Structure 4: Objects I

This unit introduces the concept of object-oriented programming and presents the code elements for working with objects.

Syntax introduced:
`class, Object`

Variables and functions are the building blocks of software. Several functions will often be used together to work on a set of related variables. Object-oriented programming, which was developed to make this process more explicit, uses objects and classes as building blocks. A class defines a group of methods (functions) and fields (variables). An object is a single instance of a class. The fields within an object are typically accessible only via its own methods, allowing an object to hide its complexity from other parts of a program. This resembles interfaces built for other complex technologies; the driver of a car does not see the complexity of the engine while in motion, although the speed and RPM are readily visible on the console. The same type of abstraction is used in object-oriented programming to make code easier to understand and reuse in other contexts.

Object-oriented programming is a different way of thinking about programming, but it builds on the previously introduced concepts. The technology for object-oriented programming existed long before the practice became popular in the mid-1980s and gradually went on to become the dominant way to think about software. Many people find it to be a more intuitive way to think about programming. In addition to providing a helpful conceptual model, object-oriented programming becomes a necessity when a program includes many elements or when it grows larger than a few pages of code. Objects can provide a powerful way to think about structuring your ideas in code, and you'll find a number of examples with objects throughout the rest of this text.

All software written in Processing consists of objects, but this fact is initially hidden so that object-oriented programming concepts can be introduced later. Unless code is made explicitly object-oriented, clicking the Run button transparently adds extra syntax that wraps a sketch as an object.

Object-oriented programming

A modular program is composed of code modules that each perform a specific task. Variables are the most basic way to think about reusing elements within a program. They allow a single value to appear many times within a program and to be easily changed. Functions abstract a specific task and allow code blocks to be reused throughout a program. Typically, one is concerned only with what a function does, not how it works. This frees the mind to focus on the goals of the program rather than on

the complexities of infrastructure. Object-oriented programming further extends the modularity of using variables and writing functions by allowing related functions to be grouped together.

It's possible to make an analogy between software objects and real-world artifacts. To get you in the spirit of thinking about the world through the object-oriented lens, we've created a list of everyday items and a few potential fields and methods for each.

```
Name      Apple
Fields    color, weight
Methods   grow(), fall(), rot()

Name      Butterfly
Fields    species, gender
Methods   flapWings(), land()

Name      Radio
Fields    frequency, volume
Methods   turnOn(), tune(), setVolume()

Name      Car
Fields    make, model, color, year
Methods   accelerate(), brake(), turn()
```

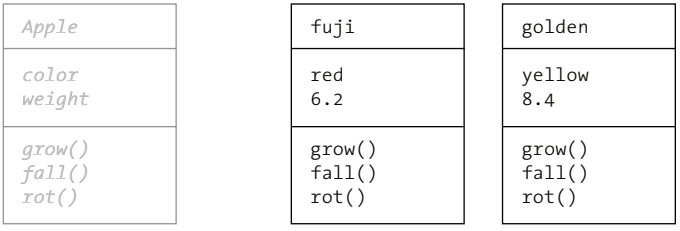
Extending the apple example reveals more about the process of thinking about the world in relation to software objects. To make a software simulation of the apple, the `grow()` method might have inputs for temperature and moisture. The `grow()` method can increase the weight field of the apple based on these inputs. The `fall()` method can continually check the weight and cause the apple to fall to the ground when the weight goes above a threshold. The `rot()` method could then take over, beginning to decrease the value of the weight field and change the color fields.

As explained in the introduction, objects are created from a class and a class describes a set of fields and methods. An instance of a class is a variable, and like other variables, it must have a unique name. If more than one object is created from a class, each must have a unique name. For example, if two objects named `fuji` and `golden` are created using the `Apple` class, each can have its own values for its fields:

```
Name      fuji
Fields    color: red
          weight: 6.2

Name      golden
Fields    color: yellow
          weight: 8.4
```

Two popular styles of class diagrams are tables and a circular format inspired by biological cells. Each diagram style shows the name of the class, the fields, and the methods. It is useful to diagram classes in this way to define their characteristics before starting to code. The diagrams are also useful because they show the components of a

class without including too much detail. Looking at the `Apple` class and the `fuji` and `golden` objects created from it, you can see how these diagrams work:



Apple class fuji object golden object

The circular diagrams reinforce *encapsulation*, the idea that an object's fields should not be accessible from the outside. The methods of an object should act as a buffer between code outside the class and the data contained within:

Apple class fuji object golden object

The fields and methods of an object are accessed with the dot operator, a period. To get the color value from the `fuji` object, the syntax `fuji.color` accesses the value of the color field inside the `fuji` object. The syntax `golden.color` accesses the value of the color field inside the golden object. The dot operator is also used to activate (or "call") the methods of the object. To run the `grow()` method inside the golden object, the syntax `golden.grow()` is used.

