What is This Course?

Introduction to Programming with Java is one of DMA's most popular courses. Literally thousands of student take this course each summer, so it's an important one. The goal of the course is to provide an introduction to programming for students using the Java programming language.

Why Java? Java can be hard! This is true, but it's also very transparent, extremely popular, and has a good combination of explicitness and abstraction. In order to make it easier, the course actually starts in a different programming language, Processing.

Processing is an open-source programming language created by some awesome people at MIT, and it's great. It is based on Java, and is in fact a simplified version of Java. It removes the need (but not the functionality) of a lot of the advanced things that Java requires, which can get in the way for a new programmer. Things like public static void main(String[] args), or declaring everything public vs private, or even containing EVERYTHING within a class.

This will allow students to start programming from minute one without having to deal with all those things. Will the course eventually go over those things? Absolutely. But not on Day 1.

Who is Taking This Course?

This course is part of DMA's Teen program, which means you can expect your class to consist of students anywhere between the ages of 12 to 18. This can present a challenging teaching environment, but you can get some tips below in the Teaching Practices section of this page.

The experience of the students in your course will also vary. Some will be completely new to programming, some will have some experience in school, some may even be pretty great at it already (but that should be rare). You will probably encounter students who have gone through the DMA online eLearning course, Fundamentals of Programming. These students will have spent several hours online learning to program with a Javascript version of Processing. You will also encounter students who are in week 2 of DMA's Academy of Programming 101, which means they will have just spent a week learning to program games in python. For both of these groups students (some may even belong to both!), day 1 may also be a little repetitive. So again, please check the Teaching Practices section.

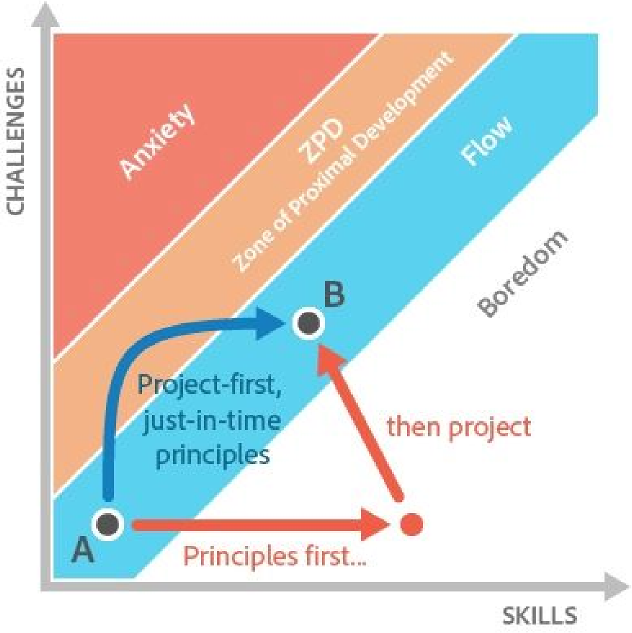
How to approach this curriulum?

Yes. You may know Java, but the lessons here in this course will provide you with a great way to teach Java, and more importantly, the core programming concepts. This course is also a part of several academies, which means that students will be moving on from this course into one of several other more advanced courses, and the curriculum in those courses will probably build on what you taught in this course. Also, there are probably things in this course that you are not familiar with, such as Processing, or how to using processing as a graphics library in Eclipse.

Does this mean you shouldn't experiment, show the students some your own projects, or even come up with your own lessons? Of course not! You are encouraged to do so. Just be sure you are covering the concepts outlined in this curriculum. And if you do create original stuff, share it with your fellow instructors!

DMA believes in the learn by doing model in which a student works through the design process in a creative, positive and supportive environment. Project based learning activities are the backbone to a successful DMA experience. Students will be engaged in hands-on learning experiences as they are challenged to address high level questions and develop solutions to problems through an iterative process.

Project Based Learning Model



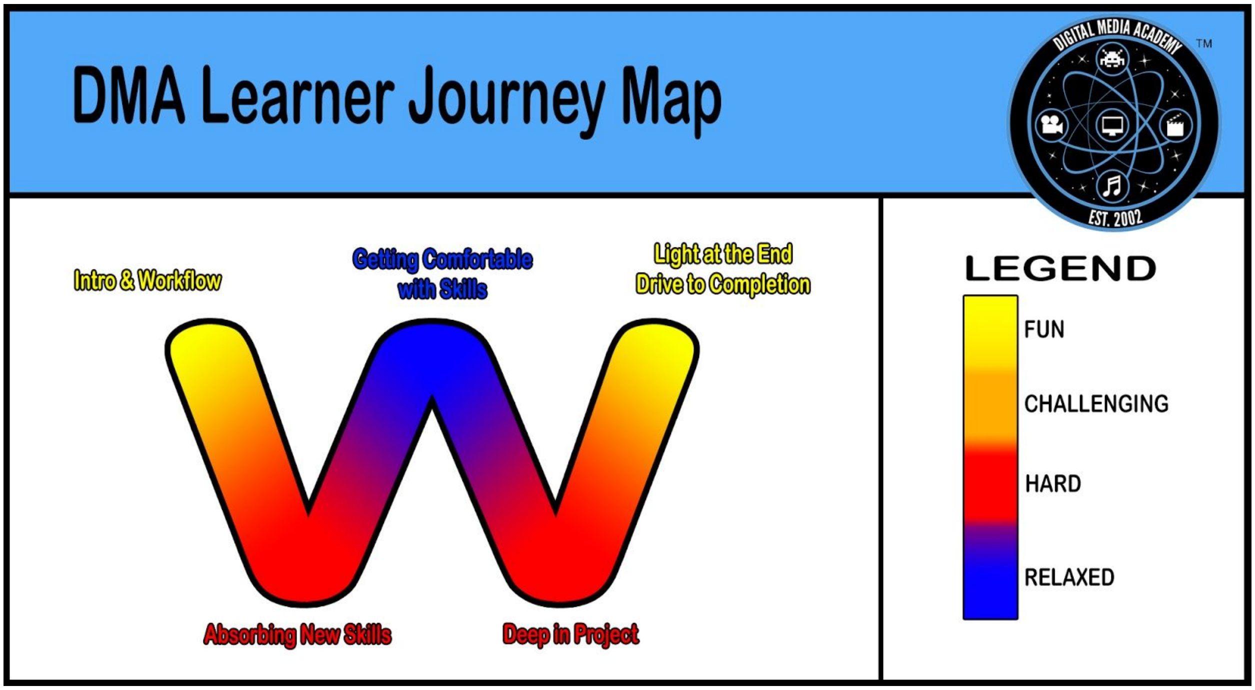
WHAT TO AVOID:

When concepts are taught first without a context for application and then followed by a project in which they are applied, there is a higher likelihood of student boredom or detachment.

WHAT TO DO:

However, when an inquiry- or project-based lesson is presented first, the just-in-time scaffolding of underlying concepts enables students to reach the zone of proximal development and a state of mental flow more quickly. In other words, students are more engaged and stay focused longer.

During a week of instruction a student will experience a range of emotions. Paying attention to how a class or even a project is structured to lead a student through this range is a fundamental step in the development process to ensure a student's success. This design is intended to provide a balance of fun, challenging, difficult and relaxed moments at the right times to keep a student engaged and motivated.



Best Practices - The DMA Model

The DMA model of watch me, do with me, and then do on your own holds very true to programming. Have students follow along with you when presenting a subject, and then allows them to practice those skills. For example, write a for loop, explain it, then have the students write another for loop following you, while re-explaining the concepts, and then give them small challenges and exercises to practice for loops.

Dealing with Advanced Students

For both advanced and struggling students, TAs are really important. Ask your TAs to challenge advanced students by giving them extra challenges that you can find in the advanced extensions of the lessons. Make sure the TAs know who those students are. If you have a large a group of advanced students, you can even assign a TA to watch over that group specifically, and even do advanced lessons (with advisement from the instructor).

Dealing with Struggling Students

Be sure to identify students who are having a hard time on Day 1. Go out of your way to engage them and make things clear. Again, point them out to your TA. You can also try to seat them next to an advanced student who doesn't mind being helpful. Use checkpoint lessons and work time to sit down with the students (or have a TA do it) to help bring them up to speed. It's important to manage your time to make this possible. Don't try to move the class to fast if a group of advanced students wants to keep going. Instead, provide them with extra challenges or tasks.

Dealing with Troublesome Students

Most of the students you meet at DMA are pretty great. Sometimes you get students who have behavioral problems. Most of the time, it is simply a matter of getting these students engaged. Try to give them assignments or even responsibilities instead of punishment. If that is not effective, simply a few warnings that the student will have to go without computer time can be helpful to encourage good behavior. Beyond that, simply notify your Staff Director and they can contact the parent for you.

Dealing with Special Needs

Since our courses deal with pretty advanced subjects, most of the special needs students you will see will be high functioning. Usually, this means they fall somewhere on an autism spectrum.Read this (Links to an external site.) guide on dealing with autistic students. Beyond that, you can really just treat them like other students, or more importantly, treat all students the same. It is NOT your position to diagnose. This is what Autistic students will respond to the best. It is NOT your position to diagnose. Parents should provide any necessary information which should show up on your roster. If you need to do something special, note that and do you best to accommodate. It is important to remain patient. Beyond that, remember that EVERY student is different :)

STEPS IN THIS LESSON

1. Introduction to Processing

Most intro programming classes begin with text-based exercises and basic command line prompts. With Processing, we will be learning to program visually through manipulating shapes. Processing syntax is very similar to Java, and a good understanding of the basic concepts of programming while in Processing will give you a head start when we jump into the more complex syntax that Java demands.

Why learn Processing?

The Processing environment is actually written in Java! Processing code is translated (compiled) into Java and is run as a Java program. Starting with Processing allows you to have a basic graphics library at your disposal without worrying about more advanced Java concepts and object-oriented programming. The goal of this lesson is to introduce you to programming and get you comfortable with drawing shapes through code.

2. Your first Program

Every new programmer, and every experienced programmer starts learning a new language by writing the same program: Hello World. The goal of this program is the print the words "Hello, World!" out on the screen.

Open Processing (a new sketch) and type:

```
print("Hello, World");
```

Make sure you type it exactly. Hit the play button on the toolbar to run your application (or use the keyboard shortcut: command + r).

Look at the bottom of the text editor, in the black part of the screen, you should see the words Hello, World. This is where you will find any text output of the program. There will also be a small window open with a gray background. That's where drawings show up. Great!

Let's get to drawing with code! Type the following into the text editor area:

line(0,0,100,100);

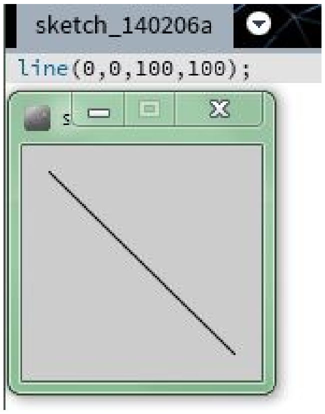
This is our first **function** (more on functions later)! The name of the function is line. The line function is followed by brackets ( ) containing 4 numbers, separated by commas. The values within the brackets are called parameters (Provide a definition of a function and parameters, simple explanation).

TEACHER TIPS

Teacher TipsIf you have an older set of students (15+) they should have learned function notation in math (i.e. f(x) = 2x +4). If they are having a hard time understanding function syntax, this might help them make the connection. Explain how f is the name of the function, and x is your parameter. The expression after the equals sign is just the code that the function executes (i.e. Multiply my parameter x by 2 and then add 4).

Notice the semi-colon (;) at the end of the statement. The semi-colon is a very important part of Java syntax, it tells the compiler that you have finished your statement. Syntax is an important part of programming. It's essentially the grammar of the code, the rules by which you write your commands. Unlike your twitter, however, programming is very strict about how you write your commands. If you don't follow the syntax rules correctly, it either doesn't work at all or doesn't work correctly.

Hit the play button on the toolbar to run your application (or use the keyboard shortcut: command + r). Your applet will open in a new window and should look something like this:



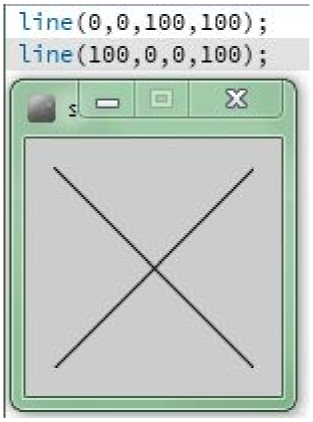
```
line(0,0,100,100);
```

Programming languages are precise and will not run if you forget a semicolon or give your function the incorrect number or type of arguments. (If you mess up the Syntax) If you receive an error message while trying to run your application, don’t panic! The compiler will catch any errors in your code and offer suggestions to help fix the problem. This is an invaluable resource as you progress through the lesson material. Even professional programmers make errors quite often and that is why it is important to test your code often!

Let’s add another line:

line (100, 0, 0, 100);

Try running the code again.



```
line(0,0,100,100);
line(100,0,0,100);
```

We have created an X shape across the screen by constructing two lines, but how exactly does this work?

The numbers in the line function represent coordinate positions. You may have learned in math class that coordinates are represented by two numbers in brackets representing the X (horizontal) and Y (vertical) position of a point. The line function takes two coordinate points as arguments and draws a line between the two points.

The coordinate system in Processing (and in the graphics package we will be using in Java) starts from the upper left hand corner (position 0, 0). This means that as your x value increases, the point moves RIGHT, and similarly, as your y value increases, your coordinate will move DOWN. This is important to remember, as it will determine the placement of all the elements coded in your application.

By default, Processing runs on a 100 x 100 pixel window (we'll see in a bit how to adjust the screen size). Our two lines are represented by two sets of coordinate points. Our first line function draws a line from point (0, 0) to point (100, 100) and the second line draws a line from point (0, 100) to point (0, 100). So we can say the structure, or the syntax of the line function is as follows:

line(x_start, y_start, x_end, y_end);

TEACHER TIPS

Teacher TipThis may be a good point to show the students processing.org and explain how the website can be used to discover and explore the functions that Processing has at its disposal. After you do that, the advanced students may want to start exploring shape-creating on their own and move ahead.

3. Rectangles and Circles (Ellipses)

Here is the code we use to draw rectangles and ellipses. We will continue to build on top of the line drawings we constructed during the previous section. This is the syntax of rectangles:

rect(x_position, y_position, width, height)

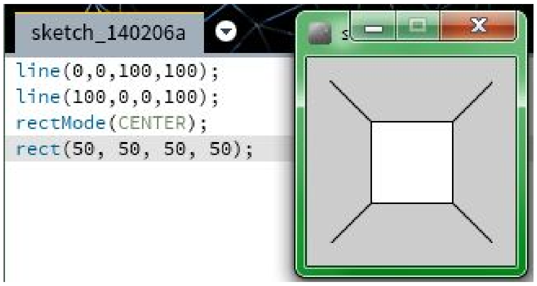
*Remember that X and Y positions start at the upper left hand corner, position (0, 0).

Let’s make some rectangles! Type this code below your line code:

rectMode(CENTER);

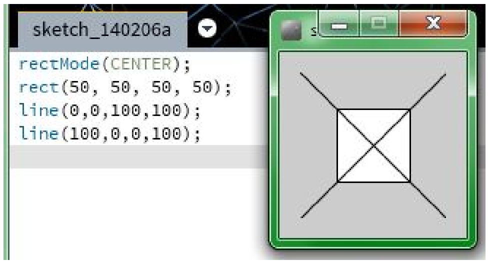
rect(50, 50, 50, 50);

Our rectangle will be drawn centered at position (50, 50) with a width of 50 and a height of 50. This effectively makes a square.



```
line(0,0,100,100);
line(100,0,0,100);
rectMode(CENTER);
rect(50, 50, 50, 50);
```

Run the code. You should see a square appear at the center of your lines. Notice that the rectangle is drawn on top of the lines. It is important to note that Processing draws images in the order that they are written in the code. If you place the rectangle code before the line code, the lines will appear on top of the rectangles. Try having the students rearrange the commands to see the result, such as drawing the square before the lines:



```
rectMode(CENTER);
rect(50, 50, 50, 50);
line(0,0,100,100);
line(100,0,0,100);
```

Next up, let’s draw an ellipse!

