Javascript!

Lexical Structure

- Javascript is written in unicode, which ultimately means that you can code with english, french, german, norwegian, hindi, tagalok, mandarin, and even emoji.
- Semicolons:
  - Do you need them? No.
  - Should you use them? 🤷‍♀️
  - Semicolons dictate the end of each line, but the engines in which we run javascript are often intelligent enough to figure out what you mean.
  - Whitespace is meaningless. Scope is determined by curly braces.
  - Javascript is case sensitive.

Literals:

- Integer: 5
- String: "I am a string" or 'I am a string'
- Boolean: true or false
- Array: [1, 2]
- Object: { firstname: 'matt', lastname: 'lane '}

Variables.

When do you mutate, when do you not (hint: NEVER MUTATE ANYTHING)

- Naming variables/constants/functions can be done with almost any unicode character. (100% recommend using the naming conventions you're comfortable with)
- Reserved words exist in javascript.
- Just like any other language

Variable Initialization

Using nothing: In javascript, it is possible to assign a value to a variable.

- MUTABLE
- Most modern environments have a strict mode enabled, which will result in an error. strict mode is not enabled in a repl:

```
> a = 5
5
> a
5
```

Variable Initialization

`var`: Variables can be declared with var.

- **MUTABLE**
  - When declared with nothing, the variable is undefined
  - Multiple variables can also be declared at the same time.
  - If declared outside of a function, var will be attached to the global object.
- Not block scoped, but function scoped.

```
> var b = "hello"
undefined
> b
'hello'
```

Variable Initialization

`let`: A newer ES6 variable that acts almost the exact same as `var`:

- **MUTABLE**
- When declared outside of a function, the variable is not attached to the global object.
- Block scoped, not function scoped.

```
> let c = 42
undefined
> c
42
> c = 'whatever'
'whatever'
```

Variable Initialization

`const`: Once initialized, a `const`'s value can never be changed.

- **IMMUTABLE**ish
  - `const` means that the reference to the identifier can't be changed.
  - Objects with sub-references can allow for a potential option of mutability.
  - Block scoped.
  - In order to avoid errors in complex projects, developers often tend to choose `const` for declaring their variables.

```
> const d = "Stay gold"
undefined
> d
'Stay gold'
> d = 3
Thrown:
TypeError: Assignment to constant variable.
```

Variables: Scoping

- **Function scoping** means that variables can all see each other so long as they're within the same function.
- **Block scoping** means that variables are scoped within the code block (that is, anything inside `{}`s)

```
function mycoolFunction(){
    if(true){
        var funkyVar = 'rhombus'       //exist in function scope
        const blockyConst = 'cube'     //exist in block scope
        let blockyLet = 'rectangular prism'   //exist in block scope

    }
    console.log(funkyVar)
    console.log(blockyConst)
    console.log(blockyLet)
}

mycoolFunction()
```

Types:

Primitives:

- Numbers
- Strings
- Boolean

Primitives - Numbers:

`Number` encompasses both floating points and integers

Primitives - String:

A sequence of characters. Strings can be enclosed in both single and double quotes, as well as ticks. As of ES6, template strings now exist (those with the ticks). Template stings allow for mid-string variable interpolation

```
const doubleHello = "HELLO"
    const singleHello = 'Hello'
const templateStringHello = `The above two sentences say
    ${doubleHello}
and
    ${singleHello}
    `
```

Primitives - Boolean:

Booleans are true and false.

- Javascript also distinguishes between truthy and falsey values.
    - falsey:
    - 0
    - Nan
    - '' (empty string)
    - null: Indicates the lack of a value
    - undefined: a variable that has not been initialized, or does not exist.
- truthy: Everything else!

