INTRODUCTION TO JAVASCRIPT

DAY 1: JAVASCRIPT FUNDAMENTALS

WHO'S THIS GUY?

SCOTT C. REYNOLDS

- Developer/Team Leader for a real long time
- Prone to tangents and rants, feel free to interrupt
- Remembers when you couldn't do anything cool with JavaScript and is salty about it
- Does not love JavaScript (but loves its power)

COURSE OBJECTIVES

- Learn what can be done with JavaScript
- Be comfortable reading and writing basic JavaScript
- Be exposed to advanced JavaScript language features so they aren't scary
- Learn how to use JavaScript libraries
- Learn to interact with APIs
- Learn to use JavaScript on the server
- Learn to use JavaScript to create Spark bots

TOOLS

- Text Editor (Brackets)
- Browser with Dev Tools (Google Chrome)
- REPL (repl.it)

- Later we'll add other tools as we explore more advanced topics (Postman, ngrok, Node, etc.)

JAVASCRIPT RESOURCES

- Mozilla Developer Network (MDN)
- StackOverflow
- JavaScript Weekly
- https://scottcreynolds.github.io/javascript-class/

A BRIEF HISTORY OF JAVASCRIPT

- 1995 - Brendan Eich at Netscape creates JavaScript in 10 days
- 1996 - Microsoft implements JScript for IE 3, kicking off our long national nightmare
- 1997 - DHTML in IE4 and Netscape Navigator 4
- 2005 - AJAX brings peak "Web 2.0". JavaScript goes from "toy" to real deal.
- 2006 - John Resig releases jQuery, we finally get productive with JavaScript
- 2009 - Ryan Dahl creates Node.js, server-side js goes mainstream

ES6

ECMAScript 6/ECMAScript 2015 is a new specification and the first update to JavaScript since ES5 was standardized in 2009. Brings a ton of great stuff to JS.

- Arrow Functions
- True classes
- String interpolation
- let/const
- iterators
- promises

WHAT IS JAVASCRIPT?

- It ain't Java
- Scripting language - interpreted not compiled
- Object-oriented
- De facto standard client-side language
- De facto standard for API communication

WHAT CAN JAVASCRIPT DO?

- Dynamic, interactive websites
- Games (Google Pac-Man)
- Hybrid and Native Mobile Apps (Ionic/React Native)
- Chat Bots (spark/slack/etc)
- Actual bots (Raspberry Pi/Arduino)
- Performant multi-threaded app server (node)
- This slideshow (Reveal.js)

WRITING AND EXECUTING JAVASCRIPT

GOODBYE WORLD

Open Chrome. Go to any page. Twitter will do if you can't think of anything else.

Open Dev Tools (right-click -> Inspect) then open the Console tab and type the following:

```
document.removeChild(document.documentElement)
```

JAVASCRIPT LANGUAGE FUNDAMENTALS

STATEMENTS

Each line in JavaScript is an instruction, and each is executed one at a time, in order (try it out in repl.it).

```
console.log("hello");
```

Multi-line statements (*blocks*) are bound by braces

```
if(true) {
  console.log("true!");
}
```

```
//this is a comment
```

```
/* this is a
 multi-line
 comment */
```

STYLE GUIDELINES

- use camelCase for multi-word variable names, otherwise lower case
- open braces on same line as block declaration
- upper case/PascalCase for object names
- descriptive naming always
- obey Principle of Least Surprise

DO WE NEED THE SEMICOLONS?

