {
  background(255);
  // Move and display Ball objects
  ball1.move();
  ball2.move();
  ball1.display();
  ball2.display();
  boolean intersecting = ball1.intersect(ball2);
  if (intersecting) {
    println("The circles are intersecting!");
  }
}
```

Assumes a function `intersect()` inside the `Ball` class that returns true or false.

Following this model and the algorithm for testing intersection, here is a function for inside the ball class itself. Notice how the function makes use of both its own location (`x` and `y`) as well as the other ball's location (`b.x` and `b.y`).

```
// A function that returns true or false based on whether two Ball objects intersect
boolean intersect(Ball b) {
  float distance = dist(x, y, b.x, b.y);
  if (distance < r + b.r) {
    return true;
  } else {
    return false;
  }
}
```

The function is testing this object's position and radius vs the position and radius for `Ball b`.

Putting it all together, I have the code in Example 10-4.

Example 10-4. Bouncing ball with intersection

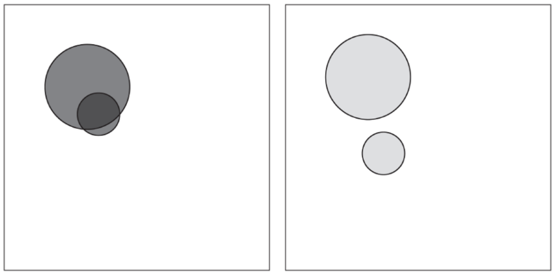


Figure 10-4

```
// Two ball variables
Ball ball1;
Ball ball2;

void setup() {
  size(400, 400);
  // Initialize Ball objects
  ball1 = new Ball(64);
  ball2 = new Ball(32);
}

void draw() {
  background(255);
  // Move and display Ball objects
  ball1.move();
  ball2.move();
  if (ball1.intersect(ball2)) {
    ball1.highlight();
    ball2.highlight();
  }
  ball1.display();
  ball2.display();
}

class Ball {
  float r; // radius
  float x, y;
  float xspeed, yspeed;
  color c = color(100, 50);

  // Constructor
  Ball(float tempR) {
    r = tempR;
    x = random(width);
    y = random(height);
    xspeed = random(-5, 5);
    yspeed = random(-5, 5);
  }

  void move() {
    x += xspeed;   // Increment x
    y += yspeed;   // Increment y

    // Check horizontal edges
    if (x > width || x < 0) {
      xspeed *= -1;
    }
    // Check vertical edges
    if (y > height || y < 0) {
      yspeed *= -1;
    }
  }

  void highlight() {
    c = color(0, 150);
  }
```

New! An object can have a function that takes another object as an argument. This is one way to have objects communicate. In this case they are checking to see if they intersect.

Whenever the circles are touching, this `highlight()` function is called and the color is darkened.

```
    // Draw the ball
    void display() {
      stroke(0);
      fill(c);
      ellipse(x ,y, r*2, r*2);
      c = color(100, 50);
    }

    // A function that returns true or false based on whether two circles intersect
    // If distance is less than the sum of radii the circles touch
    boolean intersect(Ball b) {
      float distance = dist(x, y, b.x, b.y); // Calculate distance
      if (distance < r + b.r) {              // Compare distance
        return true;                         // to sum of radii
      } else {
        return false;
      }
    }
  }
```

After the ball is displayed, the color is reset back to a darker gray.

Objects can be passed into functions as arguments too!

10-6 Part 3: The timer

Our next task is to develop a timer that executes a function every *N* seconds. Again, we will do this in two steps, first just using the main body of a program and second, taking the logic and putting it into a `Timer` class. Processing has the functions `hour()`, `second()`, `minute()`, `month()`, `day()`, and `year()` to deal with time. I could conceivably use the `second()` function to determine how much time has passed. However, this is not terribly convenient, since `second()` rolls over from 60 to 0 at the end of every minute.

For creating a timer, the function `millis()` is best. First of all, `millis()`, which returns the number of milliseconds since a sketch started, allows for a great deal more precision. One millisecond is one one-thousandth of a second (1,000 ms = 1 s). Secondly, `millis()` never rolls back to zero, so asking for the milliseconds at one moment and subtracting it from the milliseconds at a later moment will always result in the amount of time passed.

Let's say I want a sketch to change the background color to red five seconds after it started. Five seconds is 5,000 ms, so it's as easy as checking if the result of the `millis()` function is greater than 5,000.