The arguments (parameters) for drawing an ellipse are very similar to that of a rectangle:

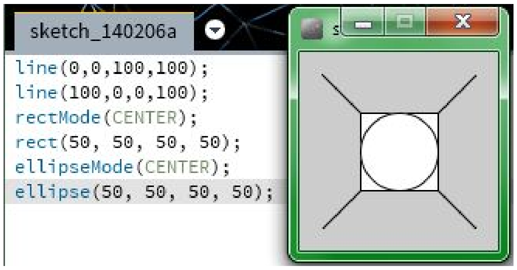
ellipse (x_position, y_position, width, height);

Type this code below your rectangle code:

ellipseMode(CENTER);

ellipse(50, 50, 50, 50);

Run this code again, and you should see a circle appear inside of your rectangle. (Here the line creation is back at the beginning).

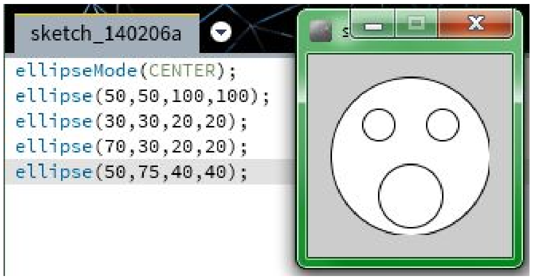


```
line(0,0,100,100);
line(100,0,0,100);
rectMode(CENTER);
rect(50, 50, 50, 50);
ellipseMode(CENTER);
ellipse(50, 50, 50, 50);
```

The code ellipseMode(CENTER) and rectMode(CENTER) mean that our coordinate positions (the first and second parameters of the function) for rectangle and ellipses are based on the center of our shape.

4. Make a surprised face!

To test your understanding of the coordinate system, the linear progression of commands, and parameter usage, try to make a simple face using ellipses (or other shapes, if you prefer). An example:



```
ellipseMode(CENTER);
ellipse(50,50,100,100);
ellipse(30,30,20,20);
ellipse(70,30,20,20);
ellipse(50,75,40,40);
```

Go ahead and save this file as something useful, like "ovalface.pde".

5. Make a traffic light!

This section introduces students to code that is a little more complex, and comments.

Open ‘traffic_light.pde’ and run the program. The code may look a bit complicated but we will cover all of this material shortly.

Run the starter code provided. You will see three different colored circles representing the traffic light colors, but they are a bit jumbled up.

The code lines that start with // are comments. They do not change the functionality of a program, but are used to provide helpful hints to the programmer. Comments are an essential part of writing good code. Effective comments let you understand what a program does before running it. This is especially useful if you plan on sharing your codes with others.

```
// this is a comment! tell me how your program works!

/*   This is a multi-lined comment.

     I could go on forever.

     Don’t forget to close me!

*/
```

Look for the comments labeled GREEN LIGHT, YELLOW LIGHT and RED LIGHT. We will be fixing the position of the circles to make a good looking traffic light.

If you are feeling adventurous, go ahead and adjust the first two arguments of ellipse for each light. Step-by-step instructions are provided below:

Let’s take a look at the green light ellipse code first.

```
ellipse (10, 20, 25, 25);
```

The vertical positioning looks fine (the green light belongs at the top), but the horizontal positioning is slightly off. Recall that increases in our x position (the first argument of ellipse) will move our circle to the right.

Let’s increase the value of x to 50:

```
ellipse (50, 20, 25, 25);
```

Run your application to double check your work. That position looks about right!

Moving on to the yellow light. The horizontal positioning looks find but the yellow light is too far down our traffic light.

To move the yellow light up, we need to decrease the y position value from its current value of 70. Let’s change it to 50:

ellipse(50, 50, 25, 25);

Much better! The red light is not positioned correctly at all, and is almost falling off of the screen. Can you guess what the correct x position is for the red light? By looking at our two previous ellipses, we can get the correct x position value of 50. When trying to adjust parts of your program, it is always useful to look back at the code you have already written, it usually contains hints about how to fix your current line of code!

ellipse(50, 100, 25, 25);

What about the vertical position of the red light? Try decreasing the value of the y position by 5 or 10 and run the code to see the difference. The correct spacing will be around 80.

The final positioning of the red light should be something similar to this:

ellipse(50, 80, 25, 25);

Run your code again to double-check your work, you should see a nicely organized traffic light!

Notice that the difference between the y position of the green light is

50 – 20 = 30 and similarly, the difference between the y position of the yellow light and the red light is 80-50 = 30. By comparing the values of the position of our shapes using simple math, we can further understand the relationship between the shape drawings relative to the dimensions of our application screen size.

6. Colors!

After the previous example, you may be curious about how the colors were created. You might have noticed the fill function listed before each ellipse function. The fill function has the following format for creating colors:

```
fill(RedValue, GreenValue, BlueValue);
```

Colors are created by combining values of red, green and blue color. You might recall from art class that all colors can be made by combining different amounts of these three primary colors. In Processing (and many other programming languages), each color can take a value between 0 and 255, representing the intensity of that color.

Make sure that your fill function is placed before the shape you wish to apply the color to.

If you would like to use a grayscale color, simply use a single number argument with a value from 0 to 255. This is particularly useful for setting background colors.

```
background(0);
```

Will set the background to black (the most extreme grayscale value).

```
background(255);
```

Will set the background to white.

TEACHER TIPS

Teacher TipWhat's an easy way to remember which number represents black and which represents white? Think of the number value as "adding" color; a higher number value indicates a "brighter" color.

Open up "colordrawing.pde" and run the application.

At this point, you should recognize all of the code in this example, except for the noStroke() function. By default, all shapes are outlined by a black line (stroke). The no stroke function removes these lines and allows the shapes to cleanly connect in this example (*expand on this later).

Play around with the values in fill for the different squares and see what colors you get by adjusting the red, green and blue values.

TEACHER TIPS

Teacher TipProcessing includes a built-in color wheel to help you get the number values for the colors you want. Select Tools -> Color Selector from the upper Toolbar. The values for R, G, B listed in columns on the right hand side. You can copy these values directly into the fill functions and see the result. If you find any colors that you like, make sure to write down the RGB color code for re-use in later sections of the course (and potentially your personal project).

7. Drawing Triangles and Other Bonus Shapes

Let's finish off our basic shape knowledge by learning how to draw a triangle. The triangle function takes 3 sets of XY coordinates, representing the three points of the triangle.

triangle(x1, y1, x2, y2, x3, y3);

Open up and run "triangle_draw.pde" to see some examples. It is recommended to have the students write this code themselves. Have them create a new sketch and quickly guide them through drawing a triangle or two.

Notice the first line of code:

size (200, 200);

The size function sets the screen size of your application (length, width). In this example, the size of the screen is set to 200 by 200 pixels.

8. Make your own creature

Now that we’ve seen how to draw lines, shapes, and add color to our drawing, we’re ready to create some drawings of our own! Take a look at the sample creature:

[IMAGE MISSING]

Let’s move on to making your own creature! If you feel comfortable with shape drawing and positioning, feel free to get started. You can also start by changing the colors and positions of the shapes in the sample application until you are happy with the results.

Here are a few fun things you can do to change up the look:

strokeWeight(number);

This will change the thickness of the lines surrounding the creature. Try changing the value and running the application to see the results.

stroke(number) OR **stroke(RedVal, GreenVal, BlueVal)**

The stroke function will change the color of the lines in the drawing. Recall that greyscale shades (from pure white 255 to solid black 0) consist of one argument, and colors 3 values between 0-255 represent the amounts of red, green and blue in the color respectively.
There are several things you can do to make this more challenging for advanced students or quick learners.

1. Ask them to draw a shapes using the line command. "Draw me a house to live in" is a good one. It should occupy them for a while.

2. Ask them to create an equilateral triangle. It will require a little bit of math for them to figure out the points.

3. For the traffic light, ask them to draw one from scratch.

4. Direct them to the processing.org website and ask them to learn how to make more advanced shapes like arcs.

Remember one of the primary objective of DMA summer camp is to get kids to think creatively. In this course, we are simply giving them the tools, and asking them to make stuff. If you've got a kid who gets the concepts of functions and parameters already, show them the processing.org website and tell him/her to go nuts and create something awesome.

Condition An if/then type structure, that allows programmers to have code execute only when certain conditions are met.

Static A program whose code executes once and does not react to anything the user does after it runs.

Dynamic A program which can react to the user as it runs.

Loop A syntax that allows the programmer to do repetitive tasks in a succinct manner.

Variable A name that holds a value in the memory of the computer.

Declare A statement in code that tells the compiler the name of a variable, and depending on the programming language, it's type.

Instantiate To give a variable a value or definition.

Null When a variable (or element) is not given a value or definition, it is referred to as being 'null'.

STEPS IN THIS LESSON

- **Mouse and Keyboard - Dynamic Mode, mouseX, mouseY**
- **Creating Functions - setup() and draw**
- **Variables - Greater Detail**
- **Conditionals - if, else with mousePressed**
- **Keyboard - the AND and OR operators**

1. Mouse and Keyboard Interaction

Static vs. Dynamic: What stays the same and what changes?

It’s time to learn about two functions that Processing uses to set up the initial screen and run dynamic graphics/code. We will start with mouse interactions and move on to keypresses near the end. User interaction through the mouse and keyboard will be an important part of designing games later on in the week.

All of the examples that we have seen so far (with the exception of the mouse drawing sample) have been static programs. When we run our code, all of the graphics that are displayed on the application screen are determined at runtime. Once the application has loaded, there are no changes to the screen.

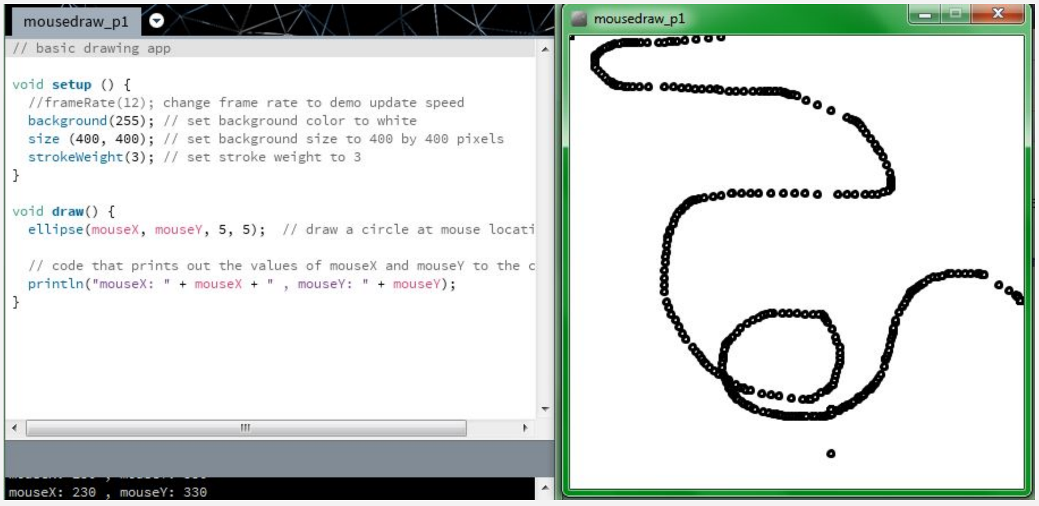
In order to make interactive drawings, we need some way of updating the application while it is running. Dynamic programs in Processing use the draw() function to handle their dynamic code. Let’s take a look at some examples.

Example: Mouse Drawing

Open up “mousedraw_p1.pde” and run the application. When you move your mouse around the application screen, you will see that the application responds by drawing a series of points.

TEACHER TIPS

Teacher TipsAlternatively, as an intructor, you can choose to write the following program line by line with the studenst and explain to the students what you are doing as you write the program. Key things to explain:The CREATION of a function requires the specific syntax used here: the keyword void, parentheses, and the curly brackets {}.void: The return type of the function. More later!parentheses (): This is where you define the parameters that a function takes! Since setup and draw take no functions, they are empty.The {} define the start and end of the function.



```
// basic drawing app

void setup () {
  //frameRate(12); change frame rate to demo update speed
  background(255); // set background color to white
  size (400, 400); // set background size to 400 by 400 pixels
  strokeWeight(3); // set stroke weight to 3
}

void draw() {
  ellipse(mouseX, mouseY, 5, 5);  // draw a circle at mouse locati

  // code that prints out the values of mouseX and mouseY to the c
  println("mouseX: " + mouseX + " , mouseY: " + mouseY);
}
```

```
mouseX: 230 , mouseY: 330
```

Let’s walk through the code to figure out what this program is doing.

2. Writing Functions

You will see that this program is divided into two main sections (functions): setup() and draw(). Don’t worry about the word ‘void’ right now, we will cover it in a later lesson this week. This is how functions are written in processing. the function is given a name, the parameters the functions takes go in the parentheses, and then the code that the function excutes goes between the curly brackets. Syntax:

```
void funcName(int param1, int param2){

    //code here

}
```

The setup() function is used to setup static elements, things that don’t change while the program is running. In this example, we set the background color, size of the application screen and stroke weight (could also be placed in draw).

```
background(255); //white background

size (400, 400); // set screen size
```

The draw() function contains the parts of the program that change. Notice that the distance between points on the screen increases as you move your mouse quickly. While it may look like the mouse position is constantly being updated, is it actually being updated at a given amount of frames per second (FPS). The default frame rate for Processing is 10 FPS, but can also be changed manually.

TEACHER TIPS

Teacher TipHere’s the key things to remember:Code inside setup() is only run once (at the start of application).Code inside draw() is updated continuously as the program runs, once every FRAME (at a default of 10 frames per second)

Let’s look at the lines of code in the draw function, responsible for drawing an ellipse at each mouse location. Recall that the 3rd and 4th arguments represent the width and height of our ellipse.

```
ellipse (mouseX, mouseY, 5, 5);
```

The position of the ellipse on our screen is determined by the X and Y position of our mouse, presented by mouseX and mouseY in Processing.

These two variables track the position of your mouse cursor as it moves across the application screen.

3. Variables, Greater Detail

Variables represent a value stored in memory. Variables work like variables in algebra. Usually in programming, however, it is best to name variables on what they represent, so the code is easier to decipher later. Variables point to a slot in the memory of the computer where the value is stored.

The function println (print line) underneath the point function prints a message to the console:

```
println("mouseX: " + mouseX + " , mouseY: " + mouseY);
```

If you look in the console area (the black area underneath your coding area) while the program runs, you will see that the console provides a constant update of your mouse position.

Lets get some more practice with variables. Go back and reopen your O-face program. What if we want to change the size of the eyes of the face? To do that we have to change the horizontal and vertical diameters of the ellipses that make up the eyes. That's FOUR parameters we have to change, that all have the same value. Let's make it a little more efficient with a variable. Declare a variable at the top the code:

```
int eyeSize = 20;
```

What did we do here? the int declaration tells processing that the variable I am about to name is an integer. eyeSize is the name of my variable. It's an appropriate name since that is what I will be using it for, the eye size of the face.

If you want to use a variable in your program, you have to declare it before you can use it. In the code above, not only do we declare the variable, but we also instantiate it, i.e. we give it a value [of 20].

Why didn't we have to declare and instantiate the variables mouseX and mouseY? Every compiler and language reserves some words or variables because they represent a value or operator that is commonly used by programmers, so the language reserves them against other use. mouseX and mouseY are reserved by processing, and they always contain the coordinates of the mouse (relative to the program window). You'll notice how processing changes the color of those words when you type them in. This tells you it is a reserved word.

TEACHER TIPS

Teacher TipWhere can you find a list of all of the reserved words in processing? Why, processing.org of course!

What is the value of a variable that has not been instantiated? We call a variable that doesn't have a value "null".

4. Conditionals

What if we wanted to draw only when we click the mouse? We'll use something called a conditional statement. It often helps to write out our code ideas in English before trying to think of how to program it in to an application.

"If the mouse is pressed, draw a point at the mouse's current position."

