Learning JavaScript

- Developed by **Brendan Eich** as LiveScript.

JavaScript Where To

- In HTML, JavaScript code is inserted between **<script></script>** tags:

```
<!DOCTYPE html>
<html>
    <body>
        <script src=""></script>
    </body>
</html>
```

- Scripts can be placed in the **body** or in the **head**.

JavaScript Output

- JavaScript can “display” data in different ways:
    1. Writing into an HTML element: **innerHTML (needs an ID)**
    2. Writing into the HTML output: **document.write() (does not need an ID)**
    3. Writing into an alert box: **alert()**
    4. Writing into the browser console: **console.log()**

JavaScript Variables

- There are 5 primitive types: Number, String, Boolean, Undefined, or Null.
    - Strings are immutable.
    - 0, -0, “”, false, undefined, NaN are **always false.**
    - If we have Boolean(“false”), it is true, since it is not an empty string.
- There are 3 wrapper objects: Number, String, and Boolean.
- Variables are declared using different keywords:
    1. nothing.
    2. var: global, re-declarable, can be updated.
    3. let: local, not re-declarable, can be updated, must be declared before use.
    4. const: local, not re-declarable, cannot be updated, must be declared before use.

This will not work:
let carName = “Volvo”;
ler carName;

JavaScript Let

- Block scope:

```
{
  let x = 2;
}
// x can NOT be used here
```

- Must be declared before use.
- Cannot be redeclared in the same scope:

```
let x = "John Doe";

let x = 0;
// this cannot be done
```

JavaScript Const

- Must be assigned a value when they are declared:

```
const PI = 3.14159265359;
```

- Block scope.
- Cannot be redeclared:

```
{
const x = 2;   // Allowed
const x = 2;   // Not allowed
}
```

- Cannot be reassigned:

```
const PI = 3.141592653589793;
PI = 3.14;          // This will give an error
PI = PI + 10;   // This will also give an error
```

JavaScript Operators

ADDITION	COMPARISON	MULTIPLICATION
+ is addition if and only if both operands are numbers.	1. If one operand is a number, and the other can be converted to a number, < is a number comparison. 2. If one operand is number, and the other cannot be converted to a number, false all the time. 3. If two operands are string, < is a string comparison (check ASCII table).	1. If operands are numbers, or all can be converted to a number, * is a number multiplication. 2. If one operand is a number, and the other cannot be converted to a number, NaN all the time.
1+2 = 3 "1"+2="12" 1+"2"="12" "1"+"2"="12" 1+"bird"="1bird" 1+2+"birds"="3birds"	11 < 2: false "11" < 2: false 11 < "2": false "11" < "2": true (check ASCII table) 11 < "bird": false	11 * 2 = 22 "11" * 2 = 22 11 * "2bird" = NaN

JavaScript Logical Assignment Operators

&&

- Evaluates expressions from left to right and returns the first falsy value encountered or the value of the last expression if all are truthy. It stops evaluating as soon as it hits a falsy value.

```
console.log(true && false); // false, because the second operand is false
console.log(true && "hello"); // "hello", because both operands are truthy, returns the last value
console.log("hi" && 123); // 123, because both operands are truthy, returns the last value
console.log(0 && "nope"); // 0, because the first operand is falsy, returns the first falsy value
```

||

- Evaluates expressions from left to right and returns the first truthy value encountered or the value of the last expression if all are falsy. It stops evaluating as soon as it hits a truthy value.

```
console.log(false || true); // true, because the second operand is true
console.log(false || ""); // "", because all operands are falsy, returns the last value
console.log("hello" || 0); // "hello", because the first operand is truthy, stops and returns the first truthy value console.log(null || "default"); // "default", because the first operand is falsy, continues and returns the next truthy value
```

!: logical NOT

```
console.log(!true); // false
console.log(!false); // true
console.log(!0); // true, because 0 is falsy
console.log(!""); // true, because an empty string is falsy
console.log(!"hello"); // false, because non-empty strings are truthy
console.log(!null); // true, because null is falsy
```

