Nature of Code

Patrick Dwyer
Fall 2005
Week 1 - September 6th

Processing Review

1. Processing Environment

   Processing is a Java based environment for creating images, animation, and audio. Using simple functions to capture user interaction and draw to the screen, it is possible to quickly develop highly interactive projects. Most ideas can be expressed in Processing using the built in functions and syntax; but for those ideas that grow beyond the basic functionality of the environment it is possible to use extension libraries written by other Processing users, or write your own libraries.

2. Setup and Drawing

   The two simplest methods to take advantage of are `setup` and `draw`. The `setup` method is called once when your Processing Applet[1] begins, before any other methods are called. It is in the `setup` method that we should define some basic attributes of our program.

```
void setup() {
      size(200, 200);
      background(0, 0, 0);
}
```

   Here we've decided that the size of our drawing area (and hence, the program's entire interface area) should be 200 pixels wide by 200 pixels tall, and that the background color of our screen should be black[2].

   Once we've used setup to create the initial conditions for our program, we can move on to the draw method, which is repeatedly called while our program is running[3].

```
void draw() {
      line(0, 0, mouseX, mouseY);
}
```

   In this draw method we're using the built in function `line()` to draw a line from the top left corner of the screen to the current position of the mouse, which Processing tracks for us using `mouseX` and `mouseY`.

---

[1] We'll end up using *Applet* and *Program* interchangeably to describe our Processing code. The term Applet derives from Java Applet, which is technically what our processing program will be when embedded in a web page.

[2] The default color mode for Processing is *R*ed *G*reen *B*lue. It is possible to change the color mode to *H*ue *S*aturation *B*rightness using the `colorMode()` method.

[3] We can define how often the draw method is called by defining a framerate for our program using the `framerate()` method. The *Processing* documentation recommends that `framerate()` be called from within the `seutp()` method.

3. Mouse and Keyboard

The primary means of interacting with the user in a Processing program is through the mouse and keyboard. For simple mouse and keyboard tracking Processing provides a number of variables that are automatically populated with data, ready for us to use in our draw method. If these variables aren't sufficient for our task, and we want to trigger specific actions or code when the mouse and keyboard are used, we can resort to methods that are called in response to user input.

Let's use a common example of drawing a series of continuous lines from the mouse. We want the user to be able to move the mouse around the program, with a line trailing behind the mouse pointer showing everywhere the mouse has been. To make things interesting we want to clear the screen whenever the user presses the *C* key on the keyboard.

```
void draw() {
      line(pmouseX, pmouseY, mouseX, mouseY);
      if (keyPressed) {
            if (key == 'c') {
                  background(220);
            }
      }
}
```

Quite a simple little draw method, but in this method we use keyboard and mouse variables that are automatically updated and tracked by Processing. To draw our line we use information about where our mouse pointer is now (mouseX, mouseY), and where it was last frame (pmouseX, pmouseY). All four of these values are provided for use each time we use our draw method. Using these values in the line method we create a line between the mouse position from the previous frame to the position in the current frame, causing a continuous line to form as each frame passes and the user moves the mouse around the screen.

To clear the screen we use two keyboard variables, keyPressed and key. The keyPressed value is true if any key on the keyboard has been pressed since the last frame, and resets to false after the draw method for this frame is finished. The value of the key that has been pressed is stored in the key variable. In this case we test to see if the key being pressed is the *C* key. The value of key will remain set to the last key pressed, and does not reset after the draw method is finished.

In most cases the variables discussed above for tracking the keyboard and mouse will be sufficient for creating simple to moderately complex Processing programs. When these variables aren't enough we move on to methods for tracking the keyboard and mouse.

