Introduction to Programming II Course Notes

Felipe Balbi

October 15, 2019

Week 1

1.101 Welcome to Introduction to Programming II

We're going to rely more on object orientation, build larger projects. Won't present a lot of syntax.

3 case studies:

- Data visualizer
- Drawing app
- Music visualizer

Assignments will center around one of these case studies and letting us extend it.

1.201 Object concepts revisited

We're going to be making extensive use of objects. A quick recap of the previous module:

We started with *hardcoded* numbers in order to draw on the screen. Like in the example below:

```
function setup()
{
  createCanvas(800, 600);
}

function draw()
{
  background(255);
  rect(30, 20, 55, 55);
}
```

As the programs we wrote started to grow, we converted those *hardcoded* numbers into **variables**. That allowed us to change sizes, colors, etc, much more easily. See below:

```
let backgroundColor = 255;

let rectX = 30;
let rectY = 20;
let rectWidth = 55;
let rectHeight = 55;

function setup()
{
  createCanvas(800, 600);
}

function draw()
{
  background(255);
  rect(rectX, rectY, rectWidth, rectHeight);
}
```

As time went by, and objects being drawn became more complex, we started grouping related variables into **objects**. The original variables then became *properties* on our objects and we referred to them using the keyword *this*, like so:

```
let backgroundColor = 255;

let myRect;

function setup()
{
  createCanvas(800, 600);
  myRect = {
    x: 30,
    y: 20,
    width: 55;
    height: 55;
    draw: function ()
    {
      rect(this.x, this.y, this.width, this.height);
    }
  };
}

function draw()
{
  background(255);
  rect(myRect.x, myRect.y, myRect.width, myRect.height);
}
```

When we decided we wanted to build multiple copies of the same **object**, we augmented our objects with **Constructor Functions**. That allowed us to quickly spawn different instances of our objects using the keyword *new*, like below:

```
let backgroundColor = 255;

let myRect;

function Rectangle(x, y, width, height)
{
  this.x = x;
  this.y = y;
  this.width = width;
  this.height = height;

  this.draw = function ()
  {
    rect(this.x, this.y, this.width, this.height);
  }
}

function setup()
{
  createCanvas(800, 600);
  myRect = new Rectangle(30, 20, 55, 55);
}

function draw()
{
  background(255);
  myRect.draw();
}
```

1.202 Objects in practice, part 1

Working through a practical example, we build it up until we get to constructor functions. During the example, which contains a flying saucer, we start with hardcoded numbers and convert those to variables. Afterwards we create a *flyingSaucer* object and move variables to become properties within our object. After doing that, we added some extra properties and animation to our flying saucer.

1.203 Objects in practice, part 2

We start by extending our flying saucer object with methods. Starting by adding a method called *hover* which computes the hovering animation of our flying saucer. After adding the method, we call it from the `draw()` function.

The second extension is a new method to add a *Tractor Beam* to the flying saucer. Again we add a new method to our object, implement it and call it from `draw()` function.

After adding a beam, we created the ability of turning the beam on and off by adding a `beam_on` property to the object and adding support to keypresses by implementing `keyPressed()` and `keyReleased()` functions. Those functions were used to modify `beam_on` property directly.

The next step here was to create multiply flying saucers. In order to do that, we added a `FlyingSaucer()` constructor function. After converting the currently flying saucer object to a constructor function, we augmented it with a `draw()` function of itself and moved the drawing code from P5JS's `draw()` to the construction function's `draw()` method. Then we called it from P5JS's `draw()` function.

When this was all working fine, we added position arguments to the constructor function so that we could create a new flying saucer at whatever position we wanted. The next step was to create an array called `flyingSaucers` and populating it with a `for` loop.

From `draw()` function we converted all flying saucer code to operate on the new `flyingSaucers` array instead.

1.205 Object orientation

Object Orientation is a programming paradigm that was created to reduce the *clutter* of multiple function calls operating on different variables. Code like this is usually referred to as *spaghetti code*.

Object Orientation deals with this problem through two principles

1. Encapsulation

   Through encapsulation, related functionality (functions and variables) are grouped together into entities called *objects*. Variables inside objects are called *properties* and functions inside objects are called *methods*.

   An object is referred to as *well-encapsulated* when it modifies strictly its own properties. A corollary of this is that other entities don't modify an object's internal state (i.e. their properties) directly; only through method calls.

   This means that *the outside world*, anything outside of the object itself, treats the object as a black box that provides functionality. We could even replace the internal implementation completely without having to modify anything outside the object, provided the **behavior** of a particular object doesn't change.

2. Abstraction

   This teaches us that only the bare minimum details about the inner workings of an object should be revealed. If a user of an object doesn't know how the object works, we allow ourselves to modify our object without affecting the rest of our code.