!!: DOUBLE NOT

```
console.log(!!true); // true
console.log(!!false); // false
console.log(!!0); // false, because 0 is falsy
console.log(!!""); // false, because an empty string is falsy
console.log(!!"hello"); // true, because non-empty strings are truthy
console.log(!!null); // false, because null is falsy
console.log(!!123); // true, because numbers other than 0 and -0 are truthy
```

JavaScript Equal and Not Equal

- == and != can do type coercion.
- === and !== cannot do type coercion.

```
"3" == 3: true
"3" === 3: false
```

```
var a = "123";
var b = "123";
if(a==b) {document.write('yes');}
      else {document.write('no');}
```

- What are the outputs?
  A. yes
  B. no
  C. error
  D. nothing

Answer:A

```
if(3!== "3") {document.write('yes');}
      else {document.write('no');}
```

- What are the outputs?
  A. yes
  B. no
  C. error
  D. nothing

Answer:A

```
var a = new String("123");
var b = new String("123");
if(a==b) {document.write('yes');}
      else {document.write('no');}
```

not equal because we are comparing the reference of each object

- What are the outputs?
  A. yes
  B. no
  C. error
  D. nothing

Answer:B

```
var a = String("123");
var b = new String("123");
if(a==b) {document.write('yes');}
      else {document.write('no');}
```

we are using cohersion because variables are not of the same type

- What are the outputs?
  A. yes
  B. no
  C. error
  D. nothing

Answer:A

JavaScript Switch

- Used to perform different actions based on different conditions.

```
switch (expression) {
      case value_1:
         // value_1 statements
      case value_2:
         // value_2 statements
      …
      [default:
         // default statements]
  }
```

JavaScript Control Statements

while: creates a loop that is executed while a condition is true

```
while (condition) {
  code block to be executed
}
```

for: defines a code block that is executed as long as a condition is true

```
for (statement 1; statement 2; statement 3) {
  code block to be executed
}
```

do…while: used when you want to run a code block at least one time

```
do {
  code block to be executed
}
while (condition);
```

JavaScript Objects

- Use **const** when you declare:
    1. A new array.
    2. A new object.
    3. A new function.
    4. A new RegExp.

- Creation of an object (the new object has no properties). Here it uses **var** but we can use **const** (the reference cannot be changed, that is, myCar always references the same object, but what is inside **can be changed**).
- **ALWAYS USE THE WORD "new" WHEN DEFINING AN OBJECT:**

```
var myObject = new Object();

/*Properties can be added anytime*/
myCar.make = "Ford";
myCar.model = "Focus";

/*Properties can be accessed by dot notation or in array notation*/
var property1 = myCar["model"];
var property1 = myCar.model;

/*This is another way of making an object*/
var myCar = {make: "Ford", model: "Contour SVT"};
```

- As said above, we cannot reassign an object:

```
const car = {type:"Fiat", model:"500", color:"white"};
car = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

JavaScript Array

- Again, arrays are created using **const**, but **cannot be reassigned**:

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element: cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");

// But:
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"]; // ERROR
```

- There are several array functions that we can use:
  - join
  - reverse
  - sort: by default converts the elements of the array to strings and compares their sequences of UTF-16 code units values.
  - concat
  - slice

```
let fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
let citrus = fruits.slice(1, 3);

// citrus contains ['Orange','Lemon']
// fruits remains unchanged
```

  - toString
  - push, pop, unshift, shift

JavaScript Function

```
function function_name([formal_parameters]) {
        - body -
        }
```

- No return? undefined is returned.
- return has no parameter? undefined is returned.

- Functions are objects.
- Functions are defined in the head of the HTML file.