For tracking the mouse, Processing provides four methods, mouseDragged, mouseMoved, mousePressed, and mouseReleased. Each of these look like our other methods so far, but each is called in response to a specific event. To respond to different actions based upon regular mouse movement and dragging of the mouse, we might use this code:

```
void mouseMoved() {
      x = x + 1;
}
void mouseDragged() {
```

```
        y = y + 1;
    }
```

The mousePressed and mouseReleased methods work in the same way; the mousePressed method is called when the mouse button is first pressed, and the mouseReleased method is called when the user releases the button:

```
    void mousePressed() {
        background(0);
    }
    void mouseReleased() {
        background(255);
    }
```

For keeping track of the keyboard we have two methods we can use; `keyPressed` and `keyReleased`. Similar to the mousePressed and mouseReleased methods, the keyboard methods are called when the user first presses a key on the keyboard, and when the key is released:

```
    void keyPressed() {
        if (key == 'z') {
            background(255, 0, 0);
        }
    }
    void keyReleased() {
        background(0);
    }
```

4. Pixels and Images

Working with pixels and images is relatively straight forward. With both we first need to create an object to place on the screen. For pixels we work with `color` object, for images `PImage` objects, both provided for us by Processing. To draw an image to the screen we first load the image, and then position it on screen:

```
    void draw() {
        PImage myImage = loadImage("myimage.jpg");
        set(0, 0, myImage);
    }
```

The `loadImage` method is given the name of an image in the data directory of our sketch folder, and loads it into a `PImage` object. Once the image is loaded we can place it anywhere on the screen using the `set()` method, which takes the x and y coordinates of where to place the image, and the image itself.

When working with pixels we need `color` objects, which store the RGB or HSB values of the pixels on the screen. To set the color of a specific pixel we use the same set command we used for placing our image on screen, but we specify a `color` object instead of a `PImage` object.

```
    void draw() {
        color myColor = color(201, 110, 42);
        set(10, 10, myColor);
    }
```

This draw method sets the pixel at (10, 10) to a deep orange. We can use similar methods to *get* the values of pixels on the screen:

```
void draw() {
      color myColor = get(10, 10);
}
```

This gets the color at the pixel (10, 10) on screen and stores it in our `myColor` object. This can be especially useful for working with the pixels of an image on screen:

```
void draw() {
      PImage myImage = loadImage("car.jpg");
      set(0, 0, myImage);
      for (int x = 10; x < 50; x++) {
            for (int y = 10; y < 50; y++) {
                  color myColor = get(x, y);
                  int myRed = red(myColor);
                  int myGreen = green(myColor);
                  int myBlue = blue(myColor);
                  myColor = color(myGreen, myRed, myBlue);
                  set(x, y, myColor);
            }
      }
}
```

In this longer section of code we draw an image to the screen (presumably an image of a car), and then for a rectangle of pixels from (10, 10) to (50, 50) we switch the red and green values of the pixels, effectively altering the image of the car on screen.

5. Shapes

Processing includes utility methods for creating a series of basic shapes; arcs, ellipses, rectangles, triangles and quadrangles. The syntax to these functions is mostly similar, and the exact variables for each can be found in the Processing reference documents. The way these shapes are drawn to the screen is controlled by the *fill* and *stroke*. Drawing a standard rectangle might look like this:

```
void draw() {
      rect(10, 10, 70, 50);
}
```

This would draw a filled rectangle with a black border, 60 pixels wide by 40 pixels high. The basic border is a single pixel wide. If we wanted instead to draw a red rectangle with a 5 pixel wide blue border with rounded corners, we would use:

```
void draw() {
      stroke(0, 0, 255);
      strokeWeight(5);
      strokeJoin(ROUND);
      fill(255, 0, 0);
      rect(10, 10, 70, 50);
}
```

The `stroke` method controls the color used for lines and borders, while `strokeWeight` controls the width of those lines. The `strokeJoin` and `strokeCap` (not featured in this code sample) methods determine how lines are joined together; in this example we set the joints of lines to be rounded. Finally the `fill` method sets the color to use when drawing the internal shape.

Further details on drawing shapes, including other shape methods, drawing styles, fill modes, curves and custom shapes can be found in the Processing documentation.

6. Text

There are two ways to display text in Processing; on screen or in the Processing debugging window. Printing text to the debugging window is simple enough:

```
void draw() {
      println("Debugging data: " + myVariable);
}
```

This will print out the text "Debugging data: " followed by the value of `myVariable`. This is useful when creating a program, to test and debug values within code, but this doesn't suffice for displaying text on screen. To display text on screen we need to follow a few steps.

