Section 1: Fundamentals

JS Fundamentals:

- **JavaScript (JS)** is a high-level, Object-Oriented programming language.
- **JS** along with **HTML** and **CSS** form the 3 main technologies that work together to create websites or applications:
    - **HTML** - responsible for content of the page (text, images, buttons, etc.)
    - **CSS** - responsible for presenting the content (styling, layout, etc)
    - **JS** - main programming language for web apps (dynamic and interactive effects). Used to manipulate **HTML** and **CSS** content.

Web Application Example
`<p></p>` – paragraph (HTML) **`p {color: red}` –** paragraph text is red (CSS) **`p.hide();` –** function to hide the paragraph (JS)
we can think of these 3 technologies like the parts of speech: HTML is the Noun - p is a paragraph (thing = noun) CSS is the Adjective - the paragraph is red (describing a noun = adjective) JS is the Verb - hide the paragraph (action = verb)

Including JavaScript in HTML document

```
<head>
  <script>
        let js = 'amazing';
        if (js === 'amazing') alert('JavaScript is FUN!');
  </script>
</head>
```

Within the **head element** we can include a **script element** to include JavaScript code that operates on our website.

This simple example declares a variable **js** and assigns it a value of **amazing.** We then check, using boolean logic, if **js is equal to amazing,** we then show an alert box with the set text.

Values and Variables:

In JavaScript, a value is a piece of data that can be used in a program. Values can be of different types, such as numbers, strings, objects, arrays, and functions. For example, **42, "Hello, World!"**, and **[1, 2, 3]** are all **values**.

A variable, on the other hand, is a **named storage location** that **can hold a value**. You can think of a variable as a **container for a value**. Variables are created using the **var, let, or const keyword.**

Value & Variable example

```
let name = "John";
const age = 30;
```

In this example, **name** and **age** are variables. "**John**" and **30** are values. The **name variable** is assigned the **value** "**John**", and the **age variable** is assigned the **value 30**.

The **let keyword** is used to declare a variable that can be reassigned later, while the **const keyword** is used to declare a variable whose value should not be reassigned after its initial assignment.

Variable naming rules:

1. Can only contain letters, numbers, underscores, and dollar signs
2. Cannot start with a number
3. Cannot use reserved JavaScript keywords

Variable naming conventions:

1. Use camelCase
2. Descriptive names

Data Types

1. **Number** - Floating point numbers, used for decimals and integers
2. **String** - Sequence of characters, used for text
3. **Boolean** - Logical type that can only be true or false, used for taking decisions
4. **Undefined** - Value taken by a variable that is not yet defined ('empty value')
5. **Null** - Also means 'empty value'
6. **Symbol (ES2015)** - Value that is unique and cannot be changed
7. **BigInt (ES2020)** - Larger integers than the Number type can hold

Primitive Data Types Examples

```
let num = 25;
let floatNum = 25.5;

let str = "Hello, World!";

let isTrue = true;
let isFalse = false;

let x;
console.log(x);                 // undefined


let y = null;

const sym = Symbol('description');

const bigInt = 1234567890123456789012345678901234567890n;
```

Non-Primitive Data Types Examples

```
let person = {
    name: "John",
    age: 30
};                                          // object


let arr = [1, 2, 3, 4, 5];                  // array

function greet() {
    console.log("Hello, World!");
}                                           // function
```

Special Values

```
let notANumber = 0/0;
console.log(notANumber);        // NaN

let infinity = 1/0;
console.log(infinity);          // Infinity

let negInfinity = -1/0;
console.log(negInfinity);       // -Infinity
```

NaN: Represents a special 'Not a Number' value. It is the result of an operation that cannot produce a normal result.

Infinity: Represents a special value that is greater than any other number.

-Infinity: Represents a special value that is less than any other number.

JavaScript has dynamic typing: We **do not** have to manually define the data type of the value stored in a variable. Instead, data types are determined automatically.

Reassigning values

```
let javascriptIsFun = true;             // declaration for bool
console.log(javascriptIsFun);           // prints true

...

javascriptIsFun = 'Yes!'
console.log(javascriptIsFun);          // prints Yes!
```

We can reassign the values that are stored in the variables by **using an assignment operator** without using the **let keyword.**

Constants

```
const birthYear = 1991;
birthYear = 1990;                      // Error - cannot reassign constant
const job;                             // Error - must assign value to constant when declared
```

Using the **const keyword** declares a variable as constant and cannot be reassigned.

A **const** variable must have a value declared.

Operators:

We have various operators we can use for numerical logic, assignment, comparison, relations, etc.

Operator examples

```
// multiplication operator
console.log(ageJohn * 2);

// division operator
console.log(ageJohn / 10);

// power operator
console.log(ageJohn ** 2);          // ageJohn to the power of 2

// concatenation operator
const firstName = 'John';
const lastName = 'Smith';
console.log(firstName + ' ' + lastName);
// concatenation operator used to join strings

// assignment operators
let x = 10 + 5;                 // 15 - addition operator
x += 10;                        // x = x + 10 = 25 - addition assignment operator
x *= 4;                         // x = x * 4 = 100 - multiplication assignment operator
x++;                            // x = x + 1 = 101 - increment operator
x--;                            // x = x - 1 = 100 - decrement operator
console.log(x);

// comparison operators
console.log(ageJohn > ageMark);      // greater than operator
console.log(ageMark >= 18);          // greater than or equal to operator
console.log(ageJohn < ageMark);      // less than   operator
console.log(ageMark <= 18);          // less than or equal to operator
```

```
// relational operators
const isFullAge = ageMark >= 18;          // true
console.log(now - 1959 > now - 1962);   // true

// operator precedence
let y, z;
y = z = 25 - 10 - 5;        // y = z = 10, y = 10
console.log(y, z);

const averageAge = (ageJohn + ageMark) / 2;
// parentheses operator has higher precedence than the division operator

console.log(ageJohn, ageMark, averageAge);
```

Strings and Template Literals:

In JavaScript, a string is a sequence of characters enclosed in single, double, or backticks (grave accents). Strings are used to store and manipulate text.

String Example

```
let singleQuotes = 'Hello, World!';
let doubleQuotes = "Hello, World!";
let backticks = `Hello, World!`;
```

Template Literals

Template literals are a more advanced form of strings that make it easier to embed expressions (variables, expressions, function calls, etc.) into strings. Template literals are enclosed by backticks **(``)** and can contain placeholders, which are indicated by **${expression}**. The expressions in the placeholders and the text between them get passed to a function.

Template Literal Example

```
let name = "John";
let age = 30;

let message = `Hello, my name is ${name}, and I am ${age} years old.`;

console.log(message);

// Output: Hello, my name is John, and I am 30 years old.
```

In this example, the name and age variables are embedded into the message string using the ${} syntax.

Multi-line Strings Example

```
let multiLineString = `This is
a multi-line
string.`;

console.log(multiLineString);

// Output:
// This is
// a multi-line
// string.
```

Tagged Template Literals

Tagged template literals allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

Tagged Template Literal Example

```
function tag(strings, ...values) {
    console.log(strings);
    console.log(values);
}

let name = "John";
let age = 30;

tag`Hello, my name is ${name}, and I am ${age} years old.`;

// Output:
// [ 'Hello, my name is ', ', and I am ', ' years old.' ]
// [ 'John', 30 ]
```

In this example, the tag function logs the strings and values arrays to the console. The strings array contains all the plain text parts of the template literal, and the values array contains all the evaluated expressions.

Type Conversion and Coercion

Type conversion, also known as type casting, is the process of converting a value from one data type to another. In JavaScript, you can convert values between different data types explicitly or implicitly.

Explicit Conversion

Explicit conversion is when you manually convert one data type to another using built-in JavaScript functions or by wrapping the value with the desired type constructor.

Explicit Conversion example

```
let str = "123";
let num = Number(str);
console.log(num);           // 123

let num2 = 123;
let str2 = String(num2);
console.log(str2);          // "123"

let bool = true;
let num3 = Number(bool);
console.log(num3);          // 1
```

Implicit Conversion

Implicit conversion, or coercion, is when JavaScript automatically converts one data type to another without you explicitly requesting the conversion. This usually happens when you perform operations on values of different data types.

Implicit Conversion example

```
let result1 = "123" + 5;
console.log(result1);       // "1235"

let result2 = "123" - 5;
console.log(result2);       // 118

let result3 = "5" * "10";
console.log(result3);       // 50
```

In the first example, the number **5** is implicitly converted to a string and concatenated with the string **"123"**, resulting in the string **"1235"**. In the second example, the string **"123"** is implicitly converted to a number and the number **5** is subtracted from it, resulting in the number **118**. In the third example, both strings **"5"** and **"10"** are implicitly converted to numbers and multiplied, resulting in the number **50.**

Type Coercion

Type coercion is a special case of implicit type conversion. It occurs when you use the **== operator** to compare values of different data types. JavaScript will automatically convert one or both of the values to the same type before performing the comparison.

Type Coercion example

```
let str = "123";
let num = 123;

console.log(str == num);      // true
console.log(str === num);     // false
```

In this example, the str variable is coerced to a number before comparing it with the num variable using the **== operator**, so the result is true. However, when using the **=== operator,** no type coercion occurs, and the values are compared as is, so the result is false.

It is generally recommended to use the **=== operator** to avoid unexpected results due to type coercion.

Truthy and Falsy Values:

In JavaScript, a value is considered "truthy" if it evaluates to true in a boolean context. Conversely, a value is considered "falsy" if it evaluates to false in a boolean context.

Falsy Values

There are only a few falsy values in JavaScript:

- false
- 0 (zero)
- "" (empty string)
- null
- undefined
- NaN (Not a Number)

Every other value in JavaScript is considered truthy.

Truthy and Falsy Values Example

```
if (0) {
    console.log("This isn't printed because 0 is falsy.");
}

if (1) {
    console.log("This is printed because 1 is truthy.");
}

let name = "";

if (name) {
    console.log("This isn't printed because an empty string is
falsy.");
}

name = "John";

if (name) {
    console.log("This is because a non-empty string is truthy.");
}
```

In this example, the first and third if statements will not be executed because **0** and **""** are **falsy** values. The second and fourth if statements will be executed because **1** and "**John**" are **truthy** values.

Important Note:

Truthy and falsy values are not the same as true and false. For example, the string "false" is actually truthy because it is a non-empty string. Similarly, the string "0" is also truthy because it is a non-empty string. To check if a value is truthy or falsy, you can use the Boolean function:

Boolean Function

```
console.log(Boolean("false"));          // true
console.log(Boolean("0"));              // true
console.log(Boolean(0));                // false
```

Equality Operators:

JavaScript has three main equality operators: **==, ===,** and **!=.**

- **== (Equality Operator):** This operator checks if two values are equal after performing any necessary type conversions. This means that it will convert the values to the same type before performing the comparison. This is also known as "loose" or "abstract" equality.

- **=== (Strict Equality Operator):** This operator checks if two values are equal without performing any type conversions. This means that the values must be of the same type to be considered equal. This is also known as "strict" equality.

- **!= (Inequality Operator):** This operator checks if two values are not equal after performing any necessary type conversions. This is the "loose" or "abstract" inequality operator.

- **!== (Strict Inequality Operator):** This operator checks if two values are not equal without performing any type conversions. This is the "strict" inequality operator.

Equality Operator Examples

```
console.log(5 == "5");              // true
console.log(null == undefined);     // true
```

```
console.log(5 === "5");             // false
console.log(null === undefined);    // false
```

```
console.log(5 != "5");              // false
console.log(null != undefined);     // false
```

```
console.log(5 !== "5");             // true
console.log(null !== undefined);    // true
```

It is generally recommended to use the **`===`** and **`!==`** operators to avoid unexpected results due to type coercion. These operators are more predictable because they do not perform any type conversions.

Switch Statement:

A switch statement in JavaScript is used to perform different actions based on different conditions. It is like a series of if...else if... statements, but it makes the code cleaner and easier to read.

Switch Statement Example Syntax

```
switch(expression) {
    case value1:
        // code block for value1
        break;
    case value2:
        // code block for value2
        break;
    ...
    default:
        // code block if no cases are true
}
```

expression: This is the value that will be compared against each case value.

case value1, value2, ...: These are the values against which the expression is compared.

break: This statement is used to exit the switch block once a case matches. If you omit the break statement, the switch block will continue to execute the next case even if the match is found.

default: (optional) This is the code block that will be executed if no case values match the expression.

Ternary Operator:

The ternary operator is a shorthand way of writing an if-else statement. It's also known as the conditional operator.

Ternary Operator Syntax

```
condition ? expression_if_true : expression_if_false
```

condition: This is the expression that is evaluated as true or false.

expression_if_true: This expression is executed if the condition is true.

expression_if_false: This expression is executed if the condition is false.