Anonymous Function

```
var f = function(x,y) {return x+y;}
f(1,2);
```

JavaScript Object.prototype

- Use it to change the template of the object.
- It affects **all the objects of the same type**.

```
function plane(newMake, newModel, newYear){
    this.make = newMake;
    this.model = newModel;
    this.year = newYear;
}
```

```
    var myPlane = new plane("Cessna", "Centurnian", "1970");

    for(var name in myPlane){
      document.write(name +": "+myPlane[name] + "<br/>");
    }
    document.write("<br/>");


    plane.prototype.price = 1000;

    for(var name in myPlane){
      document.write(name +": "+myPlane[name] + "<br/>");
    }
   //Now the price will be written too, because now all objects
created from “plane” have the property price.
```

JavaScript Class

```
class Car {
  constructor(name, year) {
          this.name = name;
          this.year = year;
  }
}
```

- Classes are **NOT OBJECTS**.
- Beware of inheritance:
    1. “extends”: inherit all methods from another class.
    2. “super”: call the parent’s constructor function.

```
<script>
            class Car{
                constructor(brand){
                    this.carname = brand;
                }
                present(){
                    return 'I have a ' + this.carname;
                }
            }
            class Model extends Car{
                constructor(brand, mod){
                    super(brand);
                    this.model = mod;
```

```
                }
                show() {
                    return this.present() + ', it is a ' +
this.model;
                }
            }
            let myCar = new Model("Ford", "Mustang");
            document.getElementById("demo").innerHTML =
myCar.show();
        </script>
```

JavaScript Pattern Matching

- Allows you to check a value against a pattern:

```
var str = “Gluckenheimer”;
var position = str.search(/heim/); → position is now 7
var matchStr = str.match(/heim/); → matchStr is now “heim”
```

- Character classes: special notations that match any symbol from a certain set of characters.
  - \d: Matches any digit (0-9).
  - \D: Matches any non-digit character.
  - \w: Matches any word character (alphanumeric character plus underscore).
  - \W: Matches any non-word character.
  - \s: Matches any whitespace character (spaces, tabs, line breaks).
  - \S: Matches any non-whitespace character.
  - Any one of which matches: [a-z]
  - Circumflex → the negation: [^0-9]
  - Period “.”: Matches any character except for a newline.

    1. /\d\.\d\d/ → Match any digit 0-9, match the period “.”, match two digits in succession (1.23).
    2. /\D\d\D/ → Match any non-digit character, match any digit, match any non-digit character (a1b, !3?).
    3. /\w\w\w/ → Match any 3 word characters (abc, 123, a1b, 1_c…).

- Quantifiers:
  - {n}: Exactly n repetitions.
  - {m,}: At least m repetitions.
  - {m,n}: At least m but not more than n repetitions
  - *: Zero or more repetitions.
  - +: One or more repetitions.
  - ?: Zero or one.

- Anchors:
  - ^: ensuring that the pattern must match from the very beginning of the string.
  - $: ensuring that the pattern must match at the very end.

    /^Lee/
    - Matches any string that starts with “Lee”.
      - **Lee Ann** matches, because the string starts with Lee.
      - **Mary Lee Ann** does not match, because the string does not start with Lee.

    /red$/
    - Matches any string that ends with “red”.
      - **A red car** does not match, because the string does not end with red.
      - **This car is red** matches, because the string ends with red.

    /^Lee/ vs /[^Lee]/
    - The former matches strings that start with “Lee”.
    - The latter negates the character set inside [], so it matches a single character that is not L, e, or e.

- Modifiers:
  - i: ignore the case of letters
    - /oak/i → matches OAK and Oak…
  - g: global matching
    - Indicates that the replacement should happen globally across the entire string.

- Pattern matching methods of *String*:
  1. string.replace(regexp, newstring)

     var str = “Some rabbits are rabid”;
     str.replace(/rab/g, “tim”); → str = Some timbits are timid

  2. string.match(regexp)

     var str = “"My 3 kings beat your 2 aces”;
     var matches = str.match(/\d/g);

  3. string.split(separator, limit)

     var str = “"grapes:apples:oranges”;
     var matches = str.split(“:”);

  4. string.search(regexp)

     var num = “123-4567”;
     var ok = num.search(/^\d{3}-\d{4}$/);
     → Matches strings that start with exactly three digits, followed by a hyphen, and then exactly four more digits, with no other characters before, in between, or after.

