ART 150 – Intro to New Media Arts
Processing Basics – Drawing Machine Assignment
Adjunct Assistant Professor Alan Perry
Office Hours by Appointment
aperry2@saic.edu or aperry30@uic.edu

THE ABSOLUTE BASICS

A Processing sketch is made of at least two parts: a `setup()` function and a `draw()` function. A function is a list of instructions are are executed when the function is "called" by the code. `setup()` is automatically called once, and `draw()` is called 60 times a second. A typical `setup()` function looks like this:

```
void setup() {
      size(400, 400);
      background(120, 255, 120);
}
```

The code between the curly brackets, `{` and `}`, is the code that is executed every time setup is called. In this example, the code calls two other functions: `size()` and `background()`. These functions set the size of the canvas, or window, and the background color, which will be a light green. The semicolon (this symbol: `;`) tells the computer that it has reached the end of the line of code. `void` is the datatype that this function returns. It is `void` in this example, and all times we write the `setup()` function, because the setup function does not return anything.

When we call a function, like `size()`, we "pass" numbers, or variables, to the function. Then the function takes those numbers to perform whatever task it is designed to do. In the case of size, the numbers we give it are the pixel width and height of the canvas, or window, that the Processing sketch will generate for us.

A QUICK NOTE ON STYLE AND SYNTAX

Programming, and especially Processing, has its own "style" – functions and variables are written in a specific, conventional way. Computer programs are very bad at reading between the lines, like humans are. You must be very specific with your code or the computer will get confused and you will get an error. To avoid this, you must make sure the capitalization of certain letters is correct – especially functions and variable names.

Processing generally uses what is called "camel case," which means that for any given function or variable with multiple words in it, you need to keep the first letter of the first word undercase and all the other first letters capitalized. For example:

Good: `camelCase`
Bad: `CamelCase`
Bad: `camelcase`

Good: `reallyLongVariableName`
Bad: `REALLYlongvariablename`
Bad: `ReallyLongVariableName`

Processing also uses certain symbols for mathematics:

```
= assigns value to a variable
* multiplies
/ divides
+ adds
- subtracts
```

Processing also uses other, somewhat familiar symbols for conditional operations, or a statement that is either true or false:

Relative Operators:

```
== equals
>  greater than
<  less than
>= greater than or equal to
<= less than or equal to
```

Logic Operators:

```
&& and
|| or
!  not
```

We can use all of these together to make non-simple operations and conditional statements. For example, if we wanted to draw something, but only when the mouse is above the 100th row of pixels and being pressed, we could use this conditional statement:

```
if ((mouseY <= 100) && (mousePressed)) {
   ellipse(mouseX, mouseY, 40, 40);
}
```

In plain human English, this is saying: if the mouse cursor position on the Y axis is less than or equal to 100, and it is true that the mouse is being pressed currently, then draw an ellipse at wherever the mouse cursor is currently.

If statements are covered more later in this handout.

READING REFERENCE DOCUMENTATION

https://processing.org/reference/

Reading reference documentation can be intimidating, especially for beginners. Developing this skill will make you a strong creative coder. You will be able to understand what functions do when you write your own code or run into code in the wild that you may want to use.

If we look at the `rect()` function's documentation, we will see some examples of the function being used and the result. Underneath that we see a description of what the function does, an example of the syntax used, and a list of the parameters used in the syntax example above. Farther down we see what the function returns (in this case `void` – meaning that it does not return anything) and two related functions, `rectMode()` and `quad()`.

If we compare the syntax and the parameters we can see what the numbers we pass to the function do. In this example, when we use the first syntax, `rect(a, b, c, d)`, we can look at the `a, b, c,` and `d` entries in the parameter section to see what those numbers do.

The parameter entry for `a` is below.

```
float: x-coordinate of the rectangle by default
```

This tells us that the first number we pass the `rect()` function is for the x-coordinate of the rectangle. Looking forward, we see that the second number is for the y-coordinate, and the 3rd and 4th numbers are for the width and height of the rectangle.