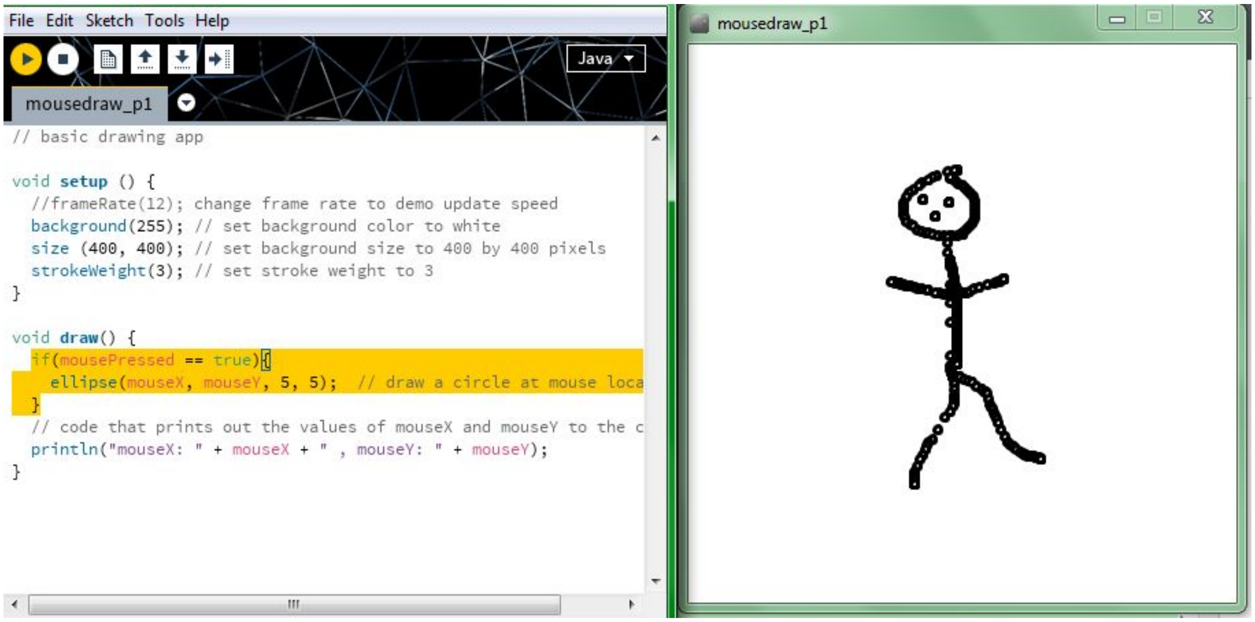
The code for this statement in Processing looks like this:

```
if(mousePressed == true)
```

Add this line in the draw function right above the ellipse function. Here's what the updated draw function should look like:

```
void draw() {

if(mousePressed == true)

ellipse(mouseX, mouseY, 5, 5);

}
```

Run your application and try moving your mouse around while clicking. If done correctly, ellipses will only been drawn when the mouse is clicked.



```
// basic drawing app

void setup () {
  //frameRate(12); change frame rate to demo update speed
  background(255); // set background color to white
  size (400, 400); // set background size to 400 by 400 pixels
  strokeWeight(3); // set stroke weight to 3
}

void draw() {
  if(mousePressed == true){
    ellipse(mouseX, mouseY, 5, 5);  // draw a circle at mouse loca
  }
  // code that prints out the values of mouseX and mouseY to the c
  println("mouseX: " + mouseX + " , mouseY: " + mouseY);
}
```

What if we want to draw lines, similar to what you would expect from a drawing application like Photoshop or Paint? We want to have a continuous line between our previous mouse position and our current mouse position. Luckily, Processing provides two additional mouse tracking variables for us to use: pmouseX and pmouseY, representing the position of the mouse in the previous frame.

Replace ellipse(mouseX, mouseY, 5, 5) with the following:

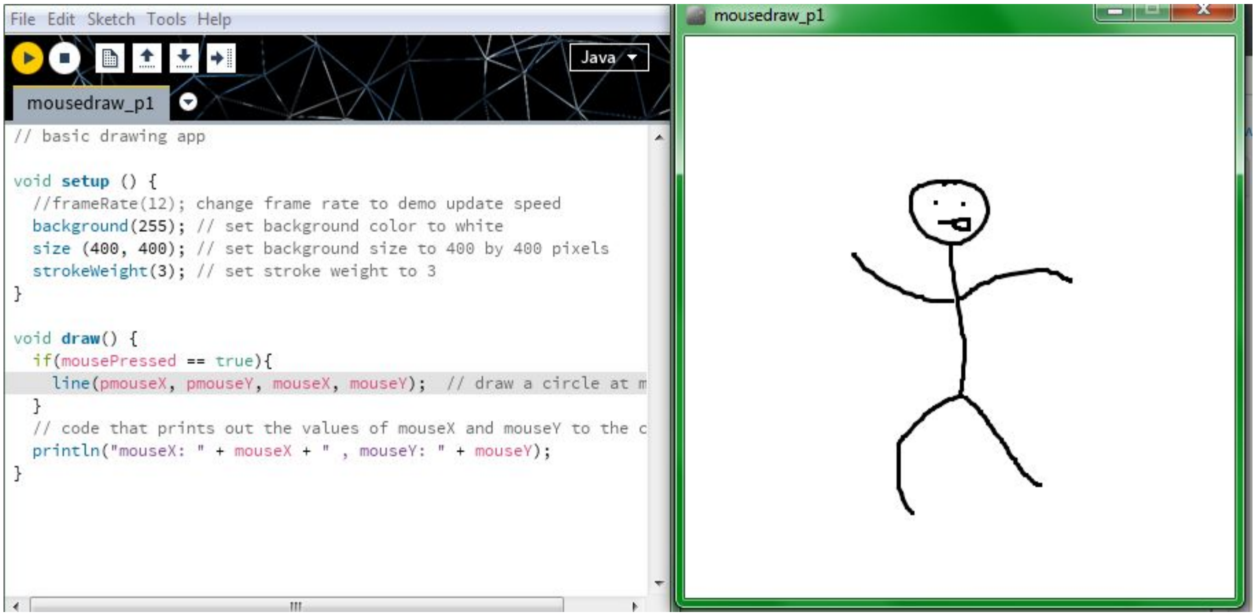
```
line (pmouseX, pmouseY, mouseX, mouseY);
```

Run the application and click to draw continuous lines based on your mouse position.

Here’s what the new draw function looks like:

```
void draw() {

   if(mousePressed == true)

         line(pmouseX, pmouseY, mouseX, mouseY);

}
```

Try changing the background color, stroke/line color and the strokeWeight (these functions can be found in the setup() function).



```
// basic drawing app

void setup () {
  //frameRate(12); change frame rate to demo update speed
  background(255); // set background color to white
  size (400, 400); // set background size to 400 by 400 pixels
  strokeWeight(3); // set stroke weight to 3
}

void draw() {
  if(mousePressed == true){
    line(pmouseX, pmouseY, mouseX, mouseY);  // draw a circle at m
  }
  // code that prints out the values of mouseX and mouseY to the c
  println("mouseX: " + mouseX + " , mouseY: " + mouseY);
}
```

Want to make it look like you’re drawing on a chalkboard? Try this out:

```
background (0);

stroke(255);
```

5. Keyboard interaction

Now that we gone over mouse movement and clicks, you are probably wondering how to track your keyboard strokes.

Take a look at “kp_1.pde”. Run the application and try pressing any key on your keyboard. While the key is pressed, you will see the message “Key Pressed!” appear on the screen. It disappears when you release the key.

Here’s the draw function:

```
void draw(){

background(168, 234, 233); //why is this in draw()?

if(keyPressed == true) {

fill(13, 185, 14);

textSize(20);

text(“Key Pressed!”, 40, 100);

}

}
```

Recall that the draw() function updates our application continuously.   If we placed background in the setup() function, results of the key press (in this case the words “Key Pressed!”) would remain on the screen after the first key press.

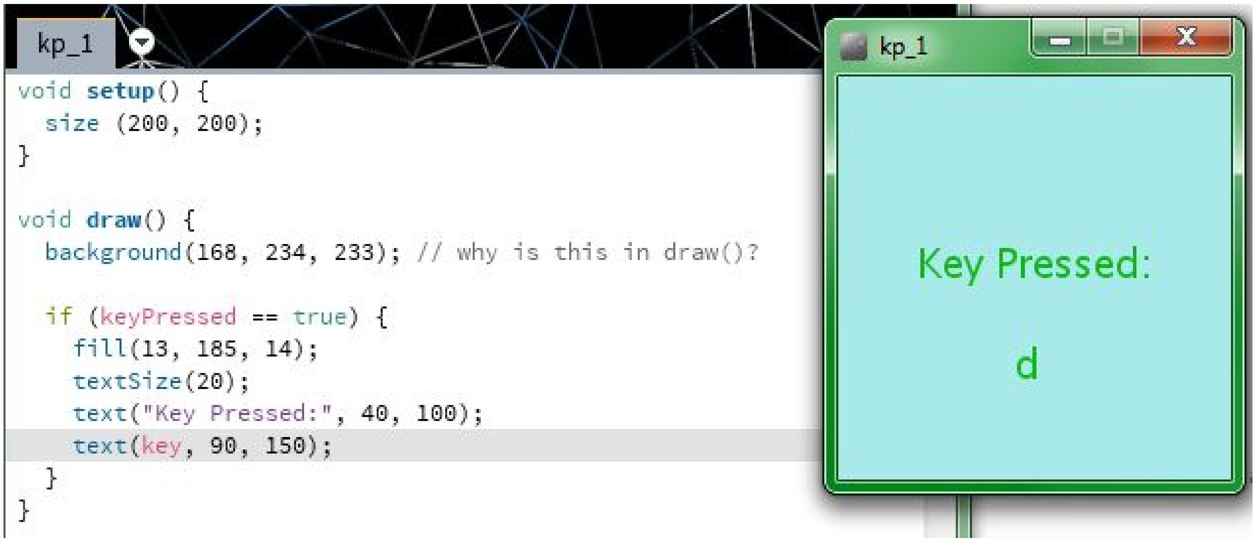
Responding to Particular Keypress Values

Let’s edit the code to display which key is pressed to the screen. We will create another text notification below our existing “Key Pressed!” text. Feel free to change the existing text to something more appropriate such as “You pressed: “.

Start by adding this line:

```
text(key, 90, 150); //display the key value at position (90, 150)
```

Run the application and check out what happens. The key that is pressed is displayed on the screen, thanks to the use of the key built-in variable that tracks which key is being pressed. This variable recognizes most symbols on the keyboard, but not any of the arrow keys or special keys like shift, command and option.



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

void draw() {
  background(168, 234, 233); // why is this in draw()?

  if (keyPressed == true) {
    fill(13, 185, 14);
    textSize(20);
    text("Key Pressed:", 40, 100);
    text(key, 90, 150);
  }
}
```

You can change the size of the text being displayed with the textSize function. Simple use a number representing the text size (in pixels) as a parameter:

```
textSize(40);  // changes the text size to 40 pixels
```

How about adding something new to our message when a certain key is pressed? Let’s use some if statements to check which key is being pressed. We can put these conditionals inside of our if(keyPressed == true) statement. This is an example of a nested conditional: one if statement inside of another! This is very common in programming, you can have as many nested if statements as you like (provided that there are no errors in your code!).

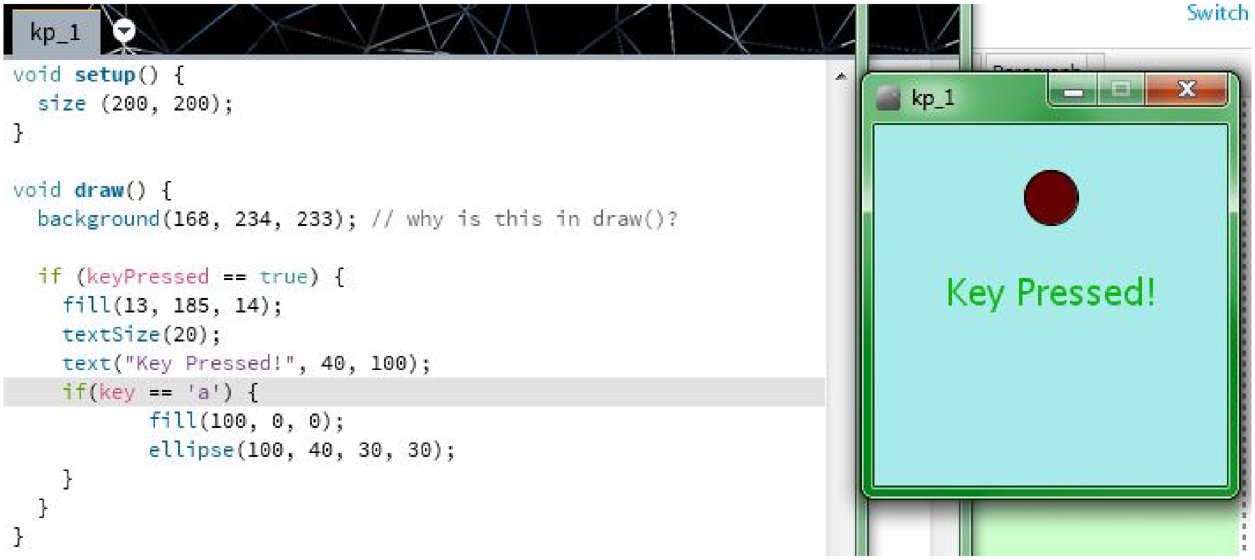
Place this code underneath the last few lines.

What if we wanted to detect a particular keypress, say the letter ‘a’? An if statement, of course!

Add the following code to your draw function and see what happens:

```
   if(key == ‘a’) {

         fill(100, 0, 0);

         ellipse(100, 40, 30, 30);

}
```

When you press the ‘a’ key, a dark red circle should appear above the text. Try writing some similar statements to make different shapes appear when you press other keys.



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

void draw() {
  background(168, 234, 233); // why is this in draw()?

  if (keyPressed == true) {
    fill(13, 185, 14);
    textSize(20);
    text("Key Pressed!", 40, 100);
    if(key == 'a') {
           fill(100, 0, 0);
           ellipse(100, 40, 30, 30);
    }
  }
}
```

```
Key Pressed!
```

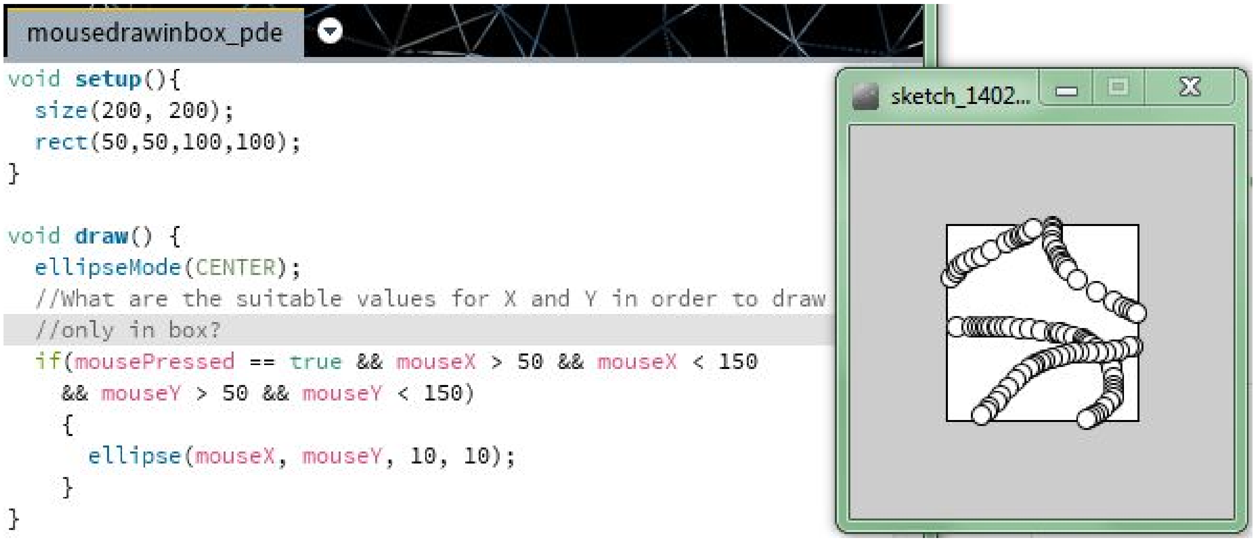
What if we wanted two keypress values to result in the same effect? We can use the OR operator (||) to indicate that either one condition OR the other needs to be true in order to execute code in the if statement.

```
     if(key == ‘a’ || key == ‘b’) {

//CODE!

}
```

This if statement will execute if key ‘a’ or key ‘b’ is pressed. Try adding some more conditional statement of your own to respond to other keyboard character values.

There is also an AND operator (&&). For and AND to be true, BOTH tests on either side of the && must be true. If only one or none are true, then the test will evaluate to false.

Can there be more than 2 tests? Can we only have something happen if 3 things are true? 4? 5? 100? You bet. Things get really fun when we start mixing ANDs and ORs.



```
void setup(){
  size(200, 200);
  rect(50,50,100,100);
}

void draw() {
  ellipseMode(CENTER);
  //What are the suitable values for X and Y in order to draw
  //only in box?
  if(mousePressed == true && mouseX > 50 && mouseX < 150
    && mouseY > 50 && mouseY < 150)
    {
      ellipse(mouseX, mouseY, 10, 10);
    }
}
```

Function A named group of code which will execute whenever it's name is called.

