INTRODUCTION TO JAVASCRIPT

with p5.js

By Deborah Orret

TABLE OF CONTENTS

UNIT ONE

Introduction to JavaScript & the p5.js library

Objectives

In this unit you will write your first sketch and learn the very basic programming fundamentals! By the end of this unit you should be able to create graphic art with shapes and colors, understand how variables work and how to use them, perform operations, use and store user input, produce output, and create animations using PS.js Code!

Outline

1.1 Welcome to Java Script! Also, What is Java Script?
Actually wait... what is programming?

Maybe you've seen movies where a hacker writes a bunch of green code on the screen, maybe you've always wondered how to build your own app, maybe you've seen the Java Plug-In update pop-up and thought to yourself "Why do I even need to update this? What is Java?"

Or maybe, you know a bit more about programming - you've heard of Python, Swift, JavaScript. Maybe you've even used a couple of these!

But programming isn't really any of that - not at its core.

Programming is the process of writing instructions for a computer to perform a task. That's it! Seems easy? It is! For the most part!

The first computer programmer didn't even own a computer. Her name was **Ada Lovelace** and she lived in the 1840s. She was writing instructions to a theoretical machine she didn't even know existed - and those instructions ended up working! That's because, at its core, programming is just a logic puzzle to best answer the question: *What are the logical steps to complete a series of given tasks?*

Programming **languages** are just like human languages - different ways to convey instructions. Some people speak English, others speak Spanish, others speak Japanese. But you can say the same thoughts in any language - with some variation. Programming languages work the same way. For the most part, the language you choose to code in is up to preference - your own, your boss's or teacher's, or of the system and program you're using.

Syntax is what we use to refer to the spelling and grammar of a programming language. Syntax can change language to language, and without the proper syntax, the computer won't be able to read the instructions you are giving it. But it is important to focus on the logic and not to get *too* hung up on the syntax. No need to work on memorizing it - with practice, you'll memorize common syntax naturally and you can always look up something if you're not sure.

In this course, we'll be using **JavaScript** and the **p5.js library**.

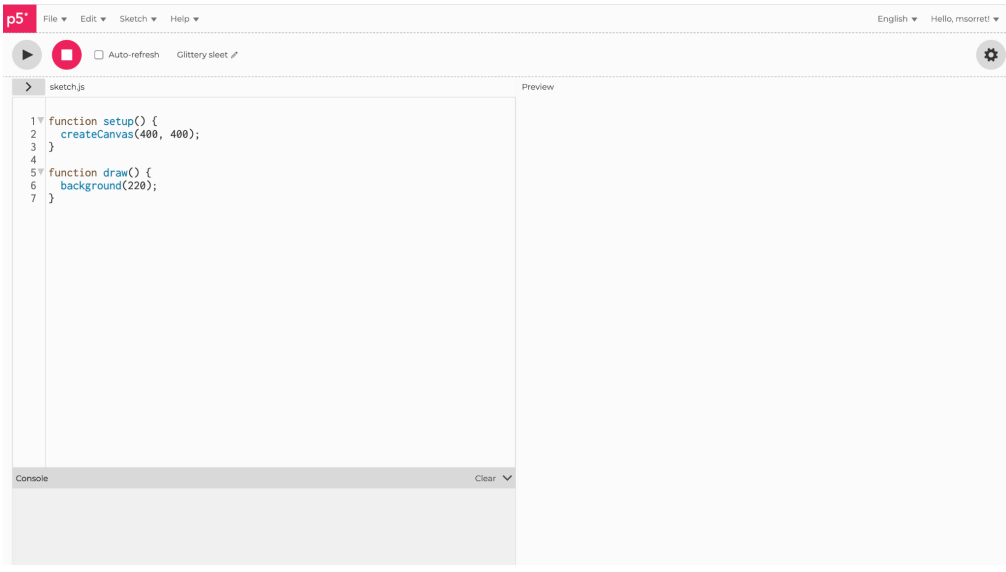
JavaScript is a programming language that can run in your web browser - which means you can code anywhere you have internet! **p5.js** is a

JavaScript **library** that makes it easy to make simple games, animations, and interactive art in the web browser. A library is just reusable code with certain functionalities that someone else wrote, so that you don't have to do it all from scratch - kind of like a cake box mix!

Let's go ahead and start!

First, go to **https://editor.p5js.org/**

Something like this should pop up:



This is the p5.js web editor - which allows you to code and access your code on any device no matter when you are, as long as you have wifi!

Take a second to **Sign Up** and make an account. Once you make an account, you'll be able to save, edit, and revisit past work.

You write code on the left side of the page. The right side of the page is where the canvas and any graphics will be displayed. The console is the gray area on the bottom left, which is used for nongraphical output like printing out "Hello World!". To save your code, File -> Save. A javascript file is called a **sketch.**

At the top we have a gray run button and a red stop button. The run button runs any code you have written in the editor and the stop button stops your code from running. Easy enough!

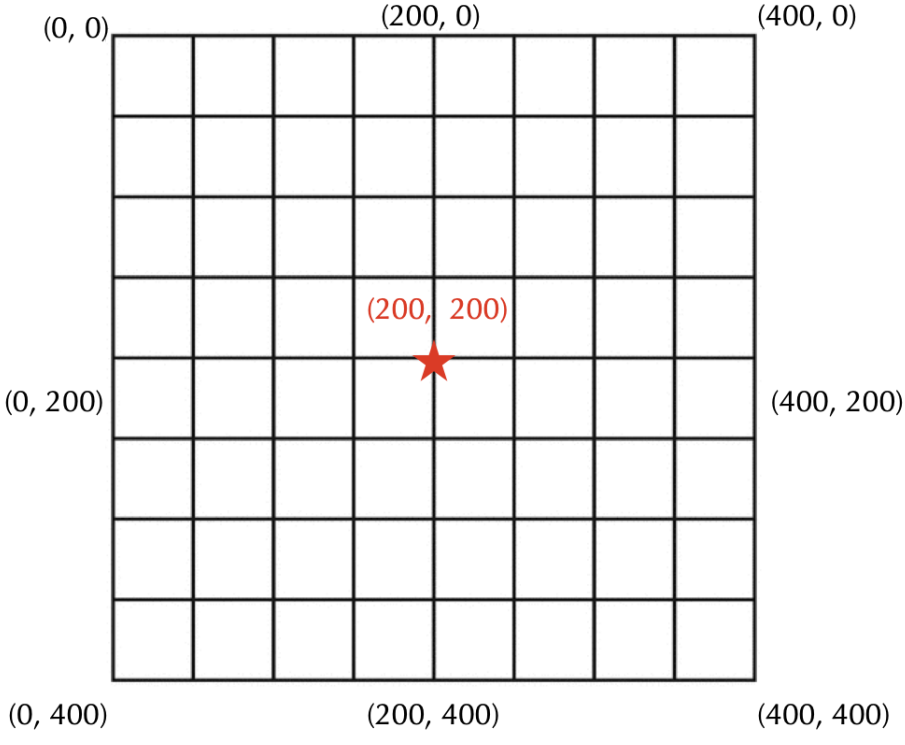
When you make a new sketch in the P5.js web editor, you'll see that there are two JavaScript **functions** already there: **setup** and **draw**. Take a second to look at the lines of code already filled in. What do you think they're for? Don't worry too much if you don't quite understand what a function is, you'll cover that soon!

For now, think about what these lines of code might do and then... google it and see if you're right! Learning how to find information is a huge part of programming. Don't know what to search? Google "setup and draw in p5js" and see what comes up!

In the next section, we'll write our first line of code!
The Canvas

The canvas is the plane that displays any graphical output and is created by the **createCanvas(width, height)** function call in the setup function. By default, the canvas is 400 pixels wide and 400 pixels tall.

The origin point (0,0) is in the top left corner. And, there are only positive values on the grid, unlike a coordinate plane you might have seen in math! Take a look at the grid below - this is how you should mentally picture the canvas and where shapes are placed!

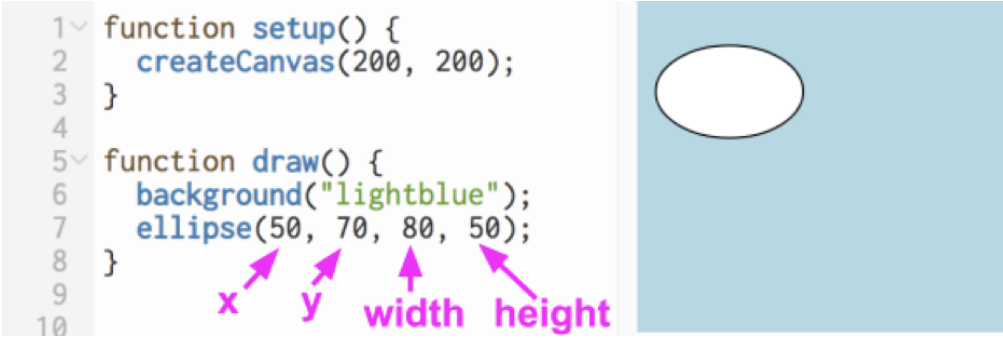


Drawing Shapes

One of the features of the p5 library is the ability to draw graphics and shapes to the canvas using prebuilt functions! You can use the draw function to place shapes all around your canvas. To fill in the body of the draw function, write your code in between the 2 curly braces!

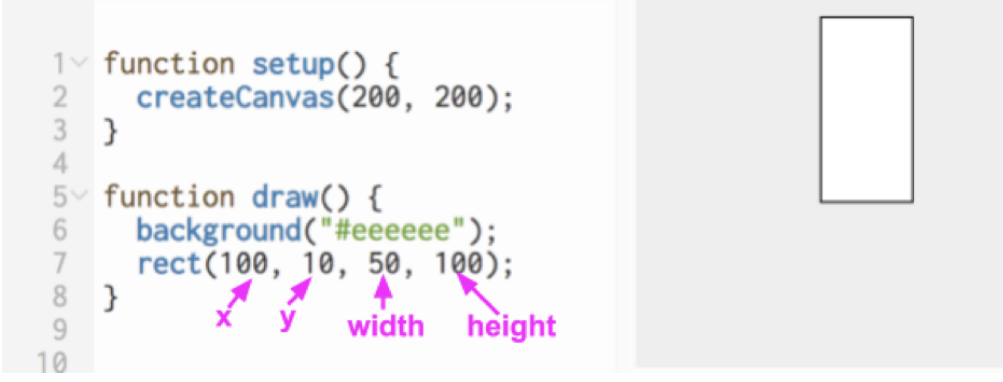
Let's start with the ellipses, rectangles, and triangles.

You can make an ellipse by calling the function **ellipse(x, y, w, h)**. The x, y, w, and h are variables that represent four numbers that you will fill in! The x and y represent the coordinates on the Canvas, and the w and h represent the width and height.



You can draw rectangles by giving the rectangle function four numbers.

You call it like this: **rect(x, y, w, h);**



Triangles are drawn by calling **triangle(x1, y1, x2, y2, x3, y3);**

Each x/y pair represents the coordinates of one of the points in the triangle. Try this one out yourself!

Filling In Shapes and Background Color

You can use the function **background(color)** to paint your entire canvas a certain color! You can use color names, hex codes, or even RGB numbers.

Color names are built in names that the computer recognizes as a color! Some examples are “red”, “blue”, “green”, “yellow”, “orange”, “purple”, “black”, “white” - and even some fun ones like “cyan”! Color names are written in all lowercase, inside quotes!

Example: background(“red”);

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background("red");
}
```

HEX codes refer to a unique code that have been assigned to every color! This code system allows the computer to recognize any color and any shade. For example, the HEX code for white is #FFFFFF and the HEX code for a nice sky blue shade is #D6FCF9. You can find the HEX codes for any color you would like on htmlcolorcodes.com! Choose the color on the color picker, and then copy down the HEX code. HEX codes are written in quotes, with the format: #HEXCODE.

Example: background(“#D6FCF9”);

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background("#D6FCF9");
}

```


RGB codes work similarly! RGB codes refer to the unique combination of three numbers from 0-255 that make up a color. You can find the RGB code on the same website, htmlcolorcodes.com! Choose the color on the color picker, and then copy down the RGB numbers. RGB codes are written inside parenthesis in the format (r, g, b).