With the concepts and terminology discussed in this unit (object, class, field, method, encapsulation, and dot operator), you are equipped to begin the journey into object-oriented programming, which is explained further in Structure 5 (p. 453).

Using classes and objects

Defining a class is creating your own data type. Unlike the primitive types `int`, `float`, and `boolean`, it's a composite type like `String`, `PImage`, and `PFont`, which means it can hold many variables and methods inside one name. When creating a class, first think carefully about what you want the code to do. It's common to make a list of variables required (these will be the fields) and figure out what type they should be.

The code on the following pages creates the same image of a white dot on the black background, but the code is written in different ways. In the first example program, a circle is positioned on screen. It needs two fields to store the location. These variables are `float` values that will provide more flexibility to control the circle's movement. The circle also needs a size, so we've created the `diameter` field to store its diameter:

`float x`	*X-coordinate of the circle*
`float y`	*Y-coordinate of the circle*
`float diameter`	*Diameter of the circle*

The name of a class should be carefully considered. The name can be nearly any word, adhering to the same naming conventions as variables (p. 40); however, class names should always be capitalized. This helps separate a class like `String` or `PImage` from the lowercase names of primitive types like `int` or `boolean`. The name `Spot` was chosen for this example because a spot is drawn to the screen (the name "Circle" also would have made sense). As with variables, it can be very helpful to give a class a name that matches its purpose.

Once the fields and name for the class definition have been determined, consider how the program would be written without the use of an object. In the following example, the data for the ellipse's position and diameter is a part of the main program. In this case, that's not a problem, but the use of several ellipses or complex motion would make the program unwieldy.

43-01

```
float x = 33;
float y = 50;
float diameter = 30;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(0);
  ellipse(x, y, diameter, diameter);
}
```

To make this code more generally useful, the next example moves the fields that pertain to the ellipse into their own class. The first line in the program declares the object `sp` of the type `Spot`. The `Spot` class is defined after the `setup()` and `draw()`. The `sp` object is constructed within `setup()`, after which its fields can be accessed and assigned. The next three lines assign values to the fields within `Spot`. These values are accessed inside `draw()` to set the position and size of the ellipse. The dot operator is used to assign and access the variables within the class.

```
Spot sp;              // Declare the object                    43-02

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot();    // Construct the object
  sp.x = 33;          // Assign 33 to the x field
  sp.y = 50;          // Assign 50 to the y field
  sp.diameter = 30;   // Assign 30 to the diameter field
}

void draw() {
  background(0);
  ellipse(sp.x, sp.y, sp.diameter, sp.diameter);
}

class Spot {
  float x, y;         // The x- and y-coordinate
  float diameter;     // Diameter of the circle
}
```

The `Spot` class as it exists is not very useful, but it's a start. This next example builds on the previous one by adding a method to the `Spot` class—this is one more step toward using object-oriented programming to its advantage. The `display()` method has been added to the class definition to draw the element to the screen:

`void display()` ***Draws the spot to the display window***

In the code below, the last line inside `draw()` runs the `display()` method for the `sp` object by writing the name of the object and the name of the method connected with the dot operator. Also notice the difference in the parameters of the ellipse function in code 43-02 and 43-03. In code 43-03, the name of the object is not used to access the `x`, `y`, and `diameter` fields. This is because the `ellipse()` function is called from within the `Spot` object. Because this line is a part of the object's `display()` function, it can access its own variables without specifying its own name.

It's important to reinforce the difference between the `Spot` class and the `sp` object in this example. Although the code might make it look like the fields `x`, `y`, and `diameter` and the method `display()` belong to `Spot`, this is just the definition for any object created from this class. Each of these elements belong to (are encapsulated by) the `sp` variable, which is one instance of the Spot data type.