Ternary Operator Example

```
var age = 20;
var type = age >= 18 ? 'Adult' : 'Minor';
console.log(type);                          // Output: Adult
```

In this example, the **condition is age >= 18**, which is true since age is 20. Therefore, the **expression_if_true** is executed, and type is assigned the value **'Adult'.**

Ternary Operator Nesting

```
var age = 20;
var type = age >= 18 ? (age >= 65 ? 'Senior' :
'Adult') : 'Minor';
console.log(type);                          // Output: Adult
```

In this example, since age is 20, the **first condition age >= 18 is true**, so the **next ternary operator** is evaluated. Since 20 is not greater than or equal to 65, type is assigned the value **'Adult'.**

Strict Mode:

Strict Mode is a feature in JavaScript that was introduced in ECMAScript 5 (ES5) to help you write more reliable and maintainable code by catching common mistakes and preventing the use of certain error-prone features. It is recommended to always use strict mode in your code.

To enable strict mode, you simply add the string **'use strict'** to the top of your script or function.

Strict Mode Declaration

```
'use strict';

let x = 10;
```

In this example, strict mode is enabled for the **entire script.**

```
function myFunction() {
    'use strict';
    let x = 10;
}
```

In this example, strict mode is only enabled for **myFunction.**

Here are some key differences between strict mode and non-strict mode (sloppy mode):

Variable Declaration: In strict mode, you must declare variables with var, let, or const before using them. In non-strict mode, variables are automatically declared in the global scope if you don't declare them.

```
'use strict';

x = 10;                    // Error: x is not defined
```

Duplicate Parameter Names: In strict mode, you cannot have duplicate parameter names in function declarations. In non-strict mode, this is allowed.

```
'use strict';

function myFunction(a, a) {
// Error: Duplicate parameter name not allowed in this context

    return a;
}
```

Read-Only Properties: In strict mode, assigning a value to a read-only property or a non-writable global variable will throw an error. In non-strict mode, the assignment will silently fail.

```
'use strict';

let obj = Object.freeze({ x: 10 });

obj.x = 20;          // Error: Cannot assign to read-only property 'x' of object '#<Object>'
```

this* Value:** In strict mode, the value of ***this is ***undefined*** in functions that are not methods or constructors. In non-strict mode, this refers to the global object.

```
'use strict';

function myFunction() {
    return this;
}

console.log(myFunction());                    // Output: undefined
```

Octal Literals: In strict mode, octal literals **(e.g., 0123)** are not allowed. In non-strict mode, octal literals are allowed.

```
'use strict';

let x = 0123;     // Error: Octal literals are not allowed in strict mode.
```

Functions:

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value—but a function can also return a value.

Function Declaration Example

```
function greet(name) {
    return 'Hello, ' + name + '!';
}

console.log(greet('Alice'));              // Output: Hello, Alice!
```

In this example, a function **greet is declared**, which takes one parameter name. The function **returns a greeting message**.

Arrow Function Example

```
let greet = (name) => 'Hello, ' + name + '!';

console.log(greet('Charlie'));           // Output: Hello, Charlie!
```

Arrow functions were introduced in ECMAScript 6 (ES6) and provide a shorter syntax for writing function expressions.

In this example, an arrow function is created and assigned to the variable **greet.** It takes in the string **name**, and then **returns** a string concatenation including the input parameter.

Immediately Invoked Function Expression

```
(function(name) {
    console.log('Hello, ' + name + '!');
})('David');                                        // Output: Hello, David!
```

An IIFE is a function that runs as soon as it is defined.
In this example, the function is executed immediately after it is created.

Callback Function

```
function greet(name, callback) {
    console.log('Hello, ' + name + '!');
    callback();
}

function sayGoodbye() {
    console.log('Goodbye!');
}

greet('Emma', sayGoodbye);

// Output:
// Hello, Emma!
// Goodbye!
```

A callback function is a function that is passed as an argument to another function and is executed after the completion of that function.

In this example, **sayGoodbye** is a callback function that is passed as an argument to the **greet** function and is executed after the **greet** function is completed.

Functions as Values

```
const calcAge2 = function (birthYear) {
        return 2037 - birthYear;
}
const age2 = calcAge2(1991);
```

In JavaScript, functions are considered values, and **we can store these values in variables**.

Arrays:

Arrays are used to store multiple values in a single variable. An array is a special type of object that is used to represent a list of values (elements). Each value (element) in the array is identified by its index, **which starts at 0.**

Array Example
`let fruits = ['Apple', 'Banana', 'Cherry'];`
In this example, an array called **fruits** is created with three elements.

Manipulating Arrays

```
let fruits = ['Apple', 'Banana', 'Cherry'];

console.log(fruits[0]);                          // Output: Apple


fruits[1] = 'Blueberry';
console.log(fruits);                             // Output: ['Apple', 'Blueberry', 'Cherry']


fruits.push('Date');
console.log(fruits);                             // Output: ['Apple', 'Blueberry', 'Cherry', 'Date']


let last = fruits.pop();
console.log(last);                               // Output: Date
console.log(fruits);                             // Output: ['Apple', 'Blueberry', 'Cherry']


let index = fruits.indexOf('Cherry');
console.log(index);                              // Output: 2
```

You can see the various ways to Manipulate elements in an array, including:

- Accessing elements with their [index number]
- Modifying elements
- Adding elements
- Removing elements
- Finding the index of an element

Looping over Arrays

```
fruits.forEach(function(item, index) {
    console.log(index, item);
});

// Output:
// 0 Apple
// 1 Blueberry
// 2 Cherry
```

In this example, the **forEach** method is used to loop over the fruits array and log each item and its index.

Merging Arrays

```
let fruits = ['Apple', 'Banana', 'Cherry'];
let vegetables = ['Carrot', 'Potato'];
let food = fruits.concat(vegetables);
console.log(food);                    // Output: ['Apple', 'Blueberry', 'Cherry', 'Carrot', 'Potato']
```

In this example, the **concat** method is used to merge the **fruits** and **vegetables** arrays into a new array; **food**.

Objects:

Objects in JavaScript are collections of key-value pairs, where the keys are strings (or symbols) and the values can be any data type. Objects are used to store related data and functions as properties and methods.

Creating Objects

```
const car = {
    make: 'Toyota',
    model: 'Corolla',
    year: 2005
};
```

In this example, an **object car** is created with three properties: **make, model, and year.**

Accessing Object Properties

```
console.log(car.make);                    // Output: Toyota
console.log(car['model']);                // Output: Corolla
```

You can access the properties of an object using **dot notation** or **bracket notation.**

Modifying Object Properties

```
car.year = 2010;
console.log(car.year);                    // Output: 2010
```

You can modify the properties of an object using **dot notation** or **bracket notation.**

Adding Object Properties

```
car.color = 'Blue';
console.log(car.color);                    // Output: Blue
```

You can add new properties to an object using **dot notation** or **bracket notation.**

Removing Object Properties

```
delete car.color;
console.log(car.color);                    // Output: undefined
```

You can remove properties from an object using the **delete** operator.

Object Methods:

Methods are functions that are stored as properties of an object.

Object Methods

```
let car = {
    make: 'Toyota',
    model: 'Corolla',
    year: 2005,
    start: function() {
        console.log('Car started');
    }
};

car.start();                                    // Output: Car started
```

In this example, a **start** method is added to the **car** object.

Nested Objects

```
let person = {
    name: 'Alice',
    age: 30,
    address: {
        street: '123 Main St',
        city: 'New York'
    }
};

console.log(person.address.city);          // Output: New York
```

Objects can be nested inside other objects. In this example, the **address property** of the **person object** is another object.

<u>More Iteration:</u>

- **For Loop**

A f**or loop** is a control flow statement that is used to repeatedly execute a block of code a specified number of times. It is commonly used for iterating over arrays or for running a block of code a certain number of times.

for loop Syntax

`for (initialization; condition; final expression) {`
`    // code to be executed`
`}`

initialization: This is executed only once at the beginning of the loop. It is typically used to declare and initialize a loop counter variable. **condition:** This is evaluated before each iteration of the loop. If the condition is true, the loop will continue to run; if it is false, the loop will stop. **final expression:** This is executed at the end of each iteration of the loop. It is typically used to update the loop counter variable.

for loop Example

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
// Output:
// 0
// 1
// 2
// 3
// 4
```

In this example, the initialization is **let i = 0**, the condition is **i < 5**, and the final expression is **i++**.

The loop will run as long as **i is less than 5**, and i will be incremented by 1 at the end of each iteration.

Looping Backwards

```
let fruits = ['Apple', 'Banana', 'Cherry'];

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}

// Output:
// Apple
// Banana
// Cherry
```

In this example, the for loop is used to iterate over the **fruits** array and log each element.

Nested Loops

```
for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 2; j++) {
        console.log('i=' + i + ', j=' + j);
    }
}

// Output:
// i=0, j=0
// i=0, j=1
// i=1, j=0
// i=1, j=1
// i=2, j=0
// i=2, j=1
```

In this example, there are two for loops: an outer loop with the counter variable i, and an inner loop with the counter variable j.

The inner loop will complete all of its iterations (j=0 and j=1) before the outer loop moves on to the next iteration (i=1, then i=2).

While Loops

```
let i = 0;

while (i < 5) {
    console.log(i);
    i++;
}

// Output:
// 0
// 1
// 2
// 3
// 4
```

In this example, the condition is **i < 5**. The loop will run as long as **i is less than 5**, and **i will be incremented by 1 at the end of each iteration.**

Section 2: HTML & CSS Fundamentals

HTML:

- **Stands for - HyperText Markup Language**

HTML syntax

```
<HTML>
  <HEAD>
    <META NAME="GENERATOR" Content="Microsoft Visual Studio">
    <TITLE>Learning HTML and CSS</TITLE>
  </HEAD>
  <BODY>
    <h1>JavaScript is fun, so is HTML & CSS!</h1>
            <p>You can learn JavaScript without HTML and CSS,
               but for DOM manipulation it's useful to have
               some basic ideas of HTML and CSS.
            </p>
  </BODY>
</HTML>
```

All **HTML** documents include an opening and closing tag

Inside the element we include the **head** and **body** sections

- The **head** is where **meta-information** about the document is located (eg. **Title**, CSS styles, FATF icons, etc)
- the **body** is where the actual content of the document goes, this is what is visible on the page (eg. **headings**, **paragraphs**, visuals, etc.)

HTML syntax 2 (more detail)

```
<!DOCTYPE html>                                  This declares the document to be HTML5
<html lang="en">        The root element of an HTML page, specifies the language as English

<head>                                  This contains meta-information about the document
    <meta charset="UTF-8">           This specifies the character encoding for the document

<meta name="viewport" content="width=device-width, initial-scale=1.0">
This makes the webpage responsive on different devices

    <title>Sample Page</title>
Sets the title of the webpage that appears on the browser tab
</head>

<body>                           This contains the content of the page that's visible to the user
    <header>        Typically used for the top section of a webpage, like a logo or navigation bar
        <h1>Welcome to My Sample Page</h1>                        This is a top-level heading
    </header>

    <nav>                                                   This is used for navigation links
        <a href="#">Home</a> |                           This is a hyperlink to the home page
        <a href="#">About</a> |                         This is a hyperlink to the about page
        <a href="#">Contact</a>                       This is a hyperlink to the contact page
    </nav>

    <main>                                      This is the main content area of the webpage
        <p>This is a sample paragraph. It contains text that provides
information to the reader.</p>                                   This is a paragraph element
    </main>