First we must create the font(s) we want to use in the data directory of our sketch. To do this we use the "Create Font" tool included in Processing. Start by opening or creating the sketch you want to work with. Once open, choose the "Create Font..." menu item from the "Tools" menu. Choose a font from the list, a font size (choose the largest font size you will display for a clean look), and click the "OK" button. Only choose the "All Characters" option if you *really* need all possible characters in the font, as it will significantly increase the size of the font files. Note the name given to the font in the "Filename" box, we'll be using this name to load the font later on.

Once our font is created it is a fairly simple matter to load the font and starting printing text to the screen:

```
PFont myFont;
void setup() {
      size(300, 300);
      myFont = loadFont("LucidaGrande-32.vlw");
      textFont(myFont, 32);
}
void draw() {
      text("Hello, world!", 10, 100);
}
```

The output of this program will be the text "Hello, world!" printed in 32 point Lucida Grande ten pixels from the left of the screen, with the baseline of the text 100 pixels from the top of the screen.

When working with fonts it is useful to note that the `stroke` methods for controlling how lines are drawn also affect text drawn to the screen.

7. Preview and Export

Preparing your program for the web is fairly straight forward. From the "File" menu choose the "Export" item. A folder will open with the prepared files. Every time the export is run these files will

be overwritten, so if you want to place the applet in your own HTML, save the HTML template you want to use elsewhere. To embed the applet in your own HTML use the following code:

```
<applet code="sketch_050904b" archive="sketch_050904b.jar"
width="300" height="300">
            <param name="image" value="loading.gif">
            <param name="boxmessage" value="Loading Processing
software...">
            <param name="boxbgcolor" value="#FFFFFF">

<!-- This is the message that shows up when people don't have
     Java installed in their browser. Any HTML can go here. -->
To view this content, you need to install Java from <A
HREF="http://java.com">java.com</A>
</applet>
```

Depending on the sketch you are embedding you will need to change the names and references of the embedding code. Make sure to upload the .JAR file of the applet, as well as the "loading.gif" file.

Object Oriented Programming

Where possible we'll make use of Object Oriented programming techniques to create easy to use and manipulate programs. Using objects in Processing will let us develop code that we can easily reuse throughout our programs.

What is an Object?

Objects are an encapsulation of properties and methods that can act upon those properties. This means that the data and functions relating to a concept can be packaged together, and work with each other. Objects can maintain their own data structures, relieving us of that task. Let's look at a quick example using an object[4] describing a circle. First let's look at functions that aren't part of an object, and then we'll wrap them up in a class and see the difference. We want to be able to calculate the area and whether or not a point lies within the circle.

First the non-object code:

```
int radius = 10;
int x = 12;
int y = 30;

// calculate the area
float myArea = area(radius);
boolean isPoint = pointInCircle(45, 13, radius, x, y);

// Point in Circle Function
boolean pointInCircle(int px, int py, int r, int cx, int cy) {
      float pointDistance = (px - cx) * (px - cx) + (py - cy) * (py -
cy);
```

[4] At times we will use the words Object and Class interchangeably. A *Class* typically defines an object.

```
        pointDistance = sqrt(pointDistance);
        if (pointDistance < (float)r) {
                return true;
        } else {
                return false;
        }
}

// Radius function
float area(int r) {
        return 3.14159 * (float)r * (float)r;
}
```

So in the non-object function we keep track of our radius, x and y, and then define and use a few functions. Seems pretty easy. Let's take a look at the same code written as an object:

```
public class circle {
        public static final int PI = 3.14159;
        public int radius = 0;
        public int x = 0;
        public int y = 0;

        public circle(int r, int nx, int ny) {
                radius = r;
                x = nx;
                y = ny;
        }

        public float area() {
                return PI * (float)radius * (float)radius;
        }

        public boolean pointInCircle(int px, int py) {
                float pointDistance = (px - x) * (px - x) + (py - y) * (py
        - y);
                pointDistance = sqrt(pointDistance);
                if (pointDistance < (float)radius) {
                        return true;
                } else {
                        return false;
                }
        }
}

circle myCircle = new circle(10, 12, 30);
float myArea = myCirlce.area();
```

```
boolean isPoint = myCircle.pointInCircle(45, 13);
```