Loop A code structure used to execute a set of code a multiple number of times.

STEPS IN THIS LESSON

- Create a Function for your Face
- Call the Function
- Draw face based on random locations
- Define Parameters for your function
- Use a For Loop to draw concentric circles
- Use a For Loop to draw many faces

Creating a new function

What is a function? You should already know this: It's a set of instructions that gets called whenever we invoke it's name. Whenever we call 'ellipse' or 'rect' or 'size' we are calling a function. It's like a command, except that command is really telling the computer to 'go do this list of other commands and then come back'. When we call a function, you will notice that we always have parentheses () after the name. Every function call in both processing and Java is followed by these parentheses.

The parentheses are where a programmer provides parameters for a function. For example, when you call the function 'size', you can't tell the computer to re-size the window without telling what actual dimensions you want. So we pass in 2 numbers, one for the width, and another for the height.

This is all review, what we want to do now is be able to CREATE a function. Technically, we've been creating functions already. Whenever we write void setup() and void draw(), we are declaring that everything that follows the open bracket { is a part of that function.

Those functions, however, are special to processing. When we write them we are telling processing to START at setup, and to do draw after and repeatedly.

TEACHER TIPS

Teacher TipIn this curriculum, the term 'function' is used, as opposed to method. Technically, for Java, they are referred to as methods. Since the Processing language and IDE are technically Java, they are also technically called methods. However, most of the languages that student will go on to and in both DMA and beyond refers to them as functions. It is important to point out both names to students.

Open up your face program or the one from the resources folder. Also open a new program. Create a setup and draw function. What we are going to do is create a program that will draw a face whenever we call the function.

Create the function declaration after the draw function, using the same syntax. Call it face. Place inside it the code to draw the face:

```
void setup(){    size(500,500);}

void draw(){

}

void face(){
ellipse(50,50,100,100);    ellipse(30,30,15,15);    ellipse(70,30,15,15);    ellipse(50,70,40,15);}
```

When you run the program, nothing shows up. This is because even though we have written the function, it does not execute unless you call it. Remember processing does setup, then draw, then repeats draw. There is no default to jump down and do face(). So we call it by writing face() in the draw function:

```
void draw(){

  face();

}
```

Now the face shows up. When the draw function gets to the face command, it jumps down to that function that we created, and then jumps to the face function, runs it, and then back to draw.

Try replacing the x and y coordinates of the ellipses with variables, and lets use the random() function to give them random values. Change face() to:

```
void face(){

   float x = random(0,500); //gives x a random value from 0 to 500

   float y = random(0,500); //gives y a random value from 0 to 500

   ellipse(x,y,100,100);

   ellipse(x-20,y-20,15,15);

   ellipse(x+20,y-20,15,15);
```

```
    ellipse(x,y+20,40,15);

}
```

Let's talk about what this does. If every ellipse in the face is drawn relative to values of x an y, then the face will be centered at x and y, no matter what x and y is. This means that we get lots of faces in lots of places. Every time draw run, face is called, and then a new random value for x and y is chosen, and then a face is drawn.

Let's change the face function to work similar to way another shape call works, with parameters, so if we called:

```
face(200,200);
```

It would draw a face at (200,200).

To do that, we have to change the function declaration to define parameters:

```
void face(float x, float y){
```

when we define parameters, we have to decide a few things. First, the order or parameters that get taken, and how many. Second, the name of each parameter. Lastly, the TYPE of each parameter. We will go more over types later, but for now, we will use float, since x and y are numbers, and float is a number data type.

This allows us to call face in the manner that we want, to put 2 numbers in the parameters of the call and have those numbers go to the function.

The number we put in the parameters will line up with the parameters we defined. The first number will go into a variable called x, and the second into y. This means the variables we created in the function we no longer need. Now it can be:

```
void face(float x, float y){

    ellipse(x,y,100,100);

    ellipse(x-20,y-20,15,15);

    ellipse(x+20,y-20,15,15);

    ellipse(x,y+20,40,15);
}
```

Now you can call face from draw, and have it draw wherever you want, without having to change the code in face. Try different numbers, random numbers, or the mouse variables. In the next exercise, we will make a lot of faces with a loop!

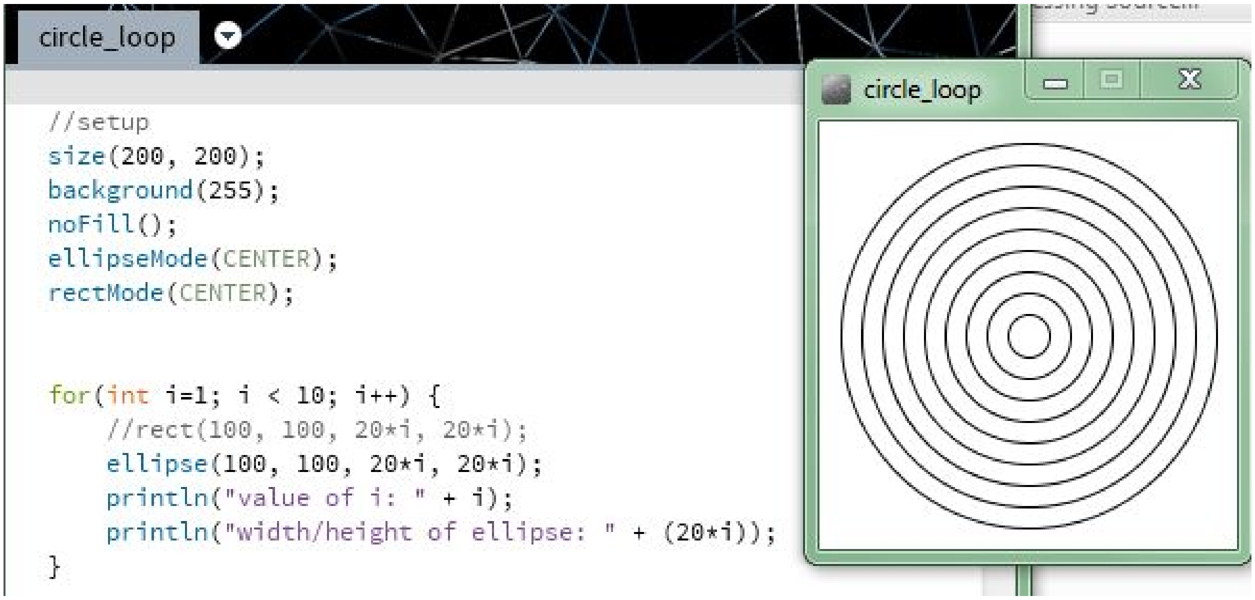
Learn to love loops!

In this exercise, you will be introduced to a super useful and extremely common concept in programming: repetition through looping. We will use a for loop, a very common function in almost every programming language. The syntax we use here is exactly the same as it is in Java.

How do we define a loop in Processing? It looks like this:

```
for ( initial condition; test condition; update) {

    // do something

}
```

Open up “circle_loop.pde” and run the application.



```
//setup
size(200, 200);
background(255);
noFill();
ellipseMode(CENTER);
rectMode(CENTER);


for(int i=1; i < 10; i++) {
    //rect(100, 100, 20*i, 20*i);
    ellipse(100, 100, 20*i, 20*i);
    println("value of i: " + i);
    println("width/height of ellipse: " + (20*i));
}
```

Take a look at the important part of the loop:

```
for(int i=1; i < 10; i++) {

        ellipse(100, 100, 20*i, 20*i);

 }
```

We have declared an integer (a positive, whole number) within the loop to use as a counter. We have given it the name ‘i’, which is a common programming convention for loops.

TEACHER TIPS

Teacher TipNote that when for the initial condition of a for loop is written, you can either use a variable that is already created, or create a new variable using the int, float, String, etc declaration.

Try playing with the code to get more circles but closer together. Experiment to see what happens when you shift the center of the ellipse with the iterations of loop, or only change one parameter, or use different shapes.

Now, use a loop to draw a lot of faces! In setup, add

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

  face(50+50*i, 250);

}
```

If you’re feeling very fancy, or have several advanced students, you can go over nested loops to create a grid, or you can wait until a later lesson.

The best thing to do here is to ask advanced students to create additional functions with extra parameters. Like, create a face function which also takes in a parameter for eye size:

```
void face(float x, float y, float eyeSize){
```

Or even colors, or type of mouth, etc.

Or functions to draw other shapes at locations, like house(), or car().

You could also asks students to create functions that do more complex things like solve a quadratic based on the values of a, b, and c:

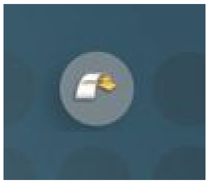
```
void quad(float a, float b, float c){