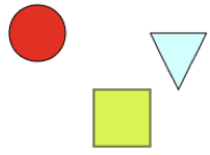
Example: background(255, 205, 0);

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(255, 205, 0);
}
```


You can use **fill** to tell P5 to “pick up a paintbrush” of a certain color and fill in your shapes. Until you call fill again with a different color, every shape you draw will be that color! You can use the same color names, HEX codes, and RGB with the fill function!

Check out the example below:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background("white");

  fill("red");
  ellipse(100, 100, 50, 50);

  fill("#D5F324");
  rect(150, 150, 50, 50);

  fill(200, 300, 400);
  triangle(200, 100, 250, 100, 225, 150);
}
```



NOTE!! Lines of code run chronologically. If you paint the background after you paint your shapes, they won’t show up! Think about it like you are actually painting!

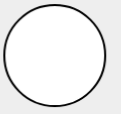
Look at the example below.

```
function setup() {
  createCanvas(200, 200);
}

function draw() {
  ellipse(50, 50, 50, 50);
  background("#eeeeee");
}
```

This shape won’t show up because the background is painted over it:

In this next example, we've fixed the problem by painting the background, then the shape. Always look at the order of your lines of code and make sure they're written in the order you want them to run.

```
function setup() {
  createCanvas(200, 200);
}

function draw() {
  background("#eeeeee");
  ellipse(50, 50, 50, 50);
}
```



Syntax and Errors

Here are just a few basic syntax components of Javascript to note!
Comments allow you to write notes for yourself or collaborators in your code that won't actually be run! You can also use comments to comment out code that you don't want to run!

```
// two slash marks start a single comment line

/* Multiline comments are enclosed by /*
   like this
   comment!
*/
```

Semicolons mark the end of a statement. They are technically optional in JavaScript... BUT, I recommend always including them, because otherwise the computer might get confused about where a statement actually ends. It's just good practice!

```
function draw() {
  background(220);
  ellipse(100, 100, 100, 100);
}
```

Case sensitivity is something you DO need to worry about with Javascript. Just like your password for your computer complains if you turn on caps lock, your code will produce an error if you aren't careful about writing your expressions in the right case.

Example: var **myVariable** is not the same as var **mYvarIabLe**

Some things always come in pairs to enclose, group, and identify information in certain ways!

{ } curly brackets
( ) parenthesis
“ “ double quotes
‘ ‘ single quotes

You might put other code or information inside these pairs, but if you see one - make sure it has a matching friend that goes with it!

On the flip side of good syntax and coding practice, we have **errors**. Errors happen when there is something wrong with your code. We also sometimes call errors **bugs**, and the process of finding and fixing errors **debugging**. The terms "bug" and "debugging" were coined by Admiral and programmer **Grace Hopper** in the 1940s. While she was working on a Mark II computer at Harvard University, her associates and her discovered a moth stuck in a relay and causing problems. She remarked that they were "debugging" the system and the term stuck!

There are three types of errors, syntax errors, runtime errors and logic errors.

Syntax errors happen when there is a syntactical mistake. Think of these like spelling or grammar errors. Usually these errors are easy to fix, notice, and lookup. Most of the time, there will be an error message, telling you the exact line the error is in and what's wrong.

Runtime errors happen at execution - which basically means when you press run. They aren't problems with the syntax, but they prevent the program from being able to run and they also produce an error message. Imagine you are trying to text a number that doesn't exist. You would get a text back that you got the wrong number. It's not really that the number format is wrong - it's just that the specific number you chose doesn't exist. That is an example of a runtime error code.

Logic errors happen when something in the instructions don't make sense. Imagine someone giving you directions on how to drive. They tell you: "First open the garage door, then turn right, then turn on your car." The sentence is grammatically correct, but it makes no sense. There are critical instructions missing and it is in the wrong order. That is an example of a logic error. Logic errors can be more tricky to find. Error messages won't pop up, the program will run anyways - it just won't do what you want it to do. Line by line debugging can be really useful when dealing with logic errors.

1.2 Variables

A **variable** is like a box in the computer's memory that stores a value for you. It serves as a placeholder for information.

Think about variables in terms of a contact on your phone. When you create a new contact, you can add a number for that contact - so that you don't have to type the number every time you want to call that person. The contact represents the information you enter for them. But, if they get a new number, you don't have to make a new contact. You can just go and change the value of the old contact to represent the new number.

You might have also learned about variables in a math class. They're kind of the same thing! Often in math, we represent variables with letters: x, y, z. In computer science you can name your variables anything you want! But you should name them something relevant - so you remember what they stand for. If you named all your contacts gibberish, it'd be pretty hard to keep track of who you're texting!

And don't forget! Javascript is case sensitive! So if you make a variable called MY_VARIABLE, and then later try to use it but type my_variable, you'll get an error!

Declare and Assign a Variable

To tell your computer to create a variable, you need to **declare it** and give it a name. This is the same as creating a new contact!

For example, this code declares a variable named **`my_variable`**:

```
var my_variable;
```

Once a variable is declared, you can store a value in it! To store a value in a variable, use the **assignment operator, =**. Imagine this as adding a number to that contact. For example, this code assigns the value 50 to `my_variable`:

```
my_variable = 50;
```

If you want, you can declare and assign a variable in the same line of code, like this:

```
var my_variable = 50;
```

But remember! Either way you do it, you always have to declare the variable before or at the same time that you assign any value to it. To continue with our contact analogy, you can't add a number to a contact that doesn't exist.

Manipulating Variables

Once you have declared a variable, it's not fixed forever! You can always change the value or type.

For example, look at the code:

```
1 var a = 10;
2 var b = 20;
3 a = 5;
4 b = 3
```

Originally the values of a and b are 10 and 20. But after lines 3 and 4 the values of a and b are 5 and 3.

To figure these types of problems out, we can just trace the code by hand one line at a time, keeping track of the values of each variable along the way! Remember, Javascript is chronological!

Data Types

In Javascript, variables themselves do not have types, but the values they hold do. The term for all the different types of values is **data types**. You might actually already be familiar with the basic data types and the differences! Here are two main ones to start with - but you'll learn a few more throughout this book!

numbers : Hold numerical information. This includes positive numbers, negative numbers, and 0

```
1 var num = 5;
2 var num = -19;
```

strings : Hold any message in single or double quotes. Strings can contain letters, symbols, and numbers.

```
1  var str = "hello";
2  var str1 = 'I am 23 years old';
3  var str2 = "23&Me";
4  var str3 = '909-555-2813';
```

To move to a new line use “\n” This would represent: hello world!

```
1  var str4 = "hello \n world!";
```

MouseX and MouseY

MouseX and **MouseY** are variables that store values that can change over time. MouseX always stores the mouse's x-coordinate, and MouseY always stores the mouse's Y-coordinate.

If we use these variables instead of just constant numbers for a shape's X and Y coordinate, the shape will move with the mouse! Essentially, the computer is constantly updating these variables to reflect the coordinates of your cursor, so that anything you use these coordinates for will follow your mouse.

Let's look at the code below:

```
1
2  function setup() {
3    createCanvas(400, 400);
4  }
5
6  // The purpose of this sketch is to show you how to use
7  // mouseX and mouseY to make shapes follow the mouse.
8  // mouseX and mouseY are called "variables"--they store
9  // values that can change over time. These two variables
10 // store the coordinates of the mouse on the canvas.
11 function draw() {
12   background("black");
13
14   fill("white");
15   // This ellipse follows the mouse exactly!
16   ellipse(mouseX, mouseY, 50, 50);
17
18   fill("purple");
19   // This ellipse will be 40 pixels to the right and
20   // 25 pixels above the mouse!
21   ellipse(mouseX + 40, mouseY - 25, 20, 20);
22
23   fill("limegreen");
24   // This ellipse will be 20 pixels to the left and
25   // 50 pixels below the mouse!
26   ellipse(mouseX - 20, mouseY + 50, 30, 30);
27 }
```

Think about what you might see on the display when the code runs. First, read the comments and try to think of what each line is doing! Then copy it down, run the code, and see if you're right! To test your understanding on mouseX and mouseY, play around with this code. See if you can make a smiley face out of the shapes that follow the mouse around.

1.3 Input and Output in Programming

Input refers to anything a user passes into our program.

Output refers to anything we display for either the user or programmer to see.

Prompting input is useful if our program relies on information that is different for every user. For example, if you're building an app that requires a password to enter, you need some form of way to prompt users for their password, then store and check what they've entered.

Displaying output is useful in pretty much any type of program that interacts with a user. This one is pretty self explanatory - imagine if we couldn't tell our users anything!

Prompting and Storing User Input

The function **prompt()** (very fittingly) **prompts** a user to input a string. Write any question or prompt you want the user to answer in between the parenthesis. Look at the example here:

```
var yourName = prompt("What is your name?");
```

This code asks the user "What is your name?" and then stores whatever the user types in the variable `yourName` *as a string*.

Make sure to always store the user answer in a variable - otherwise the computer will immediately forget it. Imagine if you asked someone a question and then immediately forgot what they said! You'd be asking the same question forever!

To prompt for a number, we need to add something to the prompt function to convert it. Look at the code below! Can you see the difference?

```
var yourAge = Number(prompt("What is your age?"));
```

Printing to the Console

Printing to the console is one type of output that can print any data type so the programmer can see its value! The function `console.log()` prints things out to the console. Between the parentheses, put whatever you

want to print. For example, to print out the string “Hi there!” or the number 5, you could write:

```
1  console.log("Hi there!"); // will print Hi There!
2  console.log(5); // will print 5
```

To print out the value of a variable, just don’t put it in quotes!

```
1  var yourName = prompt("What is your name?");
2  console.log(yourName); // will print whatever name the user answered
```

1.4 What are Operators

Operators are symbols used to signal to the computer to perform a specific operation.

Arithmetic Operators

Symbol	Description	Example	Result
+	addition	5 + 3	8
-	subtraction	5 - 3	2
*	multiplication	5 * 3	15
/	division	5/3	1.667
%	remainder	5%3	2
x++	increase by 1 (postfix)	x = 5 y = x++	x is equal to 6 y is equal to 5
++x	Increase by 1 (prefix)	x = 5 y = ++x	x is equal to 6 y is equal to 6
x--	decrease by 1 (postfix)	x = 5 y = x--	x is equal to 4 y is equal to 5
--x	decrease by 1 (prefix)	x = 5 y = --x	x is equal to 4 y is equal to 4

* The difference between the postfix and prefix operators is that when they are postfixed, the value is incremented/decremented but returns the value before doing that operation. For prefixed operators, the value is incremented/decremented, and then that new value is returned.

Assignment Operators

Symbol	Description	Example	Result
=	equals (assignment)	x = 5;	x now equals 5

Comparison Operators

Symbol	Description	Example	Result
= = =	Is the value and type on the right side equal to the left?	5 === 5	true: 5 equals 5
!==	Is the value and type on the right side not equal to the left?	5 !== 7	true: 5 does not equal 7
==	Is the value on the right side equal to the left?	5 == “5”	true: 5 = “5”
!=	Is the value on the right side not equal to the left?	“5” != “5”	true: “5” = “5”
>	greater than	6 > 5	true: 6 is greater than 5
<	less than	2 < 3	true: 2 is less than 3
>=	greater than or equal to	6 >= 5	true: 6 is greater than or equal to 5
<=	less than or equal to	3 <= 3	true: 3 is less than or equal to 3

Performing Operations

Example Problems

Problem 1

```
var a = 50;
a = 10;
var b = 2;
var c = a / b;
var d = c * 3;
console.log(a);
console.log(b);
console.log(c);
console.log(d)
```

What values will be printed to the console? Check your answer by running the code above!