- No - ASI handles it (this is why braces on same line)
- Uh, yes, ASI inserts them because they're required
- Definitely when separating statements on the same line (not common, except in for loops)

```
for(var i=0; i<10; i++)
```

- I'm gonna use them because that's how I was raised but sometimes I'll leave them out because people can change
- Follow team/codebase convention

VARIABLES

```
var i; //declare variable in local scope
```

```
i = 0; //initialize variable
```

```
var i = 0; //declare and init variable
```

```
let i = 0; //(ES6) declare variable in local scope
```

```
const i = 0; //(ES6) declare constant in local scope
```

```
newVar = 0; //Yikes. No globals!
```

ASSIGNMENT

```
var i;
i = "hello";
console.log(i);
i = "bye";
console.log(i);
```

```
let i = "hi";
console.log(i);
i = "bye";
console.log(i);
```

```
const a = "b";
a = "c";
//const c; - what happens?
```

PRIMITIVE DATA TYPES

```
var fruit = "banana"; //string
```

```
var year = 2017; //number
```

```
var pi = 3.14; //also number
```

```
var waterIsWet = true; //boolean (true/false)
```

```
var pi; //undefined
```

```
var empty = null;//null - intentionally empty
```

TRY IT

```
var fruit = "banana";
var pi;

typeof(fruit); // 'string'
typeof(pi); // 'undefined'

pi === undefined; //true
pi === null; //false
```

```
typeof(Fruit); // ???
```

```
pi = 3.14;
typeof(pi); //?
```

WORKING WITH STRINGS

CONCATENATION

```
let firstName = "Scott";
let lastName = "Reynolds";
console.log(firstName + lastName);
```

```
console.log(firstName + ' ' lastName);
```

```
console.log('My name is ' + firstName + ' ' lastName);
```

```
console.log('My name is ' + firstName + ' ' lastName +
        'and I really think pie is better than cake');
```

Protip: use variables for better readability:

```
let sentence = 'My name is ' + firstName + ' ' + lastName;
sentence += 'and I really think pie is better than cake';
console.log(sentence);
// there's gotta be a better way!
```

WORKING WITH STRINGS

INTERPOLATION WITH TEMPLATE LITERALS (ES6)

```
let firstName = "Scott";
let lastName = "Reynolds";
console.log(firstName + lastName);
```

```
let sentence = `My name is ${lastName},
        ${firstName} ${lastName}.`
```

WORKING WITH STRINGS

HELPFUL FUNCTIONS

```
var sentence = "Hi, I'm Scott";
sentence.indexOf("Scott"); //8 - Why?
sentence.indexOf("Joe"); // -1
sentence.includes("Scott"); // ES6
```

```
sentence.replace("Scott", "Joe"); //Hi, I'm Joe
console.log(sentence); // what will we get?
```

Most string methods do not alter the original object.

```
let newSentence = sentence.replace("Scott", "Joe");
```

WORKING WITH STRINGS

MORE HELPFUL FUNCTIONS

```
let fruit = "Kiwi";
fruit.length; // 4
fruit.toLowerCase(); // kiwi
fruit.toUpperCase(); // KIWI
```

```
let padded = "   weird  ";
padded.trim(); // "weird"
padded.trimLeft(); // "weird  "
padded.trimRight(); // "   weird"
console.log(padded); // "  weird  "

//remember to assign the result to keep it
padded = padded.trim();
```

WORKING WITH STRINGS

REGULAR EXPRESSIONS (REGEX)

Regex is a common syntax for text processing that is (almost) standardized across different programming languages. The basic syntax is:

```
/search text/ //matches literal "search text"
/\d*/ //matches on zero or more digits
/\w+/ //matches on one or more word chars
```

See Rubular.com for more.

WORKING WITH STRINGS

REGEX CONTINUED

```
var test = "Robin has 5 dogs";
test.search(/Robin/); // 0
test.search(/robin/); // -1 - case sensitive
test.search(/\d+/); // 10 - like what other method?
var replaced = test.replace(/Robin/, "Victoria");
```

```
var email = "joe@aol.com";
email.search(/\w+@\w+\.\w+/); // 0
var emailMatcher = /\w+@\w+\.\w+/;
test.search(emailMatcher); // -1
```

```
//use match to retrieve the match
var matched = test.match(/\d+/); // [5]
matched = test.match(/\d*/); // [''] test your matchers!
```

