Chapter 4

JavaScript

What is in This Chapter ?

This chapter will introduce us to the **JavaScript** language. We will learn how to write our first **JavaScript** program and how to run it within **node.js** and within a **browser**. We will cover the language basics including **variables**, **types**, **control structures**, **strings**, **objects** & **functions**. We will also discus **copying** of objects and **hoisting** as it related to **scope**. We end the chapter with a discussion of **higher-order functions, synchronous vs. asynchronous** code and **closures**.

4.1 The Basics

By now, we should have a decent grasp of how to create webpage content and style it to display in a more visually-appealing way. We are likely also aware that laying out a webpage in a pleasant manner does require us to have a little bit of an artistic flair. So, it will take a while before we get good at it.

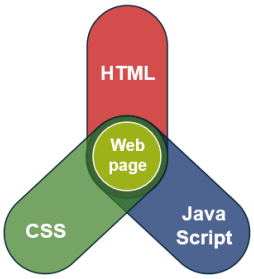
But there are limits to what we can do with just **HTML** and **CSS** because they are mainly used for static structure and styling of webpages. They are quite limited in terms of user interaction capability. Here are the limited interactive abilities with just pure **HTML** and **CSS**:

- **Basic Form Interactions**
  - Input textfields, checkboxes, radio buttons, dropdowns
  - Submitting forms (although processing still needs **JavaScript** or a backend)
  - **HTML5** validations (e.g., `required`, `pattern`, `min`, `max`)

- **Visual Interactions**
  - Hover effects (e.g., tooltips, color changes)
  - Focus styles (for accessibility/navigation)
  - Transitions and animations (**CSS** `transition` and `@keyframes`)
  - Responsive design via media queries (to adapt to various screen sizes)

- **Toggle Interfaces**
  - Using checkboxes or radio buttons combined with **CSS**:
  - Toggle menus
  - Tabs
  - Accordions (expand/collapse sections)
  - Modals/Popups (with limitations)

To do anything more than these basic interactions, we need to use something else. As it turns out, the third and last attribute for web design is **JavaScript**, which enables interactive, dynamic and animated web pages. **JavaScript** is mostly used for client-side webpage behavior, although it is also used for server-side programming as well.



Although their names sound similar, **Java** and **JavaScript** are very different in purpose, design, and use. **Java** is like a full-featured “power tool” for building big systems, while **JavaScript** is a nimble “multitool” for making websites interactive.

Here is a table explaining the differences between them:

	 **Java**	 **JavaScript**
Type	compiled into **.class** files	interpreted instantly by browser
Purpose	general-purpose programming	web scripting language
Platform	Java Virtual Machine (**JVM**)	web browser or (**Node.js**)
Language Type	strongly typed, object-oriented	loosely typed, prototype-based
Typical Use Cases	android apps, desktop s/w, large backend systems	interactive websites, front-end logic

JavaScript may seem simpler to use, but there are some downsides because it has more "dangers" or pitfalls, especially for beginners or in large codebases, due to its looser rules, dynamic nature, and inconsistent history. Here are some of the dangers:

- ✖ **Loose typing**
  - **JavaScript** doesn't enforce types, so errors can go unnoticed until runtime
- ✖ **Silent Type Coercion**
  - **JavaScript** auto-converts types unpredictably (e.g., "5" - 2 = 3, "5" + 2 = "52")
- ✖ **No Compile-Time Checking**
  - Many bugs only show up at runtime
- ✖ **Global Scope Pollution**
  - Variables can unintentionally be global if not declared properly
- ✖ **Inconsistent Browser Behavior**
  - Not all browsers support the same **JavaScript** features

The bottom line…

JavaScript gives us a lot of flexibility, but with great power comes great responsibility.

Java enforces discipline through its strict design, while **JavaScript** requires self-discipline and good tooling to ensure maintainability and safety.

So be careful and stay organized !!!

4.2 Running JavaScript Code

How do we go about writing our first **JavaScript** program? Well, the “hello world” of **JavaScript** can be done by display to the console or a pop-up window. So, we can use one of these methods:

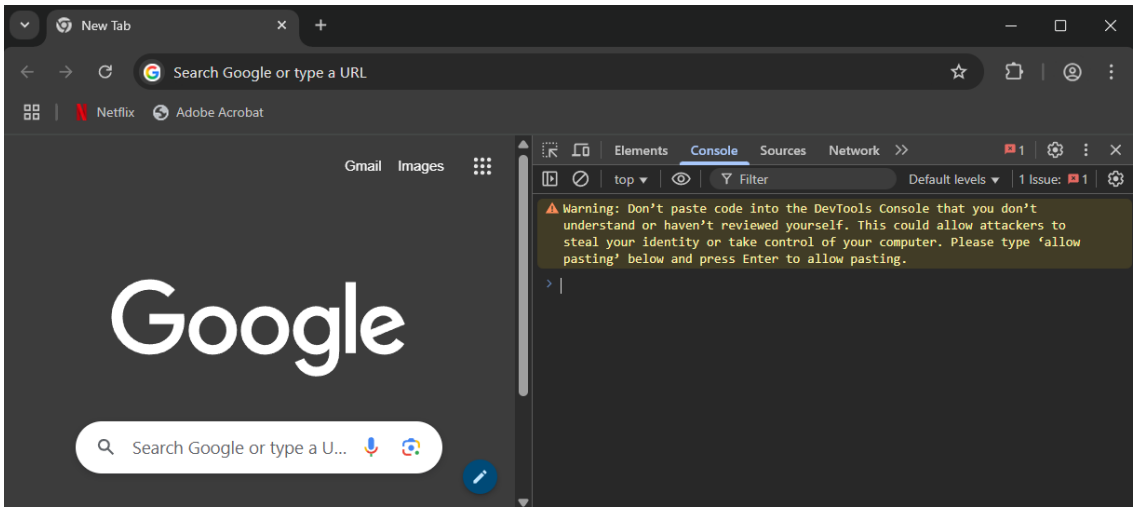
```
console.log("Hello World!");
```

or

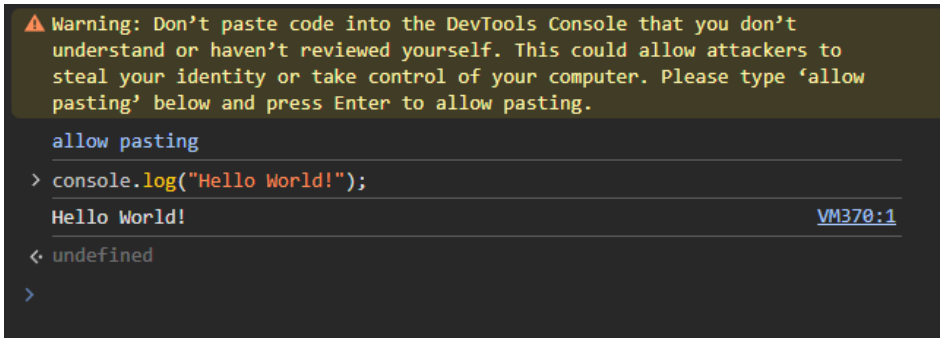
```
alert("Hi Everyone!");
```

But where do we type that? The simplest way to try this out is in the **Google Chrome Browser** itself. Follow these steps:

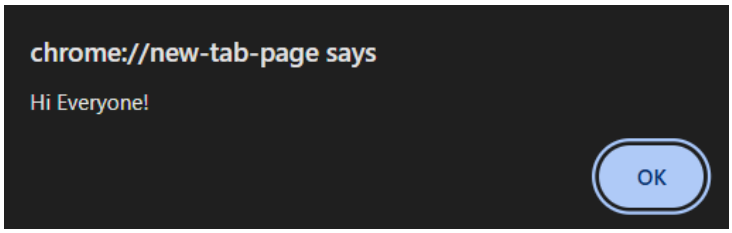
1. Open Chrome.
2. Press `Ctrl+Shift+I` to start the **DevTools** environment.
3. Click on the **Console** tab.
4. Copy `console.log("Hello World!");` from above and paste it into the console.
5. We will get a warning message about pasting (the first time):



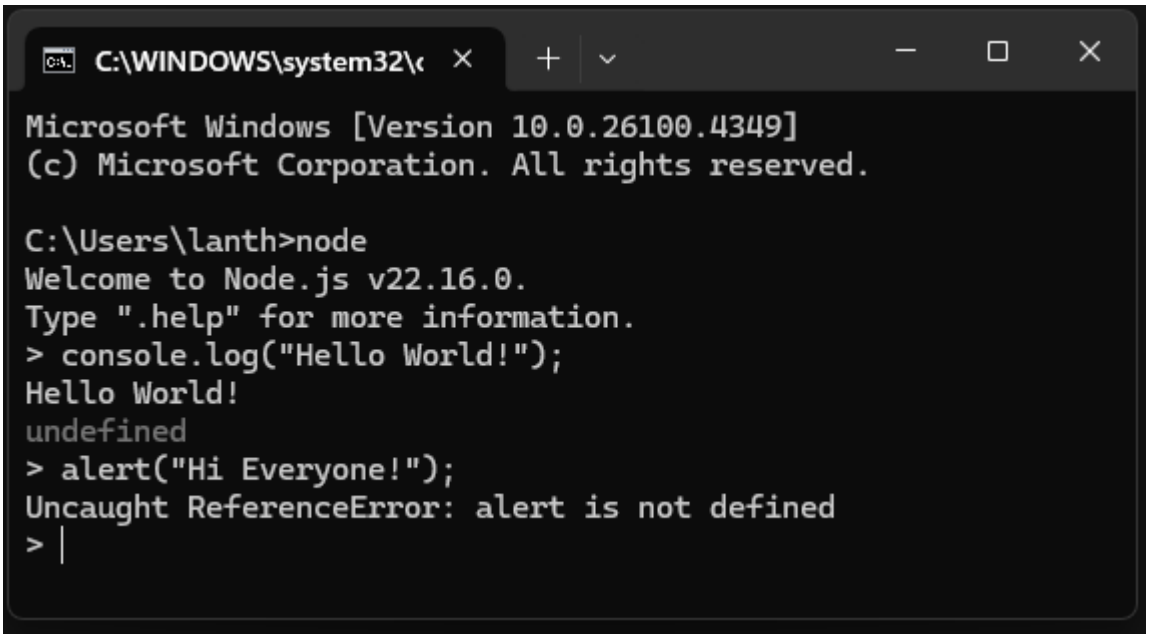
6. Type **allow pasting** and press **Enter**.
7. Then re-paste the code and press **Enter**. We should see the output as shown here:



Using the `alert("Hi Everyone!");` code will result in this popup window appearing:



We can also run our code in **node.js** … which must be installed on our system. We can go to https://nodejs.org. to download and install it for Windows, MacOS, Linux or AIX. Once installed, we can open up a console window to test things in **node.js**. On a Windows desktop, for example, we can open the command window (**WindowsKey+r** then type **cmd**). Then once in the command window, type **node** into the console. We will then be able to cut/paste in our code:



We will quickly see that some things don't work, such as the **alert** window that we used in the Chrome console. We can type **.exit** to leave the **node.js** session.

Some of the more advanced interactive web pages (e.g., a browser-based video game) may involve thousands of **JavaScript** statements. That is why **JavaScript** programs are commonly written in a separate file, typically having the **.js** file extension and then these are either:

1. Run in **node.js** or
2. Linked into an **HTML** file using the **<script> </script>** tag wrapper.

Open **vscode** (or some other editor) and create a file called **console-hello-world.js** and then just put the single line of `console.log("Hello World!");` code in there and save it somewhere.

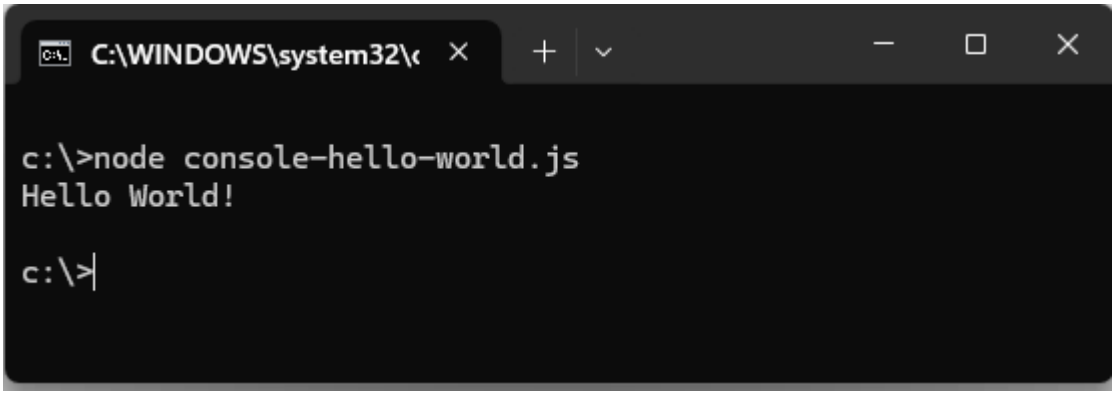
If we still have our command console window open, navigate to the directory that contains this new file that we created. In windows, we can use these simple commands to navigate:

- **`dir`** – to display the contents of the current directory
- **`cd <folder>`** – to go into a subdirectory/subfolder named **`<folder>`** (which may be an absolute path (e.g., **`cd c:\`**)) of the current directory
- **`cd ..`** – to back up to the parent directory from the current directory
- The **TAB** key (one or more times) to complete the name of a directory or cycle through the ones that partially matches what we typed so far.
- **`cls`** – to clear the console

Once we get to the directory that contains the **console-hello-world.js** file, type the following into the console:

```
node console-hello-world.js
```

Now it will load and run the file using **node.js**. This process is similar to running a java program in the console window. We will see the output:

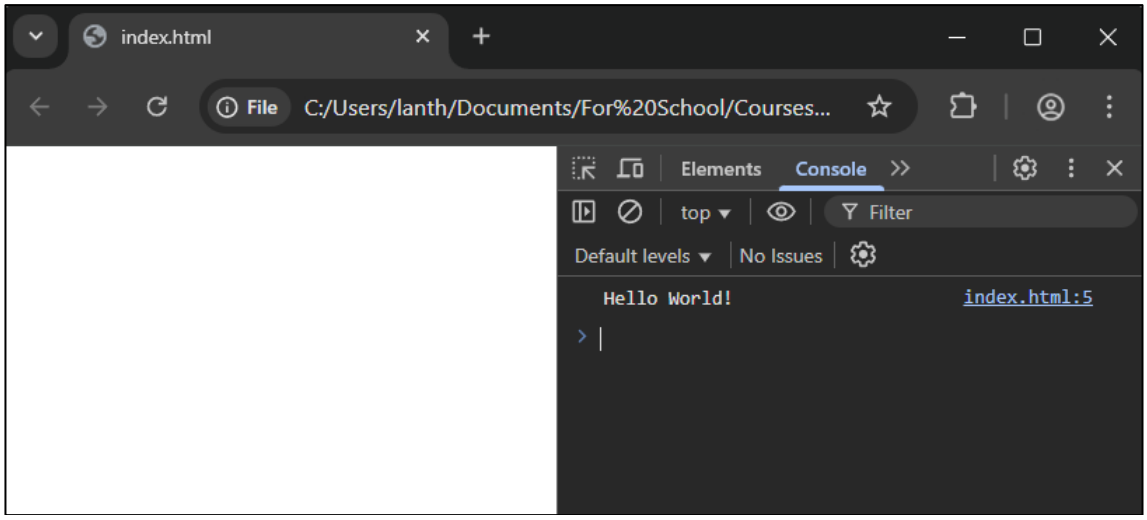


There is one more option for us … putting our code into an **HTML** file's **<`script`>** tag wrapper.

Here is the most basic **HTML5**-compliant code (saved as **index.html**) that displays the usual "Hello World":

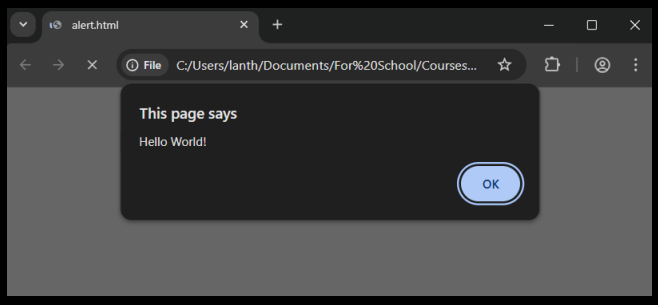
```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <script>
            console.log("Hello World!");
        </script>
    </body>
</html>
```

We can open this **HTML** file in a browser but the page will not show anything ☹. However, if we press **`Ctrl`**+**`shift`**+**`I`** to open the **DevTools** environment, we will see the output:



Of course, if we used the alert code, then we do not need to go to the **DevTools** environment to see the results. Here is a file called **alert.html**:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <script>
            alert("Hello World!");
        </script>
    </body>
</html>
```



We can also adjust our **HTML** code so that it loads the code from a **JavaScript** file. Assume that we made this one-line **JavaScript** file and saved it as **alert.js**:

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

Now, alter the **alert.html** file as shown below and save it as a file called **alert2.html** in the same directory as **alert.js**:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <script src="alert.js"></script>
    </body>
</html>
```

The result is the same. This tells us that we can write lots of code in a **JavaScript** file and then include it in a single line as shown above.

The best practice for including **JavaScript** code is to place all scripts just before the closing **</body>** tag. This will ensure that the **HTML** content loads first, so the script doesn't block/delay the loading of the webpage. Nobody likes a visually-slow loading webpage!

Another way of doing this is to place the script within the **<head>** tag wrapper but add the word **defer** to instruct the script to be loaded after the rendering of the content. Here is **alert3.html**:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="UTF-8">
        <script src="alert.js" defer></script>
    </head>
    <body>
    </body>
</html>
```

Putting the script in the **<head>** tag wrapper, along with **defer** is actually preferred:

- ✓ Keeps all scripts organized in the **<head>**
- ✓ Loads in parallel, doesn't delay rendering
- ✓ Allows managing multiple scripts without caring about **HTML** placement

4.3 Variables, Constants, Types and Conversion

In the next few sections, we will discuss the basics of the **JavaScript** language one topic at a time.

Variables are declared by using one of these keywords:

- **var** – for declaring variables with no value (not a good idea)
  - `var x;`
  - `var initialStartAmount;  // use camelCase names in JavaScript`
- **let** – for declaring a variable with an initial value
  - `let count = 10;`
  - `let cost = 947.65;`
  - `let name = "unknown";`
  - `let total = cost * count;`

Constants use **const** and must be given an initial value, but cannot be re-assigned a value:

  - `const weekDays = 5;`

Notice that variables do NOT have a type associated with them when declared, which is quite different from JAVA. That is because **JavaScript** is dynamically typed, so we can put anything into a variable:

```
let status = true;   // status starts off as a boolean
status = "invalid"; // status is now holding a string
status = 34.5;       // status is now holding a number
status = -7;         // status is still holding a number
```

Even though we do not declare the types for a variable, there are **7** primitive types of data that are used in **JavaScript**:

- **number** - any numeric value (integer or floating-point)

```
42, 3.14, -7
Infinity, -Infinity, NaN    // symbolic number values
438e4                       // 4380000
438e-4                      // 0.0438
```

- **string** - textual data (different from JAVA in that it is a primitive)

```
'hello'                     // single quotes is NOT a char
"world"                     // use double quotes
'contains "another" one'    // can use interior double quotes
"use \"backslash\" too"     // similar to JAVA
```

- **boolean** - logical value true or false

```
true, false
```

- **null** - intentional absence of any value

```
null
```

- **undefined** - a variable declared but not assigned a value

```
undefined
```

- **symbol** - unique, immutable identifier (rarely used directly)

```
Symbol('id')
```

- **bigInt** - for integers larger than `Number.MAX_SAFE_INTEGER`

```
123456789012345678901234 5n
```

Everything else is considered an object:

- **object** - General container for data & behavior (e.g., arrays, dates, functions, etc..)

We can use `typeof x` to ask for the type of a variable or expression `x`. The result will be a string containing the name of one of the primitive datatypes above or `"object"` or `"function"`.

Now, because we do not have types associated with variables, there is some automatic type conversion that occurs in our code that can lead to unexpected results.

For example, the standard equality operator (**==**) does not do the same thing as in JAVA when different types are involved. Type conversion can either be explicit or implicit.

Explicit:

Sometimes we want to convert types on purpose. Here are examples of converting **strings** and **booleans** to the **number** type:

```
Number("123");            // 123
Number("456.78");         // 456.78
parseInt("42px");         // 42
parseFloat("56.7'");      // 56.7
+"100";                   // 100
+"544.36";                // 544.36
Number(true);             // 1
Number(false);            // 0
Number("abc");            // NaN
```

Here are some examples of converting **numbers** and **booleans** to strings:

```
String(123);              // "123"
String(456.78);           // "456.78"
(123).toString();         // "123"
(456.78).toString();      // "456.78"
(456.78).toFixed(0);      // "457"
(456.78).toFixed(1);      // "456.8"
(456.78).toFixed(3);      // "456.780"
String(true);             // "true"
```

Here are some examples of converting **numbers** and **strings** to **booleans**:

```
Boolean(0);               // false
Boolean(1);               // true
Boolean(45.5);            // true

Boolean("");              // false (empty string)
Boolean(" ");             // true  (space is still a character)
Boolean("hello");         // true

Boolean(null);            // false
Boolean(undefined);       // false
Boolean(NaN);             // false
```

Implicit:

Implicit type conversion (also called ***type coercion***) often occurs in arithmetic operations, comparisons, and if statements etc.. Unfortunately, this can lead to unexpected behavior if we are not careful.

Here are some examples of string coercion using the **+** operator:

```
"5" + 1              // "51"  … number is coerced to string
true + "1"           // "true1"
null + "test"        // "nulltest"
```

Here are some examples of number coercion using arithmetic operators:

```
"10" - 2             // 8 … "10" becomes number
"4" * "2"            // 8 … both strings become numbers
true * 2             // 2 … true becomes 1
null + 1             // 1 … null becomes 0
"abc" - 1            // NaN … "abc" can't be a number
undefined + 1        // NaN … NaN + 1
true + true + 3      // 5 … 1 + 1 + 3
```

The tricky stuff comes when we start comparing using **==**:

```
5 == "5"             // true  … converts "5" to number 5
0 == false           // true  … converts false to 0
1 == true            // true  … the true becomes a 1
null == undefined    // true  … special case in JavaScript
NaN == NaN           // false … NaN is never equal to NaN
"0" == false         // true  … "0" becomes 0, false becomes 0
"" == false          // true  … "" becomes 0, false becomes 0
[] == false          // true  … [] becomes "", then 0
[] == ""             // true  … [] becomes ""
[1] == 1             // true  … [1] becomes "1", then number 1
{} == {}             // false … different object references
null == 0            // false … No coercion to 0
undefined == 0       // false … not converted to 0
```

Normally, we don't try to compare different types like this and we just want to check for equal values with the same types. For example, we normally don't want `5` to be the considered same as `"5"`. There is another operator with three equal signs (i.e., **===**) which will only return true if the types are the same AND the values are the same. Look at the same examples as above, but now with the **===** operator:

```
5 === "5"            // false
0 === false          // false
1 === true           // false
null === undefined   // false
NaN === NaN          // false
```

```
"0" === false        // false
[] === false         // false
[] === ""            // false
[1] === 1            // false
{} === {}            // false
null === 0           // false
undefined === 0      // false
```

The safe rule of thumb is to always use **===** unless we have a very specific reason to use **==**.

Here is a neat **JavaScript Equality Table** that we can use to help us see what is and isn't equal when it comes to **==** and **===**:

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

4.4 Strings, Arrays, Conditionals and Loops

Traditionally in JAVA, we would concatenate strings together using the **+** operator as shown on the examples on the left in the table below. However, **JavaScript** has something called ***template literals*** that uses the backtick (i.e., `) character to allow variables (or expression results) to be inserted into the string in a somewhat more readable manner. It also allows for multi-line strings. The code on the right of the table makes use of the template literals to obtain the same results:

With + Concatenation	With Template Literals
`const name = "Alice";` `const greeting = "Hello, " + name + "!";`	`const name = "Alice";` `const greeting = `Hello, ${name}!`;`
`const first = "Alice";` `const last = "Smith";` `const message = "Welcome " + first +` `" " + last + ", to the classroom.";`	`const first = "Alice";` `const last = "Smith";` `const message = `Welcome ${first} ${last},` `to the classroom.`;`
`const sum = "The total is " + (a + b);`	`const sum = `The total is ${a + b}`;`
`const poem = "Roses are red...\n" +` `"Violets are blue...\n" +` `"Backticks are nice...\n" +` `"And readable too.";`	`const poem = `Roses are red...` `Violets are blue...` `Backticks are nice ...` `And readable too.`;`

Of course, we have the usual useful String functions:

```
"hello".length                  // 5
"hello".toUpperCase()           // "HELLO"
"HELLO".toLowerCase()           // "hello"
```

```
"JavaScript".includes("Script")          // true
"frontend".startsWith("front")           // true
"backend".endsWith("end")                // true
"banana".indexOf("a")                    // 1
"banana".lastIndexOf("a")                // 5
"JavaScript".slice(0, 4)                 // "Java"
"JavaScript".substring(4, 10)            // "Script"
"  padded  ".trim()                      // "padded"
"  padded  ".trimStart()                 // "padded  "
"Hello world".replace("world", "there")  // "Hello there"
"aaa".replaceAll("a", "b")               // "bbb"
"one,two,three".split(",")               // ["one", "two", "three"]
"Hello".concat(" ", "world", "!")        // "Hello world!"
"cat".charAt(1)                          // "a"  … there are no char types
"cat".charCodeAt(1)                      // 97 … gives ASCII code
```

Now let's look at arrays. Unlike the C language, arrays are not fixed size, they are dynamically sized. That means that they work more like ArrayLists in JAVA, but maintain a C-like syntax.

Literal arrays are created by using the **[ ]** brackets:

```
let utensils = ["fork", "spoon", "knife"];
let empty = [];   // empty array
let numbers = [1, 2, 3];
let mixed = ["bot", 42, false];
```

We can access and modify array elements as usual:

```
numbers[2];          // returns 3
numbers[3] = 7;      // array now [1, 2, 7]
numbers[8] = 5;      // array now [1, 2, 7, <5 empty items>, 5]
numbers[9];          // returns undefined
```

Notice that **JavaScript** lets us go beyond the array boundaries … it just leaves gaps of empty items in between. It also lets us access beyond the boundaries … simply giving us `undefined` as the value there.

There are some useful array functions:

```
utensils.length;              // returns num items in array = 3
utensils.sort();              // sorts the array ascending order
utensils.reverse();           // reverses the items of the array
utensils.push("whisk");       // adds "whisk" to the end
utensils.pop();               // removes "whisk" from the end
utensils.shift();             // removes "fork" from the front
```

```
utensils.unshift("tongs");     // adds "tongs" to the front
utensils.includes("knife");    // checks if "knife" is in there
utensils.indexOf("spoon");     // returns its index (starts from 0)
```

There are some other useful functions. The **slice()** function returns a new array with the items from the "start index" to an "ending index minus 1":

```
let utensils = ["tongs", "spoon", "knife"];
let firstTwo = utensils.slice(0, 2);   // ["tongs", "spoon"]
```

The **splice()** function replaces the items from a starting index to an ending index with a new value:

```
utensils.splice(1, 1, "ladle");   // ["tongs", "ladle", "knife"]
utensils.splice(2, 2, "fork", "chopsticks");
     //["tongs", "ladle", "fork", "chopsticks"]
```

The **join()** function converts the array to a string by placing each item into the string separated by the given delimiter string:

```
utensils.join(", "); // returns "tongs, ladle, fork, chopsticks"
utensils.join("|");  // returns "tongs|ladle|fork|chopsticks"
```

The **IF/ELSE**, **SWITCH** and ternary operators statements work the same as in JAVA, as well as the ternary conditional operator. But be careful due to the different types:

```
console.log(true ?  true : false);      // true
console.log(false ?  true : false);     // false
console.log(0 ?  true: false);          // false
console.log("" ?  true: false);         // false
console.log(null ?  true: false);       // false
console.log(undefined ?  true: false); // false
console.log(NaN ?  true: false);        // false
```

As we can see … `0`, `""`, `null`, `undefined` and **NaN** are all considered `false`.

The **FOR / WHILE** loops work the same as well, but do not forget to include the `let` statement in the loop variable. Here is an example (**vowelCount.js**) of some code that counts the vowels in a string:

```
const str = "I am a string with vowels.";
const vowels = "aeiouAEIOU";
let count = 0;

for (let i=0; i<str.length; i++) {
    if (vowels.includes(str[i])) {
        count++;
    }
}

console.log(`Number of vowels: ${count}`);
```

There is also a **FOR/OF** loop that works similar to JAVA's **FOR/EACH** loop.

```
const fruits = ["apple", "banana", "cherry"];

for (let fruit of fruits) {
    console.log(fruit);
}
```

Interestingly, it is common to use `const` for the loop variable since the **FOR/OF** loop makes a new block scope for each iteration of the loop. By using `const`, we are ensuring that we will not accidentally re-assign a value to it:

```
const fruits = ["apple", "banana", "cherry"];

for (const fruit of fruits) {
    console.log(fruit);
}
```

4.5 Objects

In **JavaScript**, objects are simply a collection of properties, where each property is a ***key*** (also called a ***name***) and ***value*** pair. We can create an ***object literal*** like this:

```
let car = {
    brand: "Toyota",
    year: 2021,
    isElectric: false
};
```

We can even create an empty object literal and then add its properties to it later. The objects are dynamic … we can add or change properties at any time. If we assign a value to an existing key, that key's value will be overwritten:

```
let car = {};              // {} represents an empty object
car.brand = "Toyota";      // creates a new key with a value
car.year = 2021;           // creates a new key with a value
car.isElectric = false;    // creates a new key with a value
car.year = 2023;           // overwrites previous value for year key
```

As an alternative to `{}`, we can call a default constructor:

```
let car = new Object();
car.brand = "Toyota";
car.year = 2021;
car.isElectric = false;
```

As in JAVA, we access using the dot operator, but we can also use square brackets to access:

```
console.log(car.year);
console.log(car["year"]);      // looks for key matching the string
console.log(car[year]);        // doesn't work
```

When displayed, objects will show their key/value pairs:

```
let car = {
    brand: "Toyota",
    year: 2021,
    isElectric: false
};
console.log(car);  // { brand: 'Toyota', year: 2021, isElectric: false }
```

If we try to access components that don't exist, we will get **undefined**:

```
console.log(car.parts);  // prints undefined
```

We can access the **keys** or **values** and get back an array of them:

```
console.log(Object.keys(car));      // ['brand', 'year', 'isElectric']
console.log(Object.values(car));    // ['Toyota', 2021, false ]
```

As mentioned earlier, since the objects are dynamic, we can add properties dynamically:

```
car.color = "red";
console.log(car);
//{ brand: 'Toyota', year: 2021, isElectric: false , color: 'red' }
```

We can even use `delete` to remove a property from a particular object instance:

```
delete car.isElectric;
console.log(car);    // { brand: 'Toyota', year: 2021, color: 'red' }
```

The result is as we would expect, but just for this instance, not for all cars:

```
{ brand: 'Toyota', year: 2021, color: 'red' }
```

4.6 Functions

In **JavaScript**, functions are declared using the `function` keyword. It differs from JAVA because we do not specify a return type, nor parameter types:

```
function add(a, b) {
    return a + b;
}
```

We call functions in the same way as we do in JAVA:

```
let result = add(2, 3);
```

If we pass extra parameters to a function, our code will still run … the extra parameters are ignored.

```
result = add(2, 3, true, "help"); // result is still 5
```

If we do not pass enough parameters to a function, then the value for the missing parameters are set to **undefined**. However, we can specify some default values in the function for such cases by assigning a value when defining the parameter.

```
function add(a, b=0) {
    return a + b;
}

result = add();  // result is NaN since a is undefined
result = add(2); // result is 2
```

Another option for creating objects is to create our own constructor and call it:

```
function Car(brand, year, isElectric) {
    this.brand = brand;
    this.year = year;
    this.isElectric = isElectric;
}

let car = new Car("Toyota", 2021, false);
```

JavaScript functions are considered first-class functions, which means that they can be:

- passed as an argument to other functions
- returned by another function
- assigned as a value to a variable
- stored in an object or an array

This will be important when we look at asynchronous functions.

Here is an example of assigning an ***anonymous function*** (i.e., a function with no name) to a variable so that the variable is of type **function**:

```
let add = function(a, b) {
              return a + b;
          };   // ← notice the required semicolon

let multiply = function(a, b) {
                   return a * b;
               };
```

We call the function by using the variable name:

```
let result = add(2, 3) + multiply(5, 4);
```

In later versions of **JavaScript**, the arrow function was introduced, which allows us to create functions in another reduced way by extracting the "important stuff" from the function and using that in a simplified manner. We grab the highlighted part from the code below:

```
let add = function(a, b) {
    return a + b;
};
```

Then we place it between an arrow:

```
let add = (a, b) => a + b;
```

The syntax is shorter. When only one parameter is used, we don't need parentheses:

```
let double = a => a * 2;
```

If we need multiple lines for the code, we use the usual braces:

```
let divide = (a, b) => {
    if (b === 0) return "divide by zero";
    return a / b;
};
```

Assigning functions to variables in this way, will allow us to pass functions as arguments (e.g., as in callbacks/event handlers). We will see these kinds of functions often in this course.

4.7 Shallow and Deep Copies

When doing function calls in **JavaScript**, primitive types (including **strings**) are ***passed by value*** (i.e., a copy is made so that the original remains unchanged). However, **arrays**, **objects** and **functions** ***are passed by reference*** (i.e., no copy is made so the original can be modified within a function).

Here is an example of objects being passed by reference (**test-object-reference.js**):

```
let robot1 = {
    name: "XR-17",
    model: "Explorer",
    age: 5,
    "can fly": false    // wow!  Keys can be strings … needed if key contains spaces
};

let robot2 = {
    name: "Coptor",
    model: "drone Y-8",
    age: 2,
    "can fly": true
};
```

```
function newest(r1, r2) {
    if (r1.age < r2.age)
        return r1;
    return r2;
}

function crash(r) {
    r["can fly"] = false; // access the attribute via []
}

console.log(newest(robot1, robot2)); // both objects passed by reference

console.log(robot2);
crash(robot2);          // alters original
console.log(robot2);
```

The output is as follows:

```
{ name: 'Coptor', model: 'drone Y-8', age: 2, 'can fly': true }
{ name: 'Coptor', model: 'drone Y-8', age: 2, 'can fly': true }
{ name: 'Coptor', model: 'drone Y-8', age: 2, 'can fly': false }
```

As we can see, in the **newest()** function, the objects are passed in just like primitives. However, in the **crash()** function, we are altering the robot passed in, so afterwards it has been altered.

In some situations, we want to make copies of objects so that they do not get altered from a function call. Imagine that we have a robot definition as follows:

```
let robot3 = {
    name: "R2D2",
    model: "Droid",
    age: 12,
    battery: 60,   // percentage
    equippedTools: ["scanner", "welder"],
    missionCount: 12
};
```

Now, let's assume that we want to generate a temporary mission profile based on the robot's specs so that we can do a simulation (e.g., equip robot with extra tools, increase its battery level, etc.).

We want to do something like this:

```
function prepareMissionProfile(r) {
    r.battery = 100;               // fully charge for the mission
    r.equippedTools.push("drill"); // equip with an extra tool
    r.missionCount += 1;           // add one to number of missions

    return r;
}

let simulatedRobot = prepareMissionProfile(robot3);
```

Of course, the simulated robot is actually the same robot as the original, but now the original has been altered. In such a situation we prefer to make a copy of the object so that the original remains unaltered.

The easiest way to make a copy of an object is to use the ***spread notation*** available in the latest **JavaScript** specs. This is done by using three consecutive dots (i.e., **...**).

The following code makes a shallow copy (i.e., does not go deeper into the object to make copies if the object has other objects within it) of our robot:

```
let clone = {... robot3};
```

The clone will now be able to be altered without affecting the original. Here is our altered function to make use of this spread feature:

```
function prepareMissionProfile(r) {
    let c = { ...r };
    c.battery = 100;               // fully charge for the mission
    c.equippedTools.push("drill"); // equip with an extra tool
    c.missionCount += 1;           // add one to number of missions

    return c;
}

let simulatedRobot = prepareMissionProfile(robot3);
```

Now we will get a simulated robot that reflects the changes to the original, but the original remains unaltered.

This also works with arrays:

```
let original = [1, 2, 3];
let copy = [...original];

copy.push(4);

console.log(original); // [1, 2, 3]
console.log(copy);     // [1, 2, 3, 4]
```

Interestingly, we can use it with a string to extract the letters:

```
let word = "hello";
let letters = [...word];    // ['h', 'e', 'l', 'l', 'o']

console.log(letters);
```

And we can use **...** to merge objects:

```
let info = { name: "Steve" };
let job = { role: "Developer" };
let profile = { ...info, ...job }; // { name: "Steve", role: "Developer" }
```

Alternatively, we can use **Object.assign()** to copy over the contents from one object to a new one:

```
let newRobot = Object.assign({}, robot1);
newRobot.age = 23;
console.log(robot1.age);    // 5
console.log(newRobot.age);  // 23
```

If we want a deep copy (i.e., one that copies objects & arrays within objects), we can make use of **JSON** functions.

> **JSON** stands for ***Javascript Object Notation***

JSON is a language-independent string that is easily readable and based on **JavaScript** syntax. Here (on the right) is the **JSON** format for our robot object:

JavaScript Object	**JSON** String

`{`
`  name: "R2D2",`
`  model: "Droid",`
`  age: 12,`
`  battery: 60,`
`  equippedTools: ["scanner", "welder"],`
`  missionCount: 12`
`}`
	`'{`
	`  "name": "R2D2",`
	`  "model": "Droid",`
	`  "age": 12,`
	`  "battery": 60,`
	`  "equippedTools": ["scanner", "welder"],`
	`  "missionCount": 12`
	`}'`

Notice that it looks just like a **JavaScript** object, but it's a string in practice when used in APIs or stored in files.

We can convert an object into a **JSON** string by using **JSON.stringify()** as follows:

```
JSON.stringify(robot1)
```

The result is the string:

```
{"name":"XR-17","model":"Explorer","age":5,"can fly":false}
```

We can then use the **JSON.parse()** function to get a new object that is a deep copy of the original:

```
let newerRobot = JSON.parse(JSON.stringify(robot1));

console.log(JSON.stringify(robot1));    // {"name":"XR-17","model":"Explorer","age":5,"can fly":false}
```

But in our example, there is no need for a deep copy since there are no objects nor arrays within the `robot1` object.

We will talk more about **JSON** later in the course.

4.8 Variable Scoping

It is important to briefly discuss variable scoping. Like JAVA, variables will have:

- **global scope** – if declared outside a function
- **local scope** – if declared inside a function
- **block scope** – if declared inside a block **{ }**

`var` and `let` work in different ways. `var` cannot have block scope. A variable defined using `var` is known throughout the function it is defined in … from the start of the function. A variable defined using `let` is only known in the block **{ }** it is defined in … from the moment it is defined onward.

```
function sayHi() {
    var phrase = "Hello";   // let would have the same scope
    console.log(phrase);
}
sayHi();

console.log(phrase);        // Error, phrase is not visible here

if (true) {
    var t1 = true;          // t1 is a global variable
    let t2 = true;
}
console.log(t1);            // the variable is usable outside if
console.log(t2);            // ReferenceError: t2 is not defined

{
    let a = 123;
    console.log(a);         // 123
};
console.log(a);             // ReferenceError: a is not defined

for (var i = 0; i < 10; i++) {
    // ...
}
console.log(i);             // 10, "i" is global … visible outside loop
```

The above code is in **test-scoping.js**. It is best to avoid mixing `var` and `let`. We will use `let`.

JavaScript has a default behavior of lifting `var` declarations to the top of their scope, before the code runs. This is known as ***hoisting***. The advantage of hoisting is that we can use a variable before it is declared 😊. For example, the following code works ok even though the variable is declared at the end because the `var` declaration is ***hoisted*** to the top:

```
robotName = "R2D2";
console.log(robotName);
var robotName;
```

We have to be careful in our assumptions though, because it hoists the declarations but NOT the initializations. So, this code …

```
console.log(robotName);
var robotName = "R2D2";
```

… is actually interpreted as this:

```
var robotName;
console.log(robotName);     // will be undefined … has no value yet
robotName = "R2D2";
```

Keep in mind that `let` declarations are NOT hoisted.

Functions are also fully ***hoisted***. That means, we can have a function declared near the bottom of our file but we can use it earlier above in our code 😊.

4.9 Higher-Order Functions

A ***Higher-Order Function*** is a function that returns another function or takes another function as an incoming parameter. We also use the term ***Callback Function*** to represent the function that we pass to another function. Arrays have some useful higher-order functions.

For instance, we can iterate through the array items and do something to each item by using a **forEach()** function on the array. We simply use the array function to provide a concise way to tell it what to do with each item:

```
let utensils = ["fork", "spoon", "knife"];

utensils.forEach(utensil => console.log(utensil));
```

The above code will print out each item (Examples in this section are combined in the file **test-higher-order.js**). We can also make a modified copy of an array's items by using the **map()** function. Again, we use the arrow function to explain what we want to do with each item. The original array remains the same:

```
let uppercased = utensils.map(t => t.toUpperCase());

console.log(uppercased); // ["FORK", "SPOON", "KNIFE"]
console.log(utensils);   // still ["fork", "spoon", "knife"]
```

We can use the **filter()** function to select certain array items by passing in a boolean function:

```
let longNames = utensils.filter(t => t.length > 4);

console.log(longNames);  // ["spoon", "knife"]
```

We can use the **find()** function to look for the first item that matches what we want:

```
let found = utensils.find(u => u.startsWith("s"));

console.log(found);  // spoon
```

The **reduce()** function lets us take an array and reduce it to a single value (e.g., number, string, object, etc..) by repeatedly applying a function to each element of the array. It is a bit tricker to use. It has this format:

```
array.reduce((accumulator, currentValue) => {
  // code that returns a new accumulator
}, optionalInitialValue);
```

If we consider our utensils example, here is how we can use **reduce()** to combine all items into a single string (in this context, *reduce* means "take all this and reduce it to a single thing"):

```
utensils = ["fork", "spoon", "knife"];

let result = utensils.reduce((returnString, item) => {
    return returnString + ", " + item;
});     // no initial value, so fork is used

console.log(result);  // "fork, spoon, knife"
```

and here is the example with an initial value supplied:

```
utensils = ["fork", "spoon", "knife"];

let result = utensils.reduce((returnString, item) => {
    return returnString + ", " + item;
}, "Combined Items: ");

console.log(result);  // " Combined Items: , fork, spoon, knife"
```

Or we can perhaps convert the array to an object, where each item becomes a key and the value will be 1 (indicating that there is 1 of each utensil):

```
utensils = ["fork", "spoon", "knife"];

let obj = utensils.reduce((returnObject, item) => {
    returnObject[item] = 1;
    return returnObject;
}, {});  // initial value is a new empty object

console.log(obj);  // { fork: 1, spoon: 1, knife: 1 }
```

As we can see … there are a lot of fun things that we can do by combining the things that we learn.

4.10 Synchronous vs Asynchronous Code

JavaScript code can either be ***Synchronous*** or ***Asynchronous***:

- **Synchronous** – code is executed in sequence line-by-line. Each statement waits for the previous statement to finish before executing.

- **Asynchronous** – code doesn't have to wait. The next step in our program can start while the current step is still running.

With **Asynchronous** code, the order of completion is not guaranteed. For example, we can make 5 requests, but we won't know in what order those requests will be answered.

It is important to discuss this because a browser only has one main thread to run our **JavaScript**, and how we manage that thread directly impacts:

- how quickly the page shows up (i.e., whether the site appears fast or feels slow to load)
- how quickly it reacts when you click or type (i.e., are buttons/menus/forms laggy)
- how smoothly it handles incoming or outgoing data (i.e., fetch from server without freezing up)

Synchronous code has the advantage that it is simple and predictable. However, it blocks the browser from doing anything else. This could be bad for tasks that take a long time (e.g., loading data) because while that code runs, the browser freezes and the user interface is unresponsive … we cannot click scroll or interact! That is a bad situation.

Asynchronous code is more difficult to predict. However, it doesn't block the main thread and lets the browser remain responsive to the user. This is essential for tasks like fetching data, waiting for user input, animations and timers.

Unfortunately, **JavaScript** is single-threaded … only one instruction can be executed at a time. Therefore, it is synchronous. However, the good news is that we can manipulate **JavaScript** to behave in an asynchronous way 😊. Also, asynchronous functions do exist … generally for input/output operations.

We can use the **setTimeout()** function to delay the execution of a piece of code, without stopping the rest of the code from running (i.e., without freezing the page). This can be used for basic transitions or alert messages, among other things. The function has this format:

```
setTimeout(callbackFunction, delayInMilliseconds);
```

Once this line of code is encountered by the **JavaScript** interpreter, the interpreter will not run the function right away. Instead, it takes note of the specified delay and then continues with the next line of code in our program. After the given number of milliseconds (roughly) the interpreter will pause where it is in our program ("dropping a pin" there) and then run the code in the callback function that we supplied. When that callback function completes, it will then continue executing code at the pin that it placed.

Consider this code in **test-timeout.js**:

```
console.log( "1 - I display first at the start of the program." );

setTimeout(function() {
    console.log( "4 - I display 4th because I have the second largest delay." );
}, 2000 );

setTimeout(function() {
    console.log( "5 - I display last because I have the largest delay." );
}, 3000 );

setTimeout(function() {
    console.log( "3 - I display 3rd because my delay is smaller than those above me." );
}, 1000 );

console.log( "2 - I display quickly after the first, because I am not delayed." );
```

Do you understand why the output is as follows?

```
1 - I display first at the start of the program.
2 - I display quickly after the first, because I am not delayed.
3 - I display 3rd because my delay is smaller than those above me.
4 - I display 4th because I have the second largest delay.
5 - I display last because I have the largest delay
```

We can use the **setInterval()** function to cause some code to repeat in a loop based on a timer. It has this format:

```
setInterval(callbackFunction, intervalInMilliseconds);
```

We can add the following to our code above so that it prints out a message every half a second. Assume that we add this to the top of our program, but we can add it anywhere and then save it as **test-interval.js**:

```
setInterval(function(){
    console.log("All is going well ...");
}, 500);
```

Do you understand why the output is as follows?

```
1 - I display first at the start of the program.
2 - I display quickly after the first, because I am not delayed.
All is going well ...
All is going well ...
3 - I display 3rd because my delay is smaller than those above me.
All is going well ...
4 - I display 4th because I have the second largest delay.
All is going well ...
All is going well ...
5 - I display last because I have the largest delay.
All is going well ...
All is going well ...
All is going well ...
All is going well ...
All is going well ...
All is going well ...
```

The code actually keeps running in **node.js** until we press **Ctrl-C** to stop it.

Sometimes, we nest timeout calls. This may be done for several reasons, such as to …

- **retry something later**
  (i.e., try again to get data if the first attempt failed)
- **wait for other work to finish**
  (i.e., give the browser time to update the screen or process events first)
- **change timing on the fly**
  (i.e., adjust how often we check for data or perform an action as conditions change)

For example, we could do something like this to wait for other work to finish:

```
setTimeout(function(){
    // ... do some task ...
    console.log("First timer is up.");
    setTimeout(function(){
        // ... do another task ...
        console.log("Second timer is up.");
        setTimeout(function(){
            // ... do a final task ...
            console.log("Third timer is up.");
        }, 1000);
    }, 1000);
}, 1000);
```

In the code, each `setTimeout` is nested so that the *next* one isn't scheduled until the *previous* one has actually started its work. The tasks might vary in length. If each task takes an unpredictable amount of time, we don't want to start the next timeout until we are sure the previous task has finished. Nesting guarantees that sequence. In the code, we will see the three messages in order printer after 1, 2 and 3 seconds. This can be a very useful template for staged UI animations, step-by-step messages or dialogue timing.

It can also be useful to nest delays if the next delay (or action) ***depends on*** a result (or condition) from a previous step:

```
setTimeout(function() {
    let status = checkRobotStatus();
    console.log("Status checked");

    if (status === "ok") {
        setTimeout(function() {
            console.log("Proceeding to phase 2 ...");
        }, 1000);
    }
}, 1000);
```

It is best not to nest callbacks too deep because this becomes hard to read and debug.

Keep in mind that the time periods are not guaranteed to be exact. Since **JavaScript** is single-threaded, a timeout will not execute until the current block of code has finished executing. All the callbacks are handled by an ***event loop*** … which will be discussed later.

4.11 Closures

Closures are a core concept in **JavaScript** and powerful once we get the hang of them.

All functions have their own scope, which allows us to have local variables. Functions have access to the scope "above" them in a nesting (including global scope). For example, inner functions have access to the scope of their outer function(s).

> **A *closure* is when a function remembers the variables from the scope in which it was created, even after that scope has finished executing.**

JavaScript uses lexical scoping, which means that inner functions remember variables in the outer function based on **where they were written**, not **when they are run**. Even after the outer function has returned, the inner function retains access to its variables because the **JavaScript** engine keeps the scope "alive" for any active references.

Consider this code:

```
function createRobotCommander(robotName) {
    return function(command) {
        console.log(`Sending command to ${robotName}: ${command}`);
    };
}

// Create a commander for a specific robot
let xr17Commander = createRobotCommander("XR-17");

// These calls still remember the robot's name
xr17Commander("Activate scanning");
xr17Commander("Move forward 10 meters");
```

The output is:

```
Sending command to XR-17: Activate scanning
Sending command to XR-17: Move forward 10 meters
```

What is happening here? The `createRobotCommander()` function is called once. It returns the inner function defined there which sends commands to the robot. That function accesses the `robotName` parameter/variable from the `createRobotCommander()` function. The `xr17Commander` variable holds a reference to that inner function, so it has a function type. We can then use that variable to call that returned inner function.

Due to the closure properties of **JavaScript**, that inner function remembers the `robotName` variable even though the `createRobotCommander()` function has completed running.

Consider now an example of three functions that determine if words are small, medium or large:

```
words = ["cat", "ocelot", "tiger", "dog", "fish", "elephant",
         "walrus", "giraffe", "crockodile", "emu"];

function smallWords(words){
    return words.filter(word => {return word.length >= 0 && word.length <= 3});
}
function mediumWords(words){
    return words.filter(word => {return word.length >= 4 && word.length <= 6});
}
function largeWords(words){
    return words.filter(word => {return word.length >= 7});
}

console.log("Small Words:   ", smallWords(words));
console.log("Medium Words:  ", mediumWords(words));
console.log("Large Words:   ", largeWords(words));
```

This code produces the following output as expected:

```
Small Words:    [ 'cat', 'dog', 'emu' ]
Medium Words:   [ 'ocelot', 'tiger', 'fish', 'walrus' ]
Large Words:    [ 'elephant', 'giraffe', 'crockodile' ]
```

Do you notice anything *bad* about the above code? The logic for filtering seems to have duplicated code. It would be nice to have reusable filter logic.

To do this, we can make use of closures by defining the filter logic as its own function that will depend on the minimum and maximum word sizes:

```
function filterFactory(minsize, maxsize=Infinity){
    return arr => {
        return arr.filter(
            word => {return word.length >= minsize && word.length <= maxsize});
    }
}

sWords = filterFactory(0,3);
mWords = filterFactory(4,6);
lWords = filterFactory(7);

console.log("Small Words:   ", sWords(words));
console.log("Medium Words:  ", mWords(words));
console.log("Large Words:   ", lWords(words));
```

The output will be the same. Do you notice how much cleaner the code looks? The filtering logic is written once, so it is reusing code nicely. This only works because the `sWords()`, `mWords()` and `lWords()` functions all remember their individual `minSize` and `maxSize` values … thanks to the closure feature of **JavaScript**. Note as well that the `maxSize` has a default value of infinity so that there is no upper bound for the large words. This allows us to omit the second argument when creating the `lWords()` function.

As a last example of the benefits of closures, we will look at how we can simulate making object attributes private.

Consider a function that creates a new **utensil** object every time that it is called. The object returned will only contain functions so that the user of the object cannot access or modify the attributes of the object directly. This is similar to the notion of making an object's attributes **private**, as we do in JAVA. So, we will need to simulate something like get/set methods. Here is the code:

```
function utensil(initType, initCleanStatus) {
    let type = initType;           // e.g., "fork", "spoon"
    let isClean = initCleanStatus; // boolean

    return {
        getType: () => type,
        setType: (newType) => { type = newType; },

        isClean: () => isClean,
        wash: () => { isClean = true; },
        use: () => { isClean = false; },
    };
}
```

We create a utensil by calling this function with some initial values and the function returns an object with a bunch of functions in it. The "state" of the object is represented by the initial values passed in (i.e., the **type** of utensil and whether it **isClean**). Due to the closure feature of **JavaScript**, these values are remembered.

Notice that there is a **getType()** function that returns the type and an **isClean()** function to return its clean status. There is also a **setType()** function that allows us to alter the type. And finally, we have a **wash()** and **use()** function to adjust the cleanliness accordingly.

How do we use this? We just call the function …

```
let spoon = utensil("spoon", true);  // a clean spoon
let fork = utensil("fork", false);   // a dirty fork
let knife = utensil("knife", false); // a dirty knife
```

What is the output for this code:

```
console.log(spoon);
console.log(fork);
console.log(knife);
```

It might look strange:

```
{
  getType: [Function: getType],
  setType: [Function: setType],
  isClean: [Function: isClean],
  wash: [Function: wash],
  use: [Function: use]
}
```

```
{
  getType: [Function: getType],
  setType: [Function: setType],
  isClean: [Function: isClean],
  wash: [Function: wash],
  use: [Function: use]
}
{
  getType: [Function: getType],
  setType: [Function: setType],
  isClean: [Function: isClean],
  wash: [Function: wash],
  use: [Function: use]
}
```

All we see is the functions but we cannot see the state of each. Because the state is not stored as keys, we cannot access it directly. However, we can access via the **getType()** and **isClean()** functions as follows:

```
console.log(spoon.getType());  // spoon
console.log(spoon.isClean());  // true
console.log(fork.getType());   // fork
console.log(fork.isClean());   // false
console.log(knife.getType());  // knife
console.log(knife.isClean());  // false
spoon.use();
fork.wash();
knife.wash();
console.log(spoon.isClean());  // false
console.log(fork.isClean());   // true
console.log(knife.isClean());  // true
```

So, the closure really allows us to maintain the “state of the object” via the “remembered” function parameters 😊. We now have a way of ensuring data privacy by defining **JavaScript** objects with encapsulated state!

There is so much more to learn about **JavaScript**. If you are interested in becoming a “Good” **JavaScript** programmer, please read chapters 1-4 here: Eloquent JavaScript