  //Finds and prints the two values of x, if they exist

}
```

Welcome to Eclipse

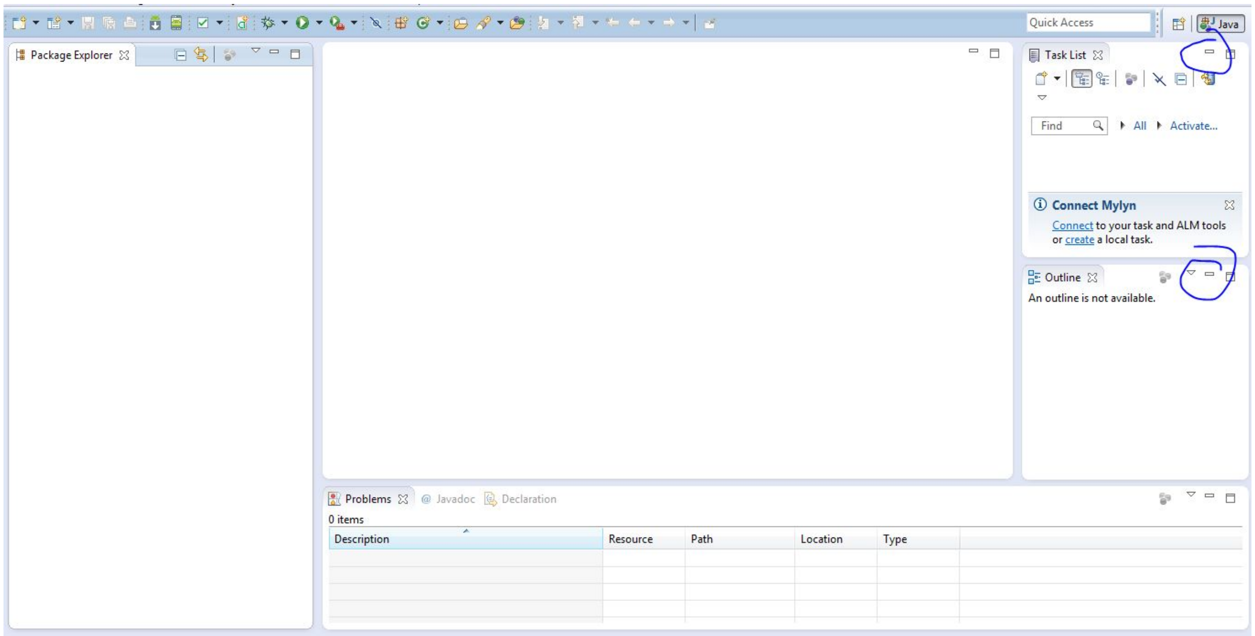
At this point we probably won't be using processing any more. Ask the students to save and close any sketches they have that they want to keep and shut all Processing windows. Now, have them open Eclipse.

When the workspace selection prompt comes up, have them create a new workspace. Click browse, ask them to create a new folder in their desktop folder called "Workspace", and then click ok. Tell them they need to make sure this is the workspace they are in whenever Eclipse opens. This prevents students from getting an old workspace from a previous week of DMA.

After Eclipse finishes loading we get the wonderful startup screen, or it goes straight to the workbench. If you get the startup screen you need to click the arrow icon to go to the workbench

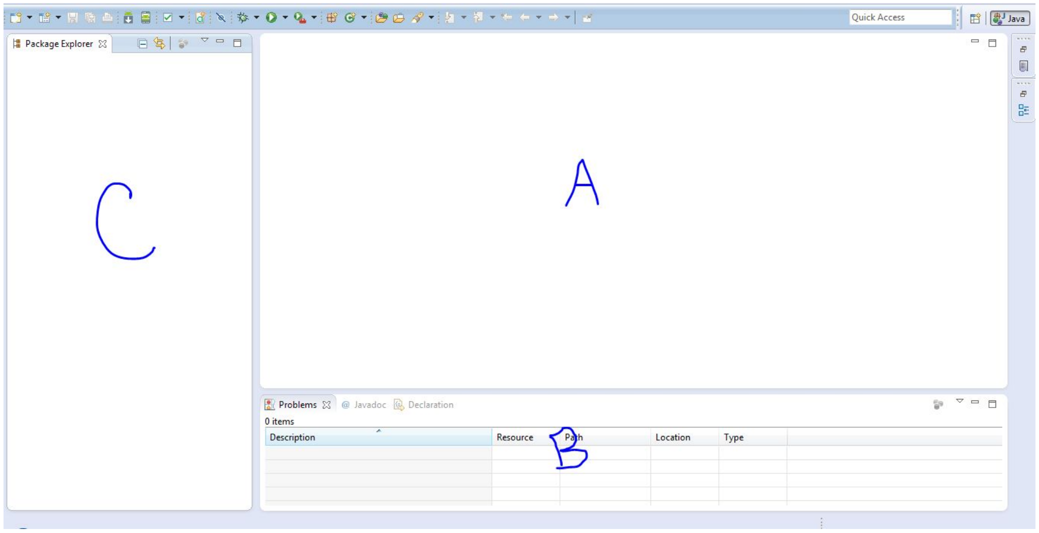


Then you get the workbench.



Have the students minimize or close the task list and the outline; they won't be necessary for this course.

Now let's talk about the 3 remaining sections of the workbench:



Section A: This is where you will be able to view and edit individual Java code files, as well as other things.

Section B: This box will report errors of your code and any text output.

Section C: This is a file explorer system, which will allow you navigate the various files and projects you are working on.

Start a new project

The 3 ways to start a project:

1) File > New > Java Project

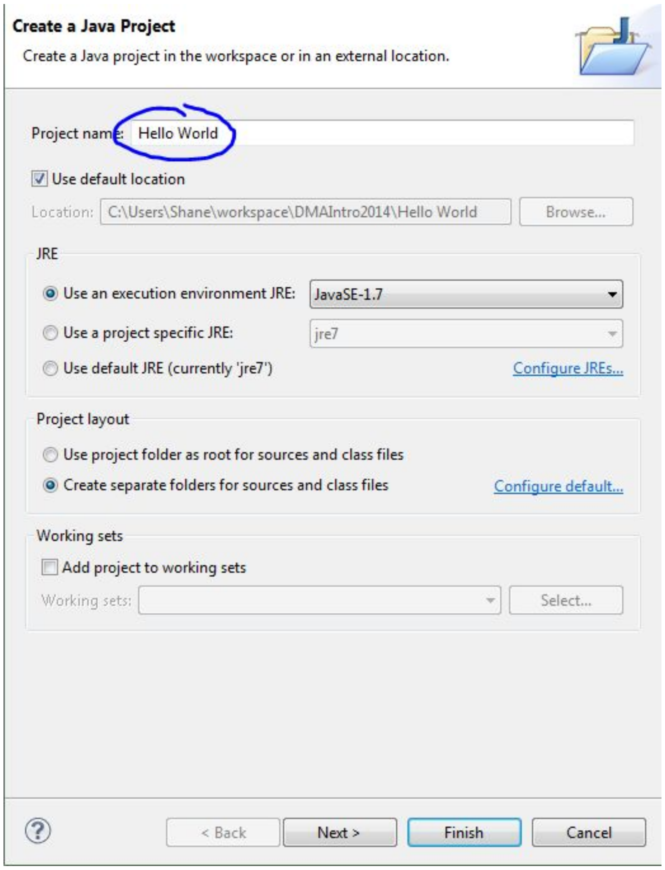
or

2) Right-click Package Explorer > New > Java Project

or

3) Click the down arrow on the ▾ button > Java Project

Then you get:



Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: Hello World

☑ Use default location

Location: C:\Users\Shane\workspace\DMAIntro2014\Hello World

JRE

● Use an execution environment JRE: JavaSE-1.7

○ Use a project specific JRE: jre7

○ Use default JRE (currently 'jre7')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

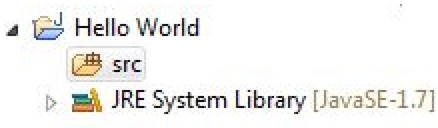
Working sets

□ Add project to working sets

Working sets:

Name the project "Hello World". Notice it creates a new folder in your workspace if you leave the location as the default. You can leave all the other settings the same. Click Finish.

In the package explorer there is a new folder icon named Hello World. The little J on the folder tells you it is a Java Project. Expand it to see 2 sub directories, a "src" folder, and JRE System Library.



Hello World
src
JRE System Library [JavaSE-1.7]

Let the students know that the src folder is where the code ends up when navigating projects in the package explorer. For bigger projects, code might end up in a bunch of locations, but for this week we'll primarily only need a src folder. We will talk more about the library later.

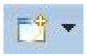
Okay, now its time to actually get some code in our project! In Java and the Eclipse IDE, all projects break up the code into single files which represent Classes. So make a new class:

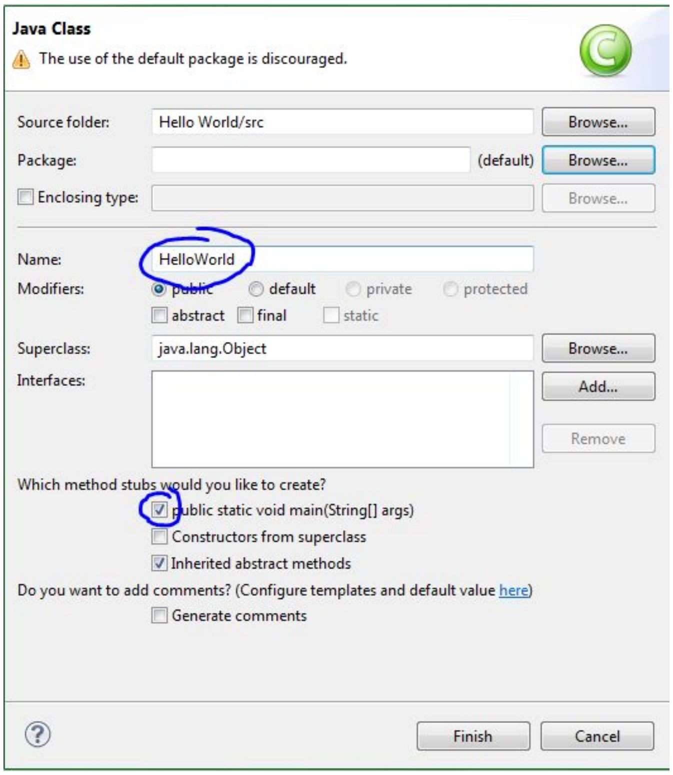
1) File > New > Class

or

2) Right Click package explorer > New > Class

or

3)  > Class



Java Class

⚠ The use of the default package is discouraged.

Source folder: Hello World/src

Package: (default)

□ Enclosing type:

Name: HelloWorld

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Name the class HelloWorld and choose to **include public static void main.** Click **Finish.**

TEACHER TIPS

Teacher TipsTell the students that naming conventions of classes is important. When we name classes, we capitalize every first letter of each word (spaces are not allowed). When we name variables and functions, we capitalize every first letter of each word EXCEPT for the first word. Like "eyeSize". This makes it easier for other programmers to quickly recognize the difference between class, function, and variable names.

After the class is created, there will be an empty class with the main() function. Delete the function header and the TODO stub, if they show up. At this point, they are just distracting.

main() is the function where the program starts, like setup() in processing. Add this to the function:

System.out.println("Hello, World!");

Click run (runbutton.PNG) and make sure to choose "Java Application", NOT APPLET. The output will appear in the Console window.

Bam, first Java program success.

Class - A Java file is usually a class - which contains everything needed to represent a thing, or to run a program. In this lesson, we are just creating a class that runs a program, which is called a Client Class

Library - A collection of classes, functions, and other code that works together that we can add to our project so we can use them. Here, we are using Processing as a library.

STEPS IN THIS LESSON

- **Creating a Processing Project**
- **The differences between Java and Processing**
- **Creating a Function**

Creating a Processing Project

So, remember when we created the hello world project, and then we made the new Class, and it had that funky line called**public static void main(String[] args)**?

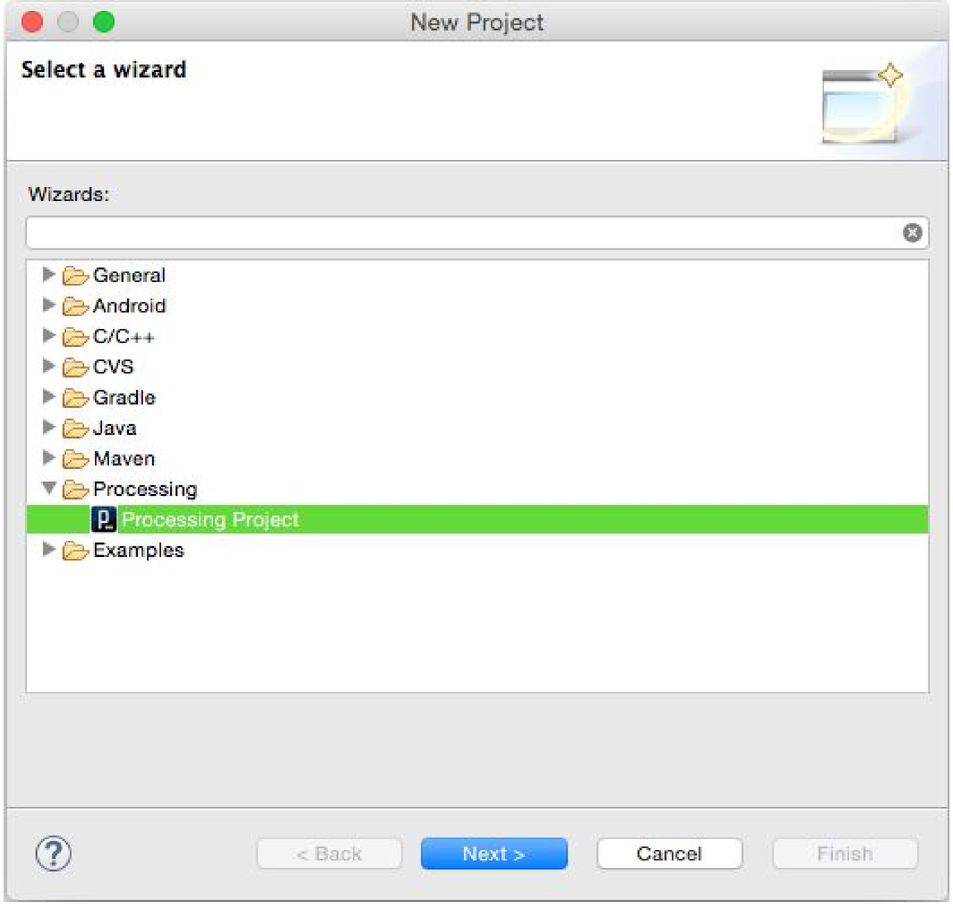
Well that line was a function declaration, and in Java, the function **main** is just like the setup function in processing. It's where the program starts. Instead of just the simple void keyword (which we still don't know what that means!) There is a bunch of other confusing things too. I agree. So, we are going to make it easier by making a new project that is going to use processing as a **Library** to make everything a little easier.

What is a library? A library is a bunch of code that is already written! It can contain things like variables and functions, and once we tell java to use a certain library, we can then call those functions and use those variables at (mostly) our discretion. There are tons of libraries and we could spend years learning how to use all of them, and more are made everyday, but for now we are just going to use processing, and later a couple extra useful ones.

Anyway, normally including processing (or any other library that's not a standard Java library) in your project can be tricky and weird if you're new to programming, but luckily we have a way to cheat a little bit that will take care of most of the work for us.

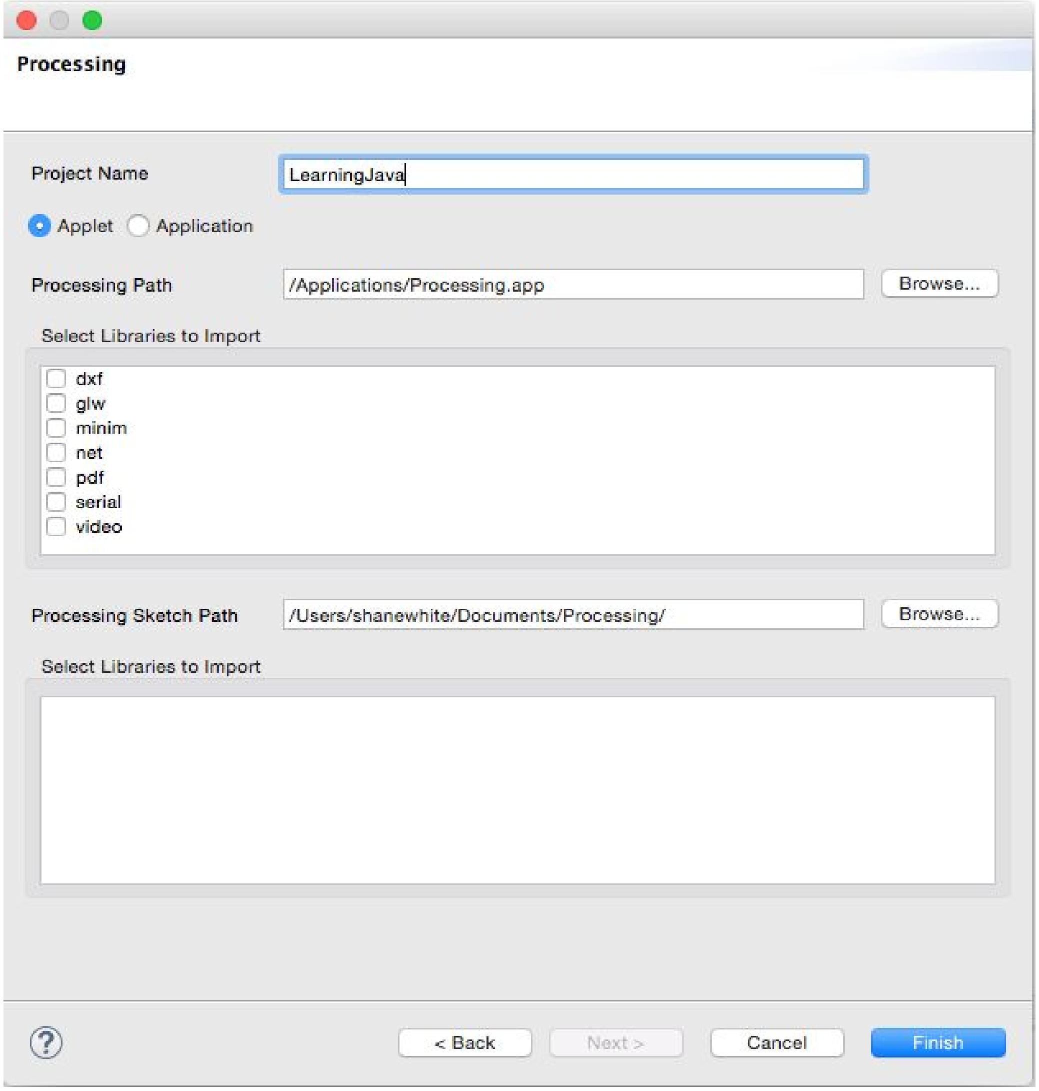
Go to File > New > Project   (**NOT** Java Project)

On the wizard window, choose the Processing folder and Processing Project



Click Next.

Name the Project "LearningJava", and make sure Applet is chosen on the next radio button. Make sure that the Processing Path is set to the application folder (like below). The sketch path doesn't matter. Don't choose any of the Libraries to import.



Processing

Project Name: LearningJava

● Applet ○ Application

Processing Path: /Applications/Processing.app

Select Libraries to Import

- □ dxf
- □ glw
- □ minim
- □ net
- □ pdf
- □ serial
- □ video

Processing Sketch Path: /Users/shanewhite/Documents/Processing/

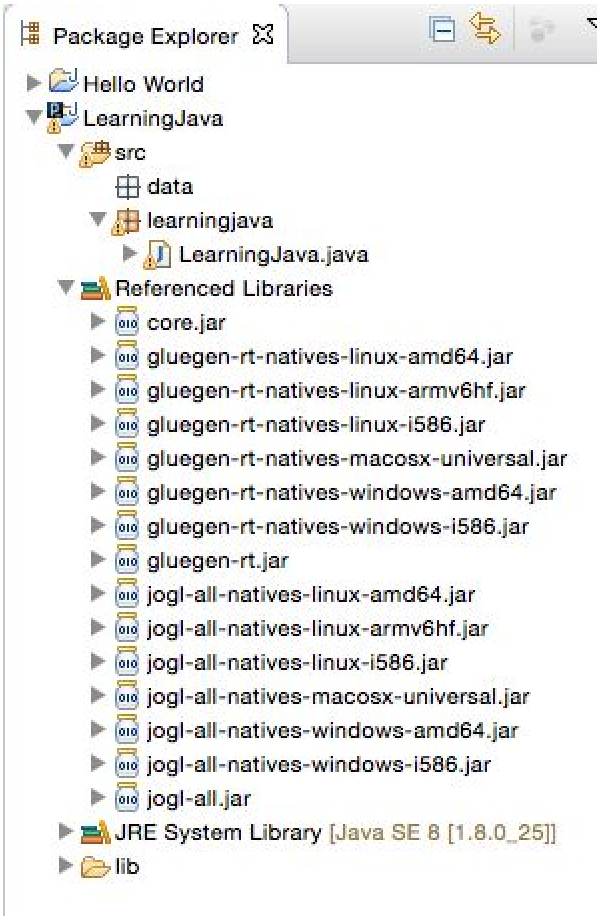
Select Libraries to Import

Click Finish!

Ok this will do several things for us.

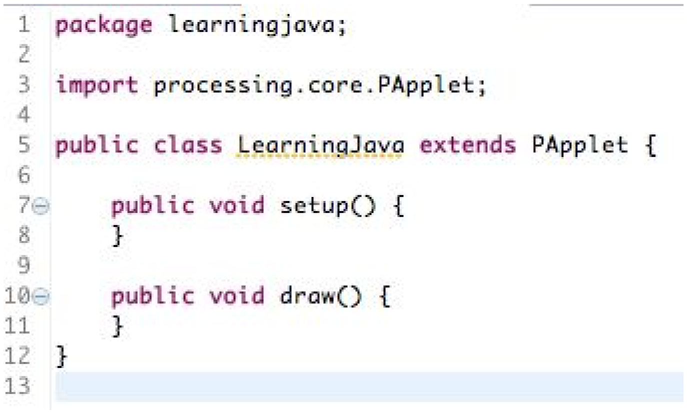
1. It will create a new project for us with the name we chose.
2. It will include the processing core library, as well as a bunch of other libraries that honestly we probably don't need but are there to support processing running on other operating systems.
3. It will create a new class file for us, called "LearningJava.java".

Expand the project in the explorer and point out all those things. Don't go into depth about the libraries or what a .jar file means as opposed to a .java file. We will talk about that at the end of the class when students export their projects. Go ahead and collaps the HelloWorld project.



- Hello World
- LearningJava
  - src
    - data
    - learningjava
      - LearningJava.java
  - Referenced Libraries
    - core.jar
    - gluegen-rt-natives-linux-amd64.jar
    - gluegen-rt-natives-linux-armv6hf.jar
    - gluegen-rt-natives-linux-i586.jar
    - gluegen-rt-natives-macosx-universal.jar
    - gluegen-rt-natives-windows-amd64.jar
    - gluegen-rt-natives-windows-i586.jar
    - gluegen-rt.jar
    - jogl-all-natives-linux-amd64.jar
    - jogl-all-natives-linux-armv6hf.jar
    - jogl-all-natives-linux-i586.jar
    - jogl-all-natives-macosx-universal.jar
    - jogl-all-natives-windows-amd64.jar
    - jogl-all-natives-windows-i586.jar
    - jogl-all.jar
  - JRE System Library [Java SE 8 [1.8.0_25]]
  - lib

Open up LearningJava.java. What do you see?



