PROCESSING
LOOPS, ARRAYS, OOP

LOOPS

Loops are repeated bits of code until something is true or false. Through this, you can repeat actions, do specific actions, or react to actions in a more specific way than with just the draw() method. There are two types of loops, a for loop and a while loop

FOR LOOP

```
for (variabe creation; if statement; increment){
  code to repeat
}

for (int i = 0; i < 10; i++){
  println(i);
}
```

This is 98-99% of the time the loop I use!

WHILE LOOP

```
while (if statement) {
   do code;
}

int i = 0;
while (i < 10){
println(i);
i++
}
```

USE

If you want to move a circle a certain amount you use a loop. If you want to add to a variable until something you use a loop. You use a loop for almost everything with cool programs. Another really cool thing with loops is to go through arrays and change their data. But what does that even mean? What is an array?

ARRAYS

Imagine you want to make five circles which each have their own X and Y coordinate and move at their own speed in some direction. Right now with what we've learned, we could only do :

```
int xPos1, xPos2, xPos3, xPos4, xPos5;
int yPos1, yPos2, yPos3, yPos4, yPos5;
ellipse(xPos1, yPos1, 100, 100);
xPos1++;
yPos1++;
```

This is horrible! How can we fix this? Well, with arrays!

ARRAYS

An array is a variable that can store a bunch of variables within it. So for instance, you can make an array that can store 10 int numbers, each with their own place to be changed. It's like a house with all the separate rooms, except they're microscopic and can be destroyed in an instant:

type name[] = new type[size];
OR
type [name];
... in void setup:
type = new type[size];

ARRAY EXAMPLE

```
int array[] = new int[5];
```

This array now stores 5 int values that can be changed on their own in one neat package. Here's how:

```
array[1] = 40;
array[2] = 4;
array[3] = 4;
array[4] = 9999;
array[5] = 30; <- THIS WON'T WORK!
```

ZERO INDEX

It's a bit strange, but arrays are zero indexed in all programming languages. This means the array's first variable or index starts at 0, not 1 like we normally think. Because of this, even though we made the array 5 variables long, we can't use [5] because it doesn't exist.

[0], [1], [2], [3], [4] is 5 long!

Now println(array.length); will still say 5 long, because it really is, but we need to remember that the indexes start at 0 and go from there, not 1.

EDITING VALUES

We can use a for loop to set values for each array in one line of code:

```
for (int i = 0; i < array.length; i++){
  array[i] = 40;
}
```

We can replace the [3] with a variable, in this case, the for loop variable and give each a value so quickly!

OOP

OOP stands for Object Oriented Programming. This changes our idea of code from being actions in a sequence to making objects with abilities that we define. This means we won't be changing the code within methods, but rather, we will be creating our own methods. Better yet, we'll be creating objects that store all these methods!

MAKING METHODS

Methods are blocks of code that are grouped together. So far, we learned that titling a group of code a certain thing can do a certain task. This isn't the main use of methods. When we create methods, we create blocks of code that aren't used until we tell the program to use it.

METHOD EXAMPLE

```
void setup(){
 size(800, 800);
}

void makeCircle(){
  ellipse(width/2, height/2, 100, 100);
}
```

If we run the code we'll see that nothing happens. Now put makeCircle(); in the setup()'s {}s

Now suddenly it makes a circle!

RETURN AND VOID

I apologize. For the longest time, I may have left you wondering what void does. It's just there! Now I feel you're ready to be taught.

When you put **void** before the method name, you're saying the method will not return a value. ... What does returning a value mean?

Well, let's do an example

RETURNING A VALUE

```
void setup(){
 size(800, 800);
 float thing = makeCircle();
 println("Thing is " + thing);
}

float makeCircle(){
  ellipse(width/2, height/2, 100, 100);
  float num = random(200);
  println("num is " + num);
  return num;
}
```

RETURNING VALUE

Woah! We set a variable equal to a method! The return replaces the method wherever it is called (told to execute, makeCircle(); in setup) with the variable that was returned (thing after the return line)

Now in the method line, we have to say what type of variable will be returned, so in this case we will put float (and not void!). If we returned a string value, we would put String, int int, boolean boolean, etc.

BACK TO THE VOID

Let me repeat **void**. Void is the word we put in front of the method name whenever we don't want to return a value. In other words, when we put void, there will be no return statement. We can't set any value equal to the method like before if it's void, because nothing will be returned. Void is used to do a task, but have no information gained back from that. Just do whatever is in your method, then bug off!

... WHY ()S?

Double apology. I now feel ready to tell you why we put ()s on all these things.

As you know, sometimes we put values in these (), like color(255), size(800, 800), rect(x,y,l,w);

This data we put inside ()s is called the parameters. We can put data in the parameters to be used. What does this mean?

PARAMETERS

```
void setup() {
  saySomething("HI!");
}

void saySomething(String word) {
  println(word);
}
```

Change the HI! to some other word. As we see, the word variable inside saySomething changes to whatever we put inside the ()s

PARAMETERS CONT

We see that in saySomething, instead of putting (), we put (String word)

This says that whenever this method is called, whatever is put inside the ()s will be this variable. It's like saying String word = "HI!" but only inside the saySomething method. From here, we can use the word variable just like normal. Again, this can be any value.