43-03

```
Spot sp;            // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot();  // Construct the object
  sp.x = 33;
  sp.y = 50;
  sp.diameter = 30;
}

void draw() {
  background(0);
  sp.display();
}

class Spot {
  float x, y, diameter;

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

The next example introduces a new programming element called a *constructor*. A constructor is a block of code activated as the object is created. The constructor always has the same name as the class and is typically used to assign values to an object's fields as it comes into existence. (If there is no constructor, the value of every numeric field is set to zero.) The constructor is like other methods except that it is not preceded with a data type or the keyword `void` because there is no return type. When the object `sp` is created, the parameters 33, 50, and 30 are assigned in order to the variables `xpos`, `ypos`, and `dia` within the constructor. Within the constructor block, these values are assigned to the object's fields `x`, `y`, and `diameter`. For the fields to be accessible within every method of the object, they must be declared outside of the constructor. Remember the rules of variable scope (p. 178)—if the fields are declared within the constructor, they cannot be accessed outside the constructor.

43-04

```
Spot sp;                        // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot(33, 50, 30);  // Construct the object
}

void draw() {
  background(0);
  sp.display();
}

class Spot {
  float x, y, diameter;

  Spot(float xpos, float ypos, float dia) {
    x = xpos;        // Assign 33 to x
    y = ypos;        // Assign 50 to y
    diameter = dia;  // Assign 30 to diameter
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

The behavior of the `Spot` class can be extended by the addition of more methods and fields to the definition. The following example extends the class so that the ellipse moves up and down the display window and changes direction when it collides with the top or bottom. Since the class will be moving, it needs a field to set the speed, and because it will change directions, it needs a field to hold the current direction. We've named these fields `speed` and `direction` to make their uses clear and the names short. We decided to make the `speed` a `float` value to give a broader range of possible speeds. The `direction` field is an `int` so that it can be easily incorporated into the math for its movement:

`float speed`	***Distance moved each frame***
`int direction`	***Direction of motion (1 is down, -1 is up)***

To create the desired motion, we need to update the position of the circle on each frame. The direction also has to change at the edges of the display window. To test for an edge, the code tests whether the y-coordinate is smaller than the circle's radius or larger than

the height of the window minus the circle's radius. Make sure to include the radius value; then the direction will change when the outer edge of the circle (rather than its center) reaches the edge. In addition to deciding what the methods need to do and what they should be called, we must also consider the return type. Because nothing is returned from this method, the keyword `void` is used:

`void move()`	***Updates the circle's position and direction***

The code within the `move()` and `display()` methods could have been combined in one method; they were separated to make the example more clear. Changing the position of the object is a separate task from drawing it to the screen, and using separate methods reflects this. These changes allow every object created from the `Spot` class to have its own size and position. The objects will also move up and down the screen, changing directions at the edge.

```
class Spot {
  float x, y;         // X-coordinate, y-coordinate
  float diameter;     // Diameter of the circle
  float speed;        // Distance moved each frame                    43-05
  int direction = 1;  // Direction of motion (1 is down, -1 is up)

  // Constructor
  Spot(float xpos, float ypos, float dia, float sp) {
    x = xpos;
    y = ypos;
    diameter = dia;
    speed = sp;
  }

  void move() {
    y += (speed * direction);
    if ((y > (height - diameter/2)) || (y < diameter/2)) {
      direction *= -1;
    }
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

To save space and to keep the focus on the reuse of objects, examples from here to the end of the unit won't reprint the code for the `Spot` class in examples that require it. Instead, when you see a comment like `// Insert Spot class`, cut and paste the code

for the class into this position to make the code work. Run the following code to see the result of the `move()` method updating the fields and the `display()` method drawing the `sp` object to the display window.

43-06

```
Spot sp; // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot(33, 50, 30, 1.5);  // Construct the object
}

void draw() {
  fill(0, 15);
  rect(0, 0, width, height);
  fill(255);
  sp.move();
  sp.display();
}