```java
package learningjava;

import processing.core.PApplet;

public class LearningJava extends PApplet {

    public void setup() {
    }

    public void draw() {
    }
}

```

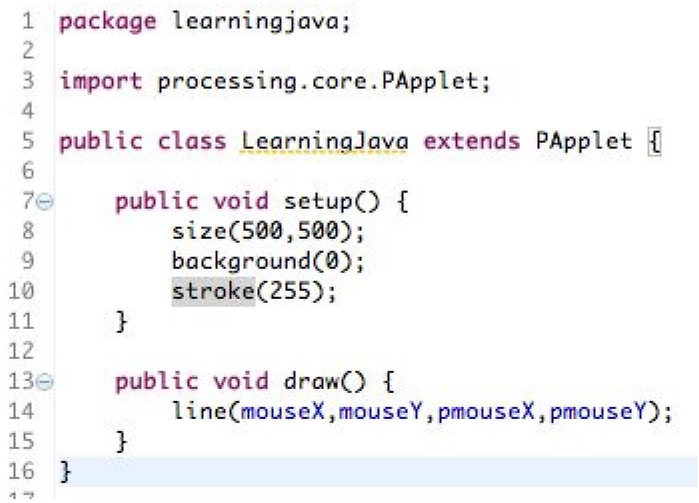
There are 3 lines of code we haven’t gone over, but there is also some things we are familiar with: a setup function and a draw function!

Let’s try them out by filling in some things we already know:

Challenge (Work with a partner[b]): In setup, set the size to 500 by 500, and then create a drawing program with a black background and white lines. Once you think you have it, have a TA check your work. DON'T RUN IT YET.

You may notice some things get underlined with red or yellow. This is one of the advantages of using eclipse rather than the simple little eclipse window. These are errors (red) or warnings (yellow) that eclipse is telling you about. For now, we can pretty much ignore the yellow ones. If there is a red one, that means you made a syntax error and the program will not run. Hover your mouse over the offending code and eclipse will try to give you more information. If you can't decipher it, ask a TA.

Here is an example solution (from now on screen shots will only include important code instead of the whole file):



```
package learningjava;

import processing.core.PApplet;

public class LearningJava extends PApplet {

    public void setup() {
        size(500,500);
        background(0);
        stroke(255);
    }

    public void draw() {
        line(mouseX,mouseY,pmouseX,pmouseY);
    }
}
```

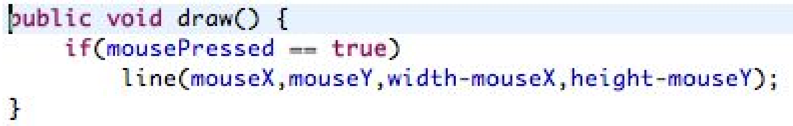
Be sure to use stroke(255) if you are drawing with a line. fill() will not affect the color of lines, only the interior of shapes.

Ok, now when you have no errors, did you go over it to think it will work? Did you trace through the code by hand? Do you know what each line will do? If you imagine the program working line by line, does it do what you want it to?[c]

When it is complete and it has been looked at by a TA or instructor, you can run it. Click the green play button, but this time, choose **Java Applet**, if it asks. *It seems to run it as an applet automagically, so hooray!*

Let’s try out some things to make this program more interesting, and make sure everything we’ve learned still works!

Try changing the draw function to this:



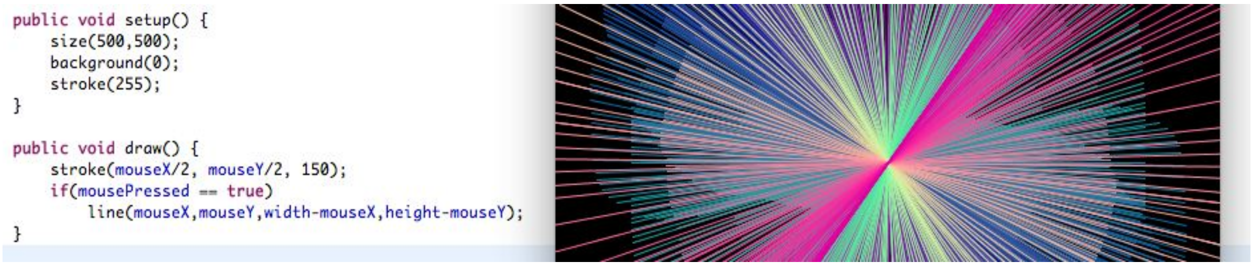
```
public void draw() {
     if(mousePressed == true)
         line(mouseX,mouseY,width-mouseX,height-mouseY);
}
```

Cool, **if** statements still work, as well some special variables we didn’t talk about before, **width** and **height**. Can you guess what those contain? What does putting **width-mouseX** in the third parameter accomplish?

What happens when you place some of these variables in the parameters of the **fill** or **background** function?

Hint: A good idea is using mouseX/2 as a parameter for color function, like fill. This only works if your window is 500x500, but doing that will scale the x value to be between 0 and 250, which is very close to the 0-255 that a color parameter should take.

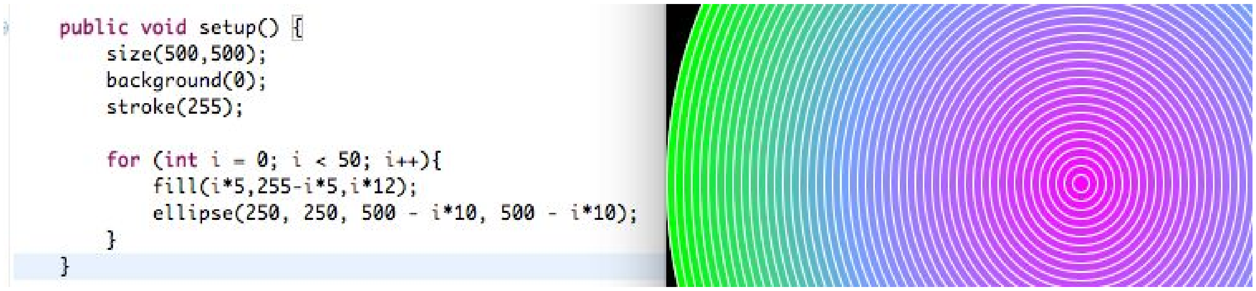
Here’s good example:



```
public void setup() {
    size(500,500);
    background(0);
    stroke(255);
}

public void draw() {
    stroke(mouseX/2, mouseY/2, 150);
    if(mousePressed == true)
        line(mouseX,mouseY,width-mouseX,height-mouseY);
}
```

Let’s see if loops still work. Write a for loop to draw several circles on the screen. As an extra challenge, make each one a different color. Remember, if you put your loop in draw, it will get run over and over, so maybe it’s a good idea to put it in setup, so it only get run once.

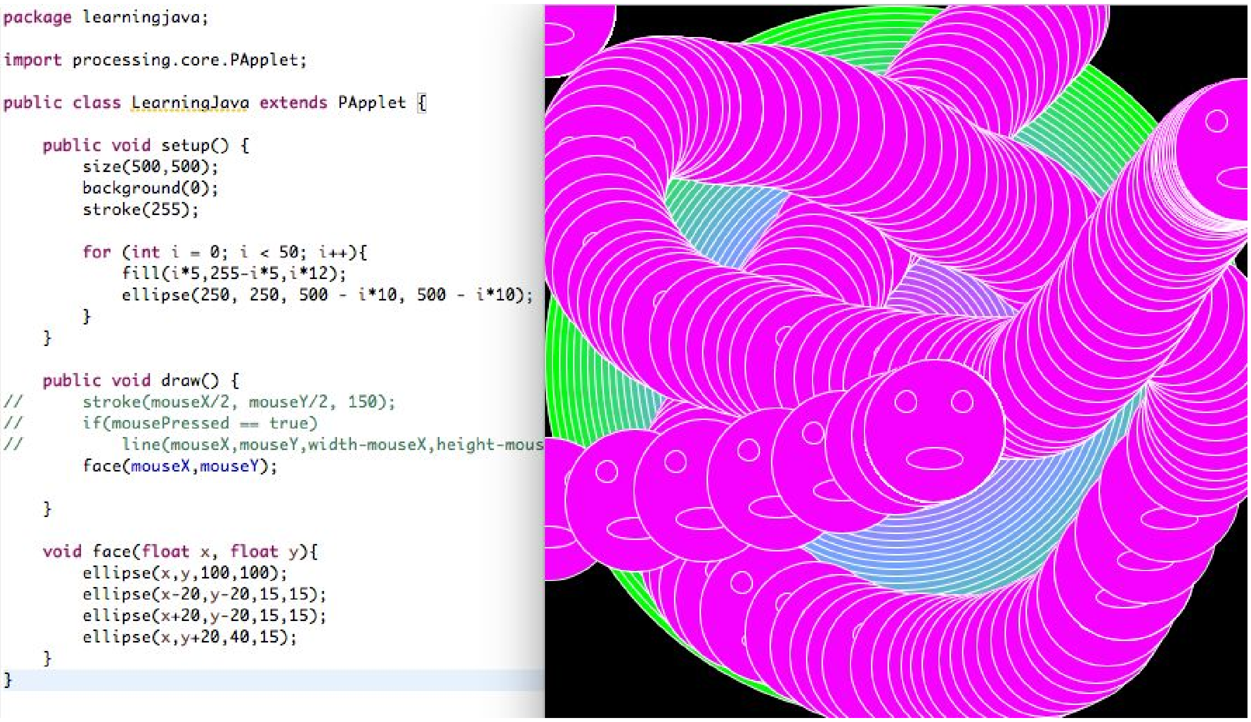
Example solution:



```
    public void setup() {
        size(500,500);
        background(0);
        stroke(255);

        for (int i = 0; i < 50; i++){
            fill(i*5,255-i*5,i*12);
            ellipse(250, 250, 500 - i*10, 500 - i*10);
        }
    }
```

Ok, now remember your face function from processing? Let’s see if that still works. Copy and paste just the function from Processing into your java program, just make sure it’s AFTER draw ends and BEFORE the class ends (the last curly bracket**}**)

Try it out!



```
package learningjava;

import processing.core.PApplet;

public class LearningJava extends PApplet {

    public void setup() {
        size(500,500);
        background(0);
        stroke(255);

        for (int i = 0; i < 50; i++){
            fill(i*5,255-i*5,i*12);
            ellipse(250, 250, 500 - i*10, 500 - i*10);
        }
    }

    public void draw() {
//      stroke(mouseX/2, mouseY/2, 150);
//      if(mousePressed == true)
//          line(mouseX,mouseY,width-mouseX,height-mous
        face(mouseX,mouseY);

    }

    void face(float x, float y){
        ellipse(x,y,100,100);
        ellipse(x-20,y-20,15,15);
        ellipse(x+20,y-20,15,15);
        ellipse(x,y+20,40,15);
    }
}
```

Awesome. Now that that all works, lets move on and learn about the TRUE power of Java, objects![d][e]

STEPS IN THIS LESSON

- **Quick Challenge: draw a rectangle**
- **Add global variables**

- **Update Variables**
- **Compartmentalize the code**

Developer's Note
The following lesson is adapted from the processing website: https://processing.org/tutorials/objects/ Some of it has been adapted to be particular to teaching in a DMA classroom and doing the exercises in eclipse, and to provide a more CAMPY feel, but the credit must go where credit due. So, here is the nice legal declaration so everyone knows where it came from: This tutorial is for Processing version 1.1+. If you see any errors or have comments, please let us know. This tutorial is from the book, Learning Processing, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright © 2008 Elsevier Inc. All rights reserved. It should also fit in nicely with the way the class is going, so you shouldn't have to make any changes to what it says, and just keep teaching the way you're teaching!

The Basics of Animation

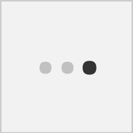
Before we really start working on objects, let's write a new program, with the goal of A) Reviewing what we know, and B) Learning how to properly animate things across the processing screen.

So, we are going to write a program that moves a rectangle across the screen, which we are going to pretend is a car.

SO, create a new Processing Project in eclipse, and call it 'Cars'.

For Review: File > New > Project >> Choose Processing > Processing Project

Quick Challenge: Can you add to the setup function to set the window size to 200x200. Can you modify the draw() function so that a rectangle with a width and height of 30x10 is drawn?



Now, how do we make this rectangle move? Remember that draw() runs over and over. So the best way to make it move, is to change the location it's being drawn every time draw() goes. How do we do that?

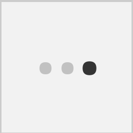
We need variables for the location of the rectangle. Namely, x and y. Declare them at the top of the program, before setup.

```
float x = 100;

float y = 100;
```

And let's use them in our draw function when we draw the rectangle. Let's also make it a black rectangle:

```
fill(0, 0, 0);

rect(x, y, 30, 10);
```

Ok, I'm getting concerned that our draw function is going to get really complicated really fast. Let's get into the habit of putting groups of code into functions. So go ahead and make a new function called display() which will draw the car. Basically it will do what draw does already.



Now every time we want to move the car, we can just change the value of x or y. We should do this every frame, so whatever changes the x should be called in the draw function.

Let’s make a new function called move(), and call it from draw. All move should do is add 1 to the value of x. Go ahead and do this now.

```
void move(){

x = x + 1;

}
```

This will draw a black line, as it will not erase the rectangle every time it draws a new one. Have the students run the program at this point without pointing this out and see if they remember how to do that. Remind the student to call background(255,255,255) as the first call in draw.

Now as another challenge, can you get the car to start back at the left side of the screen after it has gone off the right side? Remember your if statements, and what information you have available to you to test.

Solution:

```
public void move(){
   x = x + 1;
   if(x > width){
      x = -30;  //You CAN set locations to be negative!
   }
}
```

Awesome! Now what if we want to change the color? Or the speed? Well that seems pretty easy, but so that any changes will be even easier to make later on, I should have all that information stored in variables. So, add a color variable and a speed variable.

For the color, just use an int, and we will set the color in grayscale for now. The ‘color’ primitive in processing doesn’t exist in java, so later on we will have a look at the java Color class.

Here is the final Program:

```
float x = 100;
float y = 100;
float speed = 1;
float c = 0;

public void setup() {
        size(200,200);
}

public void draw() {
        background(255,255,255);
        move();
        display();
}

public void display(){
        fill(c);
        rect(x, y, 30, 10);
}

public void move(){
        x = x + speed;
        if(x > width){
                x = -30;
        }
}
```

Class an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior.

Object a particular instance of a class where the object can be a combination of variables, functions, and data structures.

STEPS IN THIS LESSON

- **What is an Object**
- **What is a Class**
- **Modeling an Object**
- **Writing a cookie cutter**
- **Using an Object**

I'm Down with OOP

Before we begin examining the details of how object-oriented programming (OOP) works, let's embark on a short conceptual discussion of "objects" themselves. Imagine you were not programming in Java, but were instead writing out a program for your day, a list of instructions, if you will. It might start out something like:

- Wake up.
- Drink coffee (or tea, or juice).
- Eat breakfast: cereal, blueberries, and soy milk if you’re a health nut, or a donut if you’re a regular person.
- Ride the subway.

What is involved here? Specifically, what things are involved? First, although it may not be immediately apparent from how we wrote the above instructions, the main thing is you, a human being, a person. You exhibit certain properties. You look a certain way; perhaps you have brown hair, wear glasses, and appear slightly nerdy. You also have the ability to do stuff, such as wake up (presumably you can also sleep), eat, or ride the subway. An object is just like you, a thing that has properties and can do stuff.

So how does this relate to programming? The properties of an object are variables; and the things an object can do are functions. Object-oriented programming is the marriage of all of the programming fundamentals: data and functionality.

Let's map out the data and functions for a very simple human object:

Human data

Height.

Weight.

Gender.

Eye color.

Hair color.

Human functions

Sleep.

Wake up.

Eat.

Ride some form of transportation.

Now, before we get too much further, we need to embark on a brief metaphysical digression. The above structure is not a human being itself; it simply describes the idea, or the concept, behind a human being. It describes what it is to be human. To be human is to have height, hair, to sleep, to eat, and so on. This is a crucial distinction for programming objects. This human being template is known as a class. A class is different from an object. You are an object. I am an object. That guy on the subway is an object. Albert Einstein is an object. We are all people, real world instances of the idea of a human being.

Think of a cookie cutter. A cookie cutter makes cookies, but it is not a cookie itself. The cookie cutter is the class, the cookies are the objects.

Using an Object

Before we look at the actual writing of a class itself, let's briefly look at how using objects in our main program (i.e., setup() and draw()) makes the world a better place.

Consider the code for the program we just wrote to make a ‘Car’ move across the screen. It consists of:

Data (Global Variables):

Car color.

Car x location.

Car y location.

Car x speed.

Setup:

Initialize car color.

Initialize car location to starting point.

Initialize car speed.

Draw:

Fill background.

Display car at location with color.

Increment car's location by speed.

To implement the above information, we defined global variables at the top of the program, initialized them in setup(), and called functions to move and display the car in draw().

Object-oriented programming allows us to take all of the variables and functions out of the main program and store them inside a car object. A car object will know about its data - color, location, speed. The object will also know about the stuff it can do, the methods (functions inside an object) - the car can drive and it can be displayed.

Using object-oriented design, the pseudo-code (the design of the code) improves to look something like this:

```
Data (Global Variables):
Car object.
Setup:
Initialize car object.
Draw:
Fill background.
Display car object.
Drive car object.
```

Notice we removed all of the global variables from the first example. Instead of having separate variables for car color, car location, and car speed, we now have only one variable, a Car variable! And instead of initializing those three variables, we initialize one thing, the Car object. Where did those variables go? They still exist, only now they live inside of the Car object (and will be defined in the Car class, which we will get to in a moment).

Moving beyond pseudocode, the actual body of the sketch might look like:

```
Car myCar;
public void setup()  {
  myCar = new Car();
}
public void draw()  {
  background(255);
  myCar.drive();
  myCar.display();
}
```

We are going to get into the details regarding the above code in a moment, but before we do so, let's take a look at how the Car class itself is written.

Class Name: The name is specified by "class WhateverNameYouChoose". We then enclose all of the code for the class inside curly brackets after the name declaration. Class names are traditionally capitalized (to distinguish them from variable names, which traditionally are lowercase).

Data: The data for a class is a collection of variables. These variables are often referred to as instance variables since each instance of an object contains this set of variables.

TEACHER TIPS

Teacher TipsProcessing does not require use public and private by default. Java and eclipse does. This course doesn’t really delve deep into public vs. private, for now it is enough to express to students that instance variables should be made private to protect them, and functions are for the most part public so other classes can call them.On another note, the code in this lesson was originally written in processing to test it, and then moved into eclipse, so some of it may be missing the public / private where it was necessary. Please email shane@dig.ma if you see any mistakes like that.

Constructor: The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give the instructions on how to set up the object. It is just like Processing's setup() function, only here it is used to create an individual object within the sketch, whenever a new object is created from this class. It always has the same name as the class and is called by invoking the new operator: "Car myCar = new Car();".

Functionality: We can add functionality to our object by writing methods.

Note that the code for a class exists as its own block and can be placed anywhere outside of setup() and draw().