WORKING WITH NUMBERS

OPERATORS

```
1 + 2  // 3
2 - 4  // -2
2 * 8  // 16
10 / 5 // 2
10 % 2 // 0
10 % 9 // 1
2 ** 3 // 8
```

```
Compound Assignment
let x = 1;
let y=2;

x += y; // x = ?
x -= y; // x = ?
y *= 2; // y = ?
y /= 2; // y = ?
```

WORKING WITH NUMBERS

INCREMENT/DECREMENT

```
let x = 1;
x++;
console.log(x); // 2
x--;
console.log(x); // 1
```

Prefix vs Postfix

```

var x = 3;
var y = x++; //y = ? x = ?

var a = 2;
var b = ++a; //b = ? a = ?
```

We use postfix (x++) in loops to operate on *current* x and then increment at the end of the iteration.

TYPE COERCION

```
var a = 2;
var b = "Barney";

a + b; // implicit coercion
```

Implicit coercion is okay for simple things like string concatenation, but when we get into things like booleans, it can be real interesting (horrible).

```
var a = 42;
var b = "3.14";

// Explicit coercion
String(a);
Number(c);
```

PARSING NUMBERS

Sometimes we need to get a number from a string that can't be coerced reliably.

```
var x = "23 Skidoo";
z = Number(x); // NaN
z = parseInt(x); // 23

var y = "3.14 pie";
z = parseFloat(y); // 3.14
z = ParseInt(y); // ?

var a = "i ate 23 pies";
z = parseInt(a); // ?
```

WORKING WITH DATES

```
var someDate = new Date();
someDate = new Date("4/24/2017");
someDate = new Date(2017,4,24); // what happens?

someDate.getDate();
someDate.getMonth();

var d = Date.parse("4/24/2017"); //ms since 1/1/70
d = Date.parse("April 24 2017");
d.getDate(); //???
typeof(d);

d = new Date(d);
d.toString();
d.toDateString();
```

WORKING WITH BOOLEANS

BOOLEAN EXPRESSIONS

```
1 === 1; // true
1 === 2; // false
1 !== 2; // true

1 <= 2; // true
1 > 2; // false

"hi" === "bye"; // false
"hi" === "HI"; // false

(true);
(false);
```

WORKING WITH BOOLEANS

TRUTHY AND FALSY

In the previous example, why did we use === ("triple equals") when most other languages use == (double-equals)?

```
1 == "1"; // remember implicit type coercion?
0 == false;

' ' == 0;
'\n\n\n' == 0;
```

Use triple-equals syntax to avoid surprises!

WORKING WITH BOOLEANS

BOOLEAN LOGIC

```
// && (and) - true if both expressions true
1 > 0 && 1 < 2; // true

// || (or) - true if either expression true
1 > 0 || 1 > 2; // true

// ! - negation
!(1 > 0 && 1 < 2); // false
```

Negation is valid but usually can be rewritten
to be more readable.

MAKING DECISIONS

IF/ELSE

```
let a = 2;

if(a > 2) {
  console.log("more than two!");
} else {
  console.log("less than or equal to two");
}
```

IF/ELSE IF

```
let a = 2;

if(a > 2) {
  console.log("more than two!");
} else if(a === 2) {
  console.log("equals two");
} else {
  console.log("less than two");
}
```

SWITCH

```
if(x === y) {
  // do something
} else if (x === z) {
  // do something else
} else if (x === a) {
  // do a third thing
} //antipattern!
```

```
var mood = "hungry";
switch(mood) {
  case "happy":
    console.log("Great day! Have ice cream!");
    break;
  case "sad":
    console.log("Bad day. Have ice cream.");
    break;
  case "hungry":
    console.log("Eat something! I suggest ice cream.");
    break;
  default:
    console.log("you know what to do");
 }
```

