Front-End Web Development

Unit 4: Dynamic Client-side Scripting
JavaScript Programming

UCD DUBLIN PROFESSIONAL ACADEMY

Course Outline

1. Getting Started
2. HTML - Structuring the Web
3. CSS - Styling the Web
4. **JavaScript - Dynamic client-side scripting**
5. CSS - Making Layouts
6. Introduction to Websites/Web Applications
7. CSS - Advanced
8. JavaScript - Modifying the Document Object Model (DOM)
9. Dynamic HTML
10. Web Forms - Working with user data
11. JavaScript - Advanced
12. Building a Web Application with JavaScript
13. Introduction to CSS Frameworks - Bootstrap
14. Building a Web Application with Svlte
15. SEO, Web Security, Performance

Assessment

- Final Project - 100% of the grade

  - Design and Build functioning Website using HTML5, CSS (including Bootstrap), JavaScript (browser only)

✔ Code will be managed in GitHub
✔ Website will be deployed to GitHub Pages
✔ All code to follow best practice and be documented

- Details and How-To-Guide are available on the course page under the section called Assessments

4. Dynamic Client-side Scripting

In This Unit

Title
Overview of JavaScript in the browser
Conditional & Loops
Strings, Arrays and Functions

Versions

Version	Name	Release Year	Features
ES1	ECMAScript 1	1997	Initial Release
ES2	ECMAScript 2	1998	Minor Editorial Changes
ES3	ECMAScript 3	1999	Added: Regular Expression, try/catch, Exception Handling, switch case and do-while
ES4	ECMAScript 4		Abandoned due to conflicts
ES5	ECMAScript 5	2009	Added: JavaScript “strict mode”, JSON support, JS getters and setters
ES6	ECMAScript 2015	2015	Added: let and const, Class declaration, import and export, for..of loop, Arrow functions
ES7	ECMAScript 2016	2016	Added: Block scope for variable, async/await, Array.includes function, Exponentiation Operator
ES8	ECMAScript 2017	2017	Added: Object.values, Object.entries, Object.getOwnPropertiesDescriptors

Versions

ES9	ECMAScript 2018	2018	Added: spread operator, rest parameters
ES10	ECMAScript 2019	2019	Added: Array.flat(), Array.flatMap(), Array.sort is now stable
ES11	ECMAScript 2020	2020	Added: BigInt primitive type, nullish coalescing operator
ES12	ECMAScript 2021	2021	Added: String.replaceAll() Method, Promise.any() Method
ES13	ECMAScript 2022	2022	Added: Top-level await, New class elements, Static block inside classes
ES14	ECMAScript 2023	2023	Added: toSorted method, toReversed, findLast, and findLastIndex methods on Array.prototype and TypedArray.prototypet

Basic syntax rules for JavaScript

- JavaScript is case-sensitive
- JavaScript ignores extra whitespace within statements
- Comments with // and /* */ syntax
- Each JavaScript statement ends with a semicolon
- Split a statement after:
  - an arithmetic or relational operator like +, -, *, /, =, ==, >, or <
  - an opening brace ({), bracket ([ ), or parenthesis
  - a closing brace (})
- Do not split a statement after:
  - an identifier, a value, or the return keyword
  - a closing bracket ( ]) or parenthesis

Rules for naming variables & other identifiers

- Identifiers can only contain letters, numbers, the underscore, and the dollar sign
- Identifiers can’t start with a number
- Identifiers are case-sensitive
- Identifiers can be any length
- Identifiers can’t be the same as reserved words
- Avoid using global properties and methods as identifiers

```
subTotal    //valid
index_1     //valid
$           //valid
$log        //valid
Sub-total   // invalid
1index      // invalid
return      // invalid
&log        // invalid
```

Naming conventions

- Two general conventions used, camelCase or snake_case/under_score notation
- Use meaningful names for identifiers
- That way, your identifiers aren't likely to be reserved words or global properties
- Be consistent: Use either camelCase or snake_case/under_score. Do not mix them
- If you're using underscore notation, use lowercase for all letters

```
taxRate                 tax_rate
calculateClick          calculate_click
emailAddress            email_address
firstName               first_name
futureValue             future_value
```

Naming Things in Code

Variables

Variables are Containers for Storing Data. Consists of a `<name> = <value>` pair JavaScript Variables can be declared in 4 ways:

- Automatically
- Using *var*
- Using *let*
- Using *const*

In this first example, x, y, and z are undeclared variables. They are automatically declared when first used:

```
x = 5;
y = 6;
z = x + y;
```

It is considered good programming practice to always declare variables before use

Variables

- The *var* keyword was used in all JavaScript code from 1995 to 2015. The *let* and *const* keywords were added to JavaScript in 2015. The *var* keyword should only be used in code written for older browsers
- Both *let* and *const* provide Block Scope in JavaScript. Variables declared inside a { } block cannot be accessed from outside the block

```
let x = 5;
let y = 6;
let z = x + y;
```

- *const* are values that cannot be changed

```
const price1 = 5;
const price2 = 6;
let total = price1 + price2;
```

Variables

When to use *var*, *let*, or *const*?