```
public void setup() {
}
public void draw() {
}
public class Car {
}
```

Using an Object: The Details

Earlier, we took a quick peek at how an object can greatly simplify the main parts of a Java sketch (i.e. setup() and draw()).

```
// Step 1. Declare an object.
public Car myCar;
public void setup()  {
  // Step 2. Initialize object.
  myCar = new Car();
}
public void draw()  {
  background(255);
  // Step 3. Call methods on the object.
  myCar.drive();
  myCar.display();
}
```

Let's look at the details behind the above three steps outlining how to use an object in your sketch.

Step 1. Declaring an object variable.

A variable is always declared by specifying a type and a name. With a primitive data type, such as an integer, it looks like this:

```
// Variable Declaration

int var;  // type name
```

Primitive data types are singular pieces of information: an integer, a float, a character, etc. Declaring a variable that holds onto an object is quite similar. The difference is that here the type is the class name, something we will make up, in this case "Car." Objects, incidentally, are not primitives and are considered complex data types. (This is because they store multiple pieces of information: data and functionality. Primitives only store data.)

Step 2. Initializing an object.

In order to initialize a variable (i.e., give it a starting value), we use an assignment operation - variable equals something. With a primitive (such as integer), it looks like this:

```
// Variable Initialization

var = 10;  // var equals 10
```

Initializing an object is a bit more complex. Instead of simply assigning it a value, like with an integer or floating point number, we have to construct the object. An object is made with the new operator.

```
// Object Initialization

myCar = new Car(); // The new operator is used to make a new object.
```

In the above example, "myCar" is the object variable name and "=" indicates we are setting it equal to something, that something being a new instance of a Car object. What we are really doing here is initializing a Car object. When you initialize a primitive variable, such as an integer, you just set it equal to a number. But an object may contain multiple pieces of data. Recalling the Car class, we see that this line of code calls the constructor, a special function named Car() that initializes all of the object's variables and makes sure the Car object is ready to go.

One other thing; with the primitive integer "var," if you had forgotten to initialize it (set it equal to 10), Java would have assigned it a default value, zero. An object (such as "myCar"), however, has no default value. If you forget to initialize an object, Java will give it the value null. null means nothing. Not zero. Not negative one. Utter nothingness. Emptiness. If you encounter an error in the message window that says "NullPointerException" (and this is a pretty common error), that error is most likely caused by having forgotten to initialize an object.

Step 3. Using an object

Once we have successfully declared and initialized an object variable, we can use it. Using an object involves calling functions that are built into that object. A human object can eat, a car can drive, a dog can bark. Calling a function inside of an object is accomplished via dot syntax: variableName.objectFunction(Function Arguments);

In the case of the car, none of the available functions has an argument so it looks like:

// Functions are called with the "dot syntax".

myCar.drive();

myCar.display();

Object Practice

The Ball Class

This challenge is straightforward but a tiny bit involved. We are going to create a class which represents a ball that can bounce around the screen.

Requirements:

- Instance Variables:
  - floats: x, y, vx, vy, size.
  - Color - We haven't gone over java.awt.Color class yet, so this can be simplified to use greyscale or use r, g, and b color variables.
- Constructor
  - Takes a new x, y, size, and color

    - Set's vx and vy to a random value between -3 and 3
- Update function to change x and y by vx and vy, respectively.
    - Include functionality to bounce off walls (will require if statements);
- Paint or display function to draw the ball.

Here is the final code (an example, names don't have to be the same):

```
public class Ball {

    private float x;
    private float y;
    private float vx;
    private float vy;
    private float size;
    private Color color;

    public Ball(float nx, float ny, float nsize, Color c){
        x = nx;
        y = ny;
        size = nsize;
        color = c;

        vx = random(-3,3);
        vy = random(-3,3);
    }

    public void update(){
        x = x + vx;
        y = y + vy;

        if(x > width || x < 0){
            vx = -vx;
        }

        if(y > height || y < 0){
            vy = -vy;
        }
    }

    public void paint(){
        noStroke();
        fill(color.getRed(), color.getGreen(), color.getBlue());
        ellipse(x, y, size, size);
    }

}
```

Another Class Ring:

```
public class Ring {
  private Color c;
    private float x;
    private float y;
    private float diam;

    public Ring(PApplet app, Color newC, float newX, float newY){
        c = newC;
        x = newX;
        y = newY;
        diam = 10;
    }

    public void update(){
        diam = diam + 1;
    }

    public void display(){
        strokeWeight(10);
        noFill();
        stroke(c.getRed() , c.getGreen(), c.getBlue(), 50f);
        ellipse(x,y,diam,diam);
    }

//  public void setDiameter(float newD){
//      diam = newD;
//  }
//          public float getDiam(){
//                      return diam;
//          }

}
```

In the last lesson, we created our Car class in the same file as the rest of our program. You may have noticed that the program we were writing is actually a class itself. If you look at the line of code before our setup function:

public class Cars extends PApplet{

All of our actual program is enclosed in the Cars class's curly brackets. In actuality, almost every file that contains Java code is itself a class, which means it can potentially be an object. In our cars example, we created a Car class WITHIN another class, Cars. Although this works, and won't cause problems, it is not very good practice and if we keep adding more functionality and more classes to the program, we will end up with a very long set of code.

So let's fix this. We are going to move the Car class into it's own file. The moving part will be easy, but because of the way the Processing core works within Java, we will have to do one tricky thing to get it working.

Step 1: Creating a new class

In eclipse, create a new class by going to New > Class. Make sure you are in the Cars project. It should create an empty class:

package cars;

public class Car {

}

In Eclipse, if the students haven't noticed already, different files that are open are in different Tabs in the big pane. Go back to the Cars class program.

Take the Car class from the Cars program, and cut it (command-X) and paste it into the Car class file, and make sure you delete one of the class headers (public class Car), you only need one, and make sure you have the right amount of curly brackets everywhere. it should look like this:

```
public class Cars extends PApplet{
```

We will talk more about the extends keyword in a minute, but what this means for us right now is that the program in the that file, Cars.java, is a PApplet type of program. A PApplet is the window that actually appears when we run the program. That window itself is an Object! There is a class for that object, and that class is called PApplet.

There should only be one PApplet at a time. This means that I can’t just go into the Car class and add the words extends PApplet and everything will be ok. That will try to create ANOTHER PApplet window. So everytime we made a Car, we would be trying to create another output window. That would be bad and cause lots of problems.

Instead, we want to tell any Car instance we create which window it should do stuff on. When the Car class was INSIDE the Cars class, it already knew. But now that we’ve separated them we need some way to pass the information on.

We will do this by pass the PApplet object as a parameter to the contructor of Car. Go to the Car class and change the constructor to have one more parameter:

```
public Car(PApplet tempP, float tempC, float tempXpos, float tempYpos, float
tempXspeed) {
```

We also need to create a class variable for that PApplet, and initialize it in the constructor. Here is the class beginning and constructor:

```
public class Car {

       float c;

       float xpos;

       float ypos;

       float xspeed;

      PApplet p;

       // The Constructor is defined with arguments.
```

```
        public Car(PApplet tempP, float tempC, float tempXpos, float tempYpos, float
tempXspeed) {

         p = tempP;

         c = tempC;

         xpos = tempXpos;

         ypos = tempYpos;

         xspeed = tempXspeed;

        }

…

}
```

There is an error on the word PApplet. “PApplet cannot be resolved to a type”. We need to tell Java where the PApplet class is so it can create an instance of that class. It exists in the processing library, and we need to tell it to go get it. At the beginning of your program, even bofore the class declaration, add:

```
import processing.core.PApplet;
```

Hint: Anytime an import statement is missing, you can hover over the error, and in the suggested fixes that Eclipse gives you, you can choose to import the missing classes.

Almost there. Now that Java knows where to find PApplet, and we are passing a PApplet into the constructor, and storing it in a class variable, we can now draw on that PApplet using processing functions. However, you will notice that those functions ARE STILL ERRORS!

Well, even though we have passed it in, PApplet p is the location where we must do the drawing, and unless we tell it every time, the car class still doesn’t know where to draw when we call stroke() or any other processing function. We do though. It needs to draw on the PApplet passed to the Car, and that PApplet is called p. So we ask p to draw what we want every time, using Dot notation. Add a ‘p.’ to the beginning of every processing function, or variable (like width and CENTER):

```
public void display() {

       p.stroke(0);

       p.fill(c);

       p.rectMode(p.CENTER);

       p.rect(xpos,ypos,20,10);

      }


      public void drive() {

       xpos = xpos + xspeed;

       if (xpos > p.width) {

        xpos = 0;

       }

}
```

Ok so the errors went away in the Car class, now we just have to actually change the calls to the Car constructor in the Cars class. Go back to Cars and look at our calls. We defined to the constructor to take in the PApplet first, so we need to place it there first, but what do we put there? Where is the PApplet for the window where drawings are showing up?

```
myCar1 = new Car(???,255,0,100,2);

myCar2 = new Car(???,100,0,10,1);
```

Well, remember, the Cars class IS actually the PApplet. This is weird, but this means that whenever we create a new Car object, the class we are making it in must pass itself if it extends PApplet. There is a shortcut to do this, by using the keyword this.

```
myCar1 = new Car(this,255,0,100,2);

myCar2 = new Car(this,100,0,10,1);
```

this passes itself to the Car class. Using this in the Cars class means that the Car class will get the Cars version of the PApplet, or the window where drawings are showing up.

This is a very tricky concept, and it’s pretty advanced. If the students don’t get it, don’t worry. It’s not the focus of this class, but this setup is how processing in Java must run. So here is the formula, which can easily be memorized:

- Each Class should have it’s own file.
- That class should have an instance variable of the type PApplet, and should have a simple variable name, like p.
- You must import the PApplet class:
- import processing.core.PApplet;
- The constructor should take in a PApplet, and assign that value to the class variable (p).
- Anytime you want to use a processing function or variable, you have to use p and dot notation:
- p.rect(x,y,w,h);
- p.width;
- p.mouseX
- etc.

Add the above set of steps to your google doc notes, it will be VERY handy for the next lesson.

In the next lesson we are going to practice this set up a lot by creating object oriented versions of Rectangles and Ellipses, and optionally triangles.

STEPS IN THIS LESSON

- **Create a new Project**
- **Create a new Class**
- **Use Color**
- **Create Accessors**
- **Create Modifiers**
- **Use Color Correctly**

Ok so now that we know how to make a class that represents something, let’s make better versions of our favorite shapes: Rectangles.

Rectangles

This is a good time to start prompting students into doing things a bit on their own. Let's do this lesson as a guided series of steps that are questions, that we encourage the students to answer as we go.

Step 1. Create a new Project

- How do we create a new project?
- File > New > Project > Processing > Processing Project
- What should be name it?
- Shapes

Step 2. Creating a class.

- How do we create a new class?
- File > New > Class > *Don't include main()*
- What do we name our class?
- We can't name it rect. That is likely to be confusing. In Java programmign convention, classes should be named with capital letters at the beginning of each word in the name. There is only one word in our recommended name:**Rectangle**.
- Step one: variables: What MUST our rectangle have to work with the processing core?
- PApplet p; //the window that this Rectangle can draw on.
- Import PApplet:
- What other variables? What kind of attributes does a rectangle have? Hint: Start with the parameters of the rect() function call:
- x coordinate: private float x;
- y coordinate: private double y;
- width: private float width;
- height: private float height;
- Anything else that we could store about a rectangle that we don't already? Something that would make it better than just calling rect()? What else do we control when we draw a rectangle, that maybe this guy can store in himself?*Suggest:*
- private Color innerColor;

Let's take break and talk about color. color as a primitive doesn't exist in Java, like it did in processing. We instead have to use the Color class from the Java libraries. You'll notice that adding the above line to our class variables give you an error. You need import the Color class, just like we did with PApplet, but if comes from a different place:

import java.awt.Color;

We will go over how to use the Color class when we get to the constructor.

- How do we create the constructor?
- *Hint: Look at your Car class: Double Hint, let’s not worry about the color yet.*
- public Rectangle(PApplet tempP, float tempX, float tempY, float tempW, float tempH){
- Set up our class variables? Start with the normal ones:
- p = tempP;
- x = tempX;
- y = tempY;
- width = tempW;
- height = tempH;
- Now for the Color? Let’s just make it white in the constructor call. This will make the default rectangle color white, like it normally is.
- A Color object is a class that was written by someone else, but we can still use it, we just have to know how. We create a Color object, just like we were creating car objects before. When we do, we should pass in three values, just like colors work when we were playing with processing (or using the fill command)
- innerColor = new Color(255,255,255); // gives us white.
- Close the constructor. Your code should look like this:

```
package shapes;

import java.awt.Color;
import processing.core.PApplet;

public class Rectangle {

  PApplet p;
        private float x;
        private float y;
        private float width;
        private float height;
        private Color innerColor;

        public Rectangle(PApplet tempP, float tempX, float tempY, float tempW, float tempH){
                p = tempP;
                x = tempX;
                y = tempY;
                width = tempW;
                height = tempH;
                innerColor = new Color(255,255,255);
        }

        public void paint(){
                p.noStroke();
           p.fill(innerColor.getRed(),innerColor.getGreen(),innerColor.getBlue()); // set the color
before we draw...
           p.rect(x,y,width,height); // use the parameters we are storing!
        }

        public void setX(float newX){
           x = newX;
        }

        public void setY(float newY){
           y = newY;
        }

        public void setWidth(float newW){
           width = newW;
        }

        public void setHeight(float newH){
           height = newH;
        }

        public float getX(){
                return x;
        }

        public float getY(){
                return x;
        }

        public float getWidth(){
                return x;
        }

        public float getHeight(){
                return x;
        }
        public Color getColor(){
                return innerColor;
        }
}
```

ArrayA list of items with indices from 0 to n
ArrayListAn object oriented approach to arrays

STEPS IN THIS LESSON

- **Learn about what an array is**
- **Update our ball program to use an array**
- **Update it again to use ArrayLists**

What is an Array

Remember our BallMania program? What if I want more than one ball? I would have to make a variable fore every ball. This is already better than trying to make a variable for every aspect of every ball, but still that's a lot of variables if I want 20 balls. A better choice is to have one variable that holds all the balls, and I could just give it a number or something to tell it which particular ball I want. So, instead of:

Ball ball1 = new Ball ...;

Ball ball2 = new Ball ...;

Ball ball3 = new Ball ...;

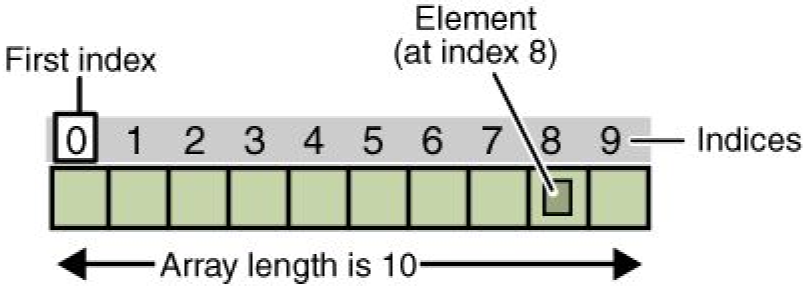
...

I could do something like

Ball balls = // 20 new Balls ...

Well, with arrays we can!

An array is group of objects (or primitives) that all have the same type. I can retrieve the value of any of the group by using a number. Each object has a different number attached to it, like an address, or we call them indexes. Think of it like a bunch of mailboxes at the post office, and each box contains an item, and each box has a number or index so I can tell which box I am using.



I create an array like this:

type[] *name* = new *type*[*size*];

For example, if I want a box with 10 slots full of integers, I can go:

int[] numbers = new int[10];

Or we can explicitly define what's in the array at the start by going:

int[] numbers = {5,6,2,3,1};

The indices of an array start at 0 and count up. So the number at index 0 in the array above is 5 (numbers[0] == 5), and the number in the last element is at index 4, and it is a '1' (numbers[4] == 1).

Let's create an array for 20 Ball objects.

Ball[] b = new Ball[20];

At this time, the array is a bunch of slots, 20 of them, with indices 0-19. But there are no Ball objects inside, since I never said new Ball(//parameters etc).

The current values of each element is 'null', meaning that there is no object there but, there is a space in memory ready to point to one.

To create 20 balls, I need to visit each spot in the array and call the Ball constructor. Should I go:

b[0] = new Ball...

b[1] = new Ball...

..etc?

No. Instead we will use a loop. A for loop can create a number which goes from 0 to the last index, and then use that number to create an object in the array:

```
for(int i =0; i < b.length; i++){

   b[i] = new Ball(this, 200,200,10, Color.GREEN );

}
```

Easy enough. However, this also means that every time draw runs, we need to access every index in the array and call update() and paint() on the ball in there.

Well:

```
for(int i =0; i < b.length; i++){

   b[i].update();

   b[i].paint();

}
```

```
package shapes;

import java.awt.Color;
import processing.core.PApplet;

public class BallMania extends PApplet {

   Ball[] b;
        public void setup() {
                background(0);
                size(400,400);

                b = new Ball[20];
                for(int i =0; i < b.length; i++){
                        b[i] = new Ball(this, 200,200,10, Color.GREEN );
                }
```

```
    }

    public void draw() {
        fill(0,0,0,15);
        noStroke();
        rect(0,0,width,height);
        for(int i =0; i < b.length; i++){
            b[i].update();
            b[i].paint();
        }


    }
}
```

ArrayList

An ArrayList is like an array, but it works like an object. This means that there is special syntax like square brackets, and instead we do all the work by using function calls and dot notation.

Why is this better? Well, array can't be re-sized without copying everything into another array. This means we can't add and remove things very easily while the program is running. With an ArrayList we can. Here's how to use an ArrayList:

Declaring, Importing.

```
ArrayList<Ball> b = new ArrayList<Ball>();  //You will then have to import ArrayList from java.util
```

Adding 20 items

Since the ArrayList is empty so far, you will have to simply go from 0 to 20, since there is no end point in the array. We simply add items to the List by using .add(Object o);

```
for(int i =0; i < 20; i++){

   b.add(new Ball(this, 200,200,10, Color.GREEN ));

}
```

Updating and Painting

You must use .size() to get the length of the array. This is better than using 20, because the size of the array can change and this for loop will still always work:

```
for(int i = 0; i < b.size(); i++){

   b.get(i).update();

   b.get(i).paint();

}
```

Here is the updated code:

```
package shapes;

import java.awt.Color;
import java.util.ArrayList;

import processing.core.PApplet;

public class BallMania extends PApplet {

   //Ball[] b;
        ArrayList<Ball> b = new ArrayList<Ball>();
        public void setup() {
                background(0);
                size(400,400);

                //b= new Ball[20];
                //for(int i =0; i < b.length; i++){
                        //b[i] = new Ball(this, 200,200,10, Color.GREEN );
                for(int i =0; i < 20; i++){
                        b.add(new Ball(this, 200,200,10, Color.GREEN ));
                }
        }

        public void draw() {
                fill(0,0,0,15);
                noStroke();
                rect(0,0,width,height);
                //for(int i =0; i < b.length; i++){
                    for(int i = 0; i < b.size(); i++){
                              b.get(i).update();
                              b.get(i).paint();
                    }
          }
}
```

Strengths and Weaknesses:

Arrays

- Fast
- Easy
- Take Less Memory
- Can't be resized.

ArrayLists

- Slower
- A little Harder to use
- Can be Resized, which is invaluable.
- Can only store objects.

Dynamic ArrayLists:

Because ArrayLists can be easily resized, we can add items to the ArrayList on a mouse click, and have lots of fun. Simply add this to draw():

```
if(mousePressed == true){

   b.add(new Ball(this, 200,200,10, Color.GREEN ));

}
```

STEPS IN THIS LESSON

- **Create Project, Import Files**
- **Add the Ball to the screen**
- **Add the Paddle to the screen**
- **Add the Bricks to the screen**

Steps in this lesson should be presented as a series of challenges. Creating breakout is simply an application of everything that we've learned so far. So, it should be taken one step at a time. Most steps also involve an advanced challenge, which is often fixing a bug.

1. Create and setup a Project

Create a new Processing Project (in eclipse) and name it 'Breakout'.

Copy and paste the Rectangle class (complete with Modifiers and Accessors) into your new project.

In setup, make the background black (for now) and the size 400 x 600

```
size(400,600);

background(0);
```

In draw, instead of calling background first, draw semitransparent rectangle over everything to create a 'blur' effect for the whole game. It looks way cooler.

```
fill(0,0,0,15);

rect(0,0,width,height);
```

Ok, now that we have Rectangle and Ball, we can make Breakout really easily.

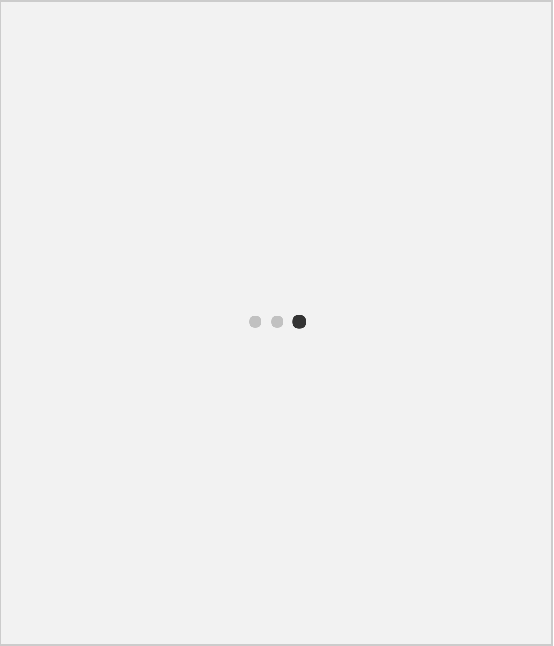
2. Adding the Ball.

Goal: *Create a new ball that bounces around the screen as the Ball class is intended to do.*

- Declare the class variable: **public Ball ball;**
-
- In setup, initialize the ball: **ball = new Ball(this, width/2, height/2, 20, Color.GREEN);**

- In draw(), update and paint the ball: **ball.update();** **ball.paint();**

Advanced: Make sure the ball always starts going down. Do not modify the Ball class.



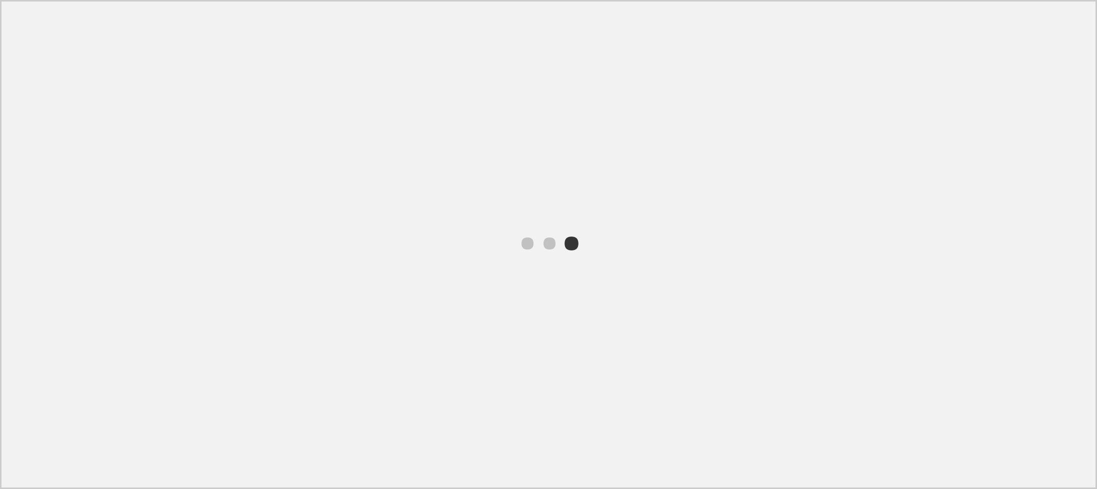
3. Adding the Paddle

Goal: Add *a paddle to the screen. Use the Rectangle class, and make it live 50 units above the bottom of the screen, have a width of 60, and a height of 20. It needs to move left and right with the mouse.*

- Declare the class variable: **public Rectangle paddle;**
- Initialize in setup(): **paddle = new Rectangle(this, 200, 550, 60, 20);**
- To keep draw() clean, let's do all the logic for updating the position and drawing the paddle in a function, which we will call from draw():

```
private void updatePaddle(){

  paddle.setX(mouseX - 30);

  paddle.paint();

}
```

Advanced: Make the paddle change color based on position.



4. Adding the Bricks

Goal: *Draw 100 Bricks on the screen. They should be arranged in a 10x10 grid. Each brick should be 15 units high, and wide enough so they fit perfectly across the width of the screen. The top row of bricks should be 100 units from the top. You can edit the paint() function of the Rectangle class to paint the outline of the rectangles a little wider than normal.*

Bricks will have to be removed from the screen when the ball hits them, and each frame we have to check if the ball is hitting them. For this reason, you should store each brick you create in an ArrayList.

- Declare and initialize the ArrayList: **public ArrayList<Rectangle> bricks = new ArrayList<Rectangle>();**
- We need to add the Bricks using a double for loop. To keep setup clean, lets package that code in a new function:

```
private void addBricks(){

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

    for(int j = 0; j < 10; j++){

      bricks.add(new Rectangle(this, 40*i, 100+15*j, 40, 15));

    }
  }

}
```

- Each frame, the bricks must be drawn on the screen. We have to loop through the array. Let's do that in a function as well, that we will call from draw():

```
private void drawBricks(){

  for(int i = 0; i < bricks.size(); i++){

    bricks.get(i).paint();

  }

}
```

Advanced: Make each row a different color.

5. Bounce off the Paddle

Now that all the elements are on the screen, we need to implement the actual mechanics of the game. Let's do all these steps in a function called updateGame().

Goal: *Get the ball to bounce off the top of the paddle.*

- What is true when the ball hits the paddle? 3 Things:
    - The center of the ball is between the left end AND the right end of the paddle. (This is really 2 things)
    - AND the bottom of the ball has moved into the top of the paddle
- So, we need an if statement with 3 conditions:

```
  if(ball.getX() > paddle.getX()                          //ball is to the right of the left side of the paddle

    && ball.getX() < paddle.getX() + 60                   //ball is to the left of right side of paddle

    && ball.getY() + ball.getSize()/2 > paddle.getY()){  //ball has crossed the top of the paddle



      ball.setVelocity(ball.getVX(), -ball.getVY());      //negate the Y velocity of the ball

  }
```

Advanced: Fix the bug where the ball will bounce constantly if it hits the side of the paddle.

6. Brick Bouncing and Destroying

Goal: *Get the ball to bounce when it hits the bottom of a brick. That brick should then be removed from the game.*

The idea of this mechanic is simple. The implementation is hard. First, the bouncing off the brick logic is actually the same as the paddle. We just need to do it to EVERY brick, EVERY frame. So the logic goes:

1. Loop through array of bricks

2. For each brick check if:
    1. The Ball is between its edges
    2. The Ball is above the bottom of the brick
3. Change the Y direction of the ball
4. Remove the brick from the array, thus preventing it from being painted again and a ball from bouncing off it again.

Here is the solution:

```
for(int i = 0; i < bricks.size(); i++){

  Rectangle b = bricks.get(i);

  if((ball.getX()+(ball.getSize()/2)) > b.getX()

   && (ball.getX()+(ball.getSize()/2)) < b.getX() + b.getWidth()

   && ball.getY()-ball.getY()/2 < (b.getY()+b.getHeight())){

    ball.setVelocity(ball.getVX(), -ball.getVY());

    bricks.remove(b);

  }

  //ADVANCED: Add another if to check the bottom of the ball

}
```

Advanced: The above solution only checks the top of ball. This will cause a bug that will prevent the logic from working correctly if the ball goes above all the bricks. To fix this, you need to add logic to test the ball bouncing off the top of a brick as well.

7. Lose Condition

Goal: Test if the user loses, and let them know.

In the same function (updateGame()) add one more chuck of logic:

```
  //check if you lost
  if(ball.getY() > height -10){

    background(0); //erase everything

    text("you lost", width/2, height/2);

    ball.setVelocity(0, 0); //stop the ball so the game will stop

  }
```

Debugging

This is the end of the lesson. The game is currently in a state where it mostly works. Fixing the bugs is left to the students who need more challenges, or who want to expand the game for their final project. After this lesson, students can begin their final project. Below is the final complete program as it was done above in the lesson.

```
package breakout;

import java.awt.Color;
import java.util.ArrayList;

import processing.core.PApplet;


public class Breakout extends PApplet {

        public ArrayList<Rectangle> bricks = new ArrayList<Rectangle>();
        public Ball ball;
        public Rectangle paddle;

        public void setup() {
                size(400,600);
                background(0);
                ball = new Ball(this, 200,300,20,Color.GREEN);
                addBricks();
                paddle = new Rectangle(this, 200, 550, 60, 20);
        }

        public void draw() {
```

```
        fill(0,0,0,15);
        rect(0,0,width,height);
        drawBricks();
        ball.update();
        ball.paint();
        updatePaddle();
        updateGame();
    }

    private void addBricks(){
        for(int i = 0; i < 10; i++){
            for(int j = 0; j < 1; j++){
                bricks.add(new Rectangle(this, 40*i, 100+15*j, 40, 15));
            }
        }
    }

    private void drawBricks(){
        for(int i = 0; i < bricks.size(); i++){
            bricks.get(i).paint();
        }
    }

    private void updatePaddle(){
        paddle.setX(mouseX - 30);
        paddle.paint();
    }

    private void updateGame(){
        //check if ball hits paddle
        if(ball.getX() > paddle.getX()
          && ball.getX() < paddle.getX() + 60
          && ball.getY() + ball.getSize()/2 > paddle.getY()){

            ball.setVelocity(ball.getVX(), -ball.getVY());
        }

        //check if ball hits a brick
        for(int i = 0; i < bricks.size(); i++){
            Rectangle b = bricks.get(i);
            if((ball.getX()+(ball.getSize()/2)) > b.getX()
              && (ball.getX()+(ball.getSize()/2)) < b.getX() + b.getWidth()
              && ball.getY()-ball.getY()/2 < (b.getY()+b.getHeight())){
                ball.setVelocity(ball.getVX(), -ball.getVY());
                bricks.remove(b);
            }
            //ADVANCED: Add another if to check the bottom of the ball
        }

        //check if you lost
        if(ball.getY() > height -10){
            background(0); //erase everything
            text("you lost", width/2, height/2);
            ball.setVelocity(0, 0); //stop the ball so the game will stop
        }
    }
}
```

Post class:

1. Attend an advanced programming DMA camp!
2. Khan academy
3. Code academy
4. Treehouse
5. CodeMaven and GameMaven
6. Lifehacker - Periodically catalogues all the code learning resources on the web
7. MUCH MORE, I'm sure you know some.
8. **Udemy. It has LOTS of courses, and most of them are free, or very very cheap.**