```
if (millis() > 5000) {
  background(255, 0, 0);
}
```

Making the problem a bit more complicated, I can expand the program to change the background to a new random color every 5 seconds.

Setup:

- Save the time at startup (note this should always be zero, but it's useful to save it in a variable anyway). Call this `savedTime`.

Draw:

- Calculate the time passed as the current time (i.e., `millis()`) minus `savedTime`. Save this as `passedTime`.
- If passedTime is greater than 5,000, fill a new random background and *reset* `savedTime` *to the current time*. This step will restart the timer.

Example 10-5 translates this into code.

Example 10-5. Implementing a timer

```
int savedTime;
int totalTime = 5000;

void setup() {
  size(200, 200);
  background(0);
  savedTime = millis();  <  Save the time.
}

void draw() {
  int passedTime = millis() - savedTime;  <  Calculate how much time has passed.

  if (passedTime > totalTime) {  <  Has five seconds passed?

    println("5 seconds have passed!");
    background(random(255)); // Color a new background
    savedTime = millis();    // Save the current time to restart the timer!
  }
}
```

With the above logic worked out, I can now move the timer into a class. Let's think about what data is involved in the timer. A timer must know the time at which it started (`savedTime`) and how long it needs to run (`totalTime`).

Data:

- `savedTime`
- `totalTime`

The timer must also be able to *start* as well as check and see if it *is finished*.

Functions:

- `start()`
- `isFinished()`—returns true or false

Taking the code from the non-object-oriented example and building it out with the above structure, I have the code shown in Example 10-6.

Example 10-6. Object-oriented timer

```
Timer timer;

void setup() {
  size(200, 200);
  background(0);
  timer = new Timer(5000);
  timer.start();
}

void draw() {
  if (timer.isFinished()) {
    background(random(255));
    timer.start();
  }
}

class Timer {
  int savedTime;  // When Timer started
  int totalTime;  // How long Timer should last

  Timer(int tempTotalTime) {
    totalTime = tempTotalTime;
  }

  // Starting the timer
  void start() {
    savedTime = millis();
  }

  boolean isFinished() {
    // Check how much time has passed
    int passedTime = millis() - savedTime;
    if (passedTime > totalTime) {
      return true;
    } else {
      return false;
    }
  }
}
```

When the timer starts it stores the current time in milliseconds.

The function `isFinished()` returns true if 5,000 ms have passed. The work of the timer is farmed out to this method.

10-7 Part 4: Raindrops

I'm almost there. I have created a `Catcher` class, I know how to test for intersection, and I have completed the `Timer` class. The final piece of the puzzle is the raindrops themselves. Ultimately, I want an array of `Drop` objects falling from the top of the window to the bottom. Since this step involves creating an array of objects that move, it's useful to approach this fourth part as a series of even smaller steps, subparts of Part 4, thinking again of the individual elements and behaviors I will need.

Part 4 Subparts:

Part 4.1: A single moving raindrop.

Part 4.2: An array of raindrop objects.

Part 4.3: Flexible number of raindrops (appearing one at a time).

Part 4.4: Fancier raindrop appearance.

Part 4.1, creating the motion of a raindrop (a simple circle for now) is easy — Chapter 3 easy.

- Increment raindrop's y position.
- Display raindrop.

Translating into code, I have ***Part 4.1 — A single moving raindrop***, shown in Example 10-7.

Example 10-7. Simple raindrop behavior

```
float x, y; // Variables for drop location

void setup() {
  size(400, 400);
  x = width/2;
  y = 0;
}

void draw() {
  background(255);
  // Display the drop
  fill(50, 100, 150);
  noStroke();
  ellipse(x, y, 16, 16);
  // Move the drop
  y++;
}
```

Again, however, I need to go a step further and make a `Drop` class — after all I will ultimately want an array of drops. In making the class, I can add a few more variables, such as speed and size, as well as a function to test if the raindrop reaches the bottom of the screen, which will be useful later for scoring the game.