TERNARY OPERATOR

Sometimes it's nice to condense things to one line:

```
var x = 0;
// condition ? valueForTrue : valueForFalse;
x > 0 ? "More!" : "Not More!"; //what output?

//always err on the side of readability
(typeOf(x) === Number && x > 0) ? document.getElementById(x) :
```

JAVASCRIPT OPERATIONS QUIZ

```
var x = 0
var y = false
if (x == y) {
  console.log("hmmm")
} else {
  console.log("okay")
} //what is output?
```

```
21 % 5 //?
```

```
var x=1;
var y = x++; //what is x? y?
var z = ++y; //what is z? y?
```

```
"string".indexOf(r); //?
```

JAVASCRIPT OPERATIONS QUIZ

- What would I use to validate the format of a string?
- What is the name of what happens in this line: `"every" + 1;`
- How would we do *explicit* coercion in the previous line?
- What is the type and value of an intentionally empty variable?
- What is the type and value of an unassigned variable?

ARRAYS

- Data structure that represents a list.
- Ordered data. Stored in the order of addition.
- Indexed data. Accessed by position. Zero-based.
- Can store any other type, and mixed types.

```
var myArray = []; // create new empty array
var myArray = ["hi","bye",1,[2,3]] // create array with elemen
myArray[0]; // "hi"
myArray.length; // 4
myArray[3].length; // 2
myArray[1].length; // why?
```

ARRAY OPERATIONS

ADDING AND REMOVING ITEMS

```
let arr = [1,2];

arr.push(3); // add element to end of array
console.log(arr); // [1,2,3] most array methods alter the arra

var i = arr.pop(); // remove (and return) last item
arr.shift(); // remove first item, shift everything left
arr.unshift(1); // add item to front of array

arr[0] = 4; // replace item at index 4
arr[4] = "hi"; // what happens at arr[3]?

arr.splice(1,0,"inserted"); // insert "inserted" at index 1
arr.splice(4,1); // remove 1 item starting at index 4
```

ARRAY OPERATIONS

MANIPULATING ARRAYS

```
let arr = ["one", "two"];

arr.reverse();
arr.indexOf("two"); // 0 because we reversed it!

arr.includes("two"); // ES6
arr.indexOf("three") >= 0; // false

let arrStr = "three,four,five";
arr = arrStr.split(','); // ["three", "four", "five"]
newStr = arr.join(':'); // "three:four:five"
```

ARRAY PRACTICE

- Initialize an array with the 8 planets of the solar system in order.
- Add Pluto because you're not a monster.
- Add Ceres between Mars and Jupiter.
- Add the Sun to the beginning.
- Remove Pluto from the end and add it to a new array of Dwarf Planets with Makemake and Haumea.
- Add the array of dwarf planets to the end of the solar system array.
- Find the position of Earth.

ARRAY PRACTICE SOLUTION

```
let solarSystem = ["Mercury","Venus","Earth","Mars",
                   "Jupiter","Saturn","Uranus","Neptune"];
solarSystem.push("Pluto");
solarSystem.splice(4,0,"Ceres");
solarSystem.unshift("Sun");
let dwarfPlanets = [solarSystem.pop(),"Makemake","Haumea"];
solarSystem.push(dwarfPlanets);
solarSystem.indexOf("Earth");
```

LOOPS

Why do we use loops?

- Performing an action a specific number of times.
- Performing an action until a condition is met.
- Iterating over the items of a collection and performing an action on each one.

LOOPS

WHILE...

You can think of a `while` loop as an `if` statement that repeats while it returns true. Useful when we can't predict the number of loops we need to execute.

```
var now = Date.now();
while(now % 2 !== 0) {
  console.log("not yet");
  now = Date.now();
}
```

LOOPS

DO...WHILE

`do...while` loops are similar to `while` loops except the condition is evaluated after the loop body, meaning the loop will always execute at least once.

```
var test = false;
while(test) {
  console.log("this will never execute");
}

do {
  console.log("but this will execute once");
} while(test);
```

LOOPS

FOR...