    <footer>    Typically used for the bottom of a webpage, like copyright information or links
        <p>Copyright © 2023. All rights reserved.</p>
    </footer>
</body>
</html>
```

Breakdown:

<!DOCTYPE html>: This declaration defines the document to be HTML5.
<html>: The root element of an HTML page.
<head>: Contains meta-information about the document.
<meta charset="UTF-8">: Specifies the character set used in the document, which is UTF-8 in this case.
<meta name="viewport" content="width=device-width, initial-scale=1.0">: Helps make the webpage look right on all devices, including mobile.
<title>: Sets the title of the webpage that appears on the browser tab.
<body>: Contains all the content of the page that's visible to the user.
<header>, <nav>, <main>, <footer>: These are semantic elements that help describe the kind of content they contain.
<h1>: A top-level heading. There are six levels of headings in HTML: <h1> to <h6>.
<a href="#">: Defines a hyperlink, which can be clicked to navigate to another page or resource.
<p>: Represents a paragraph of text.

HTML Elements:

HTML elements are the building blocks of web content. They define and structure content on the web. Each element is represented by a start tag, some content, and an end tag.

Common HTML Elements

1. **`<!DOCTYPE>:`** Declares the document type and version of HTML.
2. **`<html>:`** The root element that wraps all the content on the entire page.
3. **`<head>:`** Contains meta information about the document, such as its title and links to its scripts and styles.
4. **`<title>:`** Specifies the title of the document, which is displayed in the browser's title bar or tab.
5. **`<body>:`** Contains the visible content of the HTML document.
6. **`<h1>, <h2>, ... <h6>:`** Headings. <h1> is the highest and most important level, and <h6> is the lowest.
7. **`<p>:`** Represents a paragraph of text.
8. **`<a>:`** Anchor element, used to define hyperlinks.
9. **`<img>:`** Used to embed images. It's a self-closing tag.
10. **`<ul>:`** Unordered list, which is used with <li> (list items) to create bullet lists.
11. **`<ol>:`** Ordered list, which is used with <li> to create numbered lists.
12. **`<br>:`** Line break. It's a self-closing tag.
13. **`<hr>:`** Creates a thematic break or horizontal line. It's also a self-closing tag.
14. **`<div>:`** A generic container that doesn't have a specific semantic meaning. Often used with CSS and JavaScript.
15. **`<span>:`** Similar to <div>, but for inline elements instead of block-level elements.
16. **`<form>:`** Used to collect user input. Contains form elements like <input>, <textarea>, and <button>.
17. **`<table>:`** Represents a table. Used in conjunction with <tr> (table row), <td> (table data), and <th> (table header).
18. **`<iframe>:`** Inline frame used to embed another document within the current HTML document.
19. **`<script>:`** Used to embed or reference external JavaScript code.
20. **`<link>:`** Used to link external resources, like CSS files.
21. **`<meta>:`** Provides metadata about the HTML document, such as character set, viewport settings, and author information.

HTML Attributes:

HTML attributes provide additional information about an element and help define its properties or behavior. They are always specified in the start tag of an element and usually come in name/value pairs like name="value".

Common HTML Attributes

1. **`class:`** Specifies one or more class names for an element. Used primarily for styling and JavaScript.
2. **`id:`** Specifies a unique id for an element. It's used for styling, JavaScript, and as a fragment identifier in URLs.
3. **`style:`** Allows inline CSS styling for an element.
4. **`src:`** Specifies the source URL for elements like <img>, <script>, and <iframe>.
5. **`href:`** Specifies the URL for linked resources, used in the <a>, <link>, and <base> elements.
6. **`alt:`** Provides an alternative text for images (<img>). It's displayed if the image can't be loaded and is essential for accessibility.
7. width and height: Define the width and height of elements like <img>, <canvas>, and <iframe>.
8. **`disabled:`** Indicates that an input element, like a button or text field, is disabled and can't be interacted with.
9. **`placeholder:`** Provides a hint to the user of what can be entered in an input field (<input> or <textarea>).
10. **`required:`** Indicates that an input field must be filled out before submitting a form.
11. **`readonly:`** Indicates that an input field is read-only and its value cannot be changed.
12. **`value:`** Defines the default value or current value of input elements or buttons.
13. **`type:`** Specifies the type of input element, e.g., text, password, checkbox, radio, etc.
14. **`rel:`** Specifies the relationship between the current document and the linked resource. Commonly used with <link> and <a>.
15. **`target:`** Specifies where to open the linked resource. Common values include _blank (new tab/window) and _self (same frame).
16. **`lang:`** Specifies the language of the element's content, aiding in translation and accessibility.
17. **`title:`** Provides additional information about an element, typically displayed as a tooltip when the mouse hovers over it.
18. **`charset:`** Specifies the character encoding for linked resources or for the document itself.
19. **`colspan and rowspan:`** Used in table cells (<td> or <th>) to specify how many columns or rows a cell should span.
20. **`autocomplete:`** Specifies whether an input field has autocomplete enabled or disabled.

HTML Attributes example - Hyperlink

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
    content="width=device-width,initial-scale=1.0">
    <title>Learning HTML & CSS</title>
    </head>
    <body>
      <h1>JavaScript is fun, so is HTML & CSS!</h1>
        <p>You can learn JavaScript without HTML and CSS,
           but for DOM manipulation it's useful to have
           some basic ideas of HTML and CSS.

        <a href="enter hyperlink here">Link Display Text</a>
        </p>
    </body>
</html>
```

if we want to add a hyperlink to a page we can insert an anchor element into the body (or wherever we want the link to display)

Following HTML syntax, we use href to specify a **url** for our link, and then the text that will be displayed on the page, when clicked the user is sent to the link **url**.

HTML Attributes example - Image

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
    content="width=device-width,initial-scale=1.0">
           <title>Learning HTML & CSS</title>
           </head>
    <body>
      <h1>JavaScript is fun, so is HTML & CSS!</h1>
        <p>You can learn JavaScript without HTML and CSS,
           but for DOM manipulation it's useful to have
           some basic ideas of HTML and CSS.

        <a href="enter hyperlink here">Link Display Text</a>
        </p>

        <h2>Another Heading</h2>
          <p> Another Paragraph </p>
          <img src="Image Source" />
    </body>
</html>
```

if we want to add an image to a page we can insert an **img element** and set an image source from either the current working folder or from a **url**

the **img** element doesn't include a closing tag, after the source the element is closed with a **/>** (optional)

HTML Classes & IDs:

Both classes and IDs are attributes used to identify elements. They play crucial roles in styling (with **CSS**) and scripting (with **JavaScript**). Here's a brief description of each:

HTML Classes (class):
Purpose: Classes are used to group one or more elements that share the same styles or behaviors.

Multiple Elements: A single class can be applied to multiple elements on a page. For instance, if you want several paragraphs to have the same styling, you can assign the same class to each of them.

Multiple Classes: An element can have multiple classes. They are separated by spaces. For example: **<div class="class1 class2 class3">**.

CSS: In CSS, classes are selected using a period **(.)** followed by the class name. For example: **.myClass { color: blue; }**.

JavaScript: In JavaScript, you can select elements by their class name using methods like **getElementsByClassName.**

HTML IDs (id):
Purpose: IDs are used to identify a **single, unique element**. They are useful when you want to style or script a specific element without affecting others.

Uniqueness: Each ID should be unique within a page. No two elements should have the same ID value on a single page.

CSS: In CSS, IDs are selected using a hash **(#)** followed by the ID name. For example: **#myID { font-weight: bold; }**.

JavaScript: In JavaScript, you can select an element by its ID using the **getElementById method.**

Fragment Identifier: IDs can also be used as fragment identifiers in URLs. For example, navigating to **#section1** in a URL like: **www.example.com/page#section1** would scroll the page to the element with **id="section1".**

Key Differences:
Uniqueness: While classes can be used on multiple elements and an element can have multiple classes, each ID must be unique and can only be used once per page.

Specificity: In CSS, IDs have a higher specificity than classes. This means that styles applied using an ID will override styles applied using a class if there's a conflict.

In summary, while both classes and IDs allow for targeted styling and scripting, classes are best for grouping elements with shared characteristics, and IDs are best for pinpointing individual, unique elements.

HTML Class & ID example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Class and ID Example</title>
    <style>
        /* CSS rules for class */
        .highlight {
            background-color: yellow;
        }

        /* CSS rules for ID */
        #main-title {
            font-size: 24px;
            color: blue;
        }
    </style>
</head>
<body>

<h1 id="main-title">Welcome to My Website</h1>
<p>This is a regular paragraph.</p>
<p class="highlight">This paragraph is highlighted using a class.</p>

</body>
</html>
```

Breakdown:

Styling (CSS):

- Inside the **<head>** section, we have a **<style>** element containing **CSS** rules.
- The **.highlight** rule targets elements with the class **highlight**. Elements with this class will have a yellow background.
- The **#main-title** rule targets the element with the ID **main-title**. This element will have a font size of 24 pixels and a blue color.

Body Content:

- The **<h1>** element has an ID attribute with the value **main-title**. This means it will be styled according to the **#main-title CSS** rule.
- The first **<p>** element is a regular paragraph without any class or ID.
- The second **<p>** element has a class attribute with the value highlight. This means it will be styled according to the **.highlight CSS** rule, giving it a yellow background.

In this example, the ID **main-title** is used to uniquely identify and style the main title of the page. The class **highlight** is used to indicate that any element with this class should have a highlighted background. This allows for flexibility, as you can easily add the **highlight** class to other elements if you want them to have the same styling.

HTML Creating Boxes

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
    content="width=device-width,initial-scale=1.0">
           <title>Learning HTML & CSS</title>
           </head>
    <body>
      <h1>JavaScript is fun, so is HTML & CSS!</h1>
        <p class="first">You can learn JavaScript without HTML and CSS,
           but for DOM manipulation it's useful to have
           some basic ideas of HTML and CSS.

        <a href="enter hyperlink here">Link Display Text</a>
        </p>

        <h2>Another Heading</h2>
          <p class="second"> Another Paragraph </p>
          <img id="test-image" src="Image Source Link" />

          <div id="your-name">
            <h2>Your name here</h2>
          </div>
    </body>
</html>
```

Here we can create a box to display content by inserting a **div** element

We can then add content into the box by inserting it within our **div** element

HTML Creating Forms

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
    content="width=device-width,initial-scale=1.0">
           <title>Learning HTML & CSS</title>
           </head>
    <body>
      <h1>JavaScript is fun, so is HTML & CSS!</h1>
        <p class="first">You can learn JavaScript without HTML and CSS,
           but for DOM manipulation it's useful to have
           some basic ideas of HTML and CSS.
        <a href="enter hyperlink here">Link Display Text</a>
        </p>
        <h2>Another Heading</h2>
          <p class="second"> Another Paragraph </p>
          <img id="test-image" src="Image Source Link" />

          <div id="your-name">
            <h2>Your name here</h2>
            <p>Please fill in this form: </p>

            <input type="text" placeholder="Your name">
            <button>OK!</button>
          </div>
    </body>
</html>
```

To get information from the user we can insert an **input** element and set our type attribute to the type of input we are getting (text, file, etc.)

We also need a way to confirm the entered input, usually this is done with a **button** element **(JS provides actual implementation)**

```
<form id="your-name">
 <h2>Your name here</h2>
 <p>Please fill in this form: </p>

 <input type="text" placeholder="Your name">
 <button>OK!</button>
</form>
```

Another way to create a form for user input is to use **form** instead of **div.** this better describes what is inside the element, whereas **div** is a generic box. This is an example of **Semantic HTML.**

Semantic HTML refers to the use of **HTML** elements that convey meaning about the structure and content of the document. These elements provide a clear indication of their intended purpose, both to the developer and to browsers, search engines, and assistive technologies like screen readers.

Using semantic elements helps in:

Improving Accessibility: Assistive technologies can better understand and navigate the content.
SEO Benefits: Search engines can better index and rank content.
Maintainability: Developers can more easily understand and manage the structure of a webpage.

common semantic HTML elements:

`<header>:` Represents a container for introductory content or navigation links. Typically contains the website logo, website title, and main navigation.
`<nav>:` Represents a section with navigation links, either within the current document or to other documents.
`<main>:` Contains the main content of the document. There should be only one **<main>** per page, and it should be unique from content in other elements like **<header>, <footer>, or <aside>.**

`<article>`: Represents a self-contained composition in a document, like a blog post or a news story.
`<section>`: Represents a standalone section of a document, such as tabs, tabbed content, or any content that stands alone and can be independently distributed.
`<aside>`: Represents content that is related to the main content but can be considered separate. It could be a sidebar with related links or advertisements.
`<footer>`: Represents the footer of a document or a section, typically containing information about the author, copyright information, links, etc.
`<figure> and <figcaption>`: Used together, they represent content like illustrations, diagrams, photos, code listings, etc., with a caption.
`<time>`: Represents a specific period in time or a duration.
`<mark>`: Used to highlight parts of text, such as search terms or important content.

Using semantic **HTML** ensures that the content and structure of web pages are presented in a meaningful way. This not only aids in accessibility and **SEO** but also makes the web more inclusive and understandable for all users.

<u>Styling with CSS:</u>

- **CSS - Cascading Style Sheets**

A stylesheet language used to describe the look and formatting of a document written in **HTML**. While **HTML** provides the structure and content of a webpage, **CSS** defines its visual presentation, such as layout, colors, fonts, and spacing.

How CSS fits in with HTML:

Separation of Concerns: By using **CSS**, web developers can separate the content and structure (**HTML**) from the presentation (**CSS**). This makes websites easier to maintain and update.

Flexibility: One **CSS** file can style multiple **HTML** pages. This ensures a consistent look and feel across a website and allows for changes to be made site-wide by editing a single file.

Specificity and Cascade: CSS rules can target specific **HTML** elements, and when multiple rules conflict, the one with higher specificity wins. The "cascading" nature means that styles can inherit properties from parent elements or override them.