In the end the class for creating a circle ends up being a few lines longer (26 lines as opposed to 17 for the functions in the non-object code) in this example. But what if we wanted to make 100 circles? Given the functions from the first example our non-object code would look like this:

```
int[] radii = new int[100];
int[] xs = new int[100];
int[] ys = new int[100];
for (int i = 0; i < 100; i++) {
      radii[i] = i;
      xs[i] = i * 10;
      ys = i * 20;
}
// see if a point is in any of the circles
for (int i = 0; i < 100; i++) {
      if ( pointInCircle(12, 30, radii[i], xs[i], ys[i])) {
            // do something with this point
      }
}
```

Now we're keeping track of 300 variables our selves to make sure everything works properly. On the other had the same thing using our object would look like:

```
circle[] myCircles = new circle[100];
for (int i = 0; i < 100; i++) {
      myCirlces[i] = new circle(i, i * 10, i * 20);
}
// see if a point is in any of the circles
for (int i = 0; i < 100; i++) {
      if (myCirlces[i].pointInCircle(12, 30)) {
            // do something with this point
      }
}
```

Now we just keep track of our circles, no extra variables. With only three properties for the circle, this isn't too much trouble either way, but what if we need to keep track of 5 or 10, even 50 properties for each object. It's much easier to let the object do the work for us. Objects let us package code and data together in such a way that it makes less work for us.

But what's really happening in this "object"? Let's look at what goes into our circle class:

1. Class Definition

   ```
   public class circle {
   ...
   ```

   The first thing we do when we want to code a class is to name it. Here we've decided to call our class `circle`. The first part of the definition, `public class`, tells processing that we want all of the rest of our code to know that this class exists and can be used.

2. Class Properties

When we use our class the object we create is an **Instance** of our class, one unique copy of the data it contains. There are time, though, that we want to share one copy of a variable between all instances of our class. To do this we create Class Variables, or class properties:

```
...
public static final int PI = 3.14159;
...
```

Here we've defined a class variable called `PI`, and its value is shared between all instances of the class[5].

3. Instance Properties

   While class properties are defined and shared among all instances of a class, instance properties can have different values in different instances of the same class. In our circle class we define three instance properties:

```
...
public int radius = 0;
public int x = 0;
public in y = 0;
...
```

   We initially set them to 0, but each of our circle objects has their own values for these properties. These properties can be of any type (String, float, int, color,...), and are accessible in all of our class methods.

4. Constructor

   Every class can define special methods called Constructors, which are called when a new object is created. In our class we define a single constructor:

```
...
public circle(int r, int nx, int ny) {
            radius = r;
            x = nx;
            y = ny;
}
...
```

   You can recognize constructors by their lack of return type and their method name; it is always the same as the name of the class. Constructors are typically used to give initial values to an object. In our case we want to provide the radius, x position and y position of the circle when it is created. Creating an instance of the object looks like:

```
circle myCircle = new circle(10, 12, 30);
```

   We declare a variable called `myCircle`, which is of type `circle`. We then create a new circle object, with initial values for the radius, `x`, and y positions.

5. Methods

---

[5] Processing has already defined PI, TWO_PI, and HALF_PI for us, but we'll redefine it here to illustrate the point.

The remainder of our class is comprised of methods, the actions of our object. We define these methods just like any other function in Processing:

```
...
public float area() {
      return PI * (float)radius * (float)radius;
}

public boolean pointInCircle(int px, int py) {
      float pointDistance = (px - x) * (px - x) + (py - y) * (py
      - y);
      pointDistance = sqrt(pointDistance);
      if (pointDistance < (float)radius) {
            return true;
      } else {
            return false;
      }
}
...
```

In our circle class we have to methods, `area` and `pointInCircle`. Both of these methods can use the instance properties we defined; notice that the `area` method can use the `radius` and `PI` variables, even though they aren't directly defined in the method.