The `for` loop is your bread and butter. Used to iterate a finite number of times based on a condition such as a number of items in a collection.

```
var p = ["Red","Yellow","Blue"];

for (let i=0; i < p.length; i++) {
  console.log(p[i]);
}
```

LOOP QUIZ

- What loop would I use to calculate the balance of all customers' accounts?
- What loop would I use to output the progress of a long-running operation?

```
//when will this stop executing?
while(true) {
  console.log(Date.now())
}
```

```
//what gets logged?
var x = 0;
do {
  x++
  console.log(x);
} while (x === 0)
```

EXECUTING IN THE BROWSER

The browser is the easiest and most common way of executing interactive client-side JavaScript.

```
<html>
<head>
  <title></title>
</head>
<body>
<script type="text/javascript">
  console.log('this is inline js');
</script>
</body>
</html>
```

LINKING EXTERNAL CODE FILES

```
<head>
  <script src="class.js"></script>
</head>
```

```
//inside class.js
console.log("and this is linked javascript");
```

Any time you need to use an external library or maintain a non-trivial amount of code, you'll be linking separate files. Inline js is used sparingly.

BASIC INTERACTIVITY

Alerts and prompts can be quick and easy ways to collect and display data in the browser.

TRY IT

```
// class.js
let answer = prompt("Hey, what's your name?");
alert(answer);
```

EXERCISES

CONVERT CELSIUS TO FARENHEIT

Write a program that prompts for a temperature in celsius and alerts the temperature converted to farenheit.

Yes, you have to google the formula.

```
var celsius = prompt("Enter a temperature in celsius")
var farenheit = celsius * 9 / 5 + 32;
var message = `${celsius} celsius is ${farenheit} farenheit`;
alert(message);
```

FAVORITE COLOR

Prompt for a person's favorite color. Create an array of the colors of the rainbow in standard *ROYGBIV* order. Respond with the position (index) of their favorite color in the rainbow, or, if their favorite color isn't in the rainbow, a message telling them so.

```
let fave = prompt("What is your favorite color?");
let rainbow = ["red","orange","yellow","green","blue",
    "indigo","violet"];
let position = rainbow.indexOf(fave.toLowerCase());
if(position === -1) {
  alert(fave + " isn't in the rainbow");
} else {
  alert(fave + " is  number " + String(position + 1));
}
```

INTERMEDIATE JAVASCRIPT

- Functions
- Objects

FUNCTIONS

Functions are reusable chunks of code that can be called by other code.

```
function sayHi() {
  console.log("hi");
}

sayHi();
```

FUNCTION ARGUMENTS

Functions can be called with data, or arguments, that are used in the execution of the function.

```
function sayHi(name) {
  console.log("hi " + name);
}

sayHi("Scott");
```

```
function addNumbers(num1, num2) {
  let result = num1 + num2;
  console.log(result);
}

addNumbers(3,4);
addNumbers("scott",3); // what happens?
```

FUNCTION RETURN VALUES

Functions can return a (single) value.

```
function addNumbers(num1, num2) {
  let result = num1 + num2;
  return result; //once we return, we're done
  console.log("will never happen");
}
let sum = addNumbers(5,6);
console.log("sum");
```

FUN(CTIONS)

DEFAULT PARAMETERS

```
function multiply(x,y) {
  return x * y
}
multiply(4); //why no error?
```

```
function multiply(x,y) {
  if(y === undefined || !typeof(y) === Number) {
      y = 1;
  }
  return x * y
}
multiply(4);
```

```
function multiply(x,y=1) {
  return x * y
}
multiply(4);
```

FUN(CTIONS)

ARGUMENTS OBJECT AND REST PARAMETER

```
function rest(x) {
  console.log(arguments); //all arguments and system stuff
}
rest(1,2,3,4,5)
rest(1)
```

```
function rest(x, ...myArgs) {
  console.log(myArgs); // just the extra arguments as array
}
rest(1,2,3,4,5)
```

