Introduction to Javascript

Finally, we are programming!

We'll start with baby steps, and learn just enough to get you started:

1. Values and operators
    - Numbers, boolean, string, and operators
    - Creating and assigning value to variables
    - Objects and arrays
2. Program structure
    - Conditional (`if...else if...`)
    - `for...` loop
3. Functions
4. Testing the waters with D3

1.1 Values and Operators: Number

Numbers and arithmetics:

numbers `3.43` `2.9e6` `0`

operators `+` `-` `*` `/` `%`

special `NaN` `Infinity` `Undefined` `Null`

Use brackets to specify order of operation

```
1 + 3 * 4                (1 + 3) * 4
```

1.1 Values and Operators: String

Strings are enclosed by single quotes or double quotes

```
“This is a line of text.”
```

Strings can be concatenated with +

```
>> “Hello” + “world” + “!”
>> “Helloworld!”
>> “Hello” + “ ” + “world” + “ !”
>> “Hello world !”
```

1.1 Values and Operators: Boolean

Can be of values `true` or `false` (note the case)

Booleans values are the result of comparison operators

`>  <  >= <= == !=`

```
>> 9 >= 10
>> false
>> 8*8 + 1 > 64
>> true
>> NaN == NaN //what would this produce?
```

1.1 Values and Operators: Boolean

Logical operators apply to boolean values directly:

&&	AND operator: true only if both values are true
||	OR operator: true if one or both values are true
!	NOT operator

```
>> false && true //false
>> false || true //true
>> (8>9) && (9==9) //false
>> !(8>9) // true
>> !(0/0) //??
```

1.1 Values and Operators: “False-y” Boolean Values

The following values are “falsey” i.e. they evaluate to false

```
NaN      null  undefined 0  “”
```

This is a special case of type coercion i.e. JavaScript will convert values to types that it wants.

```
>> “5” * 2
>> 10
```

1.1 Values and Operators: Type Coersion

Use strict equality `===` or strict inequality `!==` to make sure the types are the same

```
>> “5” == 5
>> true
>> “5” === 5
>> false
```

1.2 Statements

Let's put them to use in statements, which can be thought of as commands to the interpreter to "do something".

They can be extremely simple:

```
alert("Hello world!");
```

Another simple case is if we want the program to "remember" something, which is when we declare variables:

```
const age = 28;
let daysPerYear = 365, monthsPerYear = 12;
```

1.2 Variable Declaration and Assignment

Varibles don't contain values; they point to values, and can in fact be made to point to a different value at any given time:

```
>> let greeting;
>> console.log(greeting); //???
>> greeting = "hello";
>> console.log(greeting); //hello
```

One more thing: variable names cannot be a reserved word; also, observe best practice for variable names, which should be short, descriptive, and capitalized properly*.

1.2 Variable Declaration: `const`, `let`, and `var`

Let's examine the following statements carefully

```
const name = "Siqi";

let age = 32;
//next year...
age += 1;

console.log(name + " teaches " +
numOfStudents + " students.");

var numOfStudents = 10; //note "hoisting"
of this variable
```

Aside: Template Literals

Template literals provide an easy way to construct strings from variables. Compare the following:

```
//Without template literals
console.log(name + “ teaches ” +
numOfStudents + “ students.”);

//With template literals
console.log(`${name} teaches
${numOfStudents} students`);
```

1.3 Objects and Arrays

Objects and **arrays** are **data structures**: a way of abstracting, encapsulating, and representing data.

Most programming languages implement some version of these data structures.

Objects are a fundamental building block of Javascript.

1.3 Objects and Arrays: Objects

Objects contain **properties**, which can be values, other objects, functions etc.