CSS Example Syntax

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Example</title>
    <style>
        /* CSS styles go here */
        body {font-family: Arial, sans-serif;
              background-color: #f4f4f4;}
        h1 {color: #333;}
        .highlight {background-color: yellow;}
    </style>
</head>
<body>

  <h1>Welcome to My Website</h1>
  <p>This is a regular paragraph.</p>
  <p class="highlight">This paragraph is highlighted using a CSS
     class.
  </p>

</body>
</html>
```

CSS styles are placed within the **<style>** element in our **HTML** document :

- The **body rule** sets the font for the entire webpage and gives the page a light gray background.
- The **h1 rule** changes the color of all top-level headings to a dark gray.
- The **.highlight rule** targets any element with the class "highlight" and gives it a yellow background. In this example, the second paragraph has this class and will be highlighted.

In larger websites the CSS is often placed in external stylesheets **[.css files]** then linked to the **HTML** document using the **<link>** element. This allows multiple **HTML** pages to share the same styles and promotes reusability and maintainability.

Link CSS file

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Example</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>

  <h1>Welcome to My Website</h1>
  <p>This is a regular paragraph.</p>
  <p class="highlight">This paragraph is highlighted using a CSS
     class.
  </p>

</body>
</html>
```

Instead of including a **Style** element, we can use the **link** element to connect a **CSS Stylesheet** to our **HTML** document.

```
#MyForm {
background-color: #af7caf;
    padding: 20px;
    border: 1px solid #ccc;
    border-radius: 5px;
    width: 400px; /
    margin: 0 auto;
}

.highlight { /* define a CSS class called highlight */
    background-color: coral;
}
```

In our **.css** file we can set styles for **ID**s using the **(#) hash symbol** and set styles for **classes** using the **(.) dot operator.**

<u>CSS Box Model:</u>

The Box model includes:

- **Content:** This is the actual content of the box, where text, images, and other media reside. Its size can be controlled using the width and height properties.

- **Padding:** This is the space between the content and the border. It effectively increases the size of the box. Padding does not have any color by default; it inherits the background color of the box's content. You can control its size using the padding property (or its individual properties like padding-top, padding-right, etc.).

- **Border:** This surrounds the padding (if any) and content. It can have a style (solid, dashed, etc.), width, and color. You can control its appearance using the border property (or its individual properties like border-width, border-color, etc.).

- **Margin:** This is the space outside the border. It separates the box from other boxes. Unlike padding, margins are transparent. You can control its size using the margin property (or its individual properties like margin-top, margin-right, etc.).

It's important to note that the total width and height of an element is the sum of its content, padding, border, and margin. However, when you set the width and height properties in CSS, you're only setting the width and height of the content area. If you want the width and height properties to include padding and borders, you can use the box-sizing property with the value border-box

Basic Box Model

```
.box {
  width: 100px;
  height: 100px;
  padding: 10px;
  border: 5px solid black;
  margin: 20px;
  background-color: lightblue;
}
```

In this example:

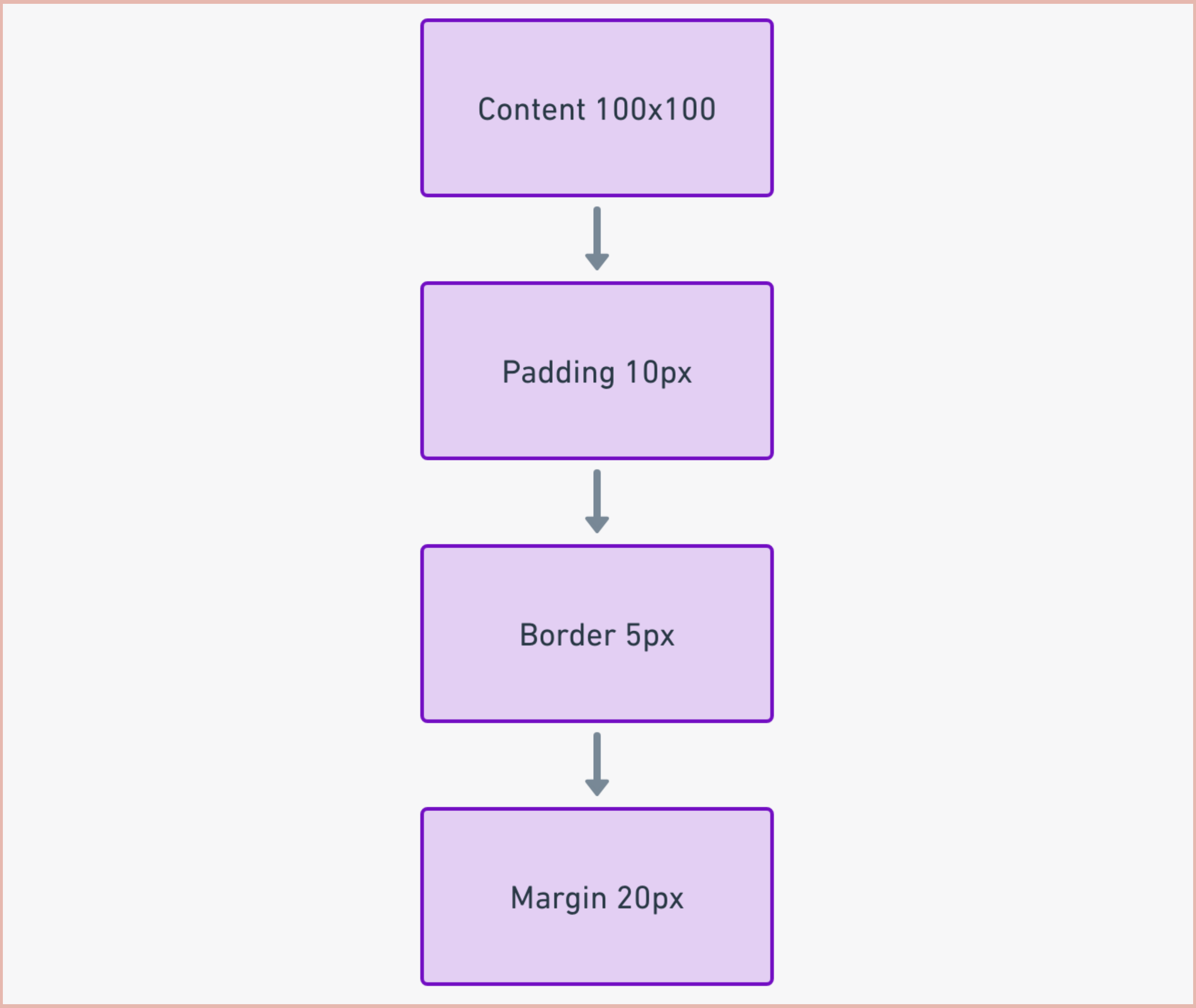
- The content area is 100x100 pixels.
- The padding adds an extra 10 pixels to each side, making the box 120x120 pixels (excluding the border).
- The border adds another 5 pixels to each side, making the box 130x130 pixels (excluding the margin).
- The margin pushes other elements away by 20 pixels on each side, but it doesn't increase the size of the box itself.

Using the (*) star selector

```
* {
   margin: 0;
   padding: 0;
}
```

the **star** or **universal selector (*)** matches and selects all elements in an HTML document.

In the above example, the universal selector is used to remove default margins and paddings from all elements, a common practice in CSS resets.



In the diagram:

- The innermost box represents the Content with a size of 100x100 pixels.
- Surrounding the content is the Padding of 10px.
- The Border of 5px surrounds the padding.
- Finally, the Margin of 20px is outside the border.

Select All Child Elements

```
div * {
    color: red;
}
```

When combined with another selector, it can target all child elements of a specific element.

Select All Elements with Specific Attributes

```
*[data-highlight="true"] {
    background-color: yellow;
}
```

The universal selector can be combined with attribute selectors to target elements based on their attributes. This selector targets all elements with a **data-highlight** attribute set to "**true**", giving them a yellow background.

Points to Consider:

Performance: Overusing the universal selector, especially in large documents, can lead to performance issues. While modern browsers are optimized to handle this efficiently, it's still a good practice to be specific with your selectors when possible.

Specificity: The universal selector has the lowest specificity. This means that if another CSS rule targets the same element and conflicts with the universal selector, the other rule will take precedence due to its higher specificity.

Inheritance: Some properties naturally inherit their values from parent elements (like **color** or **font-family**). Using the universal selector can sometimes lead to unexpected results, especially when setting properties that are typically inherited.

Section 3: DOM & Events

DOM and DOM Manipulation:

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a document as a tree of objects. Each object corresponds to a part of the document, such as an element or an attribute. The DOM provides a way for programs to access and manipulate the content, structure, and style of a document.

DOM manipulation in JavaScript involves using the JavaScript language to dynamically change the DOM, that is, to add, modify, or remove elements and attributes in a document.

Accessing an Element

```
let element = document.getElementById("myElement");
```

You can access an element in the document using various methods such as **getElementById, getElementsByClassName, getElementsByTagName, etc.**

Changing an Element

```
element.innerHTML = "New Content";
element.style.color = "red";
```

Once you have accessed an element, you can change its **content, attributes, or style.**

Adding or Removing Elements

```
// creating a new element
let newElement = document.createElement("p");

// setting its attributes and content
newElement.setAttribute("id", "newElement");
newElement.innerHTML = "This is a new paragraph";

// adding it to the document
document.body.appendChild(newElement);

// removing an existing element
document.body.removeChild(document.getElementById("myElement"));
```

In this example, we first access an element with the **id** "**myElement**", change its content and color, create a new paragraph element, set its id and content, add it to the document, and then remove the element with the id "**myElement**" from the document.

<u>Handle Click Events:</u>

In JavaScript, you can use the **addEventListener** method to attach an event handler to an element. The **addEventListener** method takes two arguments: the name of the event you want to listen for, and the function you want to call when the event occurs.

addEventListener Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Click Event Example</title>
</head>
<body>
    <button id="myButton">Click me!</button>

    <script>
        // accessing the button element
        let button = document.getElementById("myButton");

        // attaching a click event listener to the button
        button.addEventListener("click", function() {
            alert("Button was clicked!");
        });
    </script>

</body>
</html>
```

In this example, we first **access the button element with the id "myButton"**, then we **attach a click event listener to the button.** The event listener is a **function that will be called when the button is clicked**. In this case, **the function displays an alert box** with the message "Button was clicked!".

Keyboard Events:

Keyboard events in JavaScript are events that occur when a user interacts with the keyboard. The most common keyboard events are:

1. **keydown:** This event is triggered when a key is pressed down.

2. **keypress:** This event is triggered when a key is pressed down and then released. Note that this event is not triggered for all keys (e.g., ALT, CTRL, SHIFT, ESC) and is considered as a legacy event, and it's better to use keydown or keyup events instead.

3. **keyup:** This event is triggered when a key is released.

Each of these events provides an event object that contains information about the event, such as which key was pressed, whether any modifier keys (Shift, Ctrl, Alt) were held down, etc.

Keyboard Events Example

```
document.addEventListener('keydown', function(event) {
    console.log('Key down:', event.key);
});

document.addEventListener('keyup', function(event) {
    console.log('Key up:', event.key);
});

// Example HTML
// <input type="text" id="inputField">

// Using the events on a specific element
var inputField = document.getElementById('inputField');

inputField.addEventListener('keydown', function(event) {
    console.log('Key down in input field:', event.key);
});
```

In this example, whenever a key is pressed or released, the corresponding message and the key pressed will be logged to the console. The last part of the example shows how you can attach the event listener to a specific element (in this case, an input field) rather than the entire document.

<u>Selecting Elements:</u>

getElementById and **querySelector** are two different methods in JavaScript to select elements from the DOM, but there are some differences between them:

getElementById: This method is used to select an element by its **id** attribute. It is a little bit faster than **querySelector** because it directly accesses the element by its **id**, which is unique in the document.

getElementById Example

```
let element = document.getElementById('my-element');
```

In this example, **getElementById** will return the first element with the id attribute of **my-element.**

querySelector: This method is more versatile because it allows you to select elements using any valid CSS selector, not just the id attribute. You can use it to select elements by their **id, class, attribute, etc.**

querySelector Example

```
let element = document.querySelector('#my-element');
let elements = document.querySelectorAll('.my-class');
```

In the first example, **querySelector** will return the first element with the **id** attribute of **my-element**. In the second example, **querySelectorAll** will return a **NodeList** of all elements with the **class** attribute of **my-class.**

In summary, **getElementById** is faster and should be used when you want to select an element by its id, while **querySelector** is more versatile and should be used when you want to select elements using other CSS selectors.

Section 4: JavaScript Details

Types of Scope:

Scoping in JavaScript refers to the accessibility or visibility of variables, objects, and functions in different parts of your code during the runtime. There are two main types of scope in JavaScript: global scope and local scope.

1. **Global Scope:**
   Variables declared outside of any function or block are in the global scope.
   Global variables can be accessed from anywhere in the code, both inside and outside of functions or blocks.

2. **Local Scope:**
   Variables declared inside a **function or a block** (with **let** or **const** in ES6) are in the local scope of that function or block.
   Local variables can only be accessed within the function or block they were declared in, or in any nested functions or blocks.

Block Scope:
JavaScript ES6 introduced block scope variables, which are variables defined using let and const keywords. These variables are accessible only within the block (or sub-block) they were defined in.

Scoping Example

```
var globalVar = "I am global";

function myFunction() {
    var localVar = "I am local";
    console.log(globalVar); // I am global
    console.log(localVar); // I am local
}

myFunction();

console.log(globalVar); // I am global
console.log(localVar); // Uncaught ReferenceError: localVar is not defined
```

In this example, **globalVar** is in the global scope, so it can be accessed both inside and outside of **myFunction**. **localVar**, on the other hand, is in the local scope of **myFunction**, so it can only be accessed inside **myFunction**. Trying to access **localVar** outside of **myFunction** results in a **ReferenceError**.

Block Scope Example

```
function blockScopeExample() {
    if (true) {
        var x = 10;
        let y = 20;
        console.log(x, y); // 10 20
    }
    console.log(x); // 10
    console.log(y); // Uncaught ReferenceError: y is not defined
}

blockScopeExample();
```

In this example, **x** is declared using **var** and is accessible within the entire function, whereas **y** is declared using **let** and is only accessible within the **if block**. Trying to access **y** outside of the **if block** results in a **ReferenceError**.

Note:

- Variables declared with **var** are function-scoped or globally scoped if declared outside of any function. They are **not block-scoped.**
- Variables declared with **let** and **const** are block-scoped.

Hoisting:

In JavaScript, "**hoisting**" is a behavior in which variable and function declarations are moved to the top of their containing scope during the execution phase. This means you can technically use variables and functions before they are declared in your code. However, it's important to note that only the declarations are hoisted, not the initializations.

Variable Hoisting

In the case of variables declared with **var**, the declaration is hoisted but the initialization is not. If you try to use the variable before it's initialized, it will have the value **undefined**.

Variable Hoisting Example

`console.log(myVar);`
// undefined, because only the declaration is hoisted, not the initialization
 	
`var myVar = 10;`
`console.log(myVar);` // 10

In the above code, the declaration **var myVar** is hoisted to the top of the scope, but the initialization **myVar = 10** stays in place.
`console.log(myLetVar);` // ReferenceError `let myLetVar = 20;`
With **let** and **const**, the variable declaration is also hoisted, but you cannot access the variable before it is declared; trying to do so will throw a **ReferenceError**.

Function Hoisting Example

```
greet(); // Hello, World!

function greet() {
  console.log('Hello, World!');
}
```

Function declarations are fully hoisted, which means that both the name and the body of the function are hoisted.

In this code, the function **greet** can be called before it's defined because the function declaration is hoisted to the top of the scope.