```
class Drop {

  float x, y;    // Variables for location of raindrop
  float speed;   // Speed of raindrop
  color c;
  float r;       // Radius of raindrop

  Drop() {
    r = 8;                      // All raindrops are the same size
    x = random(width);          // Start with a random x location
    y = -r*4;                   // Start a little above the window
    speed = random(1, 5);       // Pick a random speed
    c = color(50, 100, 150); // Color
  }

  // Move the raindrop down
  void move() {
    y += speed;
  }

  // Check if it hits the bottom
  boolean reachedBottom() {
    if (y > height + r*4) {
      return true;
    } else {
      return false;
    }
  }

  // Display the raindrop
  void display() {
    fill(50, 100, 150);
    noStroke();
    ellipse(x, y, r*2, r*2);
  }
}
```

A raindrop object has a location, speed, color, and size.

Incrementing `y` is now in the `move()` function.

In addition, I have a function that determines if the drop leaves the window.

Before I move on to Part 4.3, the array of drops, I should make sure that a singular `Drop` object functions properly. As an exercise, complete the code in Exercise 10-3 on page 206 that would test a single drop object.

Exercise 10-3: Fill in the blanks below completing the "test drop" sketch.

```
Drop drop;

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

  _________________________________
}

void draw() {
  background(255);

  drop.____________________________

  _________________________________
}
```

Now that this is complete, the next step is to go from one drop to an array of drops — ***Part 4.2***. This is exactly the technique you perfected in Chapter 9.

```
// An array of drops
Drop[] drops = new Drop[50];

void setup() {
  size(400, 400);
  // Initialize all drops
  for (int i = 0; i < drops.length; i++) {
    drops[i] = new Drop();
  }
}

void draw() {
  background(255);
  // Move and display all drops
  for (int i = 0; i < drops.length; i++) {
    drops[i].move();
    drops[i].display();
  }
}
```

Instead of one `Drop` object, an array of 50.

Using a loop to initialize all drops.

Move and display all drops.

The problem with the above code is that the raindrops appear all at once. According to the specifications I made for the game, the raindrops should appear one at a time, every *N* seconds — I am now at ***Part 4.3 — Flexible number of raindrops (appearing one at a time)***. I can skip worrying about the timer for now and just have one new raindrop appear every frame. I should also make the array much larger, allowing for many more raindrops.

To make this work, I need a new variable to keep track of the total number of drops — `totalDrops`. Most array examples involve walking through the entire array in order to deal with the entire list. Now, I want to access a portion of the list, the number stored in `totalDrops`. Let's write some pseudocode to describe this process:

Setup:

- Create an array of drops with 1,000 spaces in it.
- Set `totalDrops` equal to 0.

Draw:

- Create a new drop in the array (at the index `totalDrops`). Since `totalDrops` starts at 0, I will first create a new raindrop in the first spot of the array.
- Increment `totalDrops` (so that the next time, a drop is created in the next spot in the array).
- If `totalDrops` exceeds the array size, reset it to zero and start over.
- Move and display all available drops (i.e., `totalDrops`).

Example 10-8 translates the above pseudocode into code.

Example 10-8. Drops one at a time

```
// An array of drops
Drop[] drops = new Drop[1000];

int totalDrops = 0;

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

void draw() {
  background(255);

  // Initialize one drop
  drops[totalDrops] = new Drop();
  // Increment totalDrops
  totalDrops++ ;
  // If totalDrops hits the end of the array
  if (totalDrops >= drops.length) {
    totalDrops = 0; //Start over
  }

  // Move and display drops
  for (int i = 0; i < totalDrops; i++) {
    drops[i].move();
    drops[i].display();
  }
}
```

New variable to keep track of total number of drops!

New! All drops are no longer displayed, rather only the `totalDrops` that are currently present in the game are shown.

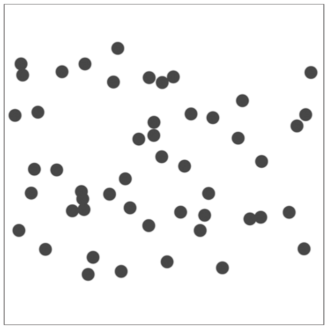


Figure 10-5

I have taken the time to figure out how the raindrop moves, created a class that exhibits that behavior, and made an array of objects from that class. All along, however, I have just been using a circle to display the drop. The advantage to this is that I was able to delay worrying about the code required for visual design and focus on the motion behaviors and organization of data and functions. Now I can focus on how the drops look — ***Part 4.4 — Finalize raindrop appearance.***

One way to create a more "drop-like" look is to draw a sequence of circles in the vertical direction, starting small, and getting larger as they move down.

Example 10-9. Fancier looking raindrop