LOOKING AT SOME FUNCTIONS

Here are some functions that we have used in class. I have taken their syntax and put explanatory words for the parameters, as a kind of cheat sheet. For more information about these functions and how they can be used, review the code on the class website (www.github.com/aperry2/UIC_ART150), play with the variables, and look at the reference documentation on them.

```
size(canvas width, canvas height);

rect(x, y, width, height);

fill(red color, green color, blue color);

fill(color variable);

line(x1, y1, x2, y2);

ellipse(x, y, width, height);

background(red color, green color, blue color);

background(color variable);

strokeWeight(line thickness);
```

```
stroke(red color, green color, blue color);

stroke(color variable);

noStroke();

save("filename in quotes and ending in .jpg");

translate(push origin this many pixels in x, push origin this many
pixels in y);

println("text to print to console " + a variable);
```

LOOKING AT SOME VARIABLES

Variables are numbers that can be passed to functions or to do some kind of math with. This is useful – necessary even – if you want any kind of change to happen in your program. Depending on where you initialize (create) your variables, you may not be able to use them in every part of the program. Global variables should be initialized first, before the `setup()` function. Variables initialized in other places will only be able to "live" in the curly brackets they are initialized, or created, in.

```
int x = 0;
```

This line of code creates an `int`, or integer, variable and assigns it a value of 0. An integer is simply a whole number, or a number without a fraction or decimal, such as 0, 1, 5, -20, etc.

```
float percent = 0.9;
```

This initializes a `float` variable, which is a variable with a floating-point number, or a number with a decimal. The value we place our `percent` variable is 0.9, which is also equal to 90%.

```
color cyan = color(0, 255, 255);
```

Color variables are assigned values by calling the `color()` function and passing it a number between 0-255 once for a gray color, or three times for an RGB color. The number range for color data is between 0 and 255 because color data is stored in a byte, which is an 8-digit binary number. The largest number a byte can store is 255. (Some trivia: Processing allows for 16,777,216 unique RGB colors in the RGB color space, which is also known as "True Color" or 24-bit color, as it uses 24 bits to store color information).

```
PImage clouds;
```

This variable is actually an object, or a complex datatype (unlike a primitive datatype like `int` or `color`). We are initializing it above, but we are not assigning any value or information to it yet. We will need to call the `loadImage()` function in the setup function to assign information to it. This will be covered more later.

```
boolean isReady;
```

`boolean` variables are variables whose value is either true or false. They can be used in `if()` statements for the computer to determine if it should do something or not.

USING VARIABLES

We can pass variables to functions, or even pass math formulae to functions. For example, if we wanted to slowly increase the size of a rectangle over time, we could do so with the following code:

```
int time = 0;

void setup() {
   size(400, 400);
   background(50, 50, 50);
}

void draw() {
   rect(40, 40, 50 + time, 50 + time);
   time = time + 1; // "time++;" would also work the same here
}
```

In this code, we initialize an integer variable named `time` and assign it a value of 0. In the `draw` loop, we increase `time` by 1 each time `draw` is called (60 times a second). We can do this in shorthand by typing `time++` but I have written a more clear version above; either will work.

We can see by looking at the 3rd and 4th variables we pass to `rect()` above that each time `time` is increased, so will the width and height of the `rect()` we are drawing. You can experiment with the last line of code – what about increasing `time` by 5, or multiplying or dividing it?

`if()` STATEMENTS AND CONDITIONALS

The two other major parts of code structure are `if()` statements, also called `if/then` statements, and `for()` loops. An `if()` statement is like a fork in a path: if something is true, the computer will do this, otherwise it will do that. This "something" is what is called the conditional. If the conditional statement is true, the code within the brackets will run. If it is false, the computer will move along. A basic `if()` statement might be:

```
if (4 > 1) { // if this is true...
   // then run this code:
   rect(40, 40, 100, 100);
}
```

With the above code, as long as 4 is greater than 1, a rectangle will be drawn at x,y 40,40 with a width and height of 100. This is quite useless as 4 is always greater than 1. Here is a more illustrative example:

```
if (size == 10) { // if this is true...
   // then run this code:
   rect(40, 40, size, size);
   size = 0;
} else {
   // if it isn’t true, run this code:
   size++;
}
```

In this example, if `size` is equal to 10, a rectangle is drawn and then `size` is reset to 0. You will notice at the bottom what is called an `else` case: this is the code that runs if the conditional statement is not true. For this example, it increases `size` by 1.