Example!

```
a = 5
var b = 'whatever'
let c = 32.6
const d = false
const e = "stuff"

console.log(`a: `, typeof a)
console.log(`b: `, typeof b)
console.log(`c: `, typeof c)
console.log(`d: `, typeof d)
console.log(`e: `, typeof e)
```

Types:

Object Types

- All non-primitive types are object types.
  - Objects: A key value pairing. The keys can be paired to variables of any type (including functions)
  - Arrays: Lists.
  - Functions: Can be invoked (discussed in detail later)

Objects:

An object is a datatype that contains key-value pairings.

```
const someObject = {
    keyNameOne: "SomeValue",
    keyNameTwo: 23,
    keyNameArray: [1,2,"three"]
    keyNameFour: {
        subObjectKey: 3.2
    }
}
```

To access these elements, you need only to use dot notation:

```
console.log(someObject.keyNameFour.subObjectKey)
```

Classes:

- ES6 introduced the idea of classes in the traditional sense:

```
class SpiceGirl {
    constructor(spice){
    this.spice = spice
    this.name = spice + ' spice'
    }

    getSpicy() {
        return "SPICE UP YOUR LIFE with " + this.spice
    }

    get name(){
        return this.name
    }

    set spice(value){
        this.spice = value
    }
}
```

Exceptions

Just like other languages try catch blocks exist to handle errors gracefully.

```
try {
    // do something bad
    const notPossible = 1/0
} catch (error) {
    console.error("There's an error!", error)
}
```

Arrays:

Arrays are lists.

```
const myList = [1,2,3,4,5]
const otherList = ['one','two','three']
const thirdList = [1,'three', 23.4, otherList]

console.log(myList)
console.log(otherList)
console.log(thirdList)
```

Functions!

- Functions are objects. **Function Objects**. In order to use a function, it must be invoked with `()`. Otherwise, simply typing the function name with no parentheses refers to the object's reference.
    - Functions in Javascript are **first class functions**. This means that functions can be passed as parameters to other functions.

Functions - Syntax

Function Declarations (older):

```
function doStuff1(thing) {
    console.log("We're doing stuff!", thing)
}
```

Functions - Syntax

Function Expressions: assignment to a variable

```
const doStuff2 = function(thing){
    console.log("We're doing stuff!", thing)
}
```

Functions - Syntax

Named Function Expressions: (these play well with stack traces)

```
const doStuff3 = function doStuff(thing){
    console.log("We're doing stuff!", thing)
}
```

Functions - Syntax

Arrow Functions: As of ES6, arrow functions have been introduced.

```
const doStuff4 = (thing) => {
    console.log("We're doing stuff!", thing)
}
```

Arrow functions allow for an implicit return for one line statements! This means that you don't have to write return variableName in specific instances.

```
const returnsHelloWorld = () => "helloWorld"
const returnedString = returnsHelloWorld()
console.log(returnedString)
```

When implicitly returning an object from an arrow function, the object needs to be wrapped in parentheses:

```
const formatSomeObject = (someObject, someParameter) => ({
  name: someObject.coolThing,
  date: someObject.time,
  hobby: someParameter
})

const myNeatObject = {
  coolThing: 'Mr Potato Head',
  time: '1996',
}

const whatDoYouDo = "Jam eyes and things into a potato"

const returnedObject = formatSomeObject(myNeatObject, whatDoYouDo)
```

Functions - Parameters:

- As of ES6 functions can now have default parameters:

```
const doStuff5 = (thing='regularCoolThing', otherThing='superCoolThing') => {

    const stuff = thing + otherThing
    console.log("We're doing stuff!", stuff)
}
```

Functions - Parameters:

- If you have just one variable in your parameter list, you don't need to wrap it in parentheses:

```
const doStuff6 = thing => {
    console.log("We're doing stuff!", thing)
}
```

Functions - Parameters:

- Functions can take in a **spread** array as a list of parameters:

```
const doStuff7 = (thing='regularCoolThing', otherThing='superCoolThing') => {
    const stuff = thing + otherThing
    console.log("We're doing stuff!", stuff)
}

const argumentsArray = ['one', 'two']

doStuff7(...argumentsArray)
```

Functions - Parameters:

- Functions can **destructure** an object to pass in arguments:

```
const doStuff8 = ({ thing ='regularCoolThing',
                    otherThing='superCoolThing' }) => {

    const stuff = thing + otherThing
    console.log("We're doing stuff!", stuff)
}

const argumentsObject = {
    thing: 'Stuff',
    otherThing: ' and things',
}

doStuff8(argumentsObject)
```

Return Values:

- Javascript cannot return more than one value, however, it can return particular data structures that can be desctructured!