```
background(255);
for (int i = 2; i < 8; i++) {
  noStroke();
  fill(0);
  ellipse(width/2, height/2 + i*4, i*2, i*2);
}
```

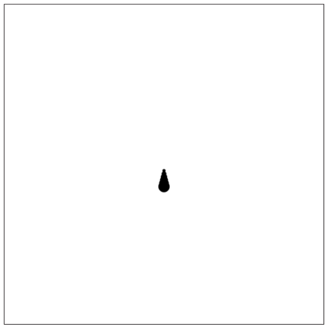


Figure 10-6

I can incorporate this algorithm in the `Drop` class from Example 10-8, using `x` and `y` for the start of the ellipse locations, and the raindrop radius as the maximum value for `i` in the `for` loop. The output is shown in Figure 10-7.

```
void display() {
  noStroke();
  fill(c);
  for (int i = 2; i < r; i++) {
    ellipse(x, y + i*4, i*2, i*2);
  }
}
```

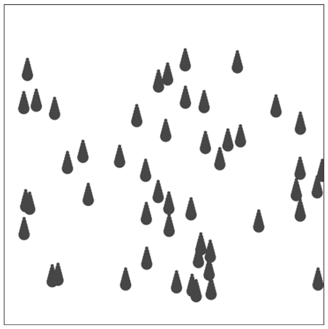
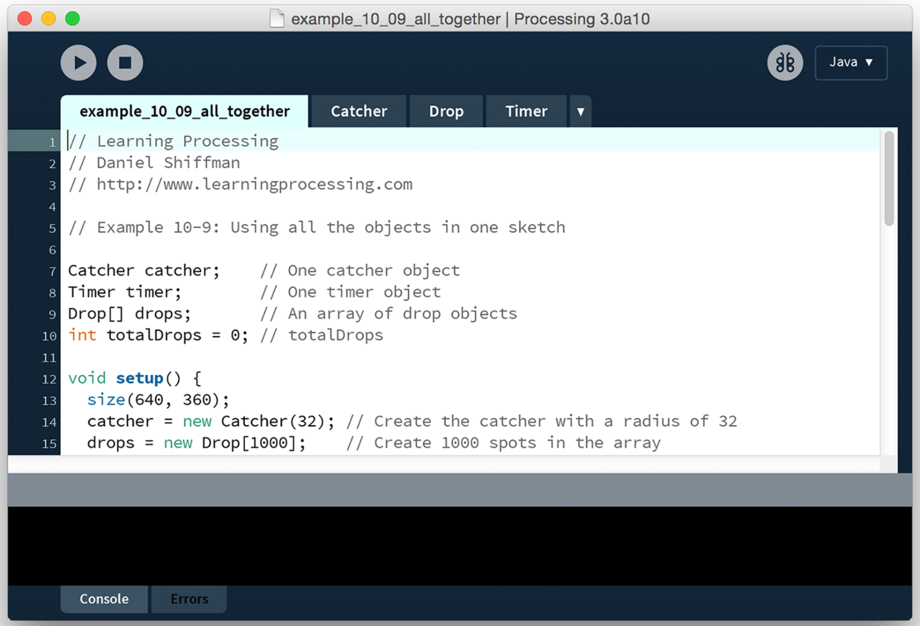


Figure 10-7

10-8 Integration: Puttin' on the ritz

It is time. Now that I have developed the individual pieces and confirmed that each one works properly, I can assemble them together in one program. The first step is to create a new Processing sketch that has four tabs, one for each of the three classes and one main program, as shown in Figure 10-8.


example_10_09_all_together | Processing 3.0a10
Java
example_10_09_all_together
Catcher
Drop
Timer
Console
Errors

```
// Learning Processing
// Daniel Shiffman
// http://www.learningprocessing.com