Problem 2

```
var a = 4 * 3;
var b = 100 / 4;
var c = a + b;
console.log(a);
console.log(b);
console.log(c)
```

What values will be printed to the console? Check your answer by running the code above!

Problem 3

```
var a = 10 % 3;
var b = 8 % 3;
var c = 15 % 3;
console.log(a);
console.log(b);
console.log(c);
```

What values will be printed to the console? Check your answer by running the code above!

String Concatenation

If you put the operator **+** between two strings, it will **concatenate** the strings - which means it will stick them together to make one string. For example:

```
var schoolAndMascot = “PHS” + “Bulldogs”;
console.log(schoolAndMascot);  // Prints “PHSBulldogs”
```

You can also use variable values when you’re concatenating strings. This can be useful when you want to make a longer message that includes a variable value. For example:

```
var song = prompt(“What is your favorite song?”);
console.log(“I also like ” + song);
```

The code above will print out “I also like ___” - where the blank will be filled in by the song that the user wrote in. So if the user answered “Despacito”, it will print out “I also like Despacito”.

But what happens if you create the string one that holds “1”... and then you create a string two that holds “2” ... and then you call console.log(one + two) ... what will be printed to the console?

You might think “3” - duh! But look closer! What does the computer think “1” and “2” are? Numbers or strings? When something is in quotes, whether it is a number, symbol, or a letter, the computer reads it as a string. When a number is written as a string, it is not stored as a *value*. The string ”1” does not represent the numerical value of 1, instead it represents a symbol that is one long vertical line.

So that means when we concatenate two strings with numbers, you add the numbers as strings! So you basically just smash 'em together into one bigger string!

```
console.log(“1” + “2”); // prints out 12
console.log(“461” + “0 meters”); // prints out 4610 meters
console.log(“xbox” + “360”); // prints out xbox360
```

It’s a bit of a weird concept to wrap your head around - so play around with different strings and variables, print them to the screen, and see the results! The more examples you see, the easier it will be to catch on how it works! Start with the examples above and add from there!

Unit 1 Review

Programming is the process of writing instructions for a computer to perform a task.

Syntax is the spelling and grammar of a programming language.

Syntax errors happen when there is a syntactical mistake.

Runtime errors happen at execution. They aren't problems with the syntax, but they prevent the program from running and produce error messages.

Logic errors is a bug that causes the program to run incorrectly, but does not produce an error message.

A **variable** is like a box in the computer's memory that stores a value for you. It serves as a placeholder for information.

MouseX and **MouseY** are variables that store values that can change over time. MouseX always stores the mouse's x-coordinate, and MouseY always stores the mouse's Y-coordinate.

Input refers to anything a user passes into our program. Output refers to anything we display to the console for a user to see.

Operators are symbols used to signal to the computer to perform a specific operation.

```
function setup() { // sets initial properties
  //body
}

function draw() { // draws to the screen whatever you place in the body
  //body
}

ellipse(x, y, width, height); // creates an ellipse at coordinates x and y,
                              // with the dimensions width/height

rect(x, y, width, height); // creates an ellipse at coordinates x and y,
                           // with the dimensions width/height

triangle(x1, y1, x2, y2, x3, y3); // creates a triangle made up of the three points:
                                  // (x1,y1), (x2, y2), and (x3,y3)

background("Color you want the background to be"); // fills the background


fill("Color you want to change your paintbrush to"); // fills shapes


prompt("Fill in what you want to ask the user here"); // prompts user for string input


console.log("Fill in what you want to print to the console"); // prints to the console

```

Test Yourself

1. Who was the first programmer?
2. What is the definition of programming?
3. How are programming languages and human languages different and the same?
4. Javascript is case sensitive and chronological. True/False?
5. Compare and contrast the three types of errors.
6. What is a variable?
7. Compare and contrast numbers and strings. Give three examples of both.
8. What is the difference between declaring and assigning a variable?
9. What is input and output?
10. What are operators? What are the three categories of operators? Give an example for each category!
11. What is the difference between ++x and x++?

Practice Problems

1. Use what you've learned about rectangles, ellipses, and colors to draw an **alien or unique character**!

2. Make a sketch that asks the user for their name, and then asks the user for their age in a separate prompt - and stores the responses in two variables, **name** and **age**. Then, print out the sentence "Your name is NAME and you are AGE years old" using the user answers!

3. You can animate a shape in p5.js if you make its x, y, width, or height depend on some variable, and then continuously change the value of that variable. Make an animation!
To do this, start in the draw loop! Declare/define a variable that equals 0! Draw a circle somewhere on the screen! Replace either the number you have in the x or y place with the variable a! Then use the increase operator ++ to add 1 to a every time the draw loop runs! See what happens!

4. The following code is supposed to add two numbers together and print the output to the console. However, there is a logic error in the code. Find the error and fix it. (Hint: Think about how *sum* is printed out.)

```
var num1 = prompt("What is the first number?");
var num2 = prompt("What is the second number?");
```

```
var sum = num1 + num2;
console.log(“The sum is: “ + sum);
```

5. Make a sketch that asks users for 2 different numbers. Then, calculate the product of these two numbers and print it out. Next, calculate the difference between the product you calculated above and the 2nd number. Print this out as well.

6. Fill in the code below in order to print out **string1** and **string2** on separate lines. Only add one line of code.

```
var string1 = “Today is Monday.”;
var string2 = “It is cold outside.”;

// Add code below this line
```

7. The following code is supposed to print out *name1*, but there are some syntax and logic errors. Find the errors and fix them.

```
var  name1 = Cara”;
console.log(“name1”;
```

UNIT TWO

Booleans & Conditionals

Objectives

In this unit you will understand how to use conditionals (if/else statements) to make decisions in your code. By the end of this unit, you should be able to write your own if/else statements and else if statements, use && and || operators in your conditional statements, and understand how to evaluate boolean expression.

Outline

2.1 Booleans

Booleans are a data type, just like strings or numbers. Booleans store one of two possible values: **true** or **false**.

For example:

```
var goesToPHS = true;
var ateBreakfast = false;
```

You can write a **boolean expression**: an expression that is evaluated to either true or false.

Remember the comparison operators from last unit? If not, don't worry - check back to Section 1.4!

You can use those to compare variables and information, as boolean expressions.

For example:

```
4 > 3;          // "4 is greater than 3" is TRUE
"Hi" === "sup";  // "'hi' is equal to 'sup'" is FALSE
10 <= 7;        // "10 is less than or equal to 7" is
FALSE
true !== false;  // "true is not equal to false" is TRUE
```

NOTE: two strings are only equal if they are **exactly the same** (in terms of spelling and capitalization). So "hi" and "Hi" are not equal!

Some functions can also return true or false. For example - let's say you want to check if a value is a whole number/integer! You can use a function that will return true if the value IS an integer and false if it's NOT.

```
Number.isInteger(3);   // is true! 3 is a whole number
Number.isInteger(0.3); // is false! 0.3 is a decimal not a
                       // whole number
```

2.2 Conditional Statements

Conditional statements are blocks of code that check if a certain condition has been met and execute if it has. We make all sorts of decisions in our day-to-day life that can be written as conditional statements.

If I wake up on time, I will eat breakfast.
If the weather is nice, I will go to the beach.
If it is Monday, I will go to school.

Programming languages use **if statements** to represent these types of conditional statements.

The syntax for an **if statement** looks like this:

```
if (condition) {
        do stuff!
}
```

The **condition** needs to be a boolean expression. It needs to evaluate to either true or false. If the condition is true, the code between the curly braces will run. If the condition is false, it will be skipped over.

For example:

```
1 // remember to always use comparison operators for conditionals (===)
2 if (wakeUpOnTime === true) {
3       var breakfast = "waffles";  // here you use = because it is
4                                   // an assignment, not a comparison
5 }
```

This code says, if wakeUpOnTime === true, then breakfast = “waffles”. If you wake up on time, you will make waffles for breakfast. Some motivation to get out of bed on time!

But what happens if you sleep through your alarm and wakeUpOnTime === false? The body of that statement will be skipped over completely. You won’t be able to make any waffles!

You might be asking yourself... is there a way to explain what to do if the first condition isn’t met? Let’s say, if you don’t wake up on time to make waffles, you want to make sure you grab a granola bar for breakfast on the way out.

An **if/else statement** allows you to do exactly that. The if part of the statement works just like the if statement we explained above, but the else allows you to let the computer know what to do if the condition isn't met.

The structure for an if/else statement looks like this:

```
if (condition) {
      do stuff;
} else {
      do other stuff;
}
```

This statement says, if the condition is met do something, if not do something else.

For example:

```
1 if (wakeUpOnTime === true) {
2
3         var breakfast = "waffles";
4 } else {
5         var breakfast = "granola bar";
6 }
7
```

If you wake up on time, breakfast will be waffles. If not, breakfast will be a granola bar. Set your alarms!

Else If

Now you might have another question...
What if you want to have a second or third choice? What if you don't wake up early enough to make waffles, but you wake up early enough to make toast. Shouldn't there be a way to have multiple options for conditions?

In these cases, you would use an **else if** statement.

The structure for an **else if** statement looks like this:

```
if (condition) {
      do stuff;
} else if (condition 2) {
      do other stuff;
}
```

For example, let's say you want to tell a user what grade they got based on their percentage on a test. 90+ is an A, 80-89 B, etc.

```
1  if (grade >= 90) {
2  console.log('A');
3  } else if (grade >= 80) {
4  console.log('B');
5  } else if (grade >= 70) {
6      console.log('C');
7  } else if (grade >= 60) {
8  console.log('D');
9  } else {
10 console.log('F');
```

Our code will check each condition, in order. As soon as it finds one that's true, it runs the code in that block and skips over the rest. So if the user entered **75** as their grade, the code would check:

- "Is 75 greater than or equal to 90? Nope. So it's not an A. Better check the next one."
- "Is 75 greater than or equal to 80? Nope. So it's not a B. Better check the next one."
- "Is 75 greater than or equal to 70? YES. So let's log "C" to the console and we're done!"

Let's run through that code!

```
var grade = 75;

if (grade >= 90) {// 75 is not greater than 90! Skip this
block!
      console.log("A");
} else if (grade >= 80) { // 75 is not greater than 80! Skip
this block!
      console.log("B");
} else if (grade >= 70) {     // 75 IS greater than 90! Run
this block!
      console.log("C");       // You found a condition that was
met!
} else if (grade >= 60) {     // So the rest of the code is
skipped over!
      console.log("D");
} else {
      console.log("F");
}
```

What would happen if I changed the order of my conditions? Check out this code!

```
var grade = 75;

if (grade >= 60) {            // 75 IS greater than 60! Run
this block!
      console.log("D"); // You found a condition that was met!
} else if (grade >= 70) {     // So the rest of the code is
skipped over!
      console.log("C"); // But is that what we want?
} else if (grade >= 80) {
      console.log("B");
} else if (grade >= 90) {
      console.log("A");
}
```

Order matters!!! Keep in mind how the computer executes your code!

Nested If-Else Statements

You can also place if-else statements within other if-else statements. When you do this it is called a nested if-else statement. This can be very useful when you have to check multiple conditions that depend on each other. The structure of a nested if-else statement looks like this:

```
if (condition) {
      // do something
      if (condition) {
            // do something
      }
} else {
      // do something
}
```

This is just one way you can write a nested if-else statement. You can also place another if-else statement within the else-block or within an elif block. You can also infinitely nest if-else statements (provided it doesn't exceed time constraints or resource limits).

Here is an example of a nested if-else statement below:

```
var grade = 75;

if (grade >= 70) {
  console.log('You are passing.');
  if (grade >= 90) {
    console.log('You have an A');
  } else if (grade >= 80) {
    console.log('You have a B.');
  } else {
    console.log('You have a C');
    }
} else {
    console.log('You are failing.');
    if (grade >= 60) {
        console.log('You have a D.');
    } else {
        console.log('You have an F.');
    }
}
```

2.3 And (&&) and OR(||)

&& (AND)

Sometimes when we make decisions, we want to know whether two things are both true at once. For example, if I am hungry AND I have enough money, I'll go buy ice cream!