Objects in Processing

We'll use objects in Processing by importing them and by writing them ourselves. Imported objects come in the form of imported libraries, which can be found in the "Sketch" menu of the Processing environment. To write our own objects, we create classes inside[6] our Processing code. We'll create a quick example here, but you could even copy the circle class above into your processing code and use it. For now, though, lets write a simple class that remembers a string of text, and draws it to the screen when we want it. We'll assume we loaded a font in our `setup` method:

```
textClass myText;
void setup() {
      // ... load font and set it as the text font ... //
      size(300, 300);
      background(0);
      myText = new textClass("Text from the textClass object!");
}

void draw() {
      text("This is from the draw method", 10, 100);
      myText.draw(10, 200);
}

public class textClass {
```

[6] In fact, what we are writing are technically called *Inner Classes*.

```
        String classText = “”;
        public textClass(String t) {
              classText = t;
        }
        public void draw(int x, int y) {
              text(classText, x, y);
        }
  }
```

Our simple class has an instance property (`classText`), a constructor (`textClass`) and a method (`draw`). Whenever we create a new `textClass` object, we give it a string to remember; and when we call the `draw` method of our object it draws the text to the screen. Note that the class we've created has access to the Processing functions for drawing to the screen.

Vectors - Basics of Motion

To animate objects in the 2-dimensional space of our screen we need to know how to represent and manipulate all of the data about how our object is moving. To do this we'll use Points and Vectors. A point, in our 2D screen space, is just a coordinate representing the position of our object (the x and y position). Vectors represent a direction and a magnitude, and are represented in our case just like a point, with two values showing movement in the x and y direction. We'll call our magnitude values X and Y. Vectors are useful because we can show a rate of movement; for instance an animated car moving across our screen from left to right might have a position of: (x = 40, y = 10) and a velocity of (x = 5, y = 0). So our car is moving at 5 pixels per frame.

In our programs we'll be using:

Position: (x, y) the object's location

Velocity: (x, y) the rate at which the object is moving

Acceleration: (x, y) the rate at which the velocity is changing

One of the more common code snippets we'll use is:

```
velocity = velocity + acceleration;
position = position + velocity;
draw object to screen;
```

This is obviously a bit of pseudo code, but forms the basis of how we'll start applying physics to our objects.

The code above doesn't take into account that each of our vectors has two components we need to work with (u and v), so considering the U and V points of each our pseudo code might more closely resemble:

```
Vx = Vx + Ax;
Vy = Vy + Ay;
Px = Px + Vx;
Py = Py + Vy;
draw object to screen;
```

This highlights a significant difference we must account for in working with vectors; instead of the normal mathematical rules we're accustom to, we'll need to use Vector math to work with our values.

Given two vectors A and B, each with two components X and Y, we need these formulas:

Vector Addition

$C = A + B : C_x = A_x + B_x ; C_y = A_y + B_y$

Vector Subtraction

$C = A - B : C_x = A_x - B_x ; C_y = A_y - B_y$

Vector Multiplication

When we multiply vectors we'll actually be multiplying one vector by a scalar (non-vector) value. In this case we'll call the value N. This is effectively scaling the length of our vector by the value N.

$C = A * N : C_x = A_x * N ; C_y = A_y * N$

Vector Division

Similar to multiplication, when dividing vectors we'll be dividing the value of one vector by a scalar value, again in this case N. This will scale down the length of the vector.

$C = A / N : C_x = A_x / N ; C_y = A_y / N$

Length of Vector

It is often useful to find the length, or magnitude, of our vector. We can use the Pythagorean theorem to do this.

Length of $A = sqrt(A_x * A_x + A_y * A_y)$

Normalized Vector

Once we know the length of a vector we can Normalize our vector, or convert it into a vector whose length is 1.0. This is useful, in that it describe our direction, without assigning a magnitude.

Normalized A = A / Length of A

Vectors in Processing

To work with vectors in Processing we'll be creating and using our own Vector class. While other classes exist to manipulate vectors (and you're welcome to use those once we get going), writing your own vector class can help to understand what's going on with the numbers. The rudimentary vector class we'll be using looks like this:

```
public class Vector2D {
      private float x;
      private float y;

      Vector2D(float nx, float ny) {
            x = nx;
            y = ny;
      }
}
```

The remainder of the code we'll fill in as we go. Notice that we used the keyword `private` instead of `public` when declaring our instance variables `x` and `y`. This creates *encapsulation,* which means we need to write methods to get and set these values. This is another strength of Object Oriented programming; it allows the coder to control who has access to values, and how those values are stored, manipulated and validated.

Homework

- Apply velocity and acceleration to something other than the motion of an object
- Create a single primitive object (square, circle, rectangle) and try and give it a "personality" using velocity and acceleration.
- Anything of your own devising that uses Vectors or Velocity/Acceleration.