// Example 10-9: Using all the objects in one sketch

Catcher catcher;    // One catcher object
Timer timer;        // One timer object
Drop[] drops;       // An array of drop objects
int totalDrops = 0; // totalDrops

void setup() {
  size(640, 360);
  catcher = new Catcher(32); // Create the catcher with a radius of 32
  drops = new Drop[1000];    // Create 1000 spots in the array
```

Figure 10-8

The first step is to copy and paste the code for each class into each tab. Individually, they will not need to change, so there is no need to revisit the code. What I do need to revisit is the main program — what goes in `setup()` and `draw()`. Referring back to the original game description and knowing how the pieces were assembled, I can write the pseudocode algorithm for the entire game.

Setup:

- Create `Catcher` object.
- Create array of `Drop` objects.
- Set `totalDrops` equal to 0.
- Create `Timer` object.
- Start timer.

Draw:

- Set catcher location to mouse location.
- Display catcher.
- Move all available drops.
- Display all available drops.
- If the catcher intersects any drop.
  — Remove drop from screen.
- If the timer is finished:
  — Increase the number of drops.
  — Restart timer.

Notice how every single step in the above program has already been worked out previously in the chapter with one exception: "Remove drop from screen." This is rather common. Even with breaking the idea down into parts and working them out one at a time, little bits can be missed. Fortunately, this piece of functionality is simple enough and with some ingenuity, you will see how I can slip it in during assembly.

One way to approach assembling the above algorithm is to start by combining all of the elements into one sketch and not worrying about how they interact. In other words, everything but having the timer trigger the drops and testing for intersection. To get this going, all I need to do is copy/paste from each part's global variables, `setup()` and `draw()`!

Here are the global variables: a `Catcher` object, an array of `Drop` objects, a `Timer` object, and an integer to store the total number of drops.

```
Catcher catcher;      // One catcher object
Timer timer;          // One timer object
Drop[] drops;         // An array of drop objects

int totalDrops = 0;   // totalDrops
```

In `setup()`, the variables are initialized. Note, however, I can skip initializing the individual drops in the array since they will be created one at a time. I will also need to call the timer's `start()` function.

```
void setup() {
  size(400, 400);

  catcher = new Catcher(32);       // Create the catcher with a radius of 32
  drops = new Drop[1000];          // Create 1000 spots in the array
  timer = new Timer(2000);         // Create a timer that goes off every 2 seconds
  timer.start();                   // Starting the timer
}
```

In `draw()`, the objects call their methods. Again, I am just taking the code from each part I did separately earlier in this chapter and pasting in sequence.

Example 10-10. Using all the objects in one sketch

```
Catcher catcher;      // One catcher object
Timer timer;          // One timer object
Drop[] drops;         // An array of drop objects

int totalDrops = 0;   // totalDrops

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

  catcher = new Catcher(32);  // Create the catcher with a radius of 32
  drops = new Drop[1000];     // Create 1000 spots in the array
  timer = new Timer(2000);    // Create a timer that goes off every 2 seconds

  timer.start();              // Starting the timer
}

void draw() {
  background(255);

  // Set catcher location
  catcher.setLocation(mouseX, mouseY);
  // Display the catcher
  catcher.display();

  // Check the timer
  if (timer.isFinished()) {
    println("2 seconds have passed!");
    timer.start();
  }

  // Initialize one drop
  drops[totalDrops] = new Drop();
  // Increment totalDrops
  totalDrops++;
  // If totalDrops hit the end of the array
  if (totalDrops >= drops.length) {
    totalDrops = 0; // Start over
  }

  // Move and display all drops
  for (int i = 0; i < totalDrops; i++) {
    drops[i].move();
    drops[i].display();
  }

}
```

From Part 1. The Catcher!

From Part 3. The Timer!

From Part 4. The Raindrops!

The next step is to take these concepts I have developed and have them work together. For example, a new raindrop should only be created when two seconds have passed (as indicated by the timer's `isFinished()` function).

```
// Check the timer
if (timer.isFinished()) {

  // Initialize one drop
  drops[totalDrops] = new Drop();
  // Increment totalDrops
  totalDrops++;
  // If totalDrops hit the end of the array
  if (totalDrops >= drops.length) {
    totalDrops = 0; // Start over
  }
  timer.start();
}
```

Concepts working together! Here when the timer "is finished," a `Drop` object is added (by incrementing `totalDrops`).

I also need to find out when the `Catcher` object intersects a drop. In Section 10-5 on page 195, I tested for intersection by calling the `intersect()` function inside the `Ball` class.