You may have a situation where you will need to test many different conditionals. This can be done with `else if()` cases, like this:

```
if (x == 1) { // if x equals one
   fill(0, 200, 0); // fill the brush with a green color
   x++; // and increase x by 1
} else if (x == 3) { // else, if x equals 3
   fill(200, 0, 0); // fill the brush with a red color
   x++; // and increase x by 1
} else if(x == 5) { // else, if x equals 5
   fill(0, 0, 200); // fill the brush with a blue color
   x = 0; // and reset x to 0
} else { // else (aka if x is not equal to 1, 3, or 5)
   x++; // increase x by 1
}
```

`for()` **LOOPS**

`for()` loops are a way to make a computer repeat a block of code a certain number of times. The basic structure is this:

```
for (int i = 0; i < 29; i++) {
  fill(i * 10, 0, i * 10);
  rect(10 * i, 10 * i, 120, 120);
}
```

The first line is the beginning of the `for` loop, which looks a lot like the `if()` statement. However, instead of a conditional operator, we really have 3 different lines of code crammed in between the parentheses. Broken down, there are:

```
int i = 0;
i < 29;
i++
```

Broken apart they look very familiar. The first line initializes an integer variable named `i` and puts the number 0 in it. The second line is a conditional operator, checking to see if `i` is less than 29. The last line increases `i` by 1. The 2nd line, `i < 29;` is the loop exit condition. As soon as `i` is no longer less than 29 (i.e. 29), the computer exits the loop.

What is so nice about `for()` loops is that you can use the incremental variable, traditionally named `i`, like any other variable. In the example code below, and on the website under `week3` as `forLoop.pde`, the `for()` loop is used to create a cascade of rectangles whose colors change with every iteration in the `for()` loop.

```
void setup() {
  size(400, 400);
  background(10, 10, 10);
  noStroke();
}

void draw() {
  for (int i = 0; i < 29; i++) {
     fill(i * 10, 0, i * 10);
     rect(10 * i, 10 * i, 120, 120);
  }
}
```

`PImage` AND IMAGES

If we want to use images in our Processing sketch, we must first create a special kind of variable called an object and load the image data into that object. We do that like initializing any other variable without assigning any value to it:

```
PImage clouds;
```

Then, in the `setup()` function, we must use `loadImage()` to load an image into the object, like so:

```
void setup() {
   clouds = loadImage("clouds.jpg");
}
```

This will load the "clouds.jpg" image into the `clouds` PImage object. We need to do this in the `setup()` function because we cannot call a function outside of another function (confusing, I know). **The other important thing here is to make sure that the file you are loading**, `clouds.jpg` in this example, **is saved in the same folder as the sketch**. The sketch must also be saved to generate the folder. You can open the sketch folder by going into the top bar menu and selecting "Sketch -> Show Sketch Folder".

So now that we have the image loaded into our PImage object. If the image we loaded into the object is the same size as the canvas, that is, the dimensions we pass to the `size()` function we call in `setup()`, then we can use the PImage object to pass it to the background as we have been doing with colors.

```
background(clouds);
```

We can also use another function, `image()`, to place the image we loaded into the PImage object anywhere on the canvas. When we call `image()`, we pass it at least 3 variables: the name of the PImage object variable, an x, and a y position. If we wanted it to draw wherever the mouse cursor is, we could do this:

```
image(clouds, mouseX, mouseY);
```

We can also scale the image by passing `image()` two more variables, a width and a height:

```
image(clouds, 0, 0, mouseX, mouseY);
```

So, if our image file is not the same dimension as the canvas, but we don't want to (or can't) edit the image, we can approximate the effect by using the following line of code at the very beginning of the `draw()` function (or the end of the `setup()` function):

```
image(clouds, 0, 0, width, height);
```

INTERACTIVITY

Interactivity can be achieved by a few different methods, but here I will explain three: the integer system variables `mouseX` and its sibling `mouseY`, the boolean system variable `mousePressed`, and event functions.