Return Values: Destructuring Arrays

```
const returnAnArrayToDestructure = () => {
    return ["Thing","other",3,4]
}

const [ first, second, third, fourth ] = returnAnArrayToDestructure()
```

Return Values: Destructuring Objects

```
const returnAnObjectToDestructure = () => {
    return {
one: 2,
    buckle: 'my',
shoe: "¯\_(ツ)_/¯",
    }
}

const {one, buckle, shoe} = returnAnArrayToDestructure()
```

Function Nesting:

- It is possible to nest one function within another. However, it should be noted that if you do nest functions inside of each other, the most internal function's scope will be limited to the outer function.

Function Nesting:

```
const externalFunction = () => {
    const internalFunction = something => {
    return something + " nothing"
    }

    const stringToReturn = internalFunction("Something is not")

    return stringToReturn
}

console.log(externalFunction())

console.log(internalFunction())
```

Functions in Objects:

- Because objects can have anything as a key value pairing, objects can have functions paired with a key:

Functions in Objects:

```
const vampire = {
    firstName: 'David',
lastName: 'Boreanas',
    pseudonym: 'Angel/Angelus',
printFullName: function() {
    console.log(this.firstName + ' ' + this.lastName)
    }
}

console.log(vampire.firstName + ' is a name')
console.log('His full name is ')
vampire.printFullName()
```

Functions in Objects

- This is a good time to discuss the differences between a `function` and an `anonymous function`

Functions in Objects

```
var human = {
    firstName: 'Sarah',
middleName: 'Michelle',
    lastName: 'Gellar',
pseudonym: 'Buffy Summers',
    sobriquet: 'The Vampire Slayer',
    printFullName: function() {
    console.log(this.firstName + ' ' + this.middleName + ' ' + this.lastName)
    },
    printSobriquet: () => {
        console.log(this.pseudonym + " " + this.sobriquet)
    }
}

console.log(human.firstName + ' is a name')
console.log('Her full name is ')
human.printFullName()
console.log('Her character name is ')
human.printSobriquet()
```

Functions in Objects

- The difference between the two function declarations is that the `this` keyword interacts differently with each:
  - `function() { ... }`: Refers to the object's enclosing object reference
  - `() => { ... }` refers to to the enclosing function context ()

First Class Functions

- Essentially, first class functions are functions that can be passed back and forth to other functions as parameters.

First Class Functions : Passing Functions

```
const doStuff = () => {
    console.log("I'm doing stuff!")
}

const doSomething  = something => {
    something()
}

doSomething(doStuff)
```

First Class Functions : Passing Functions

```
const doStuffAgain = somethingCool => {
    console.log("I'm doing ", somethingCool)
}
```

```
const doSomethingAgain  = something => {
    const someVariable = "really neat!"
    something(someVariable)
}

doSomethingAgain(doStuffAgain)
```

First Class Functions : Returning Functions

```
const functionsReturningFunctions = () => {
    const someFunction = () => {
        console.log("RETURN TO ME!")
    }
    return someFunction
}

const returnedFunction = functionsReturningFunctions()

returnedFunction()
```

First Class Functions : Returning Functions

```
const functionsReturningFunctions = () => {
    const someFunction = () => {
        console.log("RETURN TO ME!")
    }
    return someFunction
}

const returnedFunction = functionsReturningFunctions()

returnedFunction()
```

First Class Functions : Returning Functions

```
const functionsDoingCrazyStuff = () => {
    const someOtherFunction = someVariable => {
        console.log(`Some other function calls: `, someVariable)
    }
    return someOtherFunction
}

const returnedOtherFunction = functionsDoingCrazyStuff()

const someString = "HOW CRAZY!"
returnedOtherFunction(someString)
```

First Class Functions : Returning Functions

```
const crazyFunctionAgain = () => {
    const crazyOtherFunc = someVariable => {
        console.log(`Some other function calls: `, someVariable)
    }

    return crazyOtherFunc
}

crazyFunctionAgain()("DOUBLE PARENS!?")
```

Closures

When a function is run, it's executed within the scope of where it was defined, and not where it was run. (This ends up being incredibly helpful in the world of React):

```
const createAFunction = word => {
  const wordsToSay = "Well, I say " + word

  return () => console.log(wordsToSay)
}

const newFunction = createAFunction()

newFunction()
```