```
boolean intersecting = ball1.intersect(ball2);
if (intersecting) {
  println("The circles are intersecting!");
}
```

I can do the same thing here, calling an `intersect()` function in the `Catcher` class and passing through every raindrop in the system. Instead of printing out a message, I want to affect the raindrop itself, telling it to disappear, perhaps. This code assumes that the `caught()` function will do the job.

```
// Move and display all drops
for (int i = 0; i < totalDrops; i++) {
  drops[i].move();
  drops[i].display();
  if (catcher.intersect(drops[i])) {
    drops[i].caught();
  }
}
```

Concepts working together! Here, the `Catcher` object checks to see if it intersects any `Drop` object in the `drops` array.

Our `Catcher` class did not originally contain the function `intersect()`, nor did the `Drop` class include `caught()`. So these are new functions I will need to write as part of the integration process.

`intersect()` is easy to incorporate; that problem was solved already in Section 10-5 on page 195 and can be copied into the `Catcher` class (changing the parameter from a `Ball` object to a `Drop` object).

```
// A function that returns true or false based if the catcher intersects a raindrop
boolean intersect(Drop d) {
  // Calculate distance
  float distance = dist(x, y, d.x, d.y);
  // Compare distance to sum of radii
```

```
    if (distance < r + d.r) {
      return true;
    } else {
      return false;
    }
  }
```

In addition to calling functions, variables inside of an object can be accessed using the dot syntax.

When the drop is caught, I can set its location to somewhere offscreen (so that it can't be seen, the equivalent of "disappearing") and stop it from moving by setting its speed equal to 0. Although I did not work out this functionality in advance of the integration process, it's simple enough to throw in right now.

```
  // If the drop is caught
  void caught() {
    speed = 0;     // Stop it from moving by setting speed equal to zero
    y = -1000;     // Set the location to somewhere way off-screen
  }
```

And it's finished! For reference, Example 10-11 is the entire sketch. The timer is altered to execute every 300 ms, making the game ever so slightly more difficult.

Example 10-11. The raindrop catching game

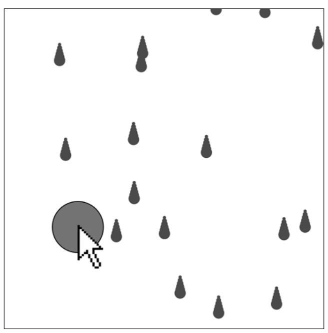


Figure 10-9

```
Catcher catcher;        // One catcher object
Timer timer;            // One timer object
Drop[] drops;           // An array of drop objects

int totalDrops = 0;     // totalDrops
void setup() {
  size(400, 400);

  // Create the catcher with a radius of 32
  catcher = new Catcher(32);
  // Create 1000 spots in the array
  drops = new Drop[1000];
  // Create and start a timer that goes off every 300
milliseconds
  timer = new Timer(300);
  timer.start();
}

void draw() {
  background(255);
  catcher.setLocation(mouseX, mouseY);   // Set catcher location
  catcher.display();                    // Display the catcher

  // Check the timer
  if (timer.isFinished()) {
    // Initialize one drop
    drops[totalDrops] = new Drop();
    // Increment totalDrops
    totalDrops++ ;
    // If totalDrops hit the end of the array
    if (totalDrops >= drops.length) {
      totalDrops = 0; // Start over
    }
    timer.start();
  }

  // Move and display all drops
  for (int i = 0; i < totalDrops; i++) {
    drops[i].move();
    drops[i].display();
    if (catcher.intersect(drops[i])) {
      drops[i].caught();
    }
  }
}

class Catcher {
  float r;     // radius
  color col;   // color
  float x, y;   // location

  Catcher(float tempR) {
    r = tempR;
    col = color(50, 10, 10, 150);
    x = 0;
    y = 0;
  }

  void setLocation(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }

  void display() {
    stroke(0);
    fill(col);
    ellipse(x, y, r*2, r*2);
  }

  // Returns true if the catcher intersects a raindrop, otherwise false
  boolean intersect(Drop d) {
    float distance = dist(x, y, d.x, d.y);     // Calculate distance
    if (distance < r + d.r) {                  // Compare distance to sum of radii
      return true;
    } else {
      return false;
    }
  }