In code, we can check whether two conditions are both true using the && operator, which means AND.

Here's how && works:

```
var hungry = true;
var money = 5;
var iceCream;
if (hungry === true && money >= 3) {
            iceCream = true;
} else {
            iceCream = false;
}
```

If you're using &&, the whole expression will be true **only** if **both** sides are true. If either side is false, the whole expression is false.

```
console.log(true && true);          // prints true
console.log(true && false);         // prints false
console.log(false && true);         // prints false
console.log(false && false);        // prints false
```

When you're using &&, each side of the && must be a complete boolean expression (that's either true or false). For example, let's say you have two numbers, a and b, and you want to know if both numbers are greater than 100. Here's the wrong way (and right way) to do it.

```
~~if (a && b > 100) { … }~~       // WRONG! DO NOT DO THIS!
if (a > 100 && b > 100) { … } // RIGHT! DO THIS!
```

II (OR)

We can also check whether at least one of two things is true, using the **|| (OR) operator**. For example, if you have money OR a get one free coupon, then you can buy ice cream. Otherwise, you can't.

Here's how **||** works:

```
var money = 5;
var coupon = true;
var iceCream;
if (money >= 3 || coupon === true) {
      iceCream = true;
} else {
      iceCream = false;
}
```

If you're using **||**, the whole expression will be true if either side is true (or both are true). The only way the whole expression is false is if both sides are false.

```
console.log(true || true);          // prints true
console.log(true || false);         // prints true
console.log(false || true);         // prints true
console.log(false || false);        // prints false
```

Here's an example of using **||** in context. If a restaurant is serving pancakes **OR** waffles today, then I'll eat breakfast there. Otherwise, I'll go somewhere else.

```
1  var servingPancakes = prompt('Are you serving pancakes today?');
2
3  var servingWaffles = prompt('Are you serving waffles today?');
4
5  if (servingPancakes === 'yes' || servingWaffles === 'yes') {
6      console.log('Great! I'll order breakfast here');
7  } else {
8      console.log('Boo. I'm going somewhere else!');
9  }
```

One interesting thing about the || operator is that if the first condition in the statement is true, the second condition is not even evaluated since only one condition needs to be true for the entire statement to be true.

Using multiple &&s and ||s

You can use as many **&&**s and **||**s as you want in the same expression. For example, you can check:

```
if (a > 100 && b > 100 && c > 100) { … }
```

One thing to keep in mind is that && has higher precedence than ||, so any &&s in your expression will be evaluated first. If you want to evaluate an || first, use parentheses! For example:

```
if ((quiz1 >= 90 || quiz2 >= 90) && (test1 >= 90 || test2 >= 90)) {…}
```

Here's a more complicated example that uses &&s and ||s:

You want to know if you got an A in your class. There are two parts of your grade: homework and tests. You can also complete extra credit. The extra credit can raise one of your grades from a B to an A. After the extra credit is added, you need to get an A on both the homework and tests to get an A in the class.

This code prompts you for your grades and if you did the extra credit. Then it will output a message telling you if you got an A overall.

Read the code below! Copy and run it! What happens?

```
// prompts for and stores user's homework grade, expected input A, B, C, D, or F
var HGrade = prompt("What grade did you get on the homework?");
// prompts for and stores user's homework grade, expected input A, B, C, D, or F
var TGrade = prompt("What grade did you get on the test?");
// prompts for and stores if user did the extra credit, expected input: yes or no
var EC = prompt("Did you do the extra credit?");

if (HGrade === "A" && TGrade === "A") { // homework and test grades are both As
    console.log("You got an A in the class!"); //print "You got an A in the class!"
}
// homework grade is a B, test grade is an A, and the extra credit was done
// OR
// homework grade is an A,  test grade is a B, and the extra credit was done
 else if ((HGrade === "B" && TGrade === "A" && EC === "yes") || (HGrade === "A" && TGrade === "B" && EC === "yes")) {
    console.log("You got an A in the class!");
}
// the grades and extra credit were not high enough to get an A
else {
    console.log("You did not get an A in the class :("); }
```

Unit 2 Review

Booleans stores one of two possible values: true or false.

Boolean expression: an expression that is either true or false.

Conditional statements are blocks of code that execute if a certain condition has been met. The condition must be a boolean expression.

Structure for an if statement:

```
if (condition) {
      do stuff!
}
```

Structure for an if/else statement:

```
if (condition) {
      do stuff;
} else {
      do other stuff;
}
```

Structure for an else if statement:

```
if (condition) {
      do stuff;
} else if (condition 2) {
      do other stuff;
}
```

&& : AND

||: OR

Test Yourself

1. What is a boolean? What do they store?
2. What is a boolean expression? Write an example!
3. How do you use a boolean expression in a conditional statement?
4. When do you use an if/else statement? When do you use an else if statement?
5. True or False: If you're using &&, the whole expression will be true **only** if both sides are true.
6. True or False: If you're using ||, the whole expression will be true **only** if both sides are true.

Practice Problems

1. Make a P5.js sketch that asks the user 3 trivia questions and checks their answer to each one. If they answer a question correctly, print "Right!". Otherwise, print "Wrong!" and print the correct answer.

2. Update your trivia sketch to allow multiple answers to be correct! (Hint: use && and ||)

3. *Trick Question!* What is the output of this code?

```
var dollars = 5;
if (dollars >=20) {

    console.log("You can buy a book")
} else if (dollars >= 10) {

    console.log("You can a baseball")
}
```

Read upside down for the answer!
Nothing prints
out! If no condition is met and there is no catch-all else statement, the computer will just skip over all the blocks of code and run nothing! If you want something to run if none of your conditions are met - always add an else statement!

4. Find the errors in the following code and fix them. (Hint: There is one syntax error and one logic error.)

```
var cats = 5;
var dogs = 5;
```

```
if (cats > dogs) {
      console.log("You have more cats than dogs.");
else {
      console.log("You have more dogs than cats.");
}
```

5. What does the code below output for the following inputs to the prompt?

Code:

```
var eggs = prompt("How many eggs do you have?");
var flour = prompt("How many cups of flour do you
have?");
var milk = prompt("How many cups of milk do you have?");

if (eggs >= 2  && flour >= 1.5 && milk >= 1.25) {
      console.log("You can make 12 or more pancakes.");
} else if (eggs >= 1 && flour > 0.75 && milk > .5) {
      console.log("You can make 6 to 12 pancakes.");
} else {
      console.log("You can't make any pancakes.");
}
```

Inputs:

a) 3 eggs, 5 cups of flour, 1 cup of milk
b) 0 eggs, 2 cups of flour, 2 cups of milk
c) 1 egg, 1 cup of flour, 1 cup of milk

6. Make a P5.js sketch that asks the user for 2 numbers and then checks if the first number is divisible by the second number. Print whether it is divisible or not.

UNIT THREE

Functions

Objectives
In this unit we will continue building on our knowledge of major programming structures with one of the most useful features- functions! We will learn what a function is and when it is useful, as well as learn how to declare, build, and use your own functions.

Outline

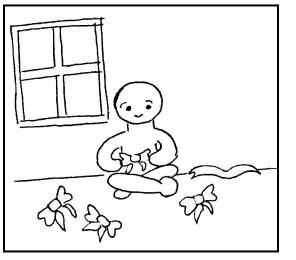
- 3.1 Function Basics
  - What Are Functions?
  - Defining a Function
  - Calling Functions
  - Function Parameters
  - Function Returns
  - Nested Functions
- 3.2 Scope
  - What is Scope?
  - Local Variables
  - Global Variables
- Review

3.1 Function Basics

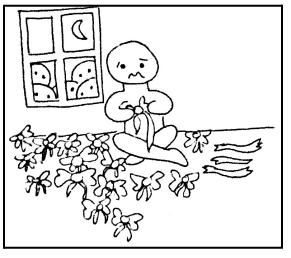
What Are Functions?

A function is a block of organized, reusable code that is used to perform a single, related action.

I'll admit - that definition isn't super helpful. Let's get into it and try to figure out what that actually means.

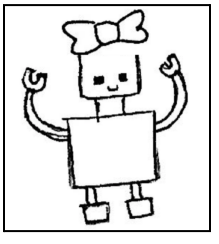


Imagine you are starting your own business. In this business, you design and make hair bows and sell them. It's easy - at first. You cut the ribbon, tie the bow, and place it in its package - over and over again.



But business starts picking up! You are getting a lot of new customers and it's starting to get overwhelming. Making the bows yourself, one by one, is taking a lot of time and space in your apartment. And it's boring - you're just doing the same thing over and over again. You need to get organized - and you need help!

And so you buy a robot that can make bows. The robot has one function (no pun intended) and every time you press the start button, your robot produces a bow. You went from having to complete three steps to one, and you can reuse the same robot to make as many bows as you want.



Think of functions like robots. They are blocks of code that are reusable, helping you do tasks simply and with less steps.

We've actually already used a lot of functions - remember those shape functions? Like ellipse(x,y,w,h), rect(x,y,w,h), etc. Without those functions, we'd have to code individual points to make up the shapes we wanted to draw. A lot of times functions will be built into libraries - so you just need to call them in order to use them. But if there's a task a function could be useful for - that isn't implicitly built into any library - you can also define one yourself.

In the next section, we'll keep exploring our hair bow business scenario and understanding the structure of functions, different types of functions, and how to do it most effectively.

Defining a Function

Before we can use a function in our code, we have to define and build it. Just like - before you can use a robot, you have to program instructions on what to do and how to do it. We call this a function definition.

The syntax of a basic function definition looks like:

```
function function_name() {

            body
}
```

The keyword function tells the computer we are building a function, and the function_name gives our function a name to be able to call it later. The body is where you would write the code to complete the task you need.

```
function my_bow_robot() {
            var bow = "~0o0~"; // creates a variable called bow
            console.log(bow); // prints our bow
}
```

Calling a Function

To use your function you need to call it. This basically means use it in the context that you now need it! We've already done this. Think of all the functions we've used so far!

Exercise: What happens when this line is executed?

```
my_bow_robot();
```

You've printed a bow and cut down a little bit of time! Now, instead of tying the bows yourself every time, you just need to tell the robot to! But is there an even better way? What if the robot could build more than one bow at a time - what if we needed it to make 3 bows?

We could just make three bows inside our function!

```
function my_bow_robot() {
            var bow = “~0o0~”;
            console.log(bow); // prints bow 1
            console.log(bow); // prints bow 2
            console.log(bow); // prints bow 3
}
```

That works - as long as you know how many bows you need before defining your function AND as long as the number of bows you need doesn’t change. But what if tomorrow you need 7 bows? And then the next day 200 bows? You could go back and update the robot’s instructions every time...but wouldn’t it be easier to just tell the same robot how many bows you need every time you call it?

By now you might have noticed our function looks like it’s missing something compared to other ones we’ve used. The parenthesis are empty, and a lot of the functions we’ve seen have information there. Some of them don’t though? What’s up with that?

Function Parameters

In between the parenthesis is where you would place **parameters** - if you need them! **Parameters** are information that you pass into the function, because you might not have it when you define it or because it might change every time. For example, we pass in an x-coordinate, a y-coordinate, width, and height every time we call the **ellipse(x,y,w,h)** function.

The syntax of a **function definition with parameters** looks like:

```
function function_name(parameter1, parameter2, …) {

            body
}
```

Let's make a bow robot function with parameters!

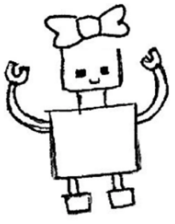
```
function my_bow_robot(num_of_bows) {

    var bow = '~0o0~';

    for (i = 0; i < num_of_bows; i++) {
        console.log(bow);
    }

}
```

This function uses a loop! We'll cover that in the next section. But basically, a loop allows us to define how many times we want a certain block of code to run - over and over again. Take a closer look and see if you can understand what's going on!
Now you can tell our robot/function to make any number of bows, and it can change every time you call it!