1. Always declare variables
2. Always use *const* if the value should not be changed
3. Always use *const* if the type should not be changed (Arrays and Objects)
4. Only use *let* if you can't use *const*
5. Only use *var* if you MUST support older versions of browsers

Data Types

JavaScript has 8 Datatypes

1. String - is a series of characters like "John Doe". Are written with either single or double quotes

```
let carName1 = "Volvo XC60"; // Using double quotes
let carName2 = 'Volvo XC60'; // Using single quotes
```

2. Number - are stored as decimal numbers (floating point). Can be written with, or without decimals. Stored as a double (64-bit floating point)

```
let x1 = 34.00; // With decimals
let x2 = 34; // Without decimals
```

3. Bigint - is a new datatype (ES2020) that can be used to store integer values that are too big to be represented by a normal JavaScript Number

```
let x = BigInt("123456789012345678901234567890");
```

Data Types

4. Boolean - can only have two values: `true` or `false`. Named after George Boole, 19th century mathematician and first professor of mathematics at Queen's College, Cork (now University College Cork)

```
let x = 5;
let y = 5;
let z = 6;
(x == y)       // Returns true
(x == z)       // Returns false
```

5. Undefined - a variable without a value, has the value undefined. The type is also undefined

```
let car;    // Value is undefined, type is undefined
car = undefined; // Value is undefined, type is undefined
```

6. null - represents the intentional absence of any object value. It is one of JavaScript's primitive values and is treated as falsy for boolean operations

Data Types

7. Symbol - Symbols are new primitive type introduced in ES6. Symbols are completely unique identifiers

```
const symbol = Symbol('description');
```

8. Object - JavaScript objects are written with curly braces {}. Object properties are written as name:value pairs, separated by commas

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

    - arrays and dates are also objects which we'll cover in later lectures

Type Conversion

- JS is both dynamically typed and weakly typed
- Statically typed means the type is enforced. All variables must be declared with a type, e.g. in Java

```
int x = 5;
String y = 'abc';
```

- Dynamically typed languages infer variable types at runtime
- This means once your code is run, the compiler/interpreter will see your variable and its value andthen decide what type it is

```
let a = 1; // integer
b = "test"; // string
```

- Weakly typed languages allow types to be inferred as another type. For example, 1 + ’2’ will result in '12’
- JS sees you’re trying to add a number with a string — an invalid operation — so it coerces your number into a string and results in the string ‘12’

Type Coercion

Type coercion is a process where a value of one type is implicitly converted into another type

```
let 1 + "1"; // "11" rather than (1 + 1 = 2)
```

Other programming languages (Python, Java) will stop the program and flag this as an error

The behavior depends on the data types involved

```
[1 ,2] + "1" // "1,21"
true + "1" // "true1"
{ a: 1 } + "1" // 1
"1" + { a: 1 } // "1[object Object]"
true + { a: 1 } // "true[object Object]"
{ a: 1 } + 1 // 1
```

Use explicit type conversion using functions such as parseInt(), parseFLoat(), Number(), String()

```
let 1 + Number("1"); // 2
```

`typeof` operator

JavaScript allows you to check the data type by using the `typeof` operator. To use the operator, you need to call it before specifying the data:

```
let x = 5;
console.log(typeof x) //'number'
console.log(typeof "Nathan") // 'string'
console.log(typeof true) // 'boolean'
```

The `typeof` operator returns the type of the data as a string. The 'number' type represents both integer and float types, the string and boolean represent their respective types

JavaScript Arithmetic Operators

Arithmetic operators perform arithmetic on numbers (literals or variables)

Operator	Description
+	Addition - e.g. let x = 100 + 50
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016) raises the first operand to the power of the second operand, e.g. x**2
/	Division
%	Modulus (Remainder) - e.g. let x = 9 % 4 // the result is x = 1 (remainder)
++	Increment number by 1 - e.g. ++x (prefix increment), x++ (postfix increment);
--	Decrement number by 1 - e.g. --x, x--;