   Choosing a good abstraction, however, takes practice.

1.206 Object orientation in practice, part 1

Continuing with the flying saucer, we put in practice the principles of encapsulation and abstraction. When we want to hide properties of an object, we turn those properties into local variables of the constructor function using one of `var`, `let` or `const` keywords.

When converting methods to be internal to the object, we start similarly by removing `this.` and turning it into `var`, `let` or `const`. However, this results in `this` inside the method no longer referring to the object itself. A simple trick for this is to create a local variable to hold a reference to `this`, like so:

```
let self = this;
```

Then, in our method, we replace all references to `this` with `self`.

We can see that in our code, the flying saucer knows nothing about cows and cows know nothing about flying saucer. When we decide we want flying saucer to be able to beam up cows, we're faced with a problem that we want to maintain encapsulation.

In order to avoid breaking encapsulation, we create a new object *CowManager* that handles the communication between cows and flying saucers.

1.207 Object orientation in practice, part 2

Implementing the beaming of cows. This brings a new set of issues which try to break our encapsulation and abstraction. We have added a `levitateCows()` method to the *CowManager*.

With all this code properly abstracted and encapsulated, it becomes easy to add more flying saucers to the scenery and everything will still behave correctly.

Week 2

1.301 Splitting across multiple files

Splitting code up into among files makes the codebase a lot more organized and easier to follow.

A good rule of function is to put each constructor function into its own file.

We want to keep our *sketch.js* as small as possible. Most of the implementation details should be left on separate files on their own constructor functions. *sketch.js* will simply combine those with a little business logic to implement our application.

We must remember to modify index.html in order to include our new javascript files. Ordering is important; we must make sure to import our constructor function files before sketch.js but after p5.min.js.

1.401 Case studies overview

Our focus is on understanding larger applications. Dividing the code up into multiple files will help us get organized and navigate through the codebase.

Moreover, we should be more comfortable reading and understanding code written by others. During this module we will write a project based on one of three different case studies.

The first of these applications will be a simple Drawing App. The second application is a Music Visualization App. The third app will be a Data Visualization App.

After completing these three case studies, we must choose one to be extended.

Week 3

2.102 Introduction to case study, part 1: drawing app

Investigation of the current features of the drawing app and an overview of the structure of the code in terms of object orientation.

The drawing app can be downloaded from 2.104 Case study 1: drawing app.

as we run the app, we will notice that some features don't work well; that's because fixing them is part of future lectures.

2.103 Introduction to case study, part 2: drawing app

Capturing Drawing App's features:

1. Pen Tool
2. Line Tool
3. Spray Can
4. Symmetry Tool
    (a) Vertical
    (b) Horizontal
5. Save Image
6. Clear Image
7. Colours

2.201 Drawing application, part 1 - under the hood

The first we notice when looking at the code for this application, is that it contains quite a few files and a considerable amount of code.

We start by opening sketch.js where we're faced with the following content:

```
//global variables that will store the toolbox colour palette
//amnd the helper functions
var toolbox = null;
var colourP = null;
var helpers = null;

//spray can object literal
sprayCan = {
  name: "sprayCanTool",
  icon: "assets/sprayCan.jpg",
  points: 13,
  spread: 10,
  draw: function(){
    //if the mouse is pressed paint on the canvas
    //spread describes how far to spread the paint from the mouse pointer
    //points holds how many pixels of paint for each mouse press.
    if(mouseIsPressed){
      for(var i = 0; i < this.points; i++){
point(random(mouseX-this.spread, mouseX + this.spread),
      random(mouseY-this.spread, mouseY+this.spread));
      }
    }
  }
};

function setup() {

  //create a canvas to fill the content div from index.html
  canvasContainer = select('#content');
  var c = createCanvas(canvasContainer.size().width,
       canvasContainer.size().height);
  c.parent("content");

  //create helper functions and the colour palette
  helpers = new HelperFunctions();
  colourP = new ColourPalette();

  //create a toolbox for storing the tools
  toolbox = new Toolbox();

  //add the tools to the toolbox.
  toolbox.addTool(new FreehandTool());
  toolbox.addTool(new LineToTool());
  toolbox.addTool(sprayCan);
  toolbox.addTool(new mirrorDrawTool());
  background(255);

}
```

```
function draw() {
  //call the draw function from the selected tool.
  //hasOwnProperty is a javascript function that tests
  //if an object contains a particular method or property
  //if there isn't a draw method the app will alert the user
  if (toolbox.selectedTool.hasOwnProperty("draw")) {
    toolbox.selectedTool.draw();
  } else {
    alert("it doesn't look like your tool has a draw method!");
  }
}
```

We can see that the canvas is created in a slightly different manner in setup:

```
canvasContainer = select('#content');
var c = createCanvas(canvasContainer.size().width,
     canvasContainer.size().height);
```

This select() method, is a P5.js DOM library method which allows us to find elements of the DOM using the elements ID, class or tag.