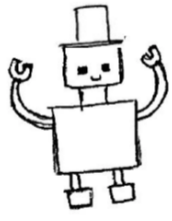
Exercise: What would be the output of this line?
my_bow_robot(6);

Parameters allow you to build less limited functions that can be used for many different things. So it's important to think about which parameters would build the most efficient, reusable function!

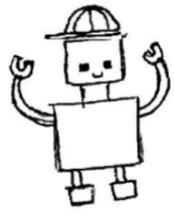
Let's look at these three functions:



```
function my_bow_robot(num_of_bows) {
  var bow = "~0o0~";
    for (i = 0; i < num_of_bows; i++) {
        console.log(bow);
    }
}
```

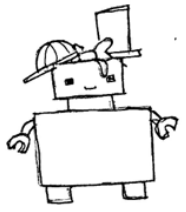


```
function my_hat_robot(num_of_hats) {
  var bow = "_II_";
    for (i = 0; i < num_of_hats; i++) {
        console.log(hat);
    }
}
```



```
function my_cap_robot(num_of_caps) {
  var cap = "__A";
    for (i = 0; i < num_of_caps; i++) {
        console.log(cap);
    }
}
```

Is there a way that you can turn these three functions into one better function by using different parameters? What if we could build a robot that could make any product - we just had to tell it what to make and how many to make? Then we'd only need one robot!



```
function my_best_robot(product, num_of_product) {
    for (i = 0; i < num_of_product; i++) {
        console.log(product);
    }
}
```

Before you build your function, think about the parameters that would be most useful. If you make your function too broad, you have too many parameters and the function body gets crowded and confusing. Make your function too specific, and you lose reusability and efficiency. It's all about balance!

Exercise: What is the output of the following code:

```
var bow = "~0o0~";
var cap = "__A";
var hat = "_II_";
my_best_robot(bow, 3);
my_best_robot(cap, 2);
```

Function Returns

Our example function that we've been using so far just prints our bow to the console. For the purpose of this example, it works! But what if we didn't want to print the bow, we just wanted the function to make some bows and then we wanted to keep the bows as a variable? How could we do that?

This is when we would create a **return** for the function. A **return value** is a value that a function returns to the calling script or function when it completes its task. That basically means the return value is the value that the function produces and can be used outside of the function. It's what the function equals after it is called!

I'll admit, it's a bit confusing. Let's look at an example.

```
function my_bow_robot() {
        var bow = '~0o0~'; // creates a variable called bow
        return bow; // returns a bow
}

var my_product = my_bow_robot();
// my_product now equals "~0o0~"
// nothing is printed to the console but now we can use my_product as a variable however we need!

console.log('Welcome to my shop! Today we are selling ' + my_product);
```

You can return any type of variable - numbers, strings, booleans, objects, etc. Just make sure to keep track of your return type, so that you use the values correctly later. It is not required to return anything from a function, but it can be useful when you need to access the result of your function later.

Exercise: Look at the functions below. What is the return value?

```
function fullName(first, last) {
    var full = first + last;
    return full;
}

function addition(num1, num2) {
    sum = num1 + num2;
    return sum;
}
```

Good Coding Practice: Function Comments

It's good coding practice (and just an all around good idea) to always add comments to your functions! Function comments detail what the function does, what the parameters are, and what the returns are. This helps you - or anyone else who might read your code - keep track of what functions do and how to use them, without having to read through every line of code! Below is a standard function comment template:

```
// function name: what the function does
// parameters: param name: description of parameter
// return: what the function returns
```

Nested Functions

A **nested function** is a function that you call within the definition of another function. Essentially it is when one function uses another function. Nested functions work a lot like nested loops!

Let's look at an example.

```
function vennDiagram(x, y, w, h) {

    noFill() //no fill so that we can see the overlapping part!

    var xToTheLeft = x - 10; // the center of the diagram is at (x,y)
    ellipse(xToTheLeft, y, w, h); // which means we have to move the
                                  // left circle a little to the left

    var xToTheRight = x + 10; // & right circle a bit to the right
    ellipse(xToTheRight, y, w, h);
}
```

We called **noFill()** and **ellipse()** within our **vennDiagram()** function definition. But these are library functions - what if we need to call one of our own functions? Do we define them inside the function? Outside the function?

You never want to *define* a function within another function definition. Define the inner function first, and then *call* the inner function within the definition of the outer function. In Javascript, you can actually call a function before its definition, but it is good practice to define it before you need to use it elsewhere.

```
function shiftX (x, L_or_R) {

    if (L_or_R === 'L') {
        return x - 10;
    } else {
        return x + 10;
    }
}

function vennDiagram(x, y, w, h) {
    noFill() // no fill so that we can see the overlapping part!
    var xLeft = shiftX(x, 'L'); // call our shiftX function!
    ellipse(xLeft, y, w, h);

    var xRight = shiftX(x, 'R');
    ellipse(xRight, y, w, h);
}
```

3.2 Scope

What is Scope?

Scope is the area of the program where an item (be it variable, constant, function, etc.) that has an identifier name is recognized.

These formal definitions are never super helpful, are they?
Let's think about scope in terms of Google Sharing.
You probably have a Google Drive that has your documents, pictures and emails. You can access all of those things because they live on your drive. Your friend also has a drive, but they can't access your information from their drive and you can't access their information from theirs.

But! You *can* share certain information to your friend's drive from your drive, so that they can see it or even edit it. They still don't have access to everything on your drive, but now they have access to a certain piece of information.

This is kind of how scope works. If a variable or function lies inside a certain scope, anything outside of that scope can't access it. There are ways to send certain information outside a scope - just like sharing a funny picture with your friend!

Let's look at an example:

```
1 function myBank() {
2     var savings = 1000;
3 }
4 savings += 200; // what happens if you try to add $200 to your
5                 // savings  outside this function?
```

You get an error: Uncaught ReferenceError: savings is not defined

What? But we did define savings! Right there in line 2.

But the variable savings was created inside the function definition of **myBank**. Outside of the function, it's like the variable doesn't exist. That means we can only access the variable savings inside the function **myBank**. In order to understand how to fix this, we first need to understand the difference between local and global scope.

Local Scope

Variables that have a limited scope, existing in only the block it is defined in, are said to have **local scope**. Think about it like a google doc that isn't shared with anyone. All the information written on that document only exists in that document.

Local variables are variables with local scope. Variables that are defined inside functions are always local variables. You can only call and manipulate these variables within a function's definition.

Global Scope

Variables without a limited scope, that are defined only once in the script and exist to be used by any function are said to have **global scope.** This is like a google doc that is shared with everyone! They can view, edit, and share the information with other people too.

Global variables are great for constants, or values that don't change over time, but a lot of times they can cause problems. Remember it's like a public google doc - so if your friend goes ahead and edits something, but you needed the information that was there before he edited it - you're out of luck. If different functions need a variable for different things, it's better to use local variables.

In this first code, savings is a local variable. It was created inside the function myBank.

```
function myBank() {
    var savings = 1000;
    var checking = 500;
}
```

In this next code, savings is a global variable. It was defined outside a function and any function can use it.

```
var savings = 1000;
var checking = 500;

function myBank() {
    console.log(savings)
    console.log(checking)
}
```

Unit 4 Review

New Vocabulary

Functions: a block of organized, reusable code that is used to perform a single, related action.

Function Parameters: information that you pass into the function

Function Returns: a value that a function returns to the calling script or function when it completes its task

Nested Functions: a function that calls another function within it's body

Scope: area of the program where an item (such as a variable, constant, function, etc.) that has an identifier name is recognized.

Local Scope: limited scope, existing in only the block it is defined in

Global Scope: without a limited scope, exist to be used by any function

New Code Structure

Structure for a function definition:

```
function func_name(parameters) {
      function body;
      return value;
}
```

Extra Help and Resources

Test Yourself

1. Why do we use functions?
2. What is the difference between a function definition and a function call?
3. What are function parameters?
4. What is a function return value?
5. Does a function need to return something?
6. What is the difference between local and global scope?

Practice Problems

1. Write a function called bakingConverter() that converts tbsp into cups, and cups into tbsp. Once you've done that, try to add some other conversions!
2. Write a function called calculator() that takes in two numbers and an operator to represent add +, subtract -, multiply *, and divide /. The function should calculate the expression and return the result!
3. Write a function that takes in a number representing an amount of money in dollars and a number representing the interest rate. The function should calculate and return the new amount of money after the interest rate has been applied.
4. Find the errors in the code below:

```
var sale = .15;
var prices = [12, 22, 18, 93, 35, 5, 19, 45];

function calculateDiscount(initial_price, discount) {
      var newPrice = initial_price * discount;
      console.log(newPrice);
}

discounted_price = calculateDiscount(prices[2], sale);
console.log("This item costs: " + new_price);
```

UNIT FOUR

Loops

Objectives

In this unit we are finally delving into one of the most useful programming components: loops! We will learn why and how to use loops, the different types of loops, and how to walk through a crazy nested loop.

Outline

4.1 Loop Basics

What are loops?

A **loop** is a sequence of instructions or a section of code that is repeated iteratively until a certain condition is met. The condition might be the number of times you need the loop to run or something that triggers when to stop running.
A loop is like a race car track - you run the track over and over again until you've completed the number of laps in the race or... until you run out of gas.
So why use a loop? A lot of times we have some code that we want to do over and over again, or that we want to do iteratively! For example, maybe you want to print out the numbers 1 - 10. You could just write console.log() out ten times...

```
console.log(1);
console.log(2);
console.log(3); … etc, etc.
```

But that takes a lot of time - and you 're kind of just doing the same thing over and over again. And what if you want to print out the numbers 1-150? There has to be a better way than to write out console.log() 150 times! That's where loops are helpful! They allow you to organize your code better and reduce the amount of code you need to write and work with.

There are two types of loops: **for-loops** and **while-loops**. They're not really two types, just two different ways to write loops.
Whatever you can do with a for-loop, you can do with a while-loop. Sometimes one method or the other might make more sense to you intuitively.

NOTE! Beware of **infinite loops** - loops that have no exit strategy and will run forever! Imagine a race where every car has unlimited gas, but there isn't a set number of laps to win. The race would never end! After reading the next two sections, think about how an infinite loop can be created and how you can avoid that problem!

There is one infinite loop that you're already using though! You might have guessed it - but the p5 draw function is actually a loop! It starts looping when you run the code and stops looping only when you stop the execution completely. The draw loop being an infinite loop allows us to animate our drawings!

You can stop the draw function from looping by calling the function noLoop() in setup. This will make it so that draw only runs once - at execution! After noLoop() stops the code in draw() from executing, redraw() causes the code inside draw() to execute once more, and loop() will cause the code inside draw() to resume looping!

For - Loops

The structure of a for loop looks like this:

```
for (start; condition; iteration)
        loop body
}
```

Start is your starting point. It is executed once before the loop starts running, and usually indicates where your iteration starts. Are you starting at 1? At 50? At 150?

It's good practice to start a loop at 0, unless there's a specific number it needs to start at. Many other data types and structures in programming start at 0, and so it's easiest to stay consistent!

Condition is the condition that needs to be met in order for the loop to keep running. This might be a limit of the number of times you want the loop to run, or the number you need stay below before the loop stops! When this condition is no longer met, the loop stops and the code goes on to the next block.

Iteration is executed each time the loop runs. It keeps track of where you are in comparison to your start and goal and increases based on this expression. Are you moving towards your goal 1 at a time, 2 at a time, 3 at a time?