// Insert Spot class
```

Like a function, a well-written class enables the programmer to focus on the resulting behavior and not the details of execution. Objects should be built for the purpose of reuse. After a difficult programming problem is solved and encoded inside an object, that code can be used later as a tool for building new code. For example, the functions and classes used in Processing grew out of many commonly used functions and classes that were part of the authors' own code.

As long as the interface to the class remains the same, the code within can be updated and modified without breaking a program that uses the object. For example, as long as the object is constructed with the x-coordinate, y-coordinate, and diameter and the names of `move()` and `display()` remain the same, the actual code inside `Spot` can be changed. This allows the programmer to refine the code for each object independently from the entire program.

Like other types of variables, additional objects are added to a program by declaring more names. The following program has three objects made from the `Spot` class. These objects, named `sp1`, `sp2`, and `sp3`, each have their own set of fields and methods. A method for each object is run by specifying its name, followed by the dot operator and the method name. For example, the code `sp1.move()` runs the `move()` method, which is a part of the `sp1` object. When these methods are run, they access the fields within their object. When `sp3` runs `move()` for the first time, the field value `y` is updated by the `speed` field value of 2.0 because that value was passed into `sp3` through the constructor.

43-07

```
Spot sp1, sp2, sp3;  // Declare the objects

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp1 = new Spot(20, 50, 40, 0.5);  // Construct sp1
  sp2 = new Spot(50, 50, 10, 2.0);  // Construct sp2
  sp3 = new Spot(80, 50, 30, 1.5);  // Construct sp3
}

void draw() {
  fill(0, 15);
  rect(0, 0, width, height);
  fill(255);
  sp1.move();
  sp2.move();
  sp3.move();
  sp1.display();
  sp2.display();
  sp3.display();
}

// Insert Spot class
```

It's difficult to summarize the basic concepts and syntax of object-oriented programming using only one example. To make the process of creating objects easier to comprehend, we've created the `Egg` class to compare and contrast with `Spot`. The `Egg` class is built with the goal of drawing an egg shape to the screen and wobbling it left and right. The `Egg` class began as an outline of the fields and methods it needed to have the desired shape and behavior:

`float x`	***X-coordinate for middle of the egg***
`float y`	***Y-coordinate for bottom of the egg***
`float tilt`	***Left and right angle offset***
`float angle`	***Used to define the tilt***
`float scalar`	***Height of the egg***
`void wobble()`	***Moves the egg back and forth***
`void display()`	***Draws the egg***

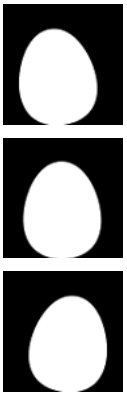
After the class requirements were established, it developed the same way as the `Spot` class. The `Egg` class started minimally, with only `x` and `y` fields and a `display()` method. The class was then added to a program with `setup()` and `draw()` to check the result. The `scale()` function was added to `display()` to decrease the size of the egg.

When this first program was working to our satisfaction, the `rotate()` method and `tilt` field were added to change the angle. Finally, the code was written to make the egg move. The `angle` field was added as a continuously changing number to set the tilt. The `wobble()` method was added to increment the angle and calculate the tilt. The `cos()` function was used to accelerate and decelerate the wobbling from side to side. After many rounds of incremental additions and testing, the final `Egg` class was working as initially planned.

43-08

```
class Egg {
  float x, y;    // X-coordinate, y-coordinate
  float tilt;    // Left and right angle offset
  float angle;   // Used to define the tilt
  float scalar;  // Height of the egg

  // Constructor
  Egg(int xpos, int ypos, float t, float s) {
    x = xpos;
    y = ypos;
    tilt = t;
    scalar = s / 100.0;
  }

  void wobble() {
   tilt = cos(angle) / 8;
   angle += 0.1;
  }

  void display() {
    noStroke();
    fill(255);
    pushMatrix();
    translate(x, y);
    rotate(tilt);
    scale(scalar);
    beginShape();
    vertex(0, -100);
    bezierVertex(25, -100, 40, -65, 40, -40);
    bezierVertex(40, -15, 25, 0, 0, 0);
    bezierVertex(-25, 0, -40, -15, -40, -40);
    bezierVertex(-40, -65, -25, -100, 0, -100);
    endShape();
    popMatrix();
  }
}
```

The Egg class is included in setup() and draw() the same way as in the Spot examples. An object of type Egg called humpty is created outside of setup() and draw(). Within setup(), the humpty object is constructed and the coordinates and initial tilt value are passed to the constructor. Within draw(), the wobble() and display() functions are run in sequence, causing the egg's angle and tilt values to update. These values are used to draw the shape to the screen. Run this code to see the egg wobble from left to right.



43-09

