Introduction to Programming II Course Notes

Felipe Balbi

October 8, 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.