JavaScript Document Object Model (DOM)

- Window object: The window in which the browser displays documents.
    - Properties:
        - document: a reference to the Document object that the window displays.
- Document object:
    - Properties:
        - forms: an array of references to the forms of the document.
        - anchors
        - links
        - images…
- Element access in JavaScript:

```
<form name = “myForm” action = “">
   <input type=”button” name=”pushMe”>
</form>
```

    - There are two ways of accessing this:
        1. document.forms[0].elements[0]
        2. document.myForm.pushMe

    - There is an easy way of checking if the checkboxes are checked:

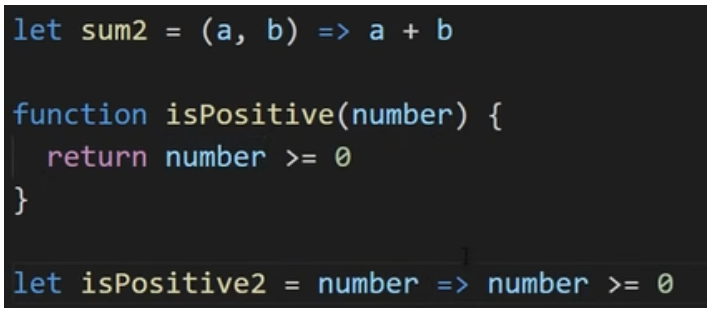
```
<form id = “topGroup”>
   <input type=”checkbox” name=”toppings”
            value = “olives”/>
   <input type=”checkbox” name=”toppings”
            value = “tomatoes”/>
</form>

let numChecked = 0;
const dom = document.getElementById(“topGroup”);
for (index=0; index < dom.toppings.length; index++)
     if (dom.toppings[index].checked)
        numChecked++;
```

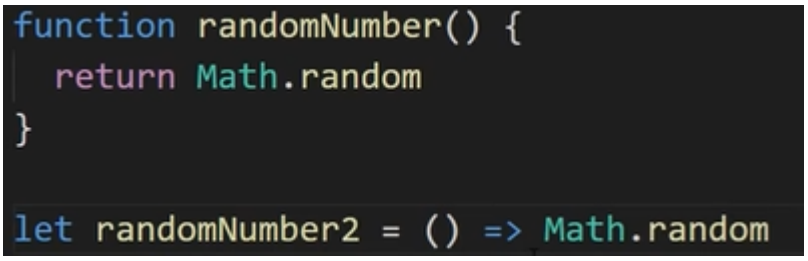
- There are also different ways of finding HTML elements:
    - getElementById()
    - getElementsByTagName() → always returns an array
    - getElementsByClassName(name)

JavaScript Arrow Function

- **Do not** have the name “function”.
- We set it as a **variable**, because normal functions by themself already create a variable.
- Then, we simply **add an arrow** “=>”.
- We can even remove the return and the brackets:



- If we were to have no parameters, we simply leave “()” to indicate that there are no parameters:



- We can return an o**bject literal enclosing in parenthesis**:

```
const createPerson = (name, age) => ({
  name: name,
  age: age
});

console.log(createPerson("Alice", 30)); //
Output: { name: 'Alice', age: 30 }
```

- We can use arrow functions as callbacks for methods:

```
const numbers = [1, 2, 3, 4];
const squares = numbers.map(x => x * x);

console.log(squares); // Output: [1, 4, 9,
16]
```

- Arrow functions **do not bind their own "this"** → They inherit it from the parent scope at the time they are defined.

```
class Button {
  constructor(label) {
    this.label = label;
    document.addEventListener('click', () =>
{
      console.log(this.label);
    });
  }
}

const myButton = new Button('Click Me!');
// When the document is clicked, "Click Me!"
will be logged to the console.
```