In Javascript, objects are always wrapped by a pair of **curly braces**.

```
const someObject = {}; //an empty object

//Non-empty object
const obj2 = {
   property1: 2,
   property2: true,
   someOtherProperty: {}
}
```

1.3 Objects and Arrays: Objects

Using the **. (dot)** notation, objects allow us to easily set and get their properties.

```
const person = {
   name: “Siqi”,
   instructor: true,
   id: 233
};

//Get
console.log(person.name);

//Set
person.residence = “New York City”;
```

1.3 Objects and Arrays: Objects

One useful way to think about objects is to think of them as a **look-up table**, a "**dictionary**", or a **map** (in programming parlance).

```
const stateCapitols = {
   AL: "Birmingham",
   AR: "Little Rock",
   ...
   MA: "Boston",
   ...
};

//How do you look up the state capitol of
MA?
```

1.3 Objects and Arrays: Arrays

Arrays can be thought of as a list. For example, something like the following

Boston New York DC Chicago

can be represented as follows:

```
const cities = [“Bosotn”, “New York”, “DC”,
“Chicago”];
```

1.3 Objects and Arrays: Arrays

As previously shown, the array syntax consisten of square brackets `[ ]`, containing values separated by comma `,`

Arrays can contain any Javascript value, from numbers, to objects, to functions, to other arrays

```
const cities = [
   {city: “Boston”, state: “MA”},
   {city: “LA”, state: “CA”}
];

//Add one item to the array
cities.push({city: “Chicago, state: “IL”});
```

1.3 Objects and Arrays: Quick Reflection

Because the job of data visualizations (and indeed most computer programs) is to manipulate / represent data, data structures play a hugely important role.

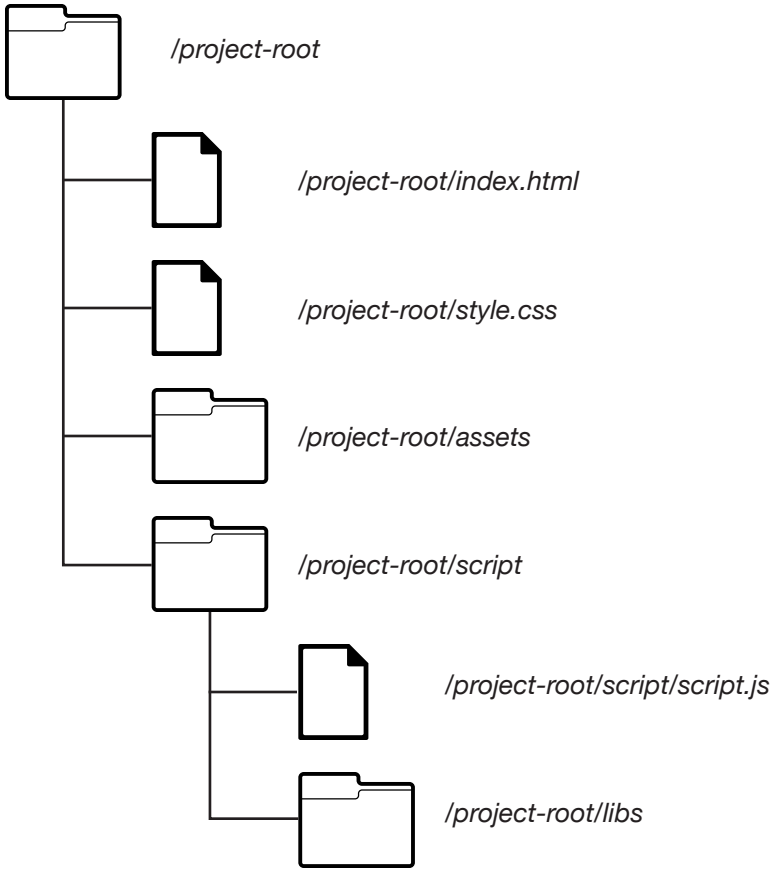
Event with relatively simple data structures like arrays and objects, we can easily model and work with real-world data sets.

Can we think of an example of real-world data that can be represented with arrays and objects (hint: where did you shop recently?)

1. Values and operators
    - Numbers, boolean, string, and operators
    - Creating and assigning value to variables
    - Objects and arrays
2. Program structure
    - Conditional (`if...else if...`)
    - `for...` loop
3. Writing functions
4. Testing the waters with D3

2. Control Structure

But when and in what order are statements run?