```
greetExpression(); // TypeError: greetExpression is not a function

var greetExpression = function() {
  console.log('Hello, World!');
};
```

Function expressions are not hoisted. If you try to call a function expression before it's defined, you'll get a **TypeError**.

In this code, the variable **greetExpression** is hoisted, but it's initialized with undefined, so trying to call it as a function results in a **TypeError**.

It's generally considered good practice to avoid relying on hoisting and to always declare your variables and functions before you use them to make your code more readable and prevent unexpected behaviors.

Temporal Dead Zone (TDZ):

The **"Temporal Dead Zone" (TDZ)** is a term used to describe a period where a variable is inaccessible, even though it has been declared in the scope. This occurs with variables declared using **let** and **const**. When you try to access a variable in its TDZ, a **ReferenceError** is thrown.

Variables declared with **let** and **const** are hoisted to the top of their block scope, but unlike **var**, they are not initialized with **undefined**. Instead, they enter a "temporal dead zone" where accessing them before they are initialized will result in a **ReferenceError**.

TDZ Example

```
console.log(myVar); // undefined - var variables are hoisted and initialized with undefined
console.log(myLetVar); // ReferenceError: Cannot access 'myLetVar' before initialization

var myVar = 10;
let myLetVar = 20;
```

In this example:

The **var myVar** declaration is hoisted to the top of the scope and initialized with **undefined**, so **console.log(myVar)** prints **undefined**.

The **let myLetVar** declaration is also hoisted to the top of the block scope, but it enters a "temporal dead zone" where it cannot be accessed until it is initialized. Trying to access it in this period results in a **ReferenceError**.

Understanding the TDZ is important to avoid reference errors in your code and to understand the differences between var, let, and const declarations in JavaScript. It's generally recommended to always initialize your variables when you declare them to avoid entering the TDZ and potentially encountering reference errors.

Section 5: Data Structures, Modern Operators & Strings

Array Destructuring:

Array destructuring allows you to unpack elements from arrays, and assign them to individual variables in a concise manner.

Array Destructuring Example

```
const colors = ['red', 'green', 'blue'];
const [red, green, blue] = colors;

console.log(red);    // 'red'
console.log(green);  // 'green'
console.log(blue);   // 'blue'
```

In the above example, the variables **red**, **green**, and **blue** are assigned to the respective elements in the **colors** array in one line of code, instead of accessing each array element individually.

Skipping Items

```
const colors = ['red', 'green', 'blue'];
const [, , blue] = colors;

console.log(blue);   // 'blue'
```

You can also skip items in the array that you're not interested in. Here, we skipped the first two items and only destructured the third item, **'blue'**.

Using Rest Parameter

```
const colors = ['red', 'green', 'blue', 'yellow', 'purple'];
const [red, ...restColors] = colors;

console.log(red);             // 'red'
console.log(restColors);      // ['green', 'blue', 'yellow', 'purple']
```

You can use the rest parameter **(...)** to collect the remaining items in an array. In this example, the variable **red** contains the first element, and **restColors** contains an array of the remaining elements.

Assigning Default Values

```
const colors = ['red'];
const [red, green = 'green'] = colors;

console.log(red);    // 'red'
console.log(green);  // 'green' (default value used)
```

You can assign default values to variables in case an element is undefined. Here, since **green** is not available in the **colors** array, it takes the default value **'green'**.

Object Destructuring:

Object destructuring in JavaScript allows you to extract properties from objects and bind them to variables. This can make your code more concise and readable.

Object Destructuring Example

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};

const { firstName, lastName, age } = person;

console.log(firstName); // 'John'
console.log(lastName);  // 'Doe'
console.log(age);       // 30
```

In this example, the properties **firstName**, **lastName**, and **age** are extracted from the **person** object and assigned to individual variables with the same names.

Renaming Variables

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};

const { firstName: first, lastName: last, age } = person;

console.log(first); // 'John'
console.log(last);  // 'Doe'
console.log(age);   // 30
```

Here, the properties **firstName** and **lastName** are extracted and assigned to new variable names, **first** and **last**, respectively.

Assigning Default Values

```
const person = {
  firstName: 'John',
  lastName: 'Doe'
};

const { firstName, lastName, age = 30 } = person;

console.log(firstName); // 'John'
console.log(lastName);  // 'Doe'
console.log(age);       // 30 (default value used)
```

In this example, since the **age** property is not available in the **person** object, it takes the default value **30**.

Nested Object Destructuring

```
const person = {
  name: {
    first: 'John',
    last: 'Doe'
  },
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};

const {
  name: { first, last },
  address: { city, country }
} = person;

console.log(first);   // 'John'
console.log(last);    // 'Doe'
console.log(city);    // 'New York'
console.log(country); // 'USA'
```

In this case, the **first** and **last** properties are extracted from the nested **name** object, and **city** and **country** are extracted from the nested **address** object.

Rest Pattern and Parameters:

Rest Pattern

The rest pattern in JavaScript is used to collect the remaining elements into an array. This pattern can be used in array destructuring, object destructuring, and with function parameters.

Rest Parameters

Rest parameters are used in function definitions to collect an indefinite number of arguments into an array.

In Array Destructuring

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];
console.log(first);        // 1
console.log(second);       // 2
console.log(rest);         // [3, 4, 5]
```

In this example, **first** and **second** take the first two elements of the array, and **rest** collects the remaining elements (3, 4, and 5) into an array.

In Object Destructuring

```
const {a, b, ...rest} = {a: 1, b: 2, c: 3, d: 4};
console.log(a);          // 1
console.log(b);          // 2
console.log(rest);       // { c: 3, d: 4 }
```

Here, **a** and **b** are destructured normally, and **rest** collects the remaining properties **(c and d**) into an object.

Rest Parameters

```
function sum(...nums) {
  return nums.reduce((accumulator, current) => accumulator + current,
0);
}

console.log(sum(1, 2, 3, 4));   // 10
```

In this example, the **sum** function accepts a rest parameter **nums**, which collects all the arguments passed to the function into an array. The **reduce** method is then used to sum up all the elements in the **nums** array.

<u>Nullish Coalescing Operator (??):</u>

In JavaScript, the nullish coalescing operator (??) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

It's useful to fall back to a default value when dealing with potential null or undefined values. This can be a better choice than the || operator which falls back on any falsy value (false, 0, '', NaN, null, undefined).

Nullish Coalescing Operator Example

```
let value = null;
let defaultValue = 42;

let result = value ?? defaultValue;
console.log(result);                                    // 42, because value is null

value = undefined;
result = value ?? defaultValue;
console.log(result);                                    // 42, because value is undefined

value = 0;
result = value ?? defaultValue;
console.log(result);                    // 0, because value is 0 (not null or undefined)

value = '';
result = value ?? defaultValue;
console.log(result);       // '', because value is an empty string (not null or undefined)
```

In this example:

- In the first case, **value** is **null**, so the nullish coalescing operator returns **defaultValue (42).**
- In the second case, **value** is **undefined**, so again, it returns **defaultValue (42).**
- In the third case, **value** is **0**, which is a falsy value but **not null** or **undefined**, so it returns **value (0)** instead of **defaultValue**.
- Similarly, in the fourth case, **value** is an **empty string (another falsy value)**, so it returns **value ('')** instead of **defaultValue**.

Logical Asssignment Operators:

In JavaScript, logical assignment operators are a combination of logical operators and assignment expressions. There are three logical assignment operators: **&&=, ||=,** and **??=.**

Logical AND Assignment

```
let a = 1;
let b = 0;

a &&= 2;                    // a = a && 2
console.log(a);             // 2, because a was truthy (1), so it's updated to 2

b &&= 2; // b = b && 2
console.log(b);             // 0, because b was falsy (0), so it remains 0
```

This operator assigns the value on its right to the variable on its left only if the variable on the left is truthy.

Logical OR Assignment

```
let x = null;
let y = 'Hello';

x ||= 'default';           // x = x || 'default'
console.log(x);            // 'default', because x was falsy (null), so it's updated to 'default'

y ||= 'default';           // y = y || 'default'
console.log(y);            // 'Hello', because y was truthy ('Hello'), so it remains 'Hello'
```

This operator assigns the value on its right to the variable on its left only if the variable on the left is falsy.

Nullish Coalescing Assignment

```
let p = null;
let q = 'World';
let r = 0;

p ??= 'default';          // p = p ?? 'default'
console.log(p);           // 'default', because p was null, so it's updated to 'default'

q ??= 'default';          // q = q ?? 'default'
console.log(q);           // 'World', because q was defined ('World'), so it remains 'World'

r ??= 'default';         // r = r ?? 'default'
console.log(r);          // 0, because r was defined (0), even though it's a falsy value, it is not
                            null or undefined, so it remains 0
```

This operator assigns the value on its right to the variable on its left only if the variable on the left is **null** or **undefined**.

For-of Loop:

The **for...of** statement creates a loop iterating over iterable objects, including: built-in **String, Array**, array-like objects (e.g., **arguments** or **NodeList**), **TypedArray**, **Map**, **Set**, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object. It is similar to the **For-each** loop from C#.

Syntax

```
for (variable of iterable) {
  statement
}
```

Here, **variable** is a reference to the current element in the **iterable** during the loop, and **iterable** is the object that has iterable properties (like arrays, strings, etc.).