VARIABLE SCOPE

Variables in JavaScript are *scoped*, or available, at the level at which they are declared and anywhere that can access that scope.

```
var outer = "outside"; // "global scope"
function testScope() { // also global scope
  console.log(outer);
  outer = "inside" // be careful of side effects!
}
testScope();
console.log(outer);
```

FUNCTION SCOPE VS BLOCK SCOPE

```
function testScope() {
  var inner = "inside"
  console.log(inner);
}
testScope();
console.log(inner); //can't get there
```

```
function testScope() {
  if(true) {
    var varBlock = "var in block";
    let letBlock = "let in block";
  }
  console.log(varBlock);
  console.log(letBlock); //can't get there
}
testScope();
```

FUNCTION PRACTICE

FIZZBUZZ

Prompt the user for a number. Create a function called FizzBuzz() that takes the user's number as an argument and prints every number from one to that number to the console log.

TWIST: if the number is a multiple of 3, it prints "Fizz" instead of the number. If the number is a multiple of 5, print "Buzz" rather than the number. If it's a multiple of 3 and 5, print "FizzBuzz".

FIZZBUZZ SOLUTION

```
function FizzBuzz(upperBound) {
  var output;
  for(var i=1;i<=upperBound;i++){
    output = ''
    if(i%3 === 0){
      output+='Fizz';
    }
    if(i%5 === 0){
      output+='Buzz';
    }
    if(output === '') {
      output+=i;
    }
    console.log(output);
  }
}
var input = prompt("Enter a number");
FizzBuzz(input);
```

OBJECTS

JavaScript is an *object-oriented (OO)* language. Objects are special data structures that allow us to encapsulate *data (properties)* and *behavior (methods)*.

Objects are generally designed to represent a cohesive concept, sometimes logical, sometimes physical. JavaScript provides some objects for us, and the ability to make our own.

```
var d = new Date(); //create instance with new keyword
```

PROPERTIES

A *property* represents data or an attribute, related to an object.

- Object: Car. Properties: Color, Transmission, DoorCount, Engine.
- Object: Polygon. Properties: Sides, Position, Area, Perimeter
- Object: Side. Properties: Length

METHODS

A *method* is a function that describes the behavior of an object.

- Object: Car. Methods: Unlock, Lock, Start, Drive
- Object: Polygon, Properties: Move(coords), Fill(color)

CREATING OBJECTS

LITERAL CONSTRUCTOR

```
var car = {}; //that's it
```

```
var car = {color: "red"};
car.color; //dot notation
car["color"]; //bracket notation
//why?? objects are key/value pairs. keys are strings
```

OBJECT CONSTRUCTOR

```
var car = new Object();
```

```
var car = new Object({color: "red"});
```

MANIPULATING OBJECTS

```
// objects are open to manipulation at runtime
        var car = {};
        car.color = "Red";
        car.engine = "V8";
        car.doors = 2;

        console.log(car);
        car.color = "Black";
        console.log(car.color);
        delete car.color;
        console.log(car);
```

OBJECT METHODS

```
var car = {color: red};
car.start = function() {
  console.log("vroom");
}
car.start();

//functions are... data?
console.log(car);
var func = function() {console.log("yep")}
func();

//functions are... OBJECTS!!
typeof(car.start);
car.start.thisIsWeird = "yep"
console.log(car.start.thisIsWeird); //but like, don't.
```

NAMESPACES

Namespacing allows you to encapsulate an object/objects within a named scope to organize and avoid collissions. You'll run into this with third party libraries like jQuery, and it's good practice when making your own libraries.

```
var JavascriptClass = { //just a top-level object
  car: {color: "red"} // where the properties are objects
}
var SomeBadClass = {
  car: {color: "rusty and bad"}
}
console.log(JavascriptClass.car);
console.log(SomeBadClass.car);
```