JavaScript Arithmetic Operators

Assignment operators assign values to JavaScript variables

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

JavaScript Operator Precedence

- Operator precedence describes the order in which operations are performed in an arithmetic expression
- Multiplication (*) and division (/) have higher precedence than addition (+) and subtraction (-)
- Expressions in parentheses are computed before the rest of the expression
- Function are executed before the result is used in the rest of the expression
- Precedence applies only when two operands compete for the same operator
- If the operators are independent, JavaScript evaluates expressions from left to right
- Parentheses may be used to change the order of operations
- https://www.w3schools.com/js/js_precedence.asp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_precedence

Functions

A function is simply a section or block of code that's written to perform a specific task. The function is executed when "something" invokes it (calls it)

The general definition of a function is

```
function name(parameter list) {
  statements in the function body
}
name(argument list); // call the function
```

where name is the name of the function, and parameter list is a list of comma-separated variables used to hold the values of each argument

You can return a value from a function by including a return statement, which is written as

```
return expression
```

where expression is a value or evaluates to a value which you want to return from the function

Examples of Functions

The following function converts Fahrenheit temperatures to their Celsius equivalent:

```
function fahrenheitToCelsius(f) { // f is called an argument
  return 5 / 9 * (f - 32);
}
fahrenheitToCelsius(79); // 61.222
```

The following function computes the area of a triangle from its base and height:

```
function triangleArea(base, height) {
  return (base * height) / 2;
}
triangleArea(10, 20); // 100
```

Concatenation operator for Strings

```
// With string data
let firstName = "Ray", lastName = "Harris";
let fullName = lastName;
fullName += ", ";
fullName += firstName;
// fullName is "Harris, Ray"

// With mixed data
let months = 120;
message = "Months: ";
message += months;
// message is "Months: 120"
```

+ Symbol - Concatenation or Addition?

- Be careful with the + operator when dealing with strings
- If both values are numbers, JavaScript adds them
- If both values are strings, JavaScript concatenates them
- If one value is a number and one is a string, JavaScript converts the number to a string and concatenates them
- The type conversion magic doesn't work in this case!

```
let addi = 3 + "4"; // when we have a string added to a number, we get concatentation, not addition

let plusi = "4" + 3; // same here... we get "43"
```

Other Arithmetic Operations

The type conversion magic does work with -, *, /

```
let multi = 3 * "4";  // here JavaScript converts the string "4" to the number 4,
                      // and multiplies it by 3, resulting in 12

let divi = 80 / "10"; // here the string "10" is converted to the number 10.
                      // Then 80 is divided by the number 10, resulting in 8

let mini = "10" - 5;  // with minus, the "10" is converted to the number 10,
                      // so we have 10 - 5, which is 5
```

Converting String values

Window object provides a number of utility methods for converting strings into numbers

```
// parseInt(string) - parse a string into an integer
// parseFloat(string) - parse a string into a float/decimal number
// examples that use these methods

let entryA = prompt("Enter any value", 12345.6789);
alert(entryA);   // displays 12345.6789  entryA =
parseInt(entryA);
alert(entryA);   // displays 12345

let entryB = prompt("Enter any value", 12345.6789);
alert(entryB);   // displays 12345.6789  entryB =
parseFloat(entryB);
alert(entryB);   // displays 12345.6789

let entryC = prompt("Enter any value", "Hello");
alert(entryC);   // displays Hello  entryC =
parseInt(entryC);
alert(entryC);   // displays NaN - Not a Number
```

NaN - Not a Number

NaN stands for “Not a Number”

A number that can’t be represented

```
let test = 0/0;
console.log(typeof test); // "number"
console.log(test);  // NaN

let test = 0/0;
console.log(typeof test); // "number"
console.log(isNaN(test));    // NaN

let test = 10/0;
console.log(typeof test); // "number"
console.log(test);  // Infinity

// examples of the isNaN method
isNaN("Harris")   // returns true
isNaN("123.45")   // returns false
```

Adding JavaScript to a HTML page

- The `<script>` tag is used to define a client-side script (JavaScript)
- The <script> tag either contains scripting statements directly, or it points to an external script file through the src attribute
- The `<script>` tag can be placed in `<head>` tag or just before the closing `</body>` tag

Internal JavaScript

JavaScript embedded in HTML page

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello World</title>
    <script>
      alert("Hello World");
    </script>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

