# A JAVASCRIPT CRASHCOURSE

## + DOM MANIPULATION

# 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
    - Operators
    - Creating and assigning value to variables
    - JavaScript objects
2. Program structure
    - Conditional (`if...else if...`)
    - `for...` loop
3. Writing functions
4. Testing the waters with D3

# I. VALUES AND OPERATORS

Numbers and arithmetics:

**numbers** `3.43  2.9e6  0`

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

**special** `NaN  Infinity`

Use brackets to specify order of operation

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

# LET'S TRY THIS OUT

Use console or repl.it

# I. VALUES: 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 !”
```

# I. VALUES: 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?
```

# I. VALUES: 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) //??
```

# I. THE SPECIAL CASE OF FALSEY 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
```

# I. THE SPECIAL CASE OF FALSEY VALUES

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

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

# STATEMENTS: "DO SOMETHING WITH VALUES"

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:

```
var age = 28;
var daysPerYear = 365, monthsPerYear = 12;
```

# DECLARING VARIABLES, AND ASSIGNING VALUES

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:

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

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*.

# OBJECTS

**Objects** are a method of abstracting, and encapsulating values

They contain **properties** (which are just values) and **methods** (functions, or the ability to do something).

Objects are always wrapped by a pair of **curly braces**.

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

What can you do with it?

# OBJECTS

1. You can explicitly set properties like this:

```
var someObject = {
      propertyName1: “Property value”,
      propertyName2: 34
   };
```

2. You can access individual properties using the dot notation

```
console.log(someObject.propertyName2);
```

3. You can set up new properties on the fly

```
someObject.propertyName3 = “A new property!”;
```

# OBJECTS

```
var newCar = {

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

# OBJECTS AS DATA STRUCTURE

```
>>console.log(newCar.color); //Silver
>>
>>newCar.owner = “Siqi”;
>>console.log(newCar.owner); //Siqi
```

Objects are incredibly useful because they provide a way to **structure data**.

Combined with **arrays**, they are a basis for data manipulation later on in the course.

Think about how our class can be represented in this structure.

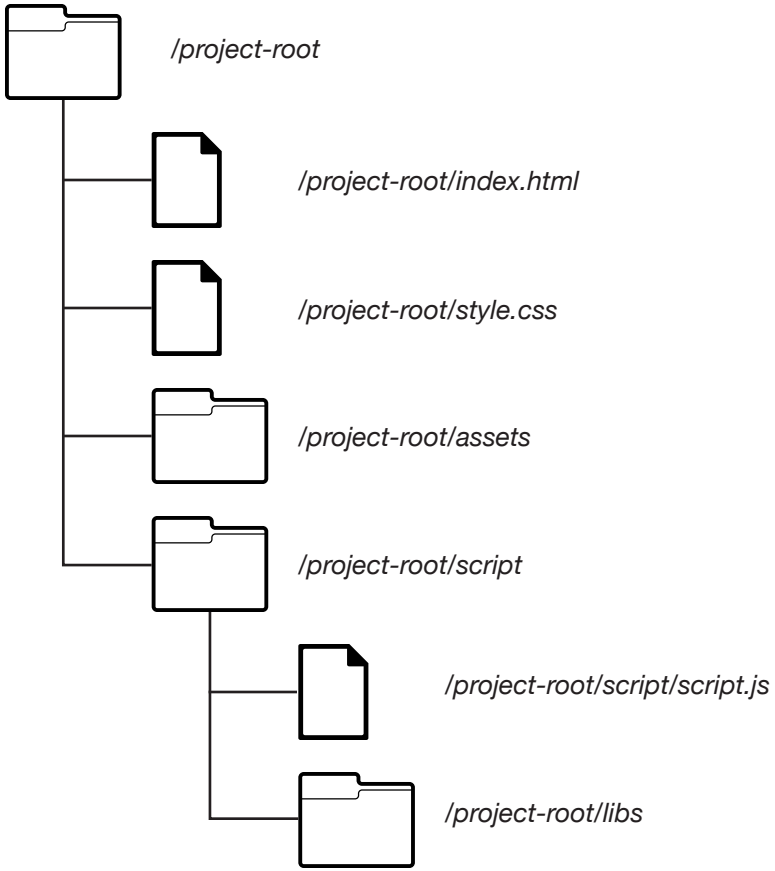
# In-Class Exercise 1: Playing with Objects and Values

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

# II PROGRAM STRUCTURE

But when and in what order are statements run?

# II PROGRAM STRUCTURE



# II PROGRAM STRUCTURE

*/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>
</html>
```

*/project-root/script/script.js*

```
//script.js
```

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

# II PROGRAM 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`)

# II PROGRAM 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");
   }
```

# II PROGRAM 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{
}
```

# II PROGRAM 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”);
}
```

# II PROGRAM STRUCTURE: FOR 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

# II PROGRAM STRUCTURE: WHILE LOOP

1. Create an initial condition
2. Run statements repeated until intial condition is no longer true

```
var counter = 0;
while(counter < 1000){
    //run statements here
    counter += 1;
}
```

# In-Class Exercise 2: Simulations

Math.random() generates a pseudo-random number between 0 and 1. For example:

```
var someNum = Math.random(); //anywhere between 0 and 1
```

But how truly random is Math.random()? Let's run a simulation to find out.

# MOVING TO MORE COMPLEX PROGRAMS...

How can we structure larger, more complex programs?

How do we deal with and take advantage of repitition?

Think of real-world analogies.

# III. FUNCTIONS: BASICS

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:

```
var 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
```

# ASIDE: FUNCTION AS PART OF AN OBJECT

```
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”);
    }
}
```

# III. 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
```

# III. FUNCTIONS: ARGUMENTS

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
```

# III. FUNCTIONS: SCOPE

Another example:

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

**variable names for arguments are arbitrary!

# III. FUNCTIONS: SCOPE

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

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

Arguments are local to functions.

# III. FUNCTIONS: SCOPE

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

# III. 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?

# III. FUNCTIONS: CLOSURE

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

var wrap1 = wrapValue(1);
```

# III. 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);
}
```

# III. 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);
}
```

# III. 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:

```
>> var height = $(“.container”).height();
```

# 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.