}

class Drop {

  float x, y;        // Variables for location of raindrop
  float speed;       // Speed of raindrop
  color c;
  float r;           // Radius of raindrop

  Drop() {
    r = 8;                      // All raindrops are the same size
    x = random(width);        // Start with a random x location
    y = -r*4;                 // Start a little above the window
    speed = random(1, 5);     // Pick a random speed
    c = color(50, 100, 150);  // Color
  }

  // Move the raindrop down
  void move() {
    y += speed; // Increment by speed
  }

  // Display the raindrop
  void display() {
    // Display the drop
    fill(c);
    noStroke();
    for (int i = 2; i < r; i++) {
     ellipse(x, y+i*4, i*2, i*2);
    }
  }

  // If the drop is caught
  void caught() {
    speed = 0; // Stop it from moving by setting speed equal to zero
    y = -1000; // Set the location to somewhere way off-screen
  }
}

class Timer {

  int savedTime; // When Timer started
  int totalTime; // How long Timer should last

  Timer(int tempTotalTime) {
    totalTime = tempTotalTime;
  }

  // Starting the timer
  void start() {
    savedTime = millis();
  }

  boolean isFinished() {
    // Check out much time has passed
    int passedTime = millis()- savedTime;
    if (passedTime > totalTime) {
      return true;
    } else {
      return false;
    }
  }

}
```

Exercise 10-4: Implement a scoring system for the game. Start the player off with 10 points. For every raindrop that reaches the bottom, decrease the score by 1. If all 1,000 raindrops fall without the score getting to zero, a new level begins and the raindrops appear faster. If 10 raindrops reach the bottom during any level, the player loses. Show the score onscreen as a rectangle that decreases in size. Do not try to implement all of these features at once. Do them one step at a time! Following is a clue to get you started, a function for the `Drop` *class to determine if the* `Drop` *object has reached the bottom of the window.*

```
boolean reachedBottom() {
  // If the drop goes a little beyond the bottom
  if (y > height + r*4) {
    return true;
  } else {
    return false;
  }
}
```

10-9 Getting ready for Act II

The point of this chapter is not to learn how to program a game of catching falling raindrops, rather it's to develop an approach to problem solving — taking an idea, breaking it down into parts, developing pseudocode for those parts, and implementing them one very small step at a time.

It's important to remember that getting used to this process takes time and it takes practice. Everyone struggles through it when first learning to program.

Before you embark on the rest of this book, let's take a moment to consider what you have learned and where you are headed. In these 10 chapters, I have focused entirely on the fundamentals of programming:

- **Data** — in the form of variables and arrays.
- **Control Flow** — in the form of conditional statements and loops.
- **Organization** — in the form of functions and objects.

These concepts are not unique to Processing and will carry you through to any and all programming languages and environments, such as C++, Python, JavaScript, and more. The syntax may change, but the fundamental concepts will not.

Starting with Chapter 13, the book will focus on some advanced concepts available in Processing, such as three-dimensional translation and rotation, image processing and video capture, networking, and sound. Although these concepts are certainly not unique to Processing, the details of their implementation will be more specific to this particular environment.

Before I move on to these advanced topics I will take a quick look at basic strategies for fixing errors in your code (Chapter 11: Debugging) as well as how to use Processing libraries (Chapter 12). Many of these advanced topics require importing libraries that come with Processing as well as libraries made by third parties. One of the strengths of Processing is its ability to be easily extended with libraries. You will read some hints of how to create your own libraries in the final chapter of this book.

Onward, ho!

Lesson Five Project

1. Develop an idea for a project that can be created with Processing using simple shape drawing and the fundamentals of programming. If you feel stuck, try making a game such as Pong or Tic-Tac-Toe.
2. Follow the strategy outlined in this chapter and break the idea down into smaller parts, implementing the algorithm for each one individually. Make sure to use object-oriented programming for each part.
3. Bring the smaller parts together in one program. Did you forget any elements or features?

Use the space provided below to sketch designs, notes, and pseudocode for your project.