/project-root
/project-root/index.html
/project-root/style.css
/project-root/assets
/project-root/script
/project-root/script/script.js
/project-root/libs

/project-root/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World</
title>
    <meta charset=“utf-8” />
    <link href=“style.css”
rel=“stylesheet” />
  </head>
  <body>
    ...
    <script ...></script>
  </body>
```

/project-root/script/script.js

```
//script.js
```

```
<script src= “script/script.js”></script>
```

2. Control Structure

Statements are generally run from top to bottom but can be altered with control structures

1. Conditional execution (`if...else if...else`)
2. Loops (`while / for`)

2. Control Structure: Conditional

Some expression that produces a boolean value

```
if( var1 > var2 ){
    console.log("var1 is greater than var2");
```

Note the space

```
>> var num = 8/12;
>> if(num > 1){
       console.log("greater than 1");
   }
>> num = num + 1;
>> if(num > 1){
       console.log("greater than 1");
   }
```

2. Control Structure: Conditional

A more complicated case, with multiple “paths” to go down:

```
if( [some boolean value “a”] ){
    //run these statements if “a” is true
}else if( [another boolean value “b”] ){
    //run these statements if “a” is false but “b” is true
}else if( [another boolean value “c”] ){
    //run these statements if “a” and “b” are false, but “c” is
true
}
...
else{
}
```

2. Control Structure: Conditional

Compare these two examples: how are they different?

```
var num = 2.5;

if(num < 5){
    console.log(“smaller than
5”);
}else if(num < 10){
    console.log(“smaller than
10”);
}
```

```
var num = 2.5;

if(num < 5){
    console.log(“smaller than
5”);
}
if(num < 10){
    console.log(“smaller than
10”);
}
```

2. Control Structure: Loop

1. Create an initial conditions
2. Create a boundary condition (boolean) to stop the loop
3. Update the state the loop at each iteration, checking against the boundary condition; stop once the boundary condition is reached

```
          1          2          3
for(var i=0; i<1000; i++){
    //statements here will run 1000 times
}
```

Note the space

2. Control Structure: Reflection

Regardless of the choice of programming languages, computer programs represent underlying algorithms.

Loops and conditional statements are important parts of many algorithms.

As a thought experiment, let's devise an algorithm and make use of these control structures. Let's say we are searching for books in a library. Consider:
- How is the information about books modeled, data structure-wise?
- How do we use the control structures we just learned about?

How can we structure larger, more complex programs?

How do we deal with and take advantage of repitition?

Think of real-world analogies.

3. Functions

Functions help to define blocks of sub-program that 1) functionally relate to each other and/or 2) can be re-used.

We can define a function just like a variable:

```
const someFunc = function(){...};
```

Defining a function will NOT run the statements inside it.
However, later this function can be called like this:

```
someFunc(); //this will run someFunc
```

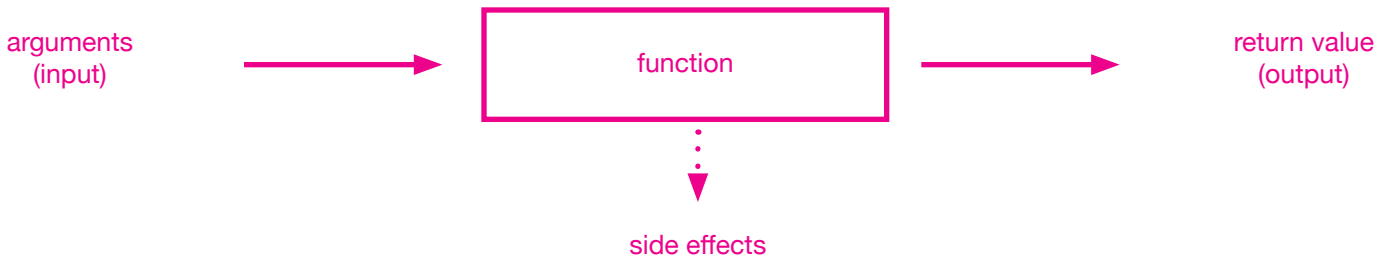
3. Functions

```
var newCar = {

    //these are properties
    make: “Subaru”,
    year: 2009,
    color: “Silver”,

    //the object contains a function; it’s
called
    //a “method”
    start: function(){
        console.log(“Vroom”);
    }
}
```

3. Functions: Basics

Let's look at a trivial example first:

```
var multiply = function(a,b){
    return a*b;
}
var num = multiply(5,8);
console.log(num); //40
```



3. Functions: Basics

a and b are **arguments**, which are initial variable values available inside the function, and are supplied by the caller.

Why the choice of these two variables?