A system variable is a variable that is determined by Processing. In the case of `mouseX` and `mouseY`, Processing determines the value of these variables by looking at where the mouse cursor is on the screen. This can be used to position an image or shape on the screen, or some other kind of interactivity.

Another system variable we can use for interactivity is the `mousePressed` variable. Not to be confused with the `mousePressed()` event function covered below, the `mousePressed` variable (with no parentheses) is a Boolean variable that returns true when the mouse is pressed and false when the mouse is not pressed. We can use it in an `if()` statement to draw a red square when the mouse is pressed, like this:

```
if (mousePressed) {
   fill(255, 0, 0);
   rect(50, 50, 200, 200);
}
```

In the reference documentation, under the "Input" section, there is a list of system variables and event functions that we can use to create interactive sketches. An event function can be added by creating a new top-level function underneath `draw()`. Here is an example that draws a green ellipse at where the mouse cursor is when the mouse is pressed:

```
void mousePressed() {
   fill(0, 255, 0);
   rect(mouseX, mouseY, 60, 60);
}
```

If you experiment with this code, you will realize that `mousePressed()` runs as soon as a mouse button is pressed, and it only runs once. You will notice that `mouseClicked()` runs only once you let go of the mouse button. Each event function runs at a different time and for a different duration. The reference documentation will explain what these differences are.

CUSTOM FUNCTIONS

Lastly, you might find yourself in a position one day when you will want or need to write your own custom function. So far, we have been using functions that are part of the Processing IDE whose inner workings are largely a mystery to us. We have been writing `draw()` and `setup()` functions, but those are really more like a "cast of characters" and "table of contents" then proper chapters that we have written ourselves. Perhaps we have used an event function, which we have had to write out ourselves as well. But all of these functions do not allow for any passed variables, and none of them return data either.

If we wanted to write a custom function that does not take any passed variables or return any data, we would write a void function, perhaps named `drawHead()`:

```
void drawHead() {
  fill(skin);
  ellipse(400, 400, 200, 400);
  fill(eyes);
  ellipse(340, 360, 80, 100);
  ellipse(460, 360, 80, 100);
}
```

When we call this function in `draw()`, it will execute all of the code in the function at that time. But, what if we want to draw this alien head at a different place each time we call the function? We can write our custom function to accept passed variables. Let's say we want to control the x,y location that this is put at. To do this, we must type certain things in between the parentheses after the name of the function (and some math on the variables we pass to the `ellipse()` functions).

```
void drawHead(int x, int y) {
  fill(skin);
  ellipse(x, y, 200, 400);
  fill(eyes);
  ellipse(x - 60, y - 40, 80, 100);
  ellipse(x + 60, y - 40, 80, 100);
}
```

What we must type in between the parentheses is the variables we will use in the body of the function. We must initialize the variables in this header before they are used in the function's code. Once our custom function is finished running, these variables will be deleted: they will NOT be stored as they are NOT global.

The last thing about custom functions is what they return. Up to this point, we have been writing functions that return `void` – or nothing. Perhaps we want to write a custom function that will add 100 units of red to a color. We will need to write a function that returns a `color` variable and takes in a passed `color` variable. Something like this:

```
color addRed(color oldColor) {
  color newColor;
  newColor = color(
     red(oldColor) + 100,
     green(oldColor),
     blue(oldColor)
  );
  return newColor;
}
```

This custom function uses three new functions: `red(), green(), and blue(),` to extract the number variables of each color from the `oldColor` variable, which was initialized in the function header and is passed to this function when we call it. We then assign new color data to the `newColor` variable, which we then `return`. However, we add 100 to the red amount of the `oldColor` variable before we `return` it.

The above code is under `customFunctions_2.pde` on our course website under the folder `week3`, if you would like to play with it.