Doesn't really make sense yet? Don't worry. Take a look at our example!

```
1  // this for loop counts the number of laps completed and
2  // outputs a message after completing every lap!
3  for (var lapNumber = 0; lapNumber < 4; lapNumber++) {
4          console.log('You just finished lap ' + lapNumber);
5  }
```

Let's iterate through the loop:

The first time the loop runs:

lapNumber = 0; // this is where we told it to start
labNumber < 4; // our condition is still met!
console.log(...); // we print that we just finished lap #0
lapNumber++; // we add one to lapNumber, now lapNumber = 1

The second time the loop runs:

lapNumber = 1; // the loop starts with lapNumber = 1
lapNumber < 4; // our condition is still met!
console.log(...); // we print that we just finished lap #1
lapNumber++; // we add one to lapNumber, now lapNumber = 2

The third time the loop runs:

lapNumber = 2; // the loop starts with lapNumber = 2
lapNumber < 4; // our condition is still met!
console.log(...); // we print that we just finished lap #2
lapNumber++; // we add one to lapNumber, now lapNumber = 3

The fourth time the loop runs:

lapNumber = 3; // the loop starts with lapNumber = 3
lapNumber < 4; // our condition is still met!
console.log(...); // we print that we just finished lap #3
lapNumber++; // we add one to lapNumber, now lapNumber = 4

The fifth time the loop runs:

lapNumber = 4; // the loop starts with lapNumber = 4
lapNumber < 4; // STOP our condition is no longer met - 4 is not LESS than 4
~~console.log(...);~~ // we skip over our loop body
~~lapNumber++;~~ // we break out of the loop, and continue with the rest of the code

We COULD have written out all this code and gotten the same result! But the loop allows us to condense line after line of code, into one simple block. Whenever you notice that you have to iteratively complete a task, see if a loop would work!

While Loops

Remember while loops are just for loops, written a little differently!

The structure of a while loop looks like this:

```
start
while (condition) {

      loop body

      iteration
}
```

Remember:

Start is your starting point. It is executed once before the loop starts running, and usually indicates where your iteration starts.

Condition is the condition that needs to be met in order for the loop to keep running. This might be a limit of the number of times you want the loop to run, or the number you need stay below before the loop stops! When this condition is no longer met, the loop stops and the code goes on to the next block.

Loop body refers to whatever you want to do within the loop. This runs as long as the condition defined above is met. Within the loop body, you may also make changes that will affect whether the condition will be met on the next iteration of the loop.

Iteration is executed each time the loop runs. It keeps track of where you are in comparison to your start and goal and increases based on this expression. Are you moving towards your goal 1 at a time, 2 at a time, 3 at a time?

The same for loop we wrote above can be written as a while loop!

```
1  var lapNumber = 0;
2
3  while (lapNumber < 4) {
4      console.log('You just finished lap ' + lapNumber);
5      lapNumber++;
6  }
```

4.2 Nested Loops

Nested loops are simple in theory! They are just a loop inside a different loop. But they can get a little complicated to think about.

In a nested loop, the inner loop runs completely through each time the outer loop runs once. Let's walk through a simple nested loop:

```
1 for (var outer = 0; outer < 2; outer++) {
2
3   for (var inner = 0; inner < 2; inner++) {
4         console.log('Inner! \n');
5     }
6
7     console.log('Outer! \n');
8 }
```

The first time the outer loop runs:

```
var outer = 0; // start at 0
outer < 2; // condition met, 0 is less than 2
```

the inner loop runs entirely through!

the first time the inner loop runs:

```
var inner = 0; // start at 0
inner < 2;  // condition met
console.log("Inner! \n"); // output Inner!
inner++;  // increment to next step in inner loop,
```

```
inner now = 1
```

the second time the inner loop runs:

```
var inner = 1; // inner = 1
inner < 1;  // condition still met
console.log("Inner! \n"); // output Inner!
inner++;  // increment to next step in inner loop,
```

```
inner now = 2
      the third time the inner loop runs:
      var inner = 2; // inner = 2
      inner <2; // condition not met, 2 isn't less than 2
      console.log("Inner! \n"); //no output
      inner++;  // no increment, break out of the loop

console.log("Outer! \n"); // output "Outer!"
outer++; //increment to next step in outer loop, outer
now = 1
```

The second time the outer loop runs:

```
var outer = 1; // outer = 1
outer < 2; // condition met, 0 is less than 2
the inner loop runs entirely through!

      the first time the inner loop runs:
      var inner = 0; // start at 0
      inner < 2;  // condition met
      console.log("Inner! \n"); // output Inner!
      inner++;  // increment to next step in inner loop,
                // inner now = 1

      the second time the inner loop runs:
      var inner = 1; // inner = 1
      inner < 1;  // condition still met
      console.log("Inner! \n"); // output Inner!
      inner++;  // increment to next step in inner loop,
                // inner now = 2

      the third time the inner loop runs:
      var inner = 2; // inner = 2
      inner <2; // condition not met, 2 isn't less than 2
      console.log("Inner! \n"); //no output
      inner++;  // no increment, break out of the loop
```

```
console.log(“Outer! \n”); // output “Outer!”
outer++; //increment to next step in outer loop, outer
         // now = 2
```

The third time the outer loop runs:

```
var outer = 2; // outer = 2
outer < 2; // condition not met, 2 is not less than  2
~~the inner loop runs entirely through!~~ // break, do not
run loop body
~~console.log(“Outer! \n”);~~ // break
~~outer++;~~ // break
```

The output of the nested loop will look like:

```
Input!
Input!
Output!
Input!
Input!
Output!
```

Unit 4 Review

New Vocabulary

Loops: a method to iteratively execute repeated blocks of code

Infinite Loops: a loop that will never end because it has no exit

2 Types of Loops: for-loops and while-loops

Nested Loops: a loop that has another loop within its body

New Code Structure
Structure for a for loop:

```
for (var counter = start_value; condition; increment/decrement
counter) {
      loop body;
}
```

Structure for a while loop:

```
var counter = start_value;
while (condition) {
      loop body;

      increment/decrement counter;
}
```

Test Yourself

1. True or False. Anytime you can use a for loop, you can also use a while loop.
2. What is an infinite loop?
3. What is a nested loop? Explain how nested loops work.

Practice Problems

1. Write a for loop that will print out "HI!" five times.
2. Write a while loop that will print the numbers 10-20, in order.
3. Write a for or while loop that will print out all the ODD numbers from 1-100, in order.
4. Write a for or while loop that will print out all the EVEN numbers from 2-100, starting from 100.
5. The following code is supposed to print out the square of the numbers between 1 to 10. However, there are errors in the code. Find the errors and fix them. (Hint: There are two logic errors and one syntax error.)

```
for (var counter = 0; counter < 10, counter++) {
      var square = counter * counter;
      console.log(counter);
}
```

6. Write a for loop that will calculate the factorial of a number inputted by the user. Print out the result. After you finish, write this as a while loop as well.
7. Look at the code below. How many times does the outer loop run? How many times does the inner loop run?

```
for (var i = 0; i < 6; i++) {
       for (var j = 0; j < 10; j++) {
              console.log(“Hi.”);
       }
}
```

8. Look at the code below. How many times does the outer loop run? How many times does the inner loop run?

```
for (var i = 0; i < 8; i++) {
       for (var j = i; j < 10; j++) {
              console.log(“Hi.”);
       }
}
```

9. The following code produces an infinite loop. Fix it so it only runs 10 times.

```
for (var counter = 0; counter < counter+1; counter++) {
        console.log(counter);
}
```

10. Write a nested loop that will produce the following output. You may use for and/or while loops.

    Expected Output:

    ```
    *
    **
    ***
    ****
    *****
    ******
    *******
    ```

UNIT FIVE

Arrays

Objectives
This unit is all about arrays! We will be looking to understand another data type called arrays - that allow us to group and work with a lot of items at once! By the end of this unit, you will understand what an array is, how to define it, how to use it, and how to use some of the other structures we've learned in combination with arrays.

Outline

5.1 Array Basics

- What is an array?
- Defining Arrays

5.2 Indexing

- How to Index an Array
- Adding/Deleting Elements
- Indexing Errors

5.3 Arrays and Loops

Review

5.1 Array Basics

What is an array?

We've talked about a few different data types so far - numbers, strings, booleans. Arrays are another data type - and they might be one of the most important ones across programming languages.

Arrays are an ordered collection of items. Pretty simple, right? But what does this actually mean?

It might help to think of arrays as a music playlist. Arrays are the playlist and the items in the array are the songs. Before we jump into the programming side, let's think a little bit more about how playlists work.

Your playlist is just a collection of songs, but there's a certain order to them. If you start listening to song number 1, and just let the playlist play, you would run through each song in the order that you added them to your playlist - or the order that you specified. But, if you wanted to just listen and access song #3, you can do that - as long as you have Spotify premium! You can also add or remove songs, even if it's been forever since you originally made the playlist!

And why do we use playlists in the first place? We use playlists to group together similar songs, or songs that we want to listen to together, so they are easier to access later. It's much easier to create a playlist of your favorite songs and let it play, than individually queue up each song when you want to listen to music.

All these features we've mentioned so far are analogous to how arrays work and why we use them. Except for one not-so-little thing!

Arrays start counting from 0! Imagine that music playlist again, and add your favorite 5 songs to it. Now, imagine that instead of saying that the first song is song 1, we just start counting at 0. So the first song is song 0, the second song is song 1, the third song is song 2, etc, etc. But the actual number of songs doesn't change, we just count them from 0-4 instead of from 1-5. Confused yet? I was too. Look at the playlists below - same songs, same order - just counted differently!

1 Watermelon Sugar
2 Falling
3 Savage - Remix
4 Blinding Lights
5 Say So

0 Watermelon Sugar
1 Falling
2 Savage - Remix
3 Blinding Lights
4 Say So

Defining Arrays

Okay playlists are great and all, but you're probably wondering how this actually works in your code!

Like every other data type, you have to declare an array before you can use it. When you declare it, you don't have to immediately define it and fill your array in with items. You can just say, here's an array, I'm planning on using it later! This is like creating a playlist, but not actually filling it with songs right away!

This is the syntax to declare an empty array:

```
var array_name = [];
```

When you want to define an array, you insert a list of items you are adding to the array into the square brackets.

```
array_name = [item1, item2, item3];
```

And, like other data types, you can declare and define arrays in the same line:

```
var array_name = [item1, item2, item3];
```

Let's make an array that holds the songs in the playlist we used as an example!

```
var myPlaylist = ["Watermelon Sugar", "Falling",
"Savage-Remix", "Blinding Lights", "Say So"];
```

// This particular array holds strings - but you can make arrays that hold anything!

```
// numerical array
var homeworkGrades = [100, 98, 85, 93, 97];

// boolean array
var trueFalseAnswers = [true, false, false, true];
```

In JavaScript, arrays can even hold things that are different data types! Other programming languages don't make it that easy for us - but luckily JavaScript does.

```
// array that holds strings, numbers, and boolean
var interviewAnswers = [“Samantha”, 24, true];
```

5.2 Indexing

How to Index an Array

Indexing is just another way of saying the method we have to count, keep track of, and access the items in our array.

Remember - JavaScript is zero-indexed, which means we Start! From! 0! I promise that idea will become more natural the more you use arrays.

To access a specific element in an array we use square brackets again and enter the index of the item you want. The syntax looks like this:

```
array_name[index];
```

Let's look at that playlist array again.

```
var myPlaylist = ["Watermelon Sugar", "Falling",
"Savage-Remix", "Blinding Lights", "Say So"];
```

If I wanted to grab the first song of the array and store it variable myFavoriteSongEver, how would I do that?

```
// "Watermelon Sugar" is the first song, but it's at index 0
var myFavoriteSongEver = myPlaylist[0];
```

Now, let's print out the value of myFavoriteSongEver:

```
// Printing out myFavoriteSongEver
console.log(myFavoriteSongEver);
```

You would see the following output:

```
Watermelon Sugar
```

It might help to write out your items with their index number like this!
At least at the beginning, when you're still getting used to counting from 0.

0 Watermelon Sugar
1 Falling
2 Savage - Remix
3 Blinding Lights
4 Say So

Exercises:
Which song is at index 1?
Which song is at index 2?
Which song is at index 4?

How many total songs are there in the array? Count carefully!

Adding/Deleting Elements to/from an Array

Javascript makes our life easy, giving us a few simple ways to work with arrays. You can add, replace, and delete elements. Let's start with adding elements.