External JavaScript

JavaScript in external file saved with .js extension. This file is then linked to the HTML page using the src attribute of the `<script>` tag

- index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello World</title>
    <script src="js/main.js"></script>
  </head>
  <body onload="helloWorld();">
    <h1>Hello World</h1>
    <button onclick="helloWorld();">Click Me!</button>
  </body>
</html>
```

External JavaScript

- main.js

```
alert("Hello World 1");

function helloWorld() {

alert("Hello World 2");

}
```

Activity

Breakout

- Join a breakout room
- Download the unit 4 exercises code from Moodle
- Modify the html and text as you like
- You have 35 minutes
- Lecturer will visit each room in turn, etc...

Conditionals and Loops

Comparison Operators

Comparison operators are used to compare two values and will return Boolean values: either true or false

Name	Example	Description
Equal	x == y	Returns true if the operands are equal
Not equal	x != y	Returns true if the operands are not equal
Strict equal	x === y	Returns true if the operands are equal and have the same type
Strict not equal	x !== y	Returns true if the operands are not equal, or have different types
Greater than	x > y	Returns true if the left operand is greater than the right operand
Greater than or equal	x >= y	Returns true if the left operand is greater than or equal to the right operand
Less than	x < y	Returns true if the left operand is less than the right operand
Less than or equal	x <= y	Returns true if the left operand is less than or equal to the right operand

Comparison Operators - Examples

```
console.log(9 == 9); // true
console.log(9 != 20); // true
console.log(2 > 10); // false
console.log(2 < 10); // true
console.log(5 >= 10); // false
console.log(10 <= 10); // true
console.log("ABC" == "ABC"); // true
console.log("ABC" == "abc"); // false
console.log("Z" == "A"); // false
```

Comparison Operators - loose vs strict

String comparisons are case-sensitive, as shown in the example above

JavaScript also has two versions of each comparison operator: loose and strict
In strict mode, JavaScript will compare the types without performing a type coercion

You need to add one more equal = symbol to the operator as follows:

```
console.log("9" == 9); // true
console.log("9" === 9); // false
console.log(true == 1); // true
console.log(true === 1); // false
```

You should use the strict comparison operators unless you have a specific reason not to

Logical Operators

The logical operators are used to check whether one or more expressions result in either True or False

There are three logical operators that JavaScript has:

Name	Example	Description
Logical AND	x `&&` y	Returns true if all operands are true, else returns false
Logical OR	x `||` y	Returns true if one of the operands is true, else returns false
Logical NOT	`!` x	Reverse the result: returns true if false and vice versa

These logical operators follow the laws of mathematical logic:

1. `&&` AND operator - if any expression returns `false`, the result is `false`
2. `||` OR operator - if any expression returns `true`, the result is `true`
3. `!` NOT operator - negates the expression, returning the opposite

Control Flows

Control flow is a feature in a programming language that allows you to selectively run specific code based on the different conditions that may arise

Using control flows allows you to define multiple paths a program can take based on the conditions present in your program

There are two types of control flows commonly used in JavaScript:

1. conditionals
    - `if...else` , `switch..case`
2. loops
    - `for` ,`while`

Control Flows (Conditionals) - `if...else`

The `if` statement allows you to create a program that runs only if a specific condition is met

The syntax for the `if` statement is as follows:

```
if (condition) {
  // code to execute if condition is true
}
```

For example, suppose you want to go on a holiday that requires 2000 euro

Using the if statement, here's how you check if you have enough balance:

```
let balance = 3000;
if (balance > 2000) {
  console.log("You have the money for this trip. Let's go!");
}
```

Control Flows (Conditionals) -`if...else`

Suppose you need to run some code only when the `if` statement condition is not fulfilled

This is where the `else` statement comes in. The `else` statement is used to run code only when the `if` statement is not fulfilled

Here's an example:

```
let balance = 3000;
if (balance > 2000) {
  console.log("You have the money for this trip. Let's go!");
} else {
  console.log("Sorry, not enough money. Save more!");
}
console.log("The end!");
```

Now change the value of balance to be less than 5000, and you'll trigger the else block in the example

Control Flows (Conditionals) -`if...else`

The `else if` statement which allows you to write another condition to check should the `if` statement condition isn't met.

Consider the example below:

```
let balance = 3000;
if (balance > 2000) {
  console.log("You have the money for this trip. Let's go!");
} else if (balance > 1000) {
  console.log("You only have enough money for a staycation");
} else {
  console.log("Sorry, not enough money. Save more!");
}
console.log("The end!");
```

When the balance amount is less than 2000, the else if statement will check if the balance is more than 1000. If it does, then the program will proceed to recommend you do a staycation

You can write as many `else if` statements as you need, and each one will be executed only if the previous statement returns false

Control Flows (Conditionals) - switch...case

Often thought of as an alternative to the `if...else` statement that gives you more readable code, especially when you have many different conditions to assess

For example:

```
let age = 15;
switch (age) {
  case 10:
    console.log("Age is 10");
    break;
  case 20:
    console.log("Age is 20");
    break;
  default:
    console.log("Age is neither 10 or 20");
}
```

Type coercion does not apply to the `switch` statement so ensure the types of the `switch` value and `case` values match

Control Flows (Conditionals) - switch...case

The switch statement body is composed of three keywords:

- `case` keyword for starting a case block
- `break` keyword for stopping the `switch` statement from running the next case
- `default` keyword for running a piece of code when no matching case is found

When your expression finds a matching case, the code following the case statement will execute until it finds the break keyword. If you omit the break keyword, then the code execution will continue to the next block

For example:

```
switch (0) {
  case 1:
    console.log("Value is one");
  case 0:
    console.log("Value is zero");
  default:
    console.log("No matching case");
}
```

Control Flows (Conditionals) - switch...case

JavaScript will print the following log:

```
> "Value is zero"
> "No matching case"
```

Without the `break` keyword, `switch` will continue to evaluate the expression against the remaining cases even when a matching case is already found

Your `switch` evaluation may match more than one `case`, so the `break` keyword is commonly used to exit the process once a match is found. Finally, you can also put expressions as `case` values:

```
switch (20) {
  case 10 + 10:
    console.log("value is twenty");
    break;
}
```

Note the value for a `case` block must exactly match the `switch` argument

Control Flows (Conditionals) - switch...case

A common mistake is thinking the `case` value gets evaluated as `true` or `false`

You need to remember the differences between the `if` and `case` evaluations:

- `if` block will be executed when the test condition evaluates to true
- `case` block will be executed when the test condition exactly matches the given `switch` argument

Rule of Thumb only use `switch` when the code is cumbersome to write using `if`

Control Flows (Loops) - for

A Loop statement is another category of control flow statement used to execute a block of code multiple times until a certain condition is met

`for` statement

Say you want to print the numbers 1 to 10, you can use the for statement and write just a single line of code as follows:

```
for (let x = 0; x < 10; x++) {
  console.log(x);
}
```

Control Flows (Loops) - for

The `for` statement is followed by parentheses (()) which contain 3 expressions:

```
for ( [initialization]; [condition]; [arithmetic expression]) {
  // As long as condition returns true,
  // This block will be executed repeatedly
}
```

- The `initialization` expression, where you declare a variable to be used as the source of the loop condition. Represented as `x = 0` in the example
- The `condition` expression, where the variable in initialization will be evaluated for a specific condition. Represented as `x < 10` in the example
- The `arithmetic` expression, where the variable value is either incremented or decremented by the end of each loop, e.g. `x++`

These expressions are separated by a semicolon (`;`)

The `for` loop is useful **when you know how many times** you need to execute a repetitive task

Control Flows (Loops) - `while`

The `while` statement or `while` loop is used to run a block of code as long as the condition evaluates to `true`

You can define the condition and the statement for the loop as follows:

```
while (condition) {
  // As long as condition returns true,
  // This block will be executed repeatedly
}
```

Just like the `for` loop, the `while` loop is used to execute a piece of code over and over again until the condition becomes `false`

Control Flows (Loops) - `while`

For example:

```
let i = 0;

while (i < 6) {

  console.log(`The value of i = ${i}`);
  i++;
}
```

Here, the `while` loop will repeatedly print the value of `i` as long as `i` is less than `6`. In each iteration, the value of `i` is incremented by 1 until it reaches 6 and the loop terminates

Keep in mind that you need to include a piece of code that eventually turns the evaluating condition to `false` or the `while` loop will be executed forever, i.e. **an infinite loop**

The `while` loop is useful **when you don't know how many times** you need to execute the code

Activity

Breakout

- Join a breakout room
- Download the unit 4 exercises code from Moodle
- Modify the html and text as you like
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

Strings, Arrays and Functions

Strings

Strings are simply data defined as a series of characters

```
let message = "Hello, Sunshine!";
console.log(message); // Hello, Sunshine!
```

A string needs to be enclosed in quotations. You can use double quotes or single quotes, but they have to match

Templates (E6) are strings enclosed in backticks (`` `This is a template string` ``). Templates allow single and double quotes inside a string and can also span multiple line:

```
let message = `He's often called "Johnny"`;
console.log(message); // He's often called "Johnny"
```

Template literals provide a feature called **string interpolation**

```
const dog1 = 'Bach', dog2 = 'Bingo';
console.log(`My two dogs are called ${dog1} and ${dog2}.`); // My two dogs are called Bach and Bingo.
```

Strings - Escape Characters

Quotes in strings can cause issues.

