JavaScript: The Basics (1)

Overview of this lecture:

- how to create a js program (Hello World)
- executuion context
- hoisting
- scope
- call stack
- syntax
- arrays
  - for loops
- anonymous funcitons
- arrow funcitons
- Events

JavaScript is not Java

- JS is a **scripting langauge** - you write it and then you run it
  - the compilation of the program happens *during* the execution not before like in C
- Came around in the mid 90s
- We see JS when on the internet
- JavaScript runs on the **client side** - for example if you are building a webpage and want to check if the users passwords match during the registration of a new account (when it asks you to make a password and then confirm it)

Creating/writing a JavaScript Program

- create a file with `.js` in a code editor of your choosing
- you do **not need** code delimiters (i.e. no curly braces)
- we include them in the script tag in one of two ways:

```
1. <script src="your JavaScript.js"></script>

2. <script> write the code here </scipt>
```

- you can either put the script in the same HTML file or a different file
  - usually **better to put it in a seperte file** and link them using the script tag
    - looks better and is efficient

Hello World in JavaScript

- stored inside a HTML file, everything is in side the `<html> </html>` tags

```
<html>

<head><title> Hello World</title></head>


<body>


    <script>
                    function doAnything(){
                            alert('Hello World');
                            }
```

```
                doAnything();
    </script>

</body>

</html>
```

- within in the script tag we have a function `doAnything`
    - again we couldve used `src` and then the filename to get the same effect

---

Execution Context

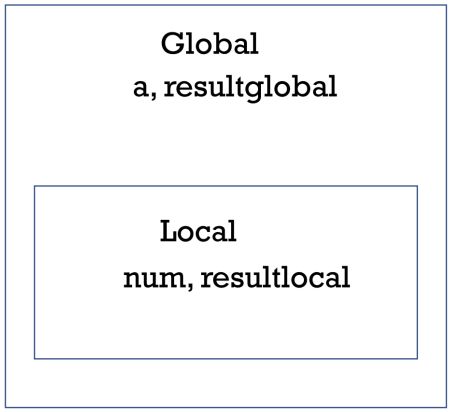
So say in the above example when the browser sees our JS code, what does it do with it?

The browser creates **a box to run out code** this box is known as the **Execution Context**

What is stored in the execution context?

- variables
- the code itself

There is both **global** and **local** execution context




```
var a = 2;

function add (num) {

  var resultlocal = num + 10;

}

var resultglobal = add(a);
```

Imagine an execution context as an environment where your JavaScript code comes to life. It's a container that holds all the necessary information for running your code, like variables, functions, and the special `this` keyword. There are two main types:

1. **Global Execution Context (GEC):** This is the first context created when your JavaScript program starts. It has access to all globally declared variables and functions (without using `var`, `let`, or `const`). Think of it as the big stage where your entire play unfolds.
2. **Local Execution Context (LEC):** Whenever you call a function, a new local execution context is created for **that specific function.** This context holds the function's arguments, local variables, and a reference to the outer context (if any). It's like a smaller stage set up for each scene in your play.

Hoisting

Hoisting is JAVAScript behaviour that allows processing declarations and functions to be at the top, before any code is executed (before compilation).

Variable Hosting:

Only the variable decelerations are hoisted, not the initialisation. e.g.

`var x = 5` here `var x` is the **declaration** whilst `x = 5` is the initialisation.

So when I say the deceleration is hoisted, it means that the variable declaration (`var x`) is processed before any code is executed.

```
console.log(x); // undefined, not ReferenceError
var x = 5;
console.log(x); // 5
```

In this example, the declaration (`var x;`) is hoisted to the top, but the initialisation (`x = 5;`) stays at its original location. That's why the first console.log prints `undefined` rather than throwing a ReferenceError.

Using `let` and `const`:
Variables declared with
`let` and `const` are also hoisted, but they do not initialise until their actual declaration is reached in the code. Attempting to access them before their declaration results in a ReferenceError.

```
javascriptCopy code
console.log(y); // ReferenceError: y is not defined
let y = 5;
console.log(y); // 5
```

`let`

The `let` keyword is used to declare variables that can be reassigned, and it has block scope. This means that a variable declared with `let` is only accessible within the block, statement, or expression where it was defined. This is different from the `var` keyword, which defines variables globally or locally to an entire function regardless of block scope.