Adding an Element
There are two different ways to add an element to the end of an array. The first one is to just assign your new element at the next index of the array using the square brackets.

```
var myPlaylist = [“Watermelon Sugar”, “Falling”,
“Savage-Remix”, “Blinding Lights”, “Say So”];
myPlaylist[5] = “Magic in the Hamptons”;

Now myPlaylist looks like :
[“Watermelon Sugar”, “Falling”, “Savage-Remix”, “Blinding
Lights”, “Say So”, “Magic in the Hamptons”];
```

This method seems easy! But it can get tricky fast. You have to be REALLY careful about your index... Imagine if you reassign an index that was already being used - let's say you messed up and wrote:

```
myPlaylist[4] = “Magic in the Hamptons”;
```

Now myPlaylist looks like: `[“Watermelon Sugar”, “Falling”, “Savage-Remix”, “Blinding Lights”, “Magic in the Hamptons”]; //looks like Say So is missing now!`

You'll end up losing data and messing the array up. So unless you actually want to reassign elements in the array, I would use the second method: the **push()** function!

```
array_name.push(“New Element”);
```

The push function is nice because you don't have to worry about indexing at all! It always just adds your element to the end of the array - no matter how long the array is.

```
// this push function adds Magic in the Hamptons to the end
myPlaylist.push("Magic in the Hamptons");
```

What if we want to insert an element somewhere else - maybe at the front? To add an element to the front we can use a similar function to push(), called **unshift().**

```
array_name.unshift("New Element");
```

Back to our playlist example! Let's add a song to the beginning!

```
myPlaylist = ["Watermelon Sugar", "Falling", "Savage-Remix",
"Blinding Lights", "Say So", "Magic in the Hamptons"];

myPlaylist.unshift("Break My Heart");
```

Now myPlaylist looks like: `["Break My Heart", "Watermelon Sugar", "Falling", "Savage-Remix", "Blinding Lights", "Say So", "Magic in the Hamptons"];`

What if I want to add an element in the middle of the array? That gets a little trickier and will make more sense if you understand how to delete an element first! We'll come back to it!

Deleting an Element

To delete an element off the end of an array, there are also two methods! The first is the push() functions opposite—the pop() function.

```
array_name.pop();
```

That little line removes the last element in the array and also returns this element! Pretty easy right?

Let's keep working with myPlaylist!

```
myPlaylist = [“Break My Heart”, “Watermelon Sugar”, “Falling”,
“Savage-Remix”, “Blinding Lights”, “Say So”, “Magic in the
Hamptons”];

var deletedSong = myPlaylist.pop();
```

Now myPlaylist looks like:
`[“Break My Heart”, “Watermelon Sugar”, “Falling”, “Savage-Remix”, “Blinding Lights”, “Say So”];`

And, deletedSong is equal to:
`“Magic in the Hamptons”`

What if you want to delete multiple elements from an array? We could call **pop()** multiple times, or even write a loop that calls **pop()**, but that could take a lot of time and lines of code. Javascript actually has a way that makes it really easy - we just change the length of the array to only hold the items we want!

The only thing you have to remember here is that the length and the index of the last element are different numbers! Remember, even though the array is indexed from 0-4 the length is actually 5!

To do this we use the length variable and reassign its value! The length variable is a property of the Array class in Javascript, which just means all arrays you define will have this property. Don’t worry if you don’t understand what this means yet, it will become clear when we learn about classes and their properties.

```
myPlaylist = [“Break My Heart”, “Watermelon Sugar”, “Falling”,
“Savage-Remix”, “Blinding Lights”, “Say So”]; // length = 6
// Modifying the length of myPlaylist
myPlaylist.length = 4;
```

Now myPlaylist looks like: `[“Break My Heart”, “Watermelon Sugar”, “Falling”, “Savage-Remix”];`
`// now length = 4, the last two elements are chopped`

Okay, cool... but what if I want to delete an element from the front of the array? What then?

We have a function for that too! The shift() function works just like the pop function, it just removes the element from the front instead!

```
myPlaylist = [“Break My Heart”, “Watermelon Sugar”, “Falling”, “Savage-Remix”];
myPlaylist.shift();
```

Now myPlaylist looks like: `[“Watermelon Sugar”, “Falling”, “Savage-Remix”];`

We’re done right? But wait...what if we want to delete an element from the middle of the array? There’s a function for that too :)

The splice function lets us delete one or more consequent elements from anywhere in the array, and then shifts all the elements accordingly. It will also change the length to reflect the updated array!

```
array_name.splice(index_to_start_at, numbers_of_elements_to_delete, optional parameters)
```

The first number is the index that you want to start deleting at, and the second number is how many numbers you want to delete from that index. It’s not the range that you want to delete! We will come back to the optional parameters at a later time.

For example, if I want to delete elements at index 3 and 4, I would call the function like:

```
array_name.splice(3, 2);
```

We want the function to start counting at 3, and we want 2 total elements to be deleted. DO NOT, DO NOT, DO NOT write the range of what you want deleted! If you write array_name.splice(3, 4); elements 3-6 will be deleted!

Let’s look at our playlist example. I want to delete “Falling”, and “Savage-Remix”.

```
myPlaylist = [“Watermelon Sugar”, “Falling”, “Savage-Remix”];
myPlaylist.splice(1, 2);
```

Now myPlaylist looks like: `[“Watermelon Sugar”];`

Remember how I said adding elements to the middle of an array would make more sense after we understand how to delete elements? We can actually use splice() to add elements too! Take a look at the next section!

Adding an Element to the Middle of an Array Using Splice

The third optional parameter for the function splice() details an element that you want to insert in the space of the elements that you are deleting. I wouldn't recommend using it for replacements - it's much easier to just reassign values. But it is the easiest way to place elements in the middle of an array.

To insert elements, without deleting any, we just need to insert 0 as the second parameter (the number of elements you want to delete). To insert using splice(), it looks like this:

```
array_name.splice(index_to_start_at, 0, "New Element");
```

This function will insert the new element right after the index that you wanted it to start at, and shift all the other elements over to make room. For example, let's say I want to insert "Say-So" back in, but I want it at index 1, right after "Watermelon Sugar".

First let's add some of our elements back into our playlist.

```
myPlaylist = ["Watermelon Sugar"];
myPlaylist.push("Falling"); // adds to end
myPlaylist.push("Savage-Remix"); // adds to end
```

Now myPlaylist looks like:

```
 ["Watermelon Sugar", "Falling", "Savage-Remix"];

myPlaylist.splice(1, 0, "Say-So");
```

Now myPlaylist looks like:

```
["Watermelon Sugar", "Say-So", "Falling", "Savage-Remix"];
```

You can also add multiple elements to the array at time using the splice() function. Here is an example:

Let's add some elements to the middle of the array after "Falling.:

```
myPlaylist.splice(3, 0, "Save Your Tears", "Heat Waves");
```

Now, myPlaylist looks like:

```
[“Watermelon Sugar”, “Say-So”, “Falling”, “Save Your Tears”,
“Heat Waves”,  “Savage-Remix”];
```

5.3 Arrays and Loops

One of the most efficient ways to work with arrays is with loops. We can loop through arrays using their indexes. Remember - arrays start at 0! So our loops have to start at 0 too!
Here's the structure of a for loop that loops through an array:

```
for (var index = 0; condition; index++) {

      the code you want done to the elements
}
```

Example:

```
for (var index = 0; index < myArray.length; index++) {
      console.log(myArray[index]); // this prints each element
}
```

Most times in loops, we use the variable i to indicate index. If you look at any sort of references, for loops might look more like this:

```
for (var i = 0; i < myArray.length; i++) {

      console.log(myArray[i]);
}
```

And, like we've said before! Whatever you can do with a for loop, you can do with a while loop! A while loop looping through an array might look like this!

```
var i = 0;
while (i < myArray.length) {
      console.log(myArray[i]);
      i++;
}
```

Unit 5 Review

New Vocabulary

Array: ordered collection of items

Index: numerical representation of an item's position in an array

Zero-Index: start the index at 0

New Structures/Functions

```
// declaring an array
arr = [];

// accessing an item at index i
arr[i];

// returns the length of the array - not the last index!
arr_name.length();

// adds element to end
arr_name.push(“New Element”);

// removes element from end
arr_name.pop();

// adds element to front
arr_name.unshift(“New El.”);

// removes element from front
arr_name.shift();

// deletes element from certain index of the array
arr_name.splice(index_to_start, num_of_elements_to_delete);

// inserts element at certain index of the array
arr_name.splice(start_index, 0, “new element”);
```

Test Yourself

1. What is an array?
2. Why do we use arrays?
3. Come up with your own analogy for arrays and items other than the one discussed.
4. What is an index?
5. What does it mean to be zero-indexed?
6. How do you add an element to the end of an array? To the front? To a middle index?
7. How do you remove an element from the end of an array? From the front? From a middle index?
8. How do you find the length of an array?
9. What is the difference between the length and the last index?
10. How do you loop through the elements in an array using a for or while loop?

Practice Problems

1. Create an array that holds all the classes you're taking this year in the form of strings.
2. Print to the console the class you have 3rd by indexing your array.
3. Remove the class you have last.
4. Remove the class you have 1st.
5. Remove the class you have 4th.
6. Add a class you want to take to the end of your schedule.
7. Add the class you have 4th back to the array and a study period right after. (They should be the 4th and 5th elements in the array.)
8. Use a loop to print your whole schedule in order.

UNIT SIX

Objects

Objectives

In this unit, we will get an introduction to objects, how to create them, how to use them, and the principles behind Object Oriented Programming. By the end of this unit, you will understand how to make a new object, how to write a class, and how to manipulate objects in your code.

Outline

6.1 Object Basics

What is an object? You tell me! We all know what an object is in real life - things like balls, pens, books, and chairs. We naturally separate similar things into categories based on functionality, appearance, and general characteristics. And this is essentially what an object in programming is too!

Real life objects have states and behaviors. The **state** is what the object looks like, the features it has, the current state it is in. The **behaviors** are things that the object can do, or functions that it has. Think of an object as a noun, states as adjectives that describe the noun, and behaviors as verbs a noun can do!

Let's think of a dog as an object. A dog's state might include its name, its fur-color, its breed, if it's hungry or not. A dog's behavior would be things like barking, eating, running, etc.

In programming, objects also have states and behaviors - which we represent as variables and functions.
Object-Oriented Programming is a type of programming that uses objects and their states/behaviors to define problems, rather than overarching functions and structures. Object-Oriented Programming might sound scary - but it's really the way we operate in the world already!

Principles of Object Oriented Programming

There are four main principles of Object Oriented Programming - encapsulation, abstraction, inheritance, and polymorphism.

Encapsulation is the principle that each object's state should be private to the object, meaning objects shouldn't be able to access other objects' states. You can remember this idea of encapsulation by thinking about each object as its own little capsule, keeping all of its information safe. The capsules can talk to each other, and might give specific access to certain information or functions, but you want to keep your own information safe and sound.

Abstraction is the principle that objects should only give access to parts of their private state that are relevant for other object's use. This is to make things simple! Another object doesn't need to be poking around under the hood - they might mess things up! They just need some sort of way to interact with the information. For example, you don't know exactly how

everything inside your phone works (or maybe you do - but I definitely don't!), but you can interact with it through buttons! Same idea!

Inheritance is the solution to one of the core problems that OOP runs into - a lot of objects are similar, they share common logic or constants, but they're not *exactly* the same. Do we really have to build a whole other object every time? Simply - no. We can create parent and child objects, which allows similar objects to share certain information. The child object can reuse all the fields and methods from its parent, but can also implement its own state and behavior. For example, we might create a parent object called Person. Person holds information like name, height, gender, etc. Then we might create a child object Doctor. Because Doctor is the child of Person, it can access all of Person's information, and it doesn't need to repeat all that information again, but it might also hold specialty, hospital, med school. Take a look at the diagram below!