For-of Example

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

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

// Output:
// apple
// banana
// cherry
```

- We define an array **fruits** with three elements.
- We create a **for...of** loop, where **fruit** is a new variable that will represent each element in the **fruits** array.
- Inside the loop, we **console.log** the current **fruit**.
- The loop iterates over each element in the **fruits** array, logging each one to the console.

Iterate Over Strings

```
const greeting = 'Hello';

for (const char of greeting) {
  console.log(char);
}

// Output:
// H
// e
// l
// l
// o
```

Here, the f**or...of** loop iterates over each **character** in the string **greeting**, logging each character to the console.

Enhanced Object Literals:

Enhanced (or extended) object literals in JavaScript are a way to create objects with more concise syntax and additional functionalities. This feature was introduced in ES6 to improve the object literal syntax and to add more power and flexibility to it.

Property Shorthand

```
let name = "John";
let age = 30;

let person = { name, age };

console.log(person);                 // Output: { name: 'John', age: 30 }
```

If you want to define an object property where the key is the same as the variable name, you can use property shorthand to define it in a more concise way.

Computed Property Names

```
let prop = 'name';
let id = '1234';

let obj = {
  [prop]: 'John',
  ['user_' + id]: 'ProfileData'
};

console.log(obj);                    // Output: { name: 'John', user_1234: 'ProfileData' }
```

You can use expressions as property names inside an object literal definition.

Method Shorthand

```
let person = {
  greet() {
    console.log('Hello!');
  }
};

person.greet();                         // Output: Hello!
```

You can define methods inside an object literal without the **function** keyword.

Defining proto Property

```
let protoObj = {
  greet() {
    console.log('Hello from prototype!');
  }
};

let obj = {
  __proto__: protoObj
};

obj.greet();                            // Output: Hello from prototype!
```

You can set the prototype of an object at the time of object creation using **__proto__** property.

Optional Chaining:

Optional Chaining is a JavaScript feature (introduced in ES2020) that allows you to access the properties of objects without causing errors if a reference or function call is **null** or **undefined**. It helps in writing cleaner and more readable code.

Syntax

```
obj?.property
obj?.[expr]
func?.(args)
```

Accessing Object Properties

```
const person = {
  name: "John",
  address: {
    city: "New York",
    zipCode: 10001,
  },
};

console.log(person.address?.city);      // Output: New York
console.log(person.address?.street);    // Output: undefined (instead of causing an
                                           error)
```

In this example, the **street** property doesn't exist, but instead of causing an error, optional chaining returns **undefined**.

Accessing Elements in Arrays

```
const array = [1, 2, 3];

console.log(array?.[0]);      // Output: 1
console.log(array?.[5]);      // Output: undefined (instead of causing an error)
```

Here, the **index 5** doesn't exist in the array, but optional chaining returns **undefined** instead of throwing an error.

Function Calls

```
const obj = {
  func: () => {
    console.log("Function called!");
  },
};

obj.func?.();                 // Output: Function called!
obj.nonExistentFunc?.();      // No output and no error (would throw an error without
                                 optional chaining)
```

In this case, calling a non-existent function property with optional chaining doesn't cause an error, preventing potential runtime issues in your JavaScript code.

Sets:

In JavaScript, a Set is a collection of unique values. Sets can be very useful when you want to store a collection of distinct items, as a set automatically ensures that no duplicate values are stored.

Creating a Set

```
const mySet = new Set([1, 2, 3, 4, 5]);
```

You can create a set using the **Set constructor.** You can optionally **pass an iterable** (like an array) to the constructor to initialize the set with a collection of values.

Adding Values to a Set

```
mySet.add(6);
mySet.add(3);                          // This will not be added again
```

You can add values to a set using the **add method**. If you try to add a value that's already in the set, it won't be added again, because sets only store unique values.

Creating a Set

```
mySet.delete(3);                       // This will remove the number 3 from the set
```

You can remove values from a set using the **delete method**.

Getting the Size of a Set

```
console.log(mySet.size);               // Output: 5
```

You can iterate over the values in a set using a for-of loop.

Converting a Set to an Array

```
const myArray = Array.from(mySet);

// or

const myArray2 = [...mySet];
```

You can convert a set to an array using the Array.from function or the **spread operator (...).**

Set Example

```
const mySet = new Set([1, 2, 3, 4, 5]);

// Adding values
mySet.add(6);
mySet.add(3);                                                    // Won't be added again

// Removing a value
mySet.delete(3);

// Checking if a value is in the set
console.log(mySet.has(3));                                       // Output: false

// Getting the size of the set
console.log(mySet.size);                                         // Output: 5

// Iterating over the set
for (const value of mySet) {
  console.log(value);
}


// Output:
// 1
// 2
// 4
// 5
// 6

// Converting the set to an array
const myArray = Array.from(mySet);
console.log(myArray);                                            // Output: [ 1, 2, 4, 5, 6 ]
```

this example demonstrates all above concepts of Sets in JavaScript

Maps:

In JavaScript, the **Map object** holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a **key** or a **value**.

Creating a Map
`const myMap = new Map();`
You can create a Map object using the new **Map()** constructor.

Setting Values
`myMap.set('name', 'Alice');`
You can add new key-value pairs to a map using the **set()** method.

Getting Values
`console.log(myMap.get('name')); // Output: Alice`
You can get the value associated with a key using the **get()** method.

Checking for Existence
`console.log(myMap.has('name')); // Output: true`
You can check if a key exists in a map using the **has()** method.

Removing Values
`myMap.delete('name');`
You can remove a key-value pair using the **delete()** method.

Iterating over a Map

```
myMap.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});

for (let [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}
```

You can iterate over a map using various methods like **forEach()** or **for...of loop.**

Map Example

```
// Creating a new Map
const myMap = new Map();

// Setting key-value pairs
myMap.set('name', 'Alice');
myMap.set('age', 25);

// Getting values
console.log(myMap.get('name'));       // Output: Alice

// Checking for existence
console.log(myMap.has('age'));        // Output: true

myMap.delete('age');// Removing a value

// Size of the Map
console.log(myMap.size);              // Output: 1

// Iterating over the Map using forEach
myMap.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});

// Iterating over the Map using for...of
for (let [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}

myMap.clear();// Clearing the Map

// Checking the size after clearing
console.log(myMap.size);              // Output: 0
```

In this example, a **Map object** is created, and various operations like adding, removing, and retrieving values are performed, along with demonstrating how to iterate over a **Map**.

Choosing a Data Structure:

There are 3 main sources of data that we have to handle in our software:

1. **From the program:** Data written directly in source code (eg. status messages)
2. **From the UI:** Data input from the user or data written in DOM (eg. tasks in todo app)
3. **From external sources:** Data fetched, for example from a web API (eg. recipe objects)

We receive these collections of data and they need to be stored somewhere within our program so they can be manipulated and referenced. This is what JavaScript built-in data structures are used for, and choosing the correct one depends on the type of data you are working with and the operations you need to perform on that data.

1. From the Program (Static or Hard-Coded Data)

data is predefined and does not change during the execution of the program.

Example 1: List of Constants

```
const countryCodes = ['US', 'CA', 'GB', 'AU'];

// or

const countryCodes = new Set(['US', 'CA', 'GB', 'AU']);
```

Scenario: A program that contains a list of country codes.
Data Structure: Array or Set.

Example 2: Configuration Settings

```
const config = {
  server: 'localhost',
  port: 8080,
  protocol: 'https'
};

// or

const config = new Map([
  ['server', 'localhost'],
  ['port', 8080],
  ['protocol', 'https']
]);
```

Scenario: A program that needs to store configuration settings.

Data Structure: Object or Map.

2. From the UI (User Input)

In this case, the data is collected through user interaction, such as forms, buttons, etc.

Example 3: Form Data

```
const formData = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'john.doe@example.com'
};
```

Scenario: A web form collecting user information.
Data Structure: Object.

Example 4: Selected Items from a List

```
const selectedItems = ['item1', 'item3', 'item4'];

// or

const selectedItems = new Set(['item1', 'item3', 'item4']);
```

Scenario: A program that stores selected items from a list.
Data Structure: Array or Set.

3. From External Sources (APIs, Databases, etc.)

In this case, the data is fetched from external sources, which could be files, APIs, databases, etc.

Example 5: Fetching Data from an API

```
fetch('https://api.example.com/users')
  .then(response => response.json())
  .then(data => {
    // data might be an array of objects if the API returns a list of users
    console.log(data);
  });
```

Scenario: A program that fetches a list of users from an API.
Data Structure: Array (for a list of items) or Object/Map (for structured data).

Example 6: Reading a JSON Configuration File

```
const config = require('./config.json');
console.log(config.server);                          // Output: 'localhost'
```

Scenario: A program that reads configuration settings from a JSON file.
Data Structure: Object.

General Tips for Choosing the Right Data Structure:

Array: Use when the order of elements matters, and you might need to access elements by their index.

Object: Use when you have structured data with named properties.

Set: Use when you want to store a collection of unique values.

Map: Use when you want to store key-value pairs where keys can be any data type, and you might need to maintain the insertion order.

Linked List, Stack, Queue: These are more advanced data structures that you might use in more complex algorithms or when you have specific performance considerations.

Working With Strings:

Strings are a constantly used data type in all programming languages. In JavaScript it is important to understand the different operations we can perform on strings and how we can extract data or manipulate them.

length Property

```
const str = "Hello, World!";
console.log(str.length);                    // Output: 13
```

used to get the length of a string.

Real-world Scenario: To validate that a user input or a password meets a certain length requirement.

charAt(index) Method

```
const str = "Hello, World!";
console.log(str.charAt(7));                 // Output: W
```

Returns the character at the specified index in a string.

Real-world Scenario: When you want to get a specific character from a string at a known index, perhaps as part of a string parsing operation.

slice(startIndex, endIndex) Method

```
const str = "Hello, World!";
console.log(str.slice(7, 12));              // Output: World
```

Extracts a section of a string and returns it as a new string.

Real-world Scenario: Trimming file extensions or extracting substrings from strings based on known index positions.

split(separator) Method

```
const str = "Hello, World!";
console.log(str.split(' '));            // Output: [ 'Hello,', 'World!' ]
```

Splits a String object into an array of strings by separating the string into substrings.

Real-world Scenario: Splitting a comma-separated values (CSV) string into individual values or breaking a sentence into words for text analysis.

replace(searchValue, replaceValue) Method

```
const str = "Hello, World!";
console.log(str.replace('World', 'JavaScript'));    // Output: Hello, JavaScript!
```

Searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced.

Real-world Scenario: Replacing sensitive information in a string before logging it or replacing placeholders in a template string with actual values.

toLowerCase() and toUpperCase() Methods

```
const str = "Hello, World!";
console.log(str.toLowerCase());      // Output: hello, world!
console.log(str.toUpperCase());      // Output: HELLO, WORLD!
```

These methods are used to convert a string to lowercase and uppercase, respectively.

Real-world Scenario: Normalizing user input for case-insensitive comparisons or displaying text in a standardized case.

trim() Method

```
const str = "   Hello, World!   ";
console.log(str.trim());              // Output: Hello, World!
```

Removes whitespace from both ends of a string.

Real-world Scenario: Cleaning up user input from a form where extra spaces at the beginning or end might cause issues.

includes(searchString) Method

```
const str = "Hello, World!";
console.log(str.includes('World'));   // Output: true
```

Determines whether one string may be found within another string, returning **true** or **false** as appropriate.

Real-world Scenario: Checking if a user's comment contains prohibited words or phrases.

startsWith(searchString) and endsWith(searchString) Methods

```
const str = "   Hello, World!   ";
console.log(str.trim());              // Output: Hello, World!
```

These methods determine whether a string begins or ends with the characters of a specified string, returning **true** or **false** as appropriate.

Real-world Scenario: File extension validation (using **endsWith**) or checking if a protocol is present in a URL (using **startsWith**).

Section 6: Functions

Default Parameters:

default parameters allow you to initialize functions with default values. If you do not provide a value for a parameter when you call the function, the default value is used instead.

Defining Default Parameters

```
function greet(name = "World") {
  console.log("Hello, " + name + "!");
}
```

In this example, the greet function has a parameter **name** which has a default value of "**World**".

Calling the Function Without Arguments

```
function greet(name = "World") {
  console.log("Hello, " + name + "!");
}

greet();   // Output: "Hello, World!"
```

If you call the function without providing an argument for the parameter with a default value, the default value will be used.

Calling the Function With Arguments

```
function greet(name = "World") {
  console.log("Hello, " + name + "!");
}

greet("Alice");   // Output: "Hello, Alice!"
```

If you provide a value for the parameter, that value will be used instead of the default value.

Multiple Parameters

```
function displayInfo(name = "Unknown", age = 30) {
  console.log("Name: " + name + ", Age: " + age);
}

displayInfo();                   // Output: "Name: Unknown, Age: 30"
displayInfo("Bob");              // Output: "Name: Bob, Age: 30"
displayInfo("Charlie", 25);      // Output: "Name: Charlie, Age: 25"
```

You can have multiple parameters with default values, and you can mix parameters with and without default values.

First Class and Higher Order Functions:

First-Class Functions

In JavaScript, functions are first-class citizens. This means that functions can be treated like any other variable: they can be assigned to variables, passed as arguments to other functions, and returned from functions.

Assigned to Variables

```
const greet = function() {
  console.log("Hello, World!");
};

greet();                    // Output: Hello, World!
```

here we assign the function to the **greet** variable.

Passed as Arguments

```
function callFunction(fn) {
  fn();
}

const sayGoodbye = function() {
  console.log("Goodbye, World!");
};

callFunction(sayGoodbye);            // Output: Goodbye, World!
```

here we pass in the **sayGoodbye** function into the **callFunction** function which in turn calls the passed in function.

Returned from Functions

```
function createMultiplier(multiplier) {
  return function(x) {
    return x * multiplier;
  };
}

const double = createMultiplier(2);
console.log(double(5));                    // Output: 10
```

here we have a **createMultiplier** function which returns **another function** , which gives the value of the **multiplier times the input**

Higher-Order Functions

A higher-order function is a function that either takes one or more functions as arguments, or returns a function as a result. This is possible because, as mentioned above, JavaScript treats functions as first-class citizens.

Taking a Function as an Argument

```
const numbers = [1, 2, 3, 4];

const squaredNumbers = numbers.map(function(x) {
  return x * x;
});

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

Many array methods, like map, filter, and reduce, are examples of higher-order functions because they take a function as an argument.

Returning a Function

```
function greaterThan(n) {
  return function(m) {
    return m > n;
  };
}

const greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));                 // Output: true
```

As shown in the first-class functions section, you can create a function that returns another function.

Immediately Invoked Function Expression(IIFE):

IIFE, is a JavaScript function that runs as soon as it is defined.