Characteristics of `let`:

- **Block Scoped**: Unlike variables declared with `var`, which are scoped to the function in which they are defined (if declared inside a function), variables declared with `let` have their scope limited to the block, statement, or expression where they are defined.
- **No Re-declaration**: In the same scope, you cannot re-declare the same variable within the same block using `let`. Attempting to do so will result in a syntax error.
- **Not Hoisted**: Variables declared by `let` are not initialized until their definition is evaluated at runtime. Accessing them before the declaration results in a `ReferenceError`.

Example of `let`:

```
javascriptCopy code
function showLetExample() {
  let x = 10;
  if (true) {
    let x = 20;  // This 'x' is local to the block
    console.log(x);  // Outputs 20
  }
  console.log(x);  // Outputs 10
}
showLetExample();
```

In this example, the variable `x` declared inside the `if` block is a separate instance from the `x` declared in the function block. They do not interfere with each other because `let` variables are block scoped.

`const`

The `const` keyword is used to declare variables that are meant to remain constant after their initial assignment. Like `let`, `const` also has block scope.

Characteristics of `const`:

- **Block Scoped**: Similar to `let`, a variable declared with `const` is confined to the scope of the block in which it was declared.
- **Cannot be Re-assigned**: Variables declared with `const` cannot be re-assigned. This is useful for defining constants that should not change throughout the execution of the program.
- **Must be Initialized**: A `const` declaration must be initialized at the time of declaration. This means you must provide a value at the same time you declare a `const` variable.

Example of `const`:

```
javascriptCopy code
function showConstExample() {
  const y = 30;
  console.log(y);  // Outputs 30

  // Uncommenting the following line will throw an error
  // y = 50;  // TypeError: Assignment to constant variable.
}
showConstExample();
```

In this example, attempting to re-assign a new value to `y` will result in a runtime error because `const` variables cannot be re-assigned once set.

Summary

- `let` is used for declaring variables that might change later. They are block scoped and not hoisted in a way that they are accessible before their declaration.
- `const` is for declaring variables that are meant to remain constant after their first assignment, also block scoped and must be initialized upon declaration.

Using `let` and `const` instead of `var` (which is function scoped and hoisted) can help avoid common bugs due to their clearer and more predictable scoping rules.

Function Hoisting

Functions are hoisted to the top of their containing Scope, this means that the entire body of the function is hoisted and can be accessed from anywhere:

```
javascriptCopy code
console.log(square(5)); // 25
function square(n) {
    return n * n;
}
```

Scope

Scope is JavaScript referes to the context in which variables and functions are accessible. This determines the visibility of your code to other parts of your code.

e.g. :

- **Global Variables** are outside functions and blocks are accessible anywhere and available to multiple functions.
- **Function Scope:** Variables that are declared inside functions are only accessible to that function
- **Block Scope:** Variables declared in if statements and for loops are only visible to those blocks

Scope Chain

When code that needs to access a variable is executed, JavaScript looks up the variable in the current scope. If it doesn't find the variable, it looks up in the outer scope, and this continues up the "scope chain" until it either finds the variable or reaches the global scope. If the variable isn't found in the global scope, a `ReferenceError` is thrown.

Call Stack

The call stack is a fundamental concept in JavaScript execution. It's a **Last-In-First-Out (LIFO)** data structure, essentially a mental stack of function calls, that keeps track of where your code is in its execution process.

Understanding the Call Stack

Imagine the call stack as a stack of plates in a cafeteria. You add plates (function calls) to the top, and you remove them (function returns) from the top as well. Here's how it works in JavaScript:

1. **Global Execution Context:** When your JavaScript program starts, a global execution context (GEC) is created. This context holds information like globally declared variables and functions. The GEC is the first "plate" placed on the stack.
2. **Function Calls:** Whenever you call a function, a new local execution context (LEC) is created and pushed onto the call stack. This LEC contains the function's arguments, local variables, and a reference to the outer context (if the function is called within another function). It's like adding a new plate on top for that specific function call.
3. **Function Execution:** The JavaScript engine starts executing the code inside the function's LEC. It can access its own local variables and arguments, as well as any variables from outer contexts (think reaching down the stack to grab a plate from below).
4. **Function Returns:** When a function finishes execution, its LEC is popped off the call stack. The JavaScript engine resumes execution from the line after the function call in the previous context. It's like removing the top plate and continuing with the plate underneath.