CLASSES

For right now, a class is a collection of methods grouped together. From here, you can create what is called an object.

First, here's the definition :
An object is an instance (a new creation) of a class that has all the methods and variables within a class.

If you're like me when you first read this, you have no idea what this means. Luckily, analogies exist for a reason! Use them!

OBJECT ANALOGY

Let's imagine we're in a factory and there's a bunch of machines.

Every machine has a button that says "New", a conveyer belt on the other end that spits out the finished product, and a big metal box that creates the thing it's designed for whenever you hit the New button. This metal box thing is the class, and the thing it spits out is the object.

ANALOGY CONT

You only make this metal box once, but you can suddenly create an infinite amount of objects (things) that come out from this. All of these that come from the same machine have the same characteristics, perhaps the ability to move on command, a color, and an X and Y, and the ability to show itself on screen.

You can mess about with each object after it's created. The class' purpose is just to make the object. Knowing this, let's ignore the proper definition and move on to creating a class

PT 1 - ESTABLISH GOAL/IDEA

Let's first see in a word sense what we're going to add. We want to say **if** the center of the circle (we can change it to the side later) goes past either side of the screen, we want it to reverse directions ... how could we say the first line of the if statement using one of the circle's variables?

PT 2 - MAKING THE IF STATEMENT

We want to add our if to the (draw() / setup()) method (Circle one!)

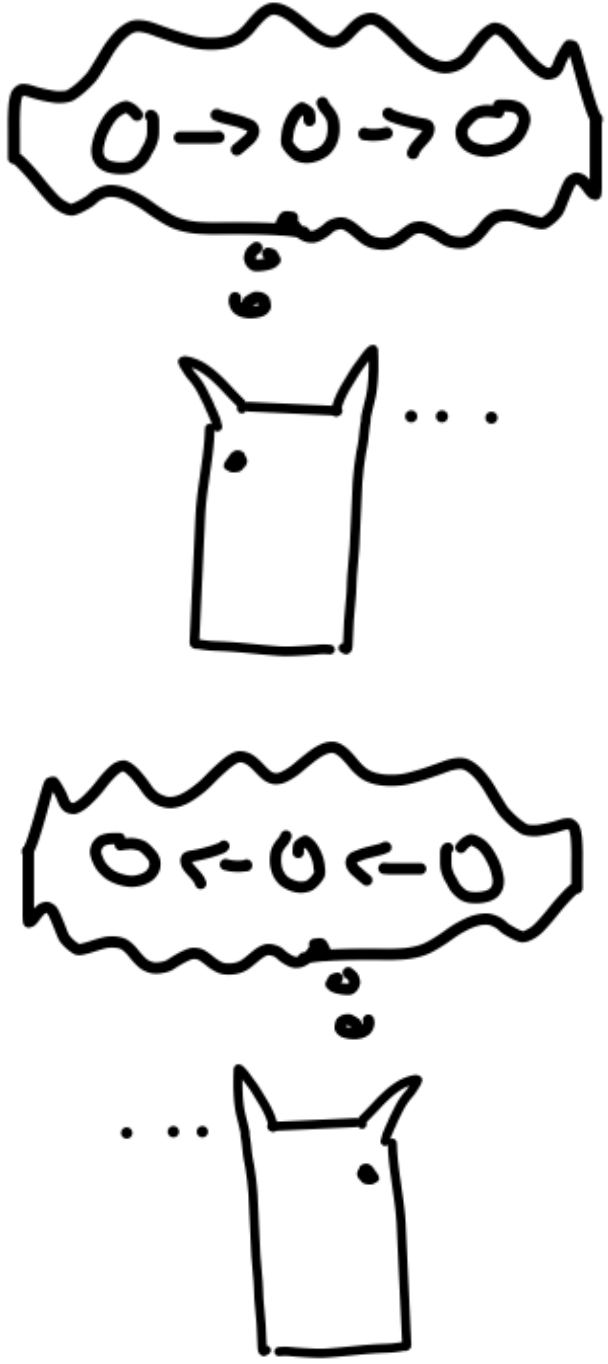
This is what it looks like:

```
if (xPos > width || xPos < 0)
  (reverse circle's direction)
```

This is saying "**If** the X of the circle is less than **or** greater than the window size, reverse its direction so the circle is going back into the window

PT 3 - REVERSING THE DIRECTION

Now we need to make the circle head in the other direction until we hit that side, then back again, then back again forever. Essentially, whenever the ball hits the wall, we want it add a number that will go in the opposite direction of where it's currently going. The best way to illustrate how we'll do this is with a math example I'll show now.



PT 3 - REVERSING THE DIRECTION

So we need to add a number to the xPos the opposite of what is currently being added. Can we just say if the xPos is greater than the width we just subtract 10? Why?

What can we use that can **change**, and can be **added** to the xPos **depending** on where the xPos is?

PT 4 - MAKING THE VARIABLE

Let's make a new variable let's say speed, directly below our xPos coordinate.

```
int speed = 10;
```

Now, using what we covered, how can we say that if the xPos is greater than or less than the sides, we reverse the directions of speed.

PT 5 - THE FINAL PART

We can make the if statement like this:

```
if (xPos < 0 || xPos > width)
 speed *= -1;
```

Now insert this in the draw function, and see what happens

PROCESSING LOGIC