Syntax

`(function() {`
`      // Your code here`
`})();`

Explanation:

Function Creation: Firstly, an anonymous function is created using function declaration syntax. This function could also accept parameters and return values like any other normal function.

Function Invocation: Right after the function definition, we add a pair of parentheses **()** to invoke the function immediately after it is defined.

Enclosing Parentheses: The entire function definition and the invoking parentheses are enclosed within another pair of parentheses. This is done to treat the function as a function expression rather than a function declaration.

IIFE Example 1

```
(function() {
    var name = "John";
    console.log("Hello, " + name);
})();
```

// The variable name is not accessible here

```
console.log(typeof name); // Output: undefined
```

- Inside the IIFE, a variable named **name** is declared and initialized to the string "**John**".
- Still inside the IIFE, a message is logged to the console which uses the **name** variable.
- Outside of the IIFE, we attempt to log the type of the **name** variable to the console. Since **name** is not defined in this scope (it is confined to the scope of the IIFE), the output is "**undefined**".

Closures:

A **closure** in JavaScript is a function that has access to its own scope, the scope of the outer function, and the global scope. In other words, a closure gives you access to an outer function's scope from an inner function. Closures are created automatically, and are not manually created objects.

Closure Example 1

```
function outerFunction(outerVariable) {
    var outerLocalVariable = 100;

    return function innerFunction(innerVariable) {
        console.log('outerVariable: ' + outerVariable);
        console.log('outerLocalVariable: ' + outerLocalVariable);
        console.log('innerVariable: ' + innerVariable);
    };
}

const newFunction = outerFunction('outside');
newFunction('inside');

// Output:
// outerVariable: outside
// outerLocalVariable: 100
// innerVariable: inside
```

In this example, **innerFunction** is a closure that encompasses its own scope, the scope of **outerFunction**, and the global scope.

Counter with Closure

```
function createCounter() {
    let count = 0;

    return {
        increment: function() {
            count++;
        },
        decrement: function() {
            count--;
        },
        getCount: function() {
            return count;
        }
    };
}

const counter = createCounter();
counter.increment();
console.log(counter.getCount());            // Output: 1
counter.decrement();
console.log(counter.getCount());            // Output: 0
```

In this example, **count** is not accessible directly from outside the **createCounter** function, but it can be accessed and modified by the methods **increment**, **decrement**, and **getCount** which form a closure over the **count** variable.

Creating Event Handlers

```
var elements = document.querySelectorAll('.my-element');

elements.forEach(function(element, index) {
    element.addEventListener('click', function() {
        console.log('Element ' + index + ' was clicked');
    });
});
```

Creating event handlers with access to variables from the scope in which they were created

Callbacks and Asynchronous Programming

```
for (var i = 0; i < 5; i++) {
    (function(i) {
        setTimeout(function() {
            console.log(i);
        }, i * 1000);
    })(i);
}
// Output: 0, 1, 2, 3, 4 (each number with a second delay)
```

Keeping track of variables when working with asynchronous code and callbacks.

Currying

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

const multiplyByTwo = multiply(2);
console.log(multiplyByTwo(5));              // Output: 10
```

Transforming a function with multiple arguments into a series of functions each taking a single argument.

Section 7: Working with Arrays

Map, Filter, Reduce Methods:

- **Map():** It iterates over each element of the array and applies the function to each element. The return value forms a new array.

- **Filter():** It iterates over each element and applies the function to each element. If the function returns true, the element is kept in the new array, otherwise, it is removed.

- **Reduce():** It iterates over each element and accumulates a value based on the return of the function. It ultimately returns a single value.

Map Method

```
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers);                    // Output: [1, 4, 9, 16, 25]
```

The **map()** method creates a new array populated with the results of calling a provided function on every element in the calling array.

Filter Method

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);                                  // Output: [2, 4]
```

The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.

Reduce Method

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, cur) => acc + cur, 0);
console.log(sum);                                         // Output: 15
```

The **reduce()** method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

Example

```
const numbers = [1, 2, 3, 4, 5];

const result = numbers
  .filter(num => num % 2 === 0)
  .map(num => num * num)
  .reduce((acc, cur) => acc + cur, 0);

console.log(result);                            // Output: 20 (2*2 + 4*4)
```

- First, we filter the array to only include even numbers, resulting in **[2, 4]**.
- Then, we map over the new array to square each number, resulting in **[4, 16]**.
- Finally, we use reduce to sum these values, resulting in **20.**

Find, Some, and Every Method:

find, some, and **every** are higher-order functions that you can use to work with arrays. These functions take a callback function as an argument and iterate over each element in the array, applying the callback function to it.

Find Method

```
const array = [1, 2, 3, 4, 5];

const found = array.find(element => element > 2);

console.log(found);                    // Output: 3
```

The **find** method returns the first element in the array that satisfies the provided testing function. If no elements satisfy the testing function, undefined is returned.

Some Method

```
const array = [1, 2, 3, 4, 5];

const hasSome = array.some(element => element > 4);

console.log(hasSome);                  // Output: true
```

The **some** method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

Every Method

`const array = [1, 2, 3, 4, 5];`
 	
`const hasEvery = array.every(element => element > 0);`
 	
`console.log(hasEvery);` // Output: true

The **every** method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

In each of these examples, the callback function takes an element from the array and tests it according to a certain condition (element > 2, element > 4, and element > 0 respectively). The method then returns a result based on whether any or all elements satisfy this condition.

Section 8: Numbers, Dates, Intl, & Timers

Converting and Checking Numbers:

Converting Strings to Numbers

parseInt(string, radix)

```
parseInt("42", 10);   // 42
```

Purpose:

- Converts a string to an integer.

Parameters:

- **string:** The string to be converted.
- **radix:** An integer between 2 and 36 that represents the base in mathematical numeral systems.

parseFloat(string)

```
parseInt("42.42"); // 42
```

Purpose:

- Purpose: Converts a string to a floating-point number.

Parameters:

- **string:** The string to be converted.

Number Checking

isNaN(value)

`isNaN("abc");` // true

Purpose:

- Determines whether a value is NaN (Not-a-Number).

Parameters:

- **value:** The value to be checked.

isFinite(value)

`isFinite(42);` // true

Purpose:

- Determines whether a value is a finite number.

Parameters:

- **value:** The value to be checked.

Other Methods and Properties in the Number Object

Number(value)

```
Number("42.42");    // 42.42
```

Purpose:

- Converts a string or boolean to a number. Can be used to convert both integers and floats.

Parameters:

- **value**: The value to be converted.

Number.isInteger(value)

```
Number.isInteger(42);   // true
```

Purpose:

- Determines whether a value is an integer.

Parameters:

- **value**: The value to be checked.

Number.isNaN(value)

```
Number.isNaN(NaN);    // true
```

Purpose:

- A more robust version of isNaN, which doesn't coerce the value to a number before checking.

Parameters:

- **value**: The value to be converted.

Number.isFinite(value)

```
Number.isFinite(42);   // true
```

Purpose:

- A more robust version of isFinite, which doesn't coerce the value to a number before checking.

Parameters:

- **value:** The value to be converted.

Number.isSafeInteger(value)

```
Number.isSafeInteger(42);   // true
```

Purpose:

- Determines whether a value is a safe integer, i.e., it is a number that can precisely represent an integer in JavaScript.

Parameters:

- **value:** The value to be converted.

Working With Dates:

Creating Date Objects

```
// Creating Date Objects
let currentDate = new Date();
console.log(currentDate);

// Date from a String
let dateFromString = new Date('September 12, 2023 10:20:30');
console.log(dateFromString);

// Date from Year, Month, Day, Hours, Minutes, Seconds, and Milliseconds
let dateFromComponents = new Date(2023, 8, 12, 10, 20, 30, 0);
console.log(dateFromComponents);

// Date from Timestamp (milliseconds since January 1, 1970, 00:00:00 UTC)
let dateFromTimestamp = new Date(1689247230000);
console.log(dateFromTimestamp);
```

Note: **Month parameter is zero-indexed** (i.e., 0 = January, 1 = February, ..., 11 = December).

Common Methods on Date Objects

getFullYear()

```
console.log(currentDate.getFullYear()); // e.g., 2023
```

Gets the full year (4 digits for 4-digit years).

getMonth()

```
console.log(currentDate.getMonth());      // e.g., 8 (for September)
```

Gets the month (0-11).

getDate()

```
console.log(currentDate.getDate());       // e.g., 12
```

Gets the day of the month (1-31).

getDay()

```
console.log(currentDate.getDay());        // e.g., 2 (for Tuesday)
```

Gets the day of the week (0-6, where 0 is Sunday).

getHours(), getMinutes(), getSeconds(), getMilliseconds()

```
console.log(currentDate.getHours());          // e.g., 10
console.log(currentDate.getMinutes());        // e.g., 20
console.log(currentDate.getSeconds());        // e.g., 30
console.log(currentDate.getMilliseconds());   // e.g., 0
```

Gets the respective components of the time.

setFullYear(year), setMonth(month), setDate(date), etc.

```
currentDate.setFullYear(2025);
currentDate.setMonth(0);
currentDate.setDate(1);
```

Sets the respective components of the date.

getTime()

```
console.log(currentDate.getTime());           // e.g., 1689247230000
```

Gets the timestamp (milliseconds since January 1, 1970).

toISOString()

```
console.log(currentDate.toISOString());       // e.g., "2023-09-12T10:20:30.000Z"
```

Gets the date as a string in ISO format.

Date and Time Formatting

toLocaleDateString()

```
console.log(currentDate.toLocaleDateString());    // e.g., "9/12/2023"
```

Formats the date as a string, using locale-specific settings.

toLocaleTimeString()

```
console.log(currentDate.toLocaleTimeString());    // e.g., "10:20:30 AM"
```

Formats the time as a string, using locale-specific settings.

toLocaleString()

```
console.log(currentDate.toLocaleString());     // e.g., "9/12/2023, 10:20:30 AM"
```

Formats both the date and time as a string, using locale-specific settings.

Difference between Dates

```
let date1 = new Date(2023, 8, 12);
let date2 = new Date(2023, 8, 15);
let differenceInMs = date2 - date1;
console.log(differenceInMs / (1000 * 60 * 60 * 24));   // 3 (difference in days)
```

To find the difference between two dates, you can subtract one date object from another, which gives the difference in milliseconds.

Section 9: Advanced DOM & Events

<u>How the DOM works:</u>

- Allows developers to make JavaScript interact with the browser
- JavaScript can create, modify, and delete HTML elements; set styles, classes, and attributes, as well as listen and respond to events
- DOM tree is generated from an HTML document which we can then interact with
- the DOM is a complex API that contains lots of useful methods and properties to interact with the DOM tree

Organization

The DOM is organized as a tree structure where each node is an object representing a part of the document. The nodes can be elements, attributes, text content, etc. Here's a basic organization:

- **Document Node:** The root node which represents the entire document.
- **Element Nodes:** Represent elements (HTML tags) in the document.
- **Attribute Nodes:** Represent attributes of elements.
- **Text Nodes:** Represent the text content within elements.
- **Comment Nodes: R**epresent comments in the document.

Common Methods

Document Methods
• **`document.getElementById(id):`** Gets an element by its ID. • **`document.getElementsByClassName(className):`** Gets elements by their class name. • **`document.getElementsByTagName(tagName):`** Gets elements by their tag name. • **`document.querySelector(selector):`** Gets the first element that matches the specified CSS selector. • **`document.querySelectorAll(selector):`** Gets all elements that match the specified CSS selector.

Element Methods

- **`element.appendChild(childElement):`** Adds a new child element.
- **`element.removeChild(childElement):`** Removes a child element.
- **`element.setAttribute(name, value):`** Sets the value of an attribute.
- **`element.getAttribute(name):`** Gets the value of an attribute.
- **`element.removeAttribute(name):`** Removes an attribute.
- **`element.classList.add(className):`** Adds a class to the element.
- **`element.classList.remove(className):`** Removes a class from the element.
- **`element.classList.toggle(className):`** Toggles a class on the element.

Creation and Insertion Methods

- **`document.createElement(tagName):`** Creates a new element.
- **`document.createTextNode(text):`** Creates a new text node.
- **`element.insertBefore(newElement, referenceElement):`** Inserts a new element before the reference element.

Common Properties

Node Properties

- **`node.childNodes:`** A NodeList of child nodes.
- **`node.parentNode:`** The parent node of the current node.
- **`node.nextSibling:`** The next sibling node.
- **`node.previousSibling:`** **`Th`**e previous sibling node.

Element Properties

- **`element.id:`** The ID of the element.
- **`element.className:`** The class of the element.
- **`element.tagName:`** The tag name of the element.
- **`element.innerHTML:`** The HTML content inside the element.
- **`element.textContent:`** The text content of the element.
- **`element.style:`** Allows to manipulate the inline style of an element.

Event Handling

JavaScript can also register various event handlers on elements in the DOM, allowing for interactive web pages.

Event Listener

`element.addEventListener(event, function):` Adds an event listener to an element.
`element.removeEventListener(event, function):` Removes an event listener from an element.

Events Example

