Game Logic

Block Fall

Overview

In this project, we'll create a falling block game, and explore a couple new programming concepts such as *objects*, *vectors*, and *lists.* If this is your first time seeing these concepts, don't worry if it takes a little bit to get used to them. This project will also introduce basic ideas of *game logic*, the rules which make up the world of a game.

To get an idea of what we're going to create, open **BlockFallSolution** and run it. Blocks will fall from the top of the screen. Move your mouse around to move your player character at the bottom, and try to avoid being hit by a falling block. Your score in blue at the top is the number of frames it's been since you've been hit by a block. In green below that is the top score you've achieved during this play session.

Getting Started

Open **BlockFall** in Processing and run it. You should see a red rectangle at the center of the window and a blue rectangle slightly below it. Take a look at the code. Nothing too complex going on just yet. We set up a 600 by 800 window, and have eight **float** variables and two **color** variables which define the properties of two rectangles. During the **draw** function, we call the necessary Processing functions to actually draw the two rectangles at the right place, size, and color.

There is a second file called **TimeTrigger**, but you won't need to pay attention to that just yet. Feel free to open it and look around, but don't worry if you don't follow.

Objects

It takes a lot of lines of code to describe just the position of two rectangles. And on top of that, those lines of code look *very* similar. If we wanted to add another rectangle, we would need to declare and initialize five more variables, and add two more lines to the **draw** function. It wouldn't be that hard to do, because it would look almost the same as the existing code, but for that same reason it would be very tedious.

Thankfully, Processing gives us a way to avoid repeating ourselves by collecting together groups of variables into what's called an *object*. In order to create an object, we first have to tell Processing exactly what kind of information it will contain. This is sort of like how you need a recipe first to make a cake, or a blueprint first to make a house. If those metaphors help you understand, focus on them. If not, don't worry! Not everyone thinks the same way. In Processing, we call the description of a kind of object a *class*. Typically, each *class definition* is put in its own file, named after the class itself. Press **shift** + **command** + **N** to create a new file in your sketch. Call it **Block**.

Defining The Block Class

To define a Processing class, use the **class** keyword. In your new **Block** file, start out by adding the following code

```
class Block {

}
```

Technically, you're done! This is a totally valid class definition. It's just not a very useful one since this class doesn't actually contain any information. What information do you need to describe a block? What is common between our red and blue block? Its position in each axis, its width and height, and its color.

Modify your Block class so that it contains that information. These lines of code are just normal variable declarations, but because they're inside the **{** and **}** that come after the class declaration, they are *part of* the Block class, and are referred to as *fields* of that class.

```
class Block {
  float positionX;
  float positionY;
  float sizeX;
  float sizeY;
  color drawColor;
}
```

Now, go back to your main sketch file. You can replace all the variable declarations with

```
Block redBlock;
Block blueBlock;
```

And just like that, tons of code repetition reduced! Of course, now your sketch is full of angry red underlines because we still refer to the old variables in the rest of the code. Let's change that! Here's how the initialization for the red block should look now that it's an object instead of a bunch of free variables. Notice that to create an object, we use the **new** keyword, and to get at the fields (variables) *inside* the object, we use a **.** character.

```
redBlock = new Block();
redBlock.positionX = 260;
redBlock.positionY = 370;
redBlock.sizeX = 80;
```

```
    redBlock.sizeY = 60;
    redBlock.drawColor = color(255, 0, 0);
```

The code for the blue block will look almost the same. Modify that too! Now, on to the **draw** function. Use the **.** character again in order to read the information we stored in the fields *inside* the blocks. Here's the code for the red block. The blue block will be almost the same.

```
fill(redBlock.drawColor);
rect(redBlock.positionX, redBlock.positionY, redBlock.sizeX, redBlock.sizeY);
```

Run your sketch again. It should look, well, exactly the same. The initialization code in **setup** is still pretty repetitive though…