Continue with the code walkthrough, the next thing that happens is:

```
helpers = new HelperFunctions();
```

We create a new object *HelperFunctions* to hold all our helpers that don't seem to be related to any particular object. Currently, here, we register a handler for the *Clear Image* button and another for the *Save Image* button.

Right after, we create a *Colour Palette* with the following line of code:

```
colourP = new ColourPalette();
```

The *colourP* object handles our colour picker area. This class provides a list colours to be shown on the HTML page. It, also, tracks the currently-selected colour so it can be styled differently.

Its method *loadColours()* will iterate over the array of colors, create a new html `<div>` for each of them and colour them with the correct colour.

It also registers a click event handler for each of the new `<div>` elements so the user can change the selected colour.

Moving on, we find our toolbox:

```
//create a toolbox for storing the tools
toolbox = new Toolbox();
```

The toolbox contains an empty list of our drawing tools. It provides a click event for the toolbar so the user can change the tool.

This class also provides a method for adding new tools to the toolbox. This method is called *addTool()*. Paired with this, the class provides a method for changing tools, assuming that we will have more than one tool to use.

On the few following lines, we actually add tools to the toolbox:

```
//add the tools to the toolbox.
toolbox.addTool(new FreehandTool());
toolbox.addTool(new LineToTool());
toolbox.addTool(sprayCan);
toolbox.addTool(new mirrorDrawTool());
```

Each and every tool are expected to follow a contract. What we mean by this is that when *Toolbox* was created, a contract was created along with it as to how tools are supposed to behave. As long as tools follow this contract, they can be added to the toolbox.

The contract states that a tool must provide:

- Tool name
- Tool icon
- `draw()` method

These contracts are commonly referred to as *Injected Dependencies.* This is commonly used by languages which are loosely typed.

And finally, we create our white background:

```
background(255);
```

2.202 drawing application, part 2 - under the hood

the line tool is peculiar because it doesn't actually "draw" until we let go of the mouse button.

the way this is implemented is by means of loadpixels() and updatepixels(). these two functions work together to create the illusion that the line tool only "draws" after the mouse button is released. *loadpixels()* will copy the current state of our canvas to a global pixels array, while *updatepixels()* will reset the screen to the values that were last stored in the pixels array.

in practice, it's almost like having a constant *undo* tool happening every time we move the mouse with the button still depressed.

2.204 introduction to p5.dom

The DOM library allows us to access other parts of the webpage outside of the P5 canvas.

The DOM (Document Object Model) is a tree representation of the components of a webpage, such as headings, paragraphs, buttons, etc.

To select a specific element from the html page, we use the DOM library's *select()* method and pass it the *id*, *class* or *tag* name as a string argument. The *select()* method will either return the element it found or *null*.

Note that when passed a *class* name argument, *select()* will return the first occurrence. If we want **every** element belonging to a class, we should use the *selectAll()* method, which will return an array with all occurrences.

2.206 P5.dom: handling events 1

In order to add a button to the webpage, we can use the *createButton()* function. This function takes a label string argument and an optional value string for the button

A slider can be created with the *createSlider()* function. The arguments to this function are the minimum value, the maximum value, an optional default value and an optional step size.

To create an area where we can input text, we can use the *createInput()* function. This function only takes two **optional** arguments: a default value and the type (text, password, etc).

The function *createSelect()* allows us to create a drop-down selectable list of options. This has one optional argument *multiple* which tells the selection box if multiple options can be selected at the same time or not. In order to add options to this selection box, we must call the *.option()* method of the object created by *createSelect()*. The argument passed to *.option()* is the option to be added.

As a sidenote, we could have used the *createColorPicker()* function instead. It takes no arguments and allows us to choose **any** color from a color wheel.

When we want to create elements inside another element, we can change its parent by calling the *.parent()* method on created elements and passing the new parent element.

2.207 P5.dom: handling events 2

During this video, the lecturer cleans up the code a bit before adding support for the button.

The button is peculiar since we need to figure out the button state at the moment the user clicks it.

For the button, we call the *.mousePressed()* method, which allows us to pass another function as an argument to it which will be the event handler for the *Mouse Pressed* event of the button.

2.208 Drawing application 2

During the video, the lecturer explains the purpose of the *populateOptions()* method, which is to add extra options as HTML elements to our own tools from the toolbox.

On the other hand, *unselectTool()* will call *updatePixels()* to reload the image of the canvas without any modifications that may have been done by the previously selected tool. For example, the mirror tool adds a red line through the middle of the canvas and when we move to another tool, we want to remove that red line.

Apart from that, *unselectTool()* will also empty the HTML element with the class *options*.