JavaScript!

What is JavaScript?

- A (traditionally) client-side scripting language
- Meant (traditionally) to run entirely on the user's browser
- Defined by the ECMAscript standard, published by the ECMA foundation
- JavaScript != Java, they are completely unrelated languages

What is it used for?

- In the context of web browsers, JS allows you to interact with the DOM (Document Object Model), so you can do things like:
  - Show and hide elements
  - Animate elements
  - Replace elements with other elements
  - Make requests to the server without reloading the page
- The DOM is a programmatic representation of all of the HTML elements on the web page.

What are the JS Data Types?

JS Data Types

- Primitives:
  - String e.g. `“This is a string”`
  - Number e.g. `12 / Infinity / 3.14` *all numbers in JS are floats
  - Boolean => `true / false`
  - `null`
  - `undefined`
  - `Symbol`
- Object e.g. `{key: value} / [1, 2, 3] / function() {}`

null vs. undefined

- `undefined` typically means that a variable has been declared but has not yet been signed to a value.
    - Functions that do not explicitly return a value will also implicitly return `undefined.`
- `null` is an assignment value. It is generally used to represent the intentional absence of any object value.

typeof operator

- The `typeof` operator is used to check the data type of a particular value. The result will be a string representing the data type of what is passed to it, for example:
    - `typeof 2 === ‘number’`
    - `typeof ‘Jon’ === ‘string’`

JS quirk: Type Coercion

Type coercion is the process of (implicitly or explicitly) converting a value from one type to another. Since JS is a weakly-typed language, type coercion can be intentional (explicit) or situational (implicit) What does this look like?

JS quirk: Type Coercion

Implicit Coercion Pop Quiz

- 2 + 2 = ?
- 2 +‘2’ = ?
- ‘2’+ 2 = ?
- ‘2’- 2 = ?

JS quirk: Type Coercion

- `2 + 2 = 4`
  - No type coercion because data types match as numbers
- `2 +‘2’ = 4`
- `‘2’+ 2 = ’22’`
  - ‘+’ is treated as ‘concat’ string operator because one of the values is typeof ‘string’
- `‘2’- 2 = 0`
  - ‘-‘ coerces the string ‘2’ to a number to properly use the subtraction operator.

JS quirk: Type Coercion

For an interesting look at how ‘==‘ can lead to some weird and unexpected coercion results, check out this link:

https://dorey.github.io/JavaScript-Equality-Table/

What are the six falsey values in JS?

Six falsey values

Using any of the following values with `‘!!’` operator, `Boolean(value)` function, or in a conditional block will coerce them to `false`:

- `0` (zero)
- `‘’` (empty string, no whitespace)
- `null`
- `undefined`
- `NaN`
- `false (of course!)`

How do we declare variables in JS?

Variable Declaration

As of ECMAscript 6 (ES6), there a three ways to declare a variable in JavaScript, each with different mechanics especially as it pertains to **scope**:

JS Variables

`var`

- The original method of declaring a variable in JavaScript
- Variable names declared in global scope can be reassigned by other `var` declarations elsewhere in your script / project if using the same variable name.
- Only contained by local (functional) scope.

JS Variables

`let`

- Introduced as a method of declaring variables as of ES6 in 2015
- Variables names declared in global scope CANNOT be reassigned by other `let` declarations of the same variable name in the same scope.
- Maintains block / lexical scope, i.e. if a variable is declared using `let` within any type of block (`if/else, for` loop) it will <u>not</u> be accessible outside of the block that it is declared in

JS Variables

`const`

- Introduced as a method of declaring variables as of ES6 in 2015
- Variables names declared CANNOT reassigned at all as they are considered constant variables (hence `const`).
- Maintains block / lexical scope as well.

JS Variables

There is a difference between variable *declaration* and variable *definition:*

- Declaration means using one of the variable declaration keywords (`var, let, const`) to declare a variable name, e.g. `let x`
- Definition means actually assigning a value to the variable that has been declared, e.g. using the previously declared variable to set `x = 1`
- Declaration and definition can, and typically does, happen in line (`let x = 1`), however there are plenty of use cases for declaring a variable that you will assign at a later time.

How do we declare functions in JS?

JS Functions

Once again, thanks in part to ES6, there are three ways to declare functions in JS:

- `function` keyword declaration
- function expression saved in a variable
- ES6 Arrow functions

JS Functions

`function`

- The most straightforward way to declare functions.
- Using `function myFunc(arg){…}` to declare the `myFunc` function will **hoist** the definition to the top of your script.
- `function` can also be used to declare anonymous functions (`function(arg){…}`)

JS Functions

Function Expression

- As JS functions are considered **first-class functions**, they can be assigned to variables and passed around like any other data type in JavaScript.
- Declaring functions this way will **not hoist** the definition to the top of your script.
- Function expressions have the benefit of allowing a function to be self-invoking, otherwise known as an IIFE(immediately invoked function expression).

JS Functions

`()=>{}`

- Introduced in ES6, Arrow function syntax (aka Fat Arrow functions) are a new way of creating functions with some special rules
- Arrow functions composed on one line will implicitly return the result of the operation taking place, however multi-line statements will still need to be wrapped in brackets and the `return` keyword is required to share any information.
- Most importantly, the Arrow functions do not create their own `this` context, instead inheriting `this` from the lexical scope in which they are created, and will go up in scope until a context is found.
- Because of `this` rules with Arrow functions, they are better suited for non-method functions and cannot be used inside of constructors.

Arrays and Objects

Arrays

- Arrays are used to hold a collection of data, and can consist of any multiples of any data type, like so:

```
[“string”, 11, [2, 3], {key: value}]
```

- Arrays can also be stored in variables, as well.

Arrays

- Once you’ve declared an array, you may want to retrieve the items inside of it using their indices.

- Arrays are zero-indexed, and an array element’s index corresponds to its position from the beginning of the array.

```
const cars = [“Porsche”, [“Camry”]
```

- In order to access “Porsche” in the cars array, you would do so by targeting the index of the value that you want from the array: `cars[0] //“Porsche”`

Arrays

- Arrays that hold other arrays are called *multi-dimensional* arrays.

```
const cars = [[“Porsche”, “Camaro”], [“Camry”,”Prius”]]
```

- To target the value “Prius”, you would target the index of the inner array AT the index of the outer array like so:

```
cars[1][1] // “Prius"
```

Objects

- A way of organizing data using key/value pairs.

```
const car = { make: “Toyota”, model: “Matrix”}
```

- Similar to arrays, you can access information using bracket notation, only what is in the bracket is the key that you wish to target.

```
car[‘make’] // “Toyota”
```

Objects

- You can also use “dot notation” to get data out of an object.

```
const user = { firstName: “Lucille”, lastName: “Bluth”}

user.firstName // “Lucille”
```

Destructuring

- New syntax introduced as of ES6, destructuring allows you to break an array into it’s elements without mutating the original array, for example

```
const arr = [1, 2, 3]

const [a, b, c] = arr

console.log(a, b, c) // 1, 2, 3
```

Destructuring

- Similar functionality exists for objects, using the key as a variable name to access the value at that key, for example:

```
const obj = {firstName: ‘Jon’, favColor: ‘blue’}

const {firstName, favColor} = obj

console.log(firstName,favColor) // ‘Jon’, ‘blue'
```