Object Constructors

In order to avoid repetition in creating an object, we can include a *constructor* in its class definition. A constructor is a special function that is called whenever we use the **new** keyword to create an object. Modify your Block class so that it reads

```
    class Block {
      float positionX;
      float positionY;
      float sizeX;
      float sizeY;
      color drawColor;

      Block(float px, float py, float sx, float sy, color dc) {
        positionX = px;
        positionY = py;
        sizeX = sx;
        sizeY = sy;
        drawColor = dc;
      }
    }
```

You don't necessarily have to accept parameters to the constructor for every field in the class. For example, if you wanted all Block objects to be green, you could modify the constructor to read

```
    Block(float px, float py, float sx, float sy) {
      positionX = px;
      positionY = py;
      sizeX = sx;
      sizeY = sy;
      drawColor = color(0, 255, 0);
```

```
}
```

Now, go back to your main sketch file. The red underlines are back. This is because a block now requires information to create, so we can't just say **new Block()** anymore. Instead, modify the initialization lines so that they use our new Block constructor! The **setup** method should now look like

```
void setup() {
  size(600, 800, P2D);
  redBlock = new Block(260, 370, 80, 60, color(255, 0, 0));
  blueBlock = new Block(260, 450, 80, 100, color(0, 0, 255));
}
```

And, just like that, many lines of initialization have been reduced. The last bit of repetition is leftover in the **draw** method. In addition to containing variables, objects can also contain functions! Just like variables which are inside objects are called *fields*, functions that are inside objects are called *methods*.

Another metaphor that might be useful for understanding the parts of an object relates them to parts of speech. An object is like a noun (a block is a *thing* in the world of our sketch). Its fields are like adjectives (the color of a block *describes* it). Its methods are like verbs, things the object can *do* (a block can *be drawn*).

Define a function inside the Block class which will handle drawing the block. The complete Block class should look like

```
class Block {
  float positionX;
  float positionY;
  float sizeX;
  float sizeY;
  color drawColor;

  Block(float px, float py, float sx, float sy, color dc) {
    positionX = px;
    positionY = py;
    sizeX = sx;
    sizeY = sy;
    drawColor = dc;
  }

  void render() {
    fill(drawColor);
    rect(positionX, positionY, sizeX, sizeY);
  }
}
```

In programming, we often use the word *render* to describe drawing something to the screen, which is why I've named the Block class's method that way.

In order to call methods inside an object, we also use the **.** character, just like we do for accessing fields inside it. In the main sketch file, the **draw** method should now read

```
void draw() {
  background(255);
  redBlock.render();
  blueBlock.render();
}
```

Now, adding a new Block to the sketch is very easy! Try adding a third Block with a different color, size, and position. It should only take three lines of code.

The process that you just went through, of taking repeated code and replacing it with a recipe for that same code (in Processing, a *class definition*), is very important in software engineering (the field of study which examines the production, quality, and organization of code). We call it *abstraction*, and it will show up time and time again. We also use abstraction when we take repeated code and put it into a function, or a loop. If you find yourself writing code that looks the same as other code in a sketch, you may want to consider whether or not some abstraction will make it simpler. In shorter words, *don't repeat yourself!*

Now on to a new topic...

Vectors

A vector, most simply, is just a pair of numbers. Usually in math we write vectors using parentheses and commas, like **(4, 5)** or **(10.5, -2)**. It's important to remember that the *order* of the numbers in a vector matters, which means that **(0, 1)** is not the same as **(1, 0)**. It's useful in a 2D game to group numbers together like this, because many values in a 2D world can be easily represented that way. For example, instead of having to hold onto two **float** variables to represent the position of something, we can just have one vector.

Because a vector is a grouping of information, we represent it in Processing with an object! Thankfully, you don't have to write the class definition for this object yourself. Because it is so common to need to use vectors, Processing comes with them built in. In Processing, the vector class is called **PVector**.