```
Egg humpty;  // Declare the object

void setup() {
  size(100, 100);
  smooth();
  // Inputs: x-coordinate, y-coordinate, tilt, height
  humpty = new Egg(50, 100, PI/32, 80);
}

void draw() {
  background(0);
  humpty.wobble();
  humpty.display();
}

// Insert Egg class
```

The Spot and Egg classes are two simple objects used to convey the basic syntax and concepts involved in object-oriented programming.

Arrays of objects

Working with arrays of objects is similar to working with arrays of other data types. Like all arrays, an array of objects is distinguished from a single object with brackets, the [and] characters. Because each array element is an object, each element of the array must be created before it can be accessed. The steps for working with an array of objects are:

1. Declare the array
2. Create the array
3. Create each object in the array

These steps are translated into code in the following example:

```
int numSpots = 6;
// Declare and create the array
Spot[] spots = new Spot[numSpots];

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  for (int i = 0; i < spots.length; i++) {
    float x = 10 + i*16;
    float rate = 0.5 + i*0.05;
    // Create each object
    spots[i] = new Spot(x, 50, 16, rate);
  }
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  for (int i = 0; i < spots.length; i++) {
    spots[i].move();     // Move each object
    spots[i].display();  // Display each object
  }
}

// Insert Spot class
```

The `Ring` class presents another example of working with arrays and objects. This class defines a circle that can be turned on, at which point it expands to a width of 400 and then stops displaying to the screen by turning itself off. When this class is added to the example below, a new ring turns on each time a mouse button is pressed. The fields and methods for `Ring` make this behavior possible:

`float x`	***X-coordinate of the ring***
`float y`	***Y-coordinate of the ring***
`float diameter`	***Diameter of the ring***
`boolean on`	***Turns the display on and off***
`void grow()`	***Increases the diameter if on is true***
`void display()`	***Draws the ring***

`Ring` was first developed as a simple class. Its features emerged through a series of iterations. This class has no constructor because its values are not set until the `start()` method is called within the program.

```
class Ring {
  float x, y;          // X-coordinate, y-coordinate
  float diameter;      // Diameter of the ring
  boolean on = false;  // Turns the display on and off

  void start(float xpos, float ypos) {
    x = xpos;
    y = ypos;
    on = true;
    diameter = 1;
  }

  void grow() {
    if (on == true) {
      diameter += 0.5;
      if (diameter > 400) {
        on = false;
      }
    }
  }

  void display() {
    if (on == true) {
      noFill();
      strokeWeight(4);
      stroke(155, 153);
      ellipse(x, y, diameter, diameter);
    }
  }
}
```

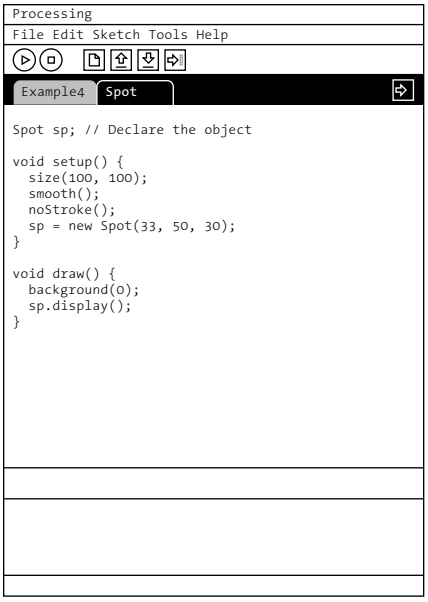
In this program, the `rings[]` array is created to hold fifty `Ring` objects. Space in memory for the `rings[]` array and `Ring` objects is allocated in `setup()`. The first time a mouse button is pressed, the first Ring object is turned on and its `x` and `y` variables are assigned the current values of the cursor. The counter variable `currentRing` is incremented by one, so the next time through the `draw()`, the `grow()` and `display()` methods will be run for the first `Ring` element. Each time a mouse button is pressed, a new `Ring` is turned on and displayed in the subsequent trip through `draw()`. When the final element in the array has been created, the program jumps back to the beginning of the array to assign new positions to earlier `Rings`.

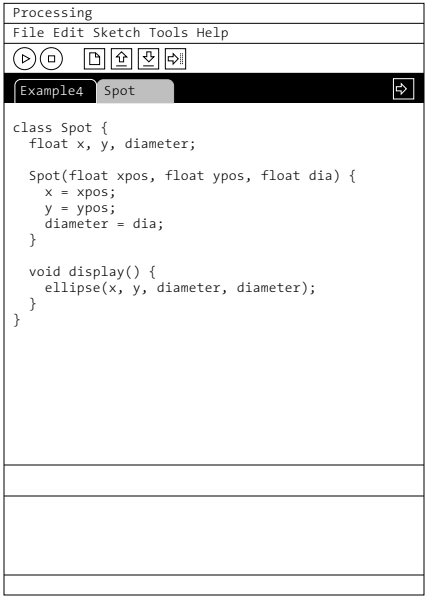
```
Ring[] rings;                   // Declare the array
int numRings = 50;
int currentRing = 0;