JavaScript Syntax

JavaScript contains many statements we are familiar with:

- `break`

- `if else`
- `for`
- `function`
- `do ... while`
- `var, let, const` - variable declarations
- `return`
- `switch`
- `throw`
- `try ... catch`
- arrays - JS allows for **mixed arrays** of different data types e.g.
  - `var array = [1, joe, 6.666]`

JavaScript Operators

- Comparison operators: `<, <=, >, >=, ==, ===`
  - the double equal (`==`) means *loosly equal* and compares two values for equality after converting both operands to a common type if the operand types are different
    - e.g. `1 == '1'` would return `true` as JS converts the string '1' into the number 1 before making the comparison
  - triple equals (`===`) means *strict equality*. This compares two values **without** performing type conversion. It returns `true` if the operands are of the same type **and** have the same value.
    - if the types of the operands are different, it returns `false` with no attempt to convert to a common type: `1 === '1'` would therefore return false even though their values are the same, because their types are different
- Logical operators: `&& || !`

You **dont need to declare types** JavaScript makes a best guess what type it is dealing with.

- sometimes it gets it wrong then you can use `parseInt` etc for integers
- many operators automatically convert types

Arrays in JavaScript

- can think of arrays as Objects with Methods

e.g.:

```
var name = "tomsanders";

console.log(name.length); = 10

OUTPUT

10 (length of the array)

console.log(name[0]; = 't'
```

Integer array:

Arrays & for loops

Take a mixed array:

```
var exampleArray = [1,tom,6.666]
```

- There are two ways to loop through and console log an array, one giving you the *index* of each element, and the other giving you the *value* of each element

```
for var element in
exampleArray {
    console.log(element);
    }
```

- `in` would console log the **index** of each element

```
for var element of
exampleArray {
    console.log(element);
    }
```

- `of` would console log the **value** of each element

map()

map: The Transformer

- **Purpose:** The `map` method creates a **new array** by applying a function to each element of the original array, thus transforming it.
- **How it works:**
    1. You provide `map` with a callback function that tells it how to transform each element.
    2. `map` iterates over the original array, calling this callback function for each element.
    3. The return value of the callback function becomes the corresponding element in the new array.

For example:

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

```
console.log(doubledNumbers);
// Output: [2, 4, 6, 8]
console.log(numbers);
// Output: [1, 2, 3, 4] (original array remains unchanged)
```

Key Points:

- `map` is **non-mutating**—it doesn't modify the original array.
- It's great for transforming arrays into new formats or structures.

pop()

pop: The Remover

- **Purpose:** The `pop` method removes the **last element** of an array and **returns** that removed element. It changes the original array in place.
- **How it works:**
  1. `pop` goes to the end of the array and removes the last item.
  2. It gives you back the value it removed as a return value.

For example:

```
const fruits = ['apple', 'banana', 'mango'];
const lastFruit = fruits.pop();

console.log(lastFruit);    // Output: 'mango'
console.log(fruits);       // Output: ['apple', 'banana']
```

- `pop` is **mutating**—it directly modifies the original array.
- It's handy when you need to grab the last item of an array and do something with it.

Anonymous Functions

- not all functions need to have a name in JavaScript

Anonymous functions are functions without a name. You define them on the fly and assign them to a variable or use them directly within an expression.

e.g. take an array: `var age = [20, 25, 27, 30]`

- lets use map which takes funciton as an argument to add 10 years to each of the values in the array:

```
age = age.map(function(any)
{
        return age + 10;
})
```

Arrow Functions

(Introduced in ES6)

Concept: Arrow functions are a more concise way to write anonymous functions. They offer a shorter syntax and implicit `return` for single-line expressions.

```
const someFunction = (parameters) => {
    // Function body (implicit return for single line)
}

// OR (for functions with multiple lines)
const someFunction = (parameters) => {
    // Function body with curly braces
    return something;
}
```

```html
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Functions</h1>
<h2>The Arrow Function</h2>

<p>This example shows the syntax of an Arrow Function,
and how to use it.</p>

<p id="demo"></p>

<script>
let hello = "";

hello = () => {
  return "Hello World!";
}

document.getElementById("demo").innerHTML = hello();
</script>
```

Events in JavaScript

- anything you do on a webpage **is an event**
  - e.g. clicking a button, hovering a mouse, entering text etc
- HMTL pages can capture these events and pass them to an **event handler** in JavaScript

An example:

```
<html>

<head> <title> Event Handlers </title> </head>

<body>

        <button onclick="alertName(event)">Button 1 </button>
       <button onclick="alertName(event)">Button 2 </button>


  <script>
        function alertName(event)
          {
            var trigger = event.srcElement;
            alert('You clicked on ' + trigger.innerHTML);
           }

   </script>

</body>

</html>
```

- the function `alertName` is passed the event