```
var multiply = function(a,b){
    return a*b;
}
var num = multiply(5,8);
console.log(num); //40
```

**variable names for arguments are arbitrary!

3. Functions: Scope

Another example:

```
const multiplier = 5;
const multiplyByTen = function(a){
    const multiplier = 10;
    return a*multiplier;
}
const num = multiplyByTen(5);
console.log(num); //50
```

3. Functions: Scope

Variables outside of any functions are **global**; they can be accessed inside any functions;

Variables created inside functions are **local** to that function--they can be accessed inside that function, but not outside;

Arguments are local to functions.

3. Functions: Scope

```
var sayHello = function(name){
   var greeting = “Hello”;
   console.log(greeting + “, ” + name);
}
console.log(name); //???
console.log(sayHello); //???
```

3. Functions: Scope

Local scopes are nested i.e. local variables (including paramters) within the “parent” function are accessible from any “child” functions contained within the parent, but NOT vice versa.

One more question: what happens to local variables when the function that created them is no longer active?

3. Functions: Closure

```
function wrapValue(n){
   return function(){
      console.log(n);
   }
}

var wrap1 = wrapValue(1);
```

3. Functions: Closure

```
function wrapValue(n){
   var localVar = n;
   return function(){
      console.log(localVar);
   }
}

var wrap1 = wrapValue(1);
wrap1(); //1
```

What is `wrap1`?

Following the code, we know

```
wrap1 = function(){
   console.log(1);
}
```

3. Functions: Closure

```
function wrapValue(n){
   var localVar = n;
   return function(){
      console.log(localVar);
   }
}

var wrap1 = wrapValue(1);
wrap1(); //1
var wrap2 = wrapValue(2);
wrap2(); //2
```

What is `wrap2`?

Local variables are re-created each time a function is called. Therefore, localVar = 2;

Following the code, we know

```
wrap2 = function(){
   console.log(2);
}
```

3. Functions: Closure

When a function “closes over” a local variable, this property is called **closure**.

In-Class Exercise 3: Write a Function

Strings values are in fact a JavaScript object, with properties like `.length` and methods like `.charAt();`

Let's write a function that counts the number of occurrence of a certain character in a string:

```
function numCharInString (string, character)
{
   //something here
}
```

...so that `numCharInString("JavaScript", "a")` returns 2

INTRODUCTION TO LIBRARIES

What we will be doing is in fact much more complicated than the examples shown so far; as examples, we will

- Add, remove, and manipulate elements dynamically;
- Import data from local files or remote servers;
- Listen to and handle user interactions or "events" (mouseclicks, drag, scroll etc.)

We will use **libraries** to accomplish these tasks much more quickly and easily.

Examples: JQuery AND D3

JQuery

- The world's most popular JavaScript library;
- Allows us to access HTML elements (DOM elements) using CSS selectors, and manipulate them;
- We'll use it extensively to handle key user interactions.

D3

- "Data Driven Documents";
- "Bring data to life using HTML, SVG and CSS"

Examples: JQuery AND D3

Observe a typical JQuery statement:

```
>> $(“.container”).addClass(“content”);
>> var width = $(“.container”).width();
>> console.log(width);
```

Examples: JQuery AND D3

The expression `$([CSS selector])` allows us to access one or more DOM elements:

```
>> $(".container").addClass("content");
```

Once we have access to these elements via JQuery, we can use any number of JQuery methods, such as those that change their CSS properties:

```
>> $(".container").css({background: "#000"
});
```

...or return information about the element:

Examples: JQuery AND D3

Finally, observe a typical statement in D3:

```
d3.select(“.container”)
    .append(“div”)
    .attr(“class”, “new-section”)
    .style(“width”, “100px”);
```

WHAT WE HAVE LEARNED

Baby steps with JavaScript

1. Values and operators
2. Program structure
3. Writing functions
4. Testing the waters with D3

Next week we'll examine **arrays** and **objects** in detail, and dive into a practical problem with D3.

Please review these concepts:

1. What is “type coersion” in JavaScript? What is an example?
2. What is the difference between `const` and `let`?
3. Can you use a variabe in a program before it’s defined? If so, how?
4. What is the syntax for an `Object`? An `Array`?
5. What is the syntax for `if...else` conditional statements?
6. What is the syntax for a `for...` loop?
7. Why use functions?
8. What are function arguments?
9. How do we retrieve values from a function?
10. Explain the idea of function “scope”.