We can use vectors to simplify our Block class. Instead of storing four **float** fields, we can store two **PVector** fields: One to hold the Block's position, and the other to hold its

dimensions. To initialize **PVector** objects, just like for Block objects, we use the **new** keyword.
Modify the Block class to use **PVector** fields.

```
class Block {
  PVector position;
  PVector size;
  color drawColor;

  Block(float px, float py, float sx, float sy, color dc) {
    position = new PVector(px, py);
    size = new PVector(sx, sy);
    drawColor = dc;
  }

  void render() {
    fill(drawColor);
    rect(position.x, position.y, size.x, size.y);
  }
}
```

The **PVector** constructor takes two values, which will be the first and second numbers in the vector. These values are used to initialize its two fields, called **x** and **y**. You can find this information about the **PVector** class in the Processing reference. You are not expected to memorize information about every class Processing provides. Almost no programmers have memorized every aspect of the language they're working in, because almost all will come with a manual like the Processing reference!

Lists

At this point, it's pretty easy to make a single new Block. But what if you wanted to add ten more, or twenty? Or what if you wanted to add a block every time a user clicks on the screen?

In Block Fall, there are often many blocks on the screen at the same time. In order to keep track of many objects of the same type, we can use a special object called a *list*. Lists can be used to store any number of objects. In Processing, there is a built in list class called **ArrayList**. When an **ArrayList** variable is declared or initialized, it needs a little bit more information than declaring or initializing the objects we've seen so far. We need to tell Processing what it is a list *of*. For example, we want a list of Block objects, so our variable declaration looks like

```
ArrayList<Block> blocks;
```

Notice that we use **<** and **>** to specify what type of objects will be *inside* the list. We have to specify this when we initialize the **ArrayList** as well.

```
blocks = new ArrayList<Block>();
```

In order to add things to our list, we have to use the **add** method from the **ArrayList** class, and in order to read things in our list, we have to use the **get** method from the **ArrayList** class. The **get** method takes one argument, the position in the list of the item we want to get (the *first* item in the list has position 0, the *second* has position 1, etc) The main sketch refactored to use lists looks like

```
ArrayList<Block> blocks;

void setup() {
  size(600, 800, P2D);
  blocks = new ArrayList<Block>();
  blocks.add(new Block(260, 370, 80, 60, color(255, 0, 0)));
  blocks.add(new Block(260, 450, 80, 100, color(0, 0, 255)));
}

void draw() {
  background(255);
  blocks.get(0).render();
  blocks.get(1).render();
}
```

So why is this useful? Now it's easy to add many blocks using **for** loops! Change the code in **setup** to the following

```
void setup() {
  size(600, 800, P2D);
  blocks = new ArrayList<Block>();

  color blockColor;
  for (int i = 0; i < 20; i++) {
    blockColor = color(random(255), random(255), random(255));
    blocks.add(new Block(random(width), random(height), 80, 60, blockColor));
  }
}
```

We declare a local **color** variable **blockColor**, then execute a **for** loop which runs 20 times, each time assigning a random color to **blockColor** and then creating a new 80 x 60 block at a random location. If you run the sketch at this point, though, you'll still only see two Blocks drawn to the screen. This because we only draw the first two still. Change the code in **draw**

```
void draw() {
  background(255);
  for (int i = 0; i < 20; i++) {
    blocks.get(i).render();
  }
}
```
Now to see how easy it is to change the number of blocks, change the number of iterations of the **for** loops from 20 to 200. Tons of blocks, and you only changed two characters of code! In order to make this even easier, you can replace the **for** loop with a special kind of loop that works on lists and always works no matter how many items are *in* the list, called a *for each* loop. It looks like

```
for (Block block : blocks) {
  block.render();
}
```

This means "For every element of the list **blocks**, call it **block** and then run the code between the **{** and **}**" which, in this case, just renders the block. Modify your sketch to use a for each loop.

Timing

So remember that other file, **TimeTrigger**? Now it's time to actually put that to use! **TimeTrigger** defines a class (called, of course, TimeTrigger). So far in Processing, you've been able to write code which runs once when the sketch is run (by putting it in the **setup** function), or code that runs every single frame (by putting it in the **draw** function) With TimeTrigger objects, code can be run periodically (over and over) without necessarily happening *every* frame.

In Block Fall,

Putting It All Together