```
// Get an element by its ID
let element = document.getElementById('myElement');

// Set the innerHTML property to change its content
element.innerHTML = 'New Content';

// Add a class to the element
element.classList.add('newClass');

// Add an event listener to the element
element.addEventListener('click', function() {
  alert('Element clicked!');
});
```

Selecting, Creating, and Deleting Elements:

Selecting DOM Elements

```
// Selecting an element by ID
let elementById = document.getElementById('myId');

// Selecting elements by class name (returns a NodeList)
let elementsByClassName = document.getElementsByClassName('myClass');

// Selecting elements by tag name (returns a NodeList)
let elementsByTagName = document.getElementsByTagName('p');

// Selecting an element using a CSS selector (returns the first match)
let elementByQuerySelector = document.querySelector('.myClass');

// Selecting elements using a CSS selector (returns a NodeList)
let elementsByQuerySelectorAll = document.querySelectorAll('.myClass');
```

Creating DOM Elements

```
// Creating a new element
let newElement = document.createElement('div');

// Creating a new text node
let newText = document.createTextNode('Hello, World!');

// Adding the text node to the new element
newElement.appendChild(newText);

// Adding a class to the new element
newElement.classList.add('myNewClass');

// Adding the new element to the body
document.body.appendChild(newElement);
```

Deleting DOM Elements

```
// Selecting an element to remove
let elementToRemove = document.getElementById('elementToRemove');

// Removing the selected element
if (elementToRemove) {
  elementToRemove.parentNode.removeChild(elementToRemove);
}

// Alternatively, you can use the remove method to remove an element
if (elementToRemove) {
  elementToRemove.remove();
}
```

Notes:

- Before deleting an element, it's a good practice to check if the element exists to avoid a **null** reference error, as demonstrated in the deleting example.
- When selecting elements by class name or tag name, or using **querySelectorAll**, a **NodeList** is returned. You can loop through this list using a **for** loop or **forEach** method to perform operations on each element in the list.
- The **createElement** and **createTextNode** methods are used to create new elements and text nodes, respectively. Once created, these can be added to the DOM using methods like **appendChild** or **insertBefore**.

Events:

In JavaScript, events are actions or occurrences that happen in the browser, often triggered by users interacting with a webpage or by the browser itself. Event handlers are functions that are called in response to these events.

Click Event

```
document.getElementById('myButton').addEventListener('click',
function() {
  alert('Button was clicked!');
});
```

Triggered when an element is clicked.

Mouseover Event

```
document.getElementById('myElement').addEventListener('mouseover',
function() {
  alert('Mouse is over the element!');
});
```

Triggered when the mouse pointer is moved onto an element.

Mouseout Event

```
document.getElementById('myElement').addEventListener('mouseout',
function() {
  alert('Mouse is out of the element!');
});
```

Triggered when the mouse pointer is moved out of an element.

Keydown Event

```
window.addEventListener('keydown', function(event) {
  alert('Key pressed: ' + event.key);
});
```

Triggered when a key is pressed down.

Keyup Event

```
window.addEventListener('keyup', function(event) {
  alert('Key released: ' + event.key);
});
```

Triggered when a key is released.

Change Event

```
document.getElementById('myInput').addEventListener('change',
function() {
  alert('Input value changed to: ' + this.value);
});
```

Triggered when the value of an input element changes.

Submit Event

```
document.getElementById('myForm').addEventListener('submit',
function(event) {
  event.preventDefault(); // Prevents the form from submitting the traditional way
  alert('Form submitted!');
});
```

Triggered when a form is submitted.

Load Event

```
window.addEventListener('load', function() {
  alert('Window loaded!');
});
```

Triggered when the window has finished loading.

Resize Event

```
window.addEventListener('resize', function() {
  alert('Window resized!');
});
```

Triggered when the window is resized.

Notes:

- In these examples, **addEventListener** is used to attach event handlers to elements or the window. The first argument is the event type (as a string), and the second argument is the event handler function.
- Inside the event handler function, you can use the **event** object to access information about the event (like which key was pressed for keyboard events).
- For the submit event, **event.preventDefault()** is used to prevent the form from being submitted in the traditional way, allowing you to handle the form submission with JavaScript instead.

Intersection Observer API:

The Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport.

Explanation

1. **Create an Intersection Observer:** You create an intersection observer by calling its constructor and passing it a callback function to be run whenever a threshold is crossed in one direction or the other.

2. **Setting Options:** You can optionally pass a set of options to the IntersectionObserver constructor, which allows you to control the circumstances under which the observer's callback is invoked.
3. **Observing Target Elements:** Once you've created the observer, you need to give it some targets to observe.

4. **Handling Intersections:** The callback function receives a list of IntersectionObserverEntry objects and the observer itself. Each entry contains information about the intersection, including the target element and whether it's intersecting.

5. **Unobserving Targets:** If you want to stop observing a target, you can call the unobserve method.

Options

- **root:** The element that is used as the viewport for checking visibility of the target. Must be the ancestor of the target. Defaults to the browser viewport if not specified or if **null**.

- **rootMargin:** Margin around the root. Can have values similar to the CSS margin property, e.g., "10px 20px 30px 40px" (top, right, bottom, left). The values can be percentages. This set of values serves to grow or shrink each side of the root element's bounding box before computing intersections.

- **threshold:** Either a single number or an array of numbers which indicate at what percentage of the target's visibility the observer's callback should be executed.

Example

```
// Step 1: Create an Intersection Observer
let observer = new IntersectionObserver((entries, observer) => {
  entries.forEach(entry => {
 // Step 4: Handling Intersections
    if (entry.isIntersecting) {
      console.log('Element is intersecting');
      entry.target.style.backgroundColor = 'green';

  // Unobserve the element if you want to observe the intersection only once
      observer.unobserve(entry.target);
    } else {
      console.log('Element is not intersecting');
      entry.target.style.backgroundColor = 'red';
    }
  });
}, {
 // Step 2: Setting Options
  root: null, // observing intersections with respect to the viewport
  rootMargin: '0px',
  threshold: 0.5 // callback will be run when 50% of the target is visible
});

// Step 3: Observing Target Elements
document.querySelectorAll('.observe').forEach(target => {
  observer.observe(target);
});
```

In this example, we have elements with the class **observe** that we want to observe. When 50% or more of an element is visible in the viewport, its background color changes to green, and when less than 50% is visible, its background color changes to red. Once an element has become 50% visible, we stop observing it.

Lazy Loading:

Lazy loading is a strategy to identify resources as non-blocking (or non-essential) and load these only when needed. It's a great way to help reduce initial page load time, lower resource usage, and speed up content delivery. In the context of web development, it's often used for images, but it can also be applied to scripts or other content types.

Lazy Loading Images

1. **HTML Structure:** Initially, set the src attribute of your images to a placeholder or set it to null, and store the actual image URL in a data attribute (like data-src).

2. **JavaScript:** Use the Intersection Observer API to observe when these images come into the viewport. When an image comes into the viewport, replace the src attribute with the actual image URL from the data-src attribute.

HTML Example

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Lazy Loading Example</title>
<style>
  .lazy {
    width: 100%;
    height: 200px;
    background-color: #f0f0f0; /* Placeholder background color */
  }
</style>
</head>
<body>

<img class="lazy" data-src="path/to/your/image1.jpg" alt="Image 1">
<img class="lazy" data-src="path/to/your/image2.jpg" alt="Image 2">
<!-- Add more images as needed -->

<script src="path/to/your/script.js"></script>
</body>
</html>
```

Here we have images with the **class lazy**, and the **data-src** attribute contains the URL of the actual image. The src attribute is not set, so the images won't be loaded initially.

JavaScript Example

```
document.addEventListener("DOMContentLoaded", function() {
  let lazyImages =
[].slice.call(document.querySelectorAll("img.lazy"));
  let imageObserver = new IntersectionObserver((entries, observer) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        let lazyImage = entry.target;
        lazyImage.src = lazyImage.dataset.src;
        lazyImage.classList.remove("lazy");
        imageObserver.unobserve(lazyImage);
      }
    });
  });

  lazyImages.forEach((lazyImage) => {
    imageObserver.observe(lazyImage);
  });
});
```

Here we set up an Intersection Observer to observe elements with the class **lazy**. When an image comes into the viewport (i.e., **isIntersecting** is true), we set its **src** attribute to the value of the **data-src** attribute, causing the image to be loaded. We then remove the **lazy** class and stop observing the image.

This way, images are only loaded as they come into the viewport, saving bandwidth and improving page load time.

Section 10: OOP in JavaScript

Overview of OOP in JavaScript:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of properties, and code, in the form of methods. In JavaScript, OOP can be implemented using various approaches including using object literals, constructor functions, classes (introduced in ES6), and object prototypes.

- **Objects:** Collections of key-value pairs. They can contain properties and methods.
- **Classes:** Introduced in ES6, classes are a blueprint for creating objects with a specific structure and behavior.
- **Inheritance:** Allows you to create a new class that shares the properties and methods of another class.
- **Encapsulation:** The bundling of data and the methods that operate on that data, restricting the access to some of the object's components.
- **Polymorphism:** The ability to present the same interface for different data types.

Object Literals

```
let person = {
  firstName: 'John',
  lastName: 'Doe',
  getFullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

console.log(person.getFullName());              // Output: John Doe
```

Constructor Functions

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;

  this.getFullName = function() {
    return this.firstName + ' ' + this.lastName;
  };
}

let person = new Person('John', 'Doe');
console.log(person.getFullName());                // Output: John Doe
```

Classes (ES6)

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

let person = new Person('John', 'Doe');
console.log(person.getFullName());               // Output: John Doe
```

Inheritance

```
class Employee extends Person {
  constructor(firstName, lastName, position) {
    super(firstName, lastName);
    this.position = position;
  }

  getEmployeeProfile() {
    return `${this.getFullName()} - ${this.position}`;
  }
}

let employee = new Employee('John', 'Doe', 'Developer');
console.log(employee.getEmployeeProfile());      // Output: John Doe - Developer
```

Prototypes

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function() {
  return this.firstName + ' ' + this.lastName;
};

let person = new Person('John', 'Doe');
console.log(person.getFullName());                 // Output: John Doe
```

<u>**Prototypes:**</u>

Prototypes are a mechanism by which objects can inherit properties and methods from other objects. This prototype-based inheritance is a core concept in JavaScript and is the foundation of object-oriented programming in the language.

Prototype Chain: Every object in JavaScript has a prototype. When you try to access a property or method of an object, JavaScript will first look on the object itself, and if it doesn't find it there, it will look on the object's prototype, and so on up the prototype chain, until it either finds the property/method or reaches an object with a null prototype.

Prototype Property: Functions in JavaScript have a prototype property (which is an object) that is used when new objects are created from that function using the new keyword. This prototype property is not to be confused with the prototype of the function itself.

Object Prototypes: In JavaScript, objects can share a prototype. If you add a property or method to an object's prototype, that property or method will be available to all objects that share that prototype.

Prototype Example

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

// Adding a method to the Person prototype
Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};

// Creating a new instance of Person
let person1 = new Person('John', 'Doe');

// The getFullName method is available to person1, even though it's not a property of person1 itself
console.log(person1.getFullName()); // Output: John Doe

// Creating another instance of Person
let person2 = new Person('Jane', 'Smith');

// The getFullName method is also available to person2, because person2 shares the prototype of
person1
console.log(person2.getFullName()); // Output: Jane Smith

// Adding a new method to the Person prototype
Person.prototype.greet = function() {
  console.log(`Hello, I am ${this.firstName} ${this.lastName}`);
};

person1.greet(); // Output: Hello, I am John Doe
person2.greet(); // Output: Hello, I am Jane Smith
```

In this example, the **Person function** is a **constructor function**, and we add methods to its prototype that are then available to all instances of **Person**. This is a powerful feature of JavaScript, as it allows you to define methods in one place and have them available to all instances, which can save memory and keep your code organized.

<u>Prototypal Inheritance and Delegation:</u>

Prototypal Inheritance

- **Creating Objects:** When you create an object in JavaScript, it has a prototype property that points to another object. This prototype object, in turn, has its prototype, forming a prototype chain.

- **Inheritance:** When you try to access a property or method on an object, JavaScript will first look on the object itself. If it doesn't find it there, it will look on the object's prototype, and so on up the prototype chain, until it either finds the property/method or reaches an object with a null prototype.

- **Constructor Prototype:** When you create an object using a constructor function, the prototype of the created object is set to the prototype property of the constructor function.

Prototypal Delegation

- **Delegation:** This is the process where an object delegates behavior to another object linked through its prototype chain. If an object doesn't have a particular property or method, it delegates the search to its prototype.

- **Dynamic Dispatch:** This refers to the process where the JavaScript engine determines at runtime which method or property to invoke. It allows objects to dynamically inherit behavior from their prototypes.

- **Method Overriding:** In prototypal delegation, you can override inherited methods by defining a method with the same name on the object itself.

Prototypal Inheritance/Delegation Example

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};

function Employee(firstName, lastName