```
let text = "We are the so-called "Vikings" from the north."; // "We are the so-called "
```

To solve this problem, you can use an backslash escape character

```
let text = "We are the so-called \"Vikings\" from the north.";
let text= 'It\'s alright.';
let text = "The character \\ is called backslash.";
```

Strings - Escape Characters

These 6 escape characters are also valid but were originally designed to control typewriters, teletypes, and fax machines

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

Strings - Methods

- A string is a primitive data type and so does not have methods and properties
- When you call a method or property on a string, JavaScript generates a wrapper object under the hood to allow the method or property to be performed
- This wrapper is disposed of afterwards

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
console.log(text); // ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Strings - Extracting Characters

There are 4 methods for extracting string characters:

- The `at(position)` method, e.g. `let char = text.at(0); // A`
- The `charAt(position)` method, e.g. `let char = text.charAt(0); // A`
- The `charCodeAt(position)` method, e.g. `let char = text.charCodeAt(0); // A`
- Using property access `[]` like in arrays, e.g. `let char = text[0]; // A`

`concat()` method

Similar to + and += operators. Concatenates one or more strings together

```
let a = 'When candles are out,';
let b = 'all cats are grey.';
let c = a.concat(' ', b);
console.log(c); // 'When candles are out, all cats are grey.'
```

Strings - Methods

`toLowerCase()` & `toUpperCase()` methods

`toLowerCase()` and `toUpperCase()` convert a string to lowercase and uppercase letters, respectively

```
let sentence = 'Always look on the bright side of life';
console.log(sentence.toLowerCase()); // always look on the bright side of life
console.log(sentence.toUpperCase()); // ALWAYS LOOK ON THE BRIGHT SIDE OF LIFE
```

`includes()` method

Checks if a specified string, passed as an argument, is present inside another string. The search is case-sensitive and returns a boolean. Can also specify a second argument stating the index at which to start searching

```
let sentence = 'Always look on the bright side of life';
sentence.includes('look up'); // false
sentence.includes('look on'); // true
sentence.includes('look on', 8); // false
```

Strings - Methods

`indexOf()` method

Searches for a substring and returns the first occurrence of the substring inside the calling string. It takes an optional parameter, indicating a specific index to start searching

```
let sentence = 'Always look on the bright side of life';
sentence.indexOf('l'); // 1
sentence.indexOf('l', 2); // 7
sentence.indexOf('l', 8); // 34
sentence.indexOf('L'); // -1
```

indexOf() returns the index of the first occurrence of the substring. If the substring is not found, it returns -1. Keep in mind that the search is case-sensitive

Strings - Methods

`startsWith()` & `endsWith()` methods

`startsWith()` method checks if a string begins with a specific sequence of characters and returns a boolean value. The search is case-sensitive. Takes an optional argument indicating the position in which to search from

```
let dish = 'Lemon curry';
dish.startsWith('lem'); // false
dish.toLowerCase().startsWith('lem'); // true
dish.startsWith('cu'); // false
dish.startsWith('cu', 6); // true
```

`endsWith()` method checks if a string ends with a specific sequence of characters, returning a boolean value and is case-sensitive. Takes an optional argument indicating the expected end position of the specified substring (the index of the expected final character + 1)

```
let dish = 'Lemon curry';
dish.endsWith('ry'); // true
dish.endsWith('on', 5); // true
```

Strings - Methods

`slice()` & `substring()` methods

Both methods return a portion of a string. The first argument passed to each method is the index of the first character to include in the string to extract. The second argument is the index of the first character to exclude:

```
let sentence = 'Always look on the bright side of life';
sentence.slice(7); // 'look on the bright side of life'
sentence.substring(7); // 'look on the bright side of life'
sentence.slice(0, 6); // 'Always'
sentence.substring(0, 6); // 'Always'
```

If the first index passed to `substring()` is greater than the second index, the two arguments are exchanged so that a string is still returned. In the same scenario, the `slice()` method returns an empty string instead:

```
let sentence = 'Always look on the bright side of life';
sentence.substring(11, 7); // 'look'
sentence.slice(11, 7); // ''
```

Strings - Methods

`split()` method

Takes a separator argument and breaks a string up, according to the occurrence of the separator character inside the string. Then, it returns an array of strings

Takes an optional argument, indicating the maximum number of items to put inside the array

```
let sentence = 'Always look on the bright side of life';
sentence.split(' '); // ['Always', 'look', 'on', 'the', 'bright', 'side', 'of', 'life']
sentence.split(' ', 5); // ['Always', 'look', 'on', 'the', 'bright']
```

Strings - Methods

`match()` method

Searches for a specific pattern – passed as a regular expression – inside a string, and returns an array containing the matching results. See https://www.freecodecamp.org/news/regular-expressions-for-beginners/ for a guide to regular expressions

```
const tongueTwister = "How much wood would a woodchuck chuck if a woodchuck could chuck wood?"
const regex1 = /(w|c)o*(ul)?d/g;
const regex2 = /wool/g;

tongueTwister.match(regex1); // ['wood', 'would', 'wood', 'wood', 'could', 'wood']
tongueTwister.match(regex2); // null
```

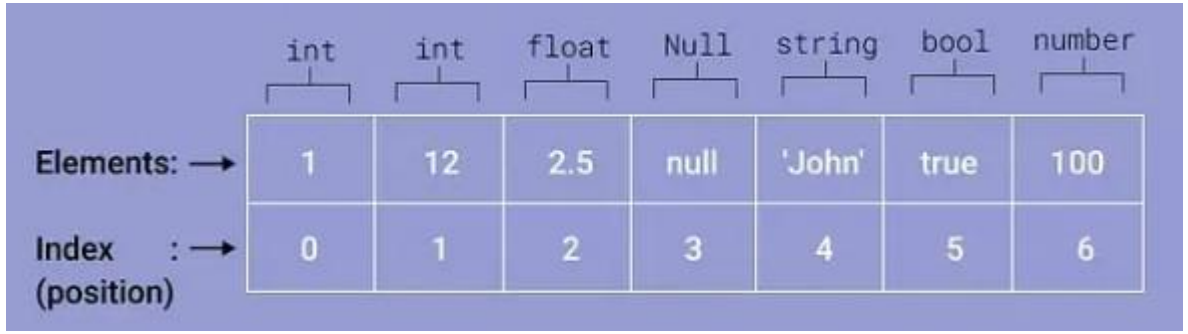
If you only need to know if a pattern is present or not inside a string, you should use `test()` which returns a `boolean` .

Arrays

An array is an object data type that can be used to hold more than one value and uses numbered indexes. An array can be a list of strings, numbers, booleans, objects, or a mix of them all

To create an array, there are two options square brackets `[]` or `new Array()` and separate the items using a comma

```
let emptyArray = [] // empty array
let myArray = [1, 12, 2.5, null, 'John', true, 100]; // preferred method
// or
let myArray = new Array(1, 12, 2.5, null, 'John', true, 100);
```



Each array element has an associated index number starting from zero

Arrays - Accessing and changing elements

To access or change the value of an array, you need to add the square brackets notation [x] next to the array name, where x is the index number of that element

```
console.log(myArray[0]); // 1
console.log(myArray[4]); // John
console.log(myArray.at(0)); // 1 - similar to [0], introduced in ES2022
```

To change an element:

```
myArray[0] = 'Sean';
console.log(myArray[0]); // Sean
```

To get the `length` of an array:

```
console.log(myArray.length); // 7
```

Note: `length` is a property and not a method

Arrays - Methods

`push()`

use `push()` to add a new element to the end of the array:

```
myArray.push('Jane');
console.log(myArray); // 'Sean', 12, 2.5, null, 'John', true, 100, 'Jane'
```

`pop()`

use `pop()` to remove an element from the end of the array:

```
myArray.pop();
console.log(myArray); // 'Sean', 12, 2.5, null, 'John', true, 100
```

- use `unshift()` to add a new element to the start of the array
- use `shift()` to remove the first element of the array

Arrays - Methods

`toString()` converts an array to a string of (comma separated) array values

```
console.log(myArray.toString()); // Sean,12,2.5,null,John,true,100
```

`join()` behaves like `toString()` but allows you to specify the separator

```
console.log(myArray.join(' ')); // Sean 12 2.5 null John true 100
```

`delete()` deletes the specified element from the array but will leave an empty hole in the array. Use pop() or shift() instead