Polymorphism means that objects can take on more than one form depending on the context. It doesn't make too much sense until you get into the details of inheritance...so we'll move on for now! Once you've finished this unit and understand more about objects, it might be worth coming back to this topic - especially if you're interested in an object oriented language such as Java, C++, or C#!
Okay, I won't keep you waiting any longer. Let's make some objects!

6.2 Classes

What is a Class?
A **class** is a template, or maybe a better word is blueprint, for your object. Classes are how we define objects so we can go ahead and use them later.

Think about it like a cookie cutter and a cookie - a class is the cookie cutter and the actual object is the cookie!

Remember that idea of encapsulation? We want to take everything it means to be an object and pack it all away to keep it safe! This means variables, functions, data types, and anything else you need for your object.

In order to define a class we need a **class declaration.** We use the keyword class and the initial properties are assigned a **constructor()**. Let's look at the basic structure of a class declaration:

```
class Person { // note: no need for parenthesis here!
      constructor() { // defines the "state"
            .
            .
            .
      }
      class_functions() { //defines the "behavior"
            .
            .
            .
      }

}
```

And one more thing to note! This will make more sense later, but it's so hard to remember - so we might as well start drilling it in now. Whenever you declare a variable, or even use a variable within your class definition, you have to use the **this** keyword. It looks something like:

```
this.variable_name = "i belong to your object!".
```

Notice the **dot notation**! The keyword and the variable name are separated by a . not a space.

This is because you want to assign that variable to the class and to the class only. Remember we talked about keeping our information safe!! Again, we'll go more in depth later, but start keeping this in mind. Pun intended :-)

Constructors

Okay, time to dig deep. I just threw a bunch of words around - classes, constructors, class functions, class variables. But what do they actually mean? Let's start with constructors.

Constructors are essentially the instructions to actually make the object initially. This is where we put all the information about what makes an object THAT object. Remember when we talked about state and behaviors? Constructors define the initial state. Usually, this is in the form of defining certain initial variables that are attached to the object.

And remember! Always use the **this.** notation when working with variables inside a class!

```
class Samantha {
    constructor() { // defines the "state"

        this.name = 'Samantha';
        this.age = 24;
        this.job = 'engineer';

    }
}
```

We made a Samantha! Every time you create or use an object Samantha, the computer will look at the constructor and say, "How do we make a Samantha? Well first we name her Samantha, then we make her 24 years old, and then we give her a job as an engineer." It's almost like we're playing the Sims.

But there's a problem - we can only make Samanthas. What if I want to make a Joe object? Do I have to make a whole other class? Not necessarily.

Remember when we talked about function parameters and how we can use parameters to make more abstract, efficient, and reusable functions? We can use class parameters to do the same thing!

```
class Person {
    constructor(name, age, job) {

        this.name = name;
        this.age = age;
        this.job = job;

    }
}
```

Note - when you pass parameters into the class, we are actually passing them into the constructor. So in the class definition, we place the parameters into the constructor parenthesis! Remember - classes don't have parentheses when you define them!

Now we can make objects that can be 24 year old Samanthas or 72 year old Joes! The most important thing to think about when writing constructors is to fully answer the question: What makes this object *this* object? Think about how to best categorize your objects to make your classes as reusable and useful as possible!

Class Functions

Okay, now we know how to define the state of an object. But there were two parts- state and behavior. How do we define the behavior of an object? What if it's Samantha's birthday and she's aging a year?
This is where we use **class functions** - or functions defined within a class that detail an object's behaviors.

They look and work like normal functions! Sometimes these are called **member functions**!

```
class Person {
    constructor(name, age, job) {

        this.name = name; // assigns name passed in to this.name
        this.age = age; // assigns age passed in to this.age
        this.job = job; // assigns job passed in to this.job
    }

    birthday() { // don't use the function keyword inside a class!
        this.age += 1;
        console.log('Today is my birthday! I turn ' + this.age);
    }

}
```

Notice I don't need to pass in the age parameter, this is because the function is part of the class and already has access to the object state as defined within the constructor.

So if I make a new object, and then call the birthday function, our birthday message will print to the console!

```
var Samantha = new Person("Samantha", 24, "engineer");
Samantha.birthday(); // prints "Today is my birthday! I turn 25
```

Hold up... that syntax for our function looks different... Confused? The next section is all about how we actually use objects in our code!

6.3 Using Objects

Object Instances

An **object instance** is just a fancy way of saying a new object! Think of it like making a new cookie with your cookie cutter! Creating a new object is pretty easy - but it isn't exactly the same as creating a new variable or function.

To create a new object - or instance - you need to declare a variable first, and then assign your object to that variable!

Let's look at that example again:

We declare a variable Samantha and then create and assign a Person object to this variable. The variable Samantha is now a Person object with the characteristics we passed in.

```
var Samantha = new Person("Samantha", 24, "engineer");
```

But notice there's a new keyword in there! Whenever you create a new object you need to use the **new** keyword.

Dot Notation

Using class functions looks a little different too.
Similarly to how we use dot notation for the this keyword, we have to use dot notation for class functions to indicate which object's function we are calling.

Specifically, you should separate the name of the object and the name of the function you are calling with a dot.

```
objectName.function_name();
```

Back to our example:

```
Samantha.birthday();
```

Example Class: What's the Output?

```
class Person {
    constructor(name, age, job) {

        this.name = name;
        this.age = age;
        this.job = job;
    }
    sayName() {
        console.log('My name is ' + this.name + '\n');

    }
    sayAge() {
        console.log('I am ' + this.age + ' years old. \n');

    }
    sayJob() {
        console.log('I am a(n) ' + this.job + '\n');

    }
    birthday() {
        this.age += 1;
        console.log("Today's my birthday! I am " + this.age + "\n");
    }
    switch_jobs(new_job) {
        this.job = new_job;
        console.log('New job! I am now a(n) ' + this.job + '\n');

    }

}
var Samantha = new Person('Samantha', 24, 'engineer');
Samantha.sayName();
Samantha.sayAge();
Samantha.sayJob();
Samantha.birthday();
Samantha.switch_jobs('manager');
```

Another Example Class: Using Graphics

```
class SmileyFace {
    constructor(x,y,color) { // passes in user coordinates + color

        this.x = x;
            this.y = y;
            this.color = color;
    }
    show() { // class function that creates/shows the :)

        strokeWeight(5);
        fill(this.color);
        ellipse(this.x, this.y, 50, 50);
        fill("■black");
        strokeWeight(7);
        point(this.x + 8, this.y - 5);
        point(this.x - 8, this.y - 5);
        strokeWeight(5);
        point(this.x + 2, this.y + 15);
        strokeWeight(2);
        arc(this.x, this.y + 5, 10, 5, 0, PI + QUARTER_PI, OPEN)

    }

    followTheCursor() { // makes the :) follow the cursor

    this.x = mouseX;
    this.y = mouseY;

    }
}

function setup() { // new objects made in setup()
  createCanvas(400, 400);
  smiley = new SmileyFace(200, 200, '■yellow');
}

function draw() { // graphics-related class functions called in draw()
  background(220);
  smiley.show();
  smiley.followTheCursor();
}
```

Note: Variables created in draw() or setup() can be used within these functions. However, typically we cannot access variables that are declared in one function within a different function. The variable would need to be defined globally in order to access it from all functions.

6.4 Arrays and Objects

Now that we understand both objects AND arrays AND loops, we can put everything together and make collections of objects.

But wait! Before we do that - take a second. You've learned a LOT so far! You've learned all the basics of what you need to program. This is complicated stuff - but you made it here, you're ready. So if you're feeling overwhelmed because it's all coming together, just take a breath. You know what an object is, you know what an array is, you know what a loop is, you know what functions and variables and data types are. Just break it down little by little - and we'll figure it out!

Breathe in. Breathe out. Ready?

Look at the SmileyFace class from the previous section! This class creates a smiley face object, with the functionality of being able to be displayed on the screen and follow your cursor. For this example, we don't need it to follow our cursor - so let's get rid of that function and make it simple!

```
class SmileyFace {
    constructor(x,y,color) { // passes in user coordinates + color

        this.x = x;
            this.y = y;
            this.color = color;
    }
    show() { // class function that creates/shows the :)

        strokeWeight(5);
        fill(this.color);
        ellipse(this.x, this.y, 50, 50);
        fill("■black");
        strokeWeight(7);
        point(this.x + 8, this.y - 5);
        point(this.x - 8, this.y - 5);
        strokeWeight(5);
        point(this.x + 2, this.y + 15);
        strokeWeight(2);
        arc(this.x, this.y + 5, 10, 5, 0, PI + QUARTER_PI, OPEN)

    }

}
```

What if we want to make 10 and display them on our screen like we're looking out into a crowd of our smiley faces. Do we really have to repeat the line of code smileyFace = new SmileyFace(x,y,color); Over and over again?

We talked about something we can use when we notice we have to perform the same task over and over again - and we just learned to use it with arrays. Loops! Let's loop through and create an array full of new smiley faces!

```
function draw() { // graphics-related class functions called in draw()

  for (var i = 0; i < 10; i++) {
    smileyFacesArray[i] = new SmileyFace(200, 200, 'yellow');
  }

}
```

Each time the loop runs through, it adds a new Smiley Face object to the array! Cool!

But all our objects are exactly the same, they have the same coordinates. If we were to display them, they would all be right on top of each other... Not ideal.

Let's modify our loop a little bit so every time the loop runs through, the coordinates change just a bit.

```
function draw() { // graphics-related class functions called in draw()

  var shift = 0; // use this variable to shift our x over
  var x = 5; // starting x coordinate
  var y = 200; // starting y coordinate
  var smileyFacesArray = [];
  for (var i = 0; i < 10; i++) {
      smileyFacesArray[i] = new SmileyFace(x + shift, y, 'yellow');
      shift += 50;
  }
}
```

And now we can use another loop in the draw function to display our faces using the class function show().

```
  for (var i = 0; i < 10; i++) { // when displaying graphics, everything
    smileyFacesArray[i].show();
  }
```

Unit 6 Review

New Vocabulary

Object: an abstract data type with state (data) and behavior(functionality)

Object Oriented Programming (OOP): a type of programming that uses objects and their states/behaviors to define problems

Encapsulation: the principle that each object's state should be private to the object

Abstraction: the principle that objects should only give access to parts of their private state that is relevant for other object's use

Inheritance: the mechanism of basing an object or class upon another object

Polymorphism: objects can take on more than one form depending on the context

Class: a template or blueprint for your object

Constructor: an instructional function that define the state

Class Function: or functions defined within a class that detail an objects behaviors

Object Instance: a new object variable

Dot Notation: using a dot to indicate attachment of variables or functions to as class

New Code Structure

Structure for a class:

```
class ClassName {
      constructor(parameters) {
            this.var = parameters;
            .
            .
      }
      class_function(parameters) {
            .
            .
            .
      }
}

var x = new className(constructor parameters); // outside of
setup() use the var keyword
function setup() {
      y = new className(constructor parameters); // inside of
      setup() do not use the var keyword
}
```

Extra Help and Resources

Test Yourself

1. What is an object?
2. What are the four principles of object oriented programming? Explain each one in your own words.
3. What is a class?
4. Why do we need a constructor?
5. When do we use the this keyword?
6. What are class functions?
7. What is an object instance? What keyword do we have to use with object instances?
8. What is dot notation?

Practice Problems

1. Write down what would be the state and behaviors for the following objects: flower, teacher, cat
2. For each of the objects above, write down the parameters you could pass into the constructor.
3. For each of the objects above, write at least one function that fulfills a behavior.
4. Pick one of the objects above and write a full class for it.
5. Write a **PUSDStudent** class that stores a student's name, grade, ID Number, and number of classes taken. Then create an object instance that reflects your information.
6. Add some functions to the PUSDStudent class you created in Problem 5. Some possible functions are: updating the number of classes taken, updating a student's grade, etc.
7. Write a class function in the PUSDStudent class that will print out the student's name, grade, ID Number, and the number of classes taken.
8. Use the functions you created above to modify your object instance. Then, use the function you created in number 7 to print out the student's name, grade, ID number, and the number of classes they have taken.