void setup() {
  size(100, 100);
  smooth();
  rings = new Ring[numRings];  // Create the array
  for (int i = 0; i < numRings; i++) {
    rings[i] = new Ring();     // Create each object
  }
}

void draw() {
  background(0);
  for (int i = 0; i < numRings; i++) {
    rings[i].grow();
    rings[i].display();
  }
}

// Click to create a new Ring
void mousePressed() {
  rings[currentRing].start(mouseX, mouseY);
  currentRing++;
  if (currentRing >= numRings) {
    currentRing = 0;
  }
}

// Insert Ring class
```

As modular code units, objects can be utilized in diverse ways according to the desires of different people for the needs of different projects. This is one of the exciting things about programming with objects.

Multiple files

The programs written before this unit have used one file for all of their code. As programs become longer, a single file can become inconvenient. When programs grow to hundreds and thousands of lines, breaking programs into modular units helps manage different parts of the program. Processing manages files with the Sketchbook, and each sketch can have multiple files that are managed with tabs.





Multiple Files

Programs can be divided into different files and represented as tabs within the PDE. This makes it easier to manage complicated programs.

The arrow button in the upper-right corner of the Processing Development Environment (PDE) is used to manage these files. Clicking this button reveals options to create a new tab, rename the current tab, and delete the current tab. If a project has more than one tab, each tab can also be hidden and revealed with this button. Hiding a tab temporarily removes that code from the sketch.

Code 43-04 can be divided into separate files to make it into a more modular program. First open or retype this program into Processing and name it "Example4." Now click on the arrow button and select the New Tab option from the menu that appears. A prompt asking for the name of the new tab appears. Type the name you want to assign to the file and click "OK" to continue. Because we'll be storing the `Spot` class in this file, use the name "Spot." You now have a new file called *Spot.pde* in your sketch folder. Select "Show Sketch Folder" from the Sketch menu to see this file.

Next, click on the original tab and and select the text for the `Spot` class. Cut the text from this tab, change to the `Spot` tab, and paste. Save the sketch for good measure, and press the Run button to see how the two files combine to create the final program. The file *Spot.pde* can be added to any sketch folder to make the `Spot` class accessible for that sketch.

When a sketch is created with multiple files, one of the files must have the same name as the folder containing the sketch to be recognized by Processing. This file is the main file for the sketch and always appears as the leftmost tab. The `setup()` and `draw()` methods for the sketch should be in this file. Only one of the files in a sketch can have a `setup()` and `draw()`. The other tabs appear in alphabetical order from left to right. When a sketch is run, all of the PDE files that comprise a sketch are converted to one file before the code is compiled and run. Additional functions and variables in the additional tabs have access to all global variables in the main file with `setup()` and `draw()`. Advanced programmers may want a different behavior, and a more detailed explanation can be found in the reference.

Exercises

1. *Write your own unique* `Spot` *class that has a different behavior than the one presented in the example. Design a kinetic composition with 90 of your* `Spots`.
2. *Design a class that displays, animates, and defines the behavior of an organism in relation to another object made from the same class.*
3. *Create a class to define a software puppet that responds to the mouse.*