```
myArray.delete(2); // 'Sean', 12, empty, null, 'John', true, 100
```

Arrays – Search Methods

`indexOf()`

Searches an array for an element value and returns its position. Returns -1 if the element is not found

```
let position = myArray.indexOf('Sean') + 1; // 1
```

`lastIndexOf()` is the same as `indexOf()` , but returns the position of the last occurrence of the specified element

`includes()`

`includes()` (ES2016) allows us to check if an element is present in an array (including NaN, unlike indexOf)

```
let foundSean = myArray.includes('Sean'); // true
let foundJane = myArray.includes('Jane'); // false
```

Arrays - Methods

`sort()`

Sorts an array alphabetically:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort(); // Apple,Banana,Mango,Orange
```

To sort an array containing numbers or objects you will need to write a compare function See https://www.w3schools.com/js/js_array_sort.asp for details

`reverse()`

reverses the elements in an array:

```
fruits.reverse(); // Mango,Apple,Orange,Banana
```

Use `toSorted()` and `toReversed()` to return a new array rather than modifying the original array

Functions

As we saw previously, a function is a block of code designed to perform a particular task. The function is executed when "something" invokes it (calls it)

```
function name(parameter list) {
  statements in the function body
  return value;
}
let value = name(argument list); // call the function

// Expression; the function is anonymous but assigned to a variable
const name2 = function (parameter list) {
  statements in the function body
  return value;
}
let value2 = name2(argument list); // call the function
```

Functions – Default Parameters

You can set a default parameter for any parameter.

```
function greet(name = "Nathan") {
  console.log(`Hello, ${name}!, Nice weather today, right?");
}

console.log(greet()); // Hello, Nathan! Nice weather today, right?
console.log(greet("Jack")); // Hello, Jack! Nice weather today, right?
```

When you pass undefined to a function that has a default parameter, the default parameter will be used:

```
function greet(name = "John"){
  console.log(name);
}
greet(undefined); // John
greet(null); // null
```

But when you pass null to the function, the default parameter will be ignored

Functions – Variable scope

A variable declared inside a function can only be accessed from that function. This is because that variable has a local scope

A variable declared outside of any block is known as a global variable because of its global scope

```
let globalString = "This is a global variable";
function greet() {
  console.log(globalString); // This is a global variable
  let localString = "This is a local variable";
}

greet();
console.log(myString); // ReferenceError: myString is not defined
```

In practice, you rarely need to declare the same variable in different scopes:

- Any variable declared outside a function shouldn't be used inside a function without passing them as parameters
- A variable declared inside a function should never be referred to outside of that function

Functions – Rest parameter

The rest parameter is a parameter that can accept any number of data as its arguments. The arguments will be stored as an array

```
function printArguments(...args){
    console.log(args);
}

printArguments("A", "B", "C"); // [ 'A', 'B', 'C' ]
```

A function can only have one rest parameter, and the rest parameter must be the last parameter in the function.
Useful when your function needs to work with an indefinite number of arguments

Functions - First-call Function

A programming language is said to have First-class functions when functions in that language are treated like any other variable

In such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable

```
function sayHello() { // in this instance, sayHello() is called a callback function
  return "Hello, ";
}
function greeting(helloMessage, name) {
  console.log(helloMessage() + name);
}
// Pass `sayHello` as an argument to `greeting` function
greeting(sayHello, "JavaScript!");
// Hello, JavaScript!
```

Functions - Arrow functions

The arrow function syntax allows you to write a JavaScript function with a shorter, more concise syntax

```
function greetings(name) {
  console.log(`Hello, ${name}!`);
}
greetings("John"); // Hello, John!

const greetings = (name) => {
  console.log("");
  console.log(`Hello, ${name}!`);
};
const greetings = (name) => console.log(`Hello, ${name}!`);

greetings("John"); // Hello, John!
```

Note:

- curly brackets can be omitted for single statements
- the round brackets can be omitted when you have exactly one parameter for the function
- round brackets are required when you have no parameters or more than one parameter

Activity

Breakout

- Join a breakout room
- Download the unit 4 exercises code from Moodle
- Modify the html and text as you like
- You have 35 minutes
- Lecturer will visit each room in turn, etc...

Summary

Completed this Week

- Overview of JavaScript in the browser
- Conditionals and Loops
- Strings, Arrays and Functions

For next Week

- Complete the remaining exercises for unit 4 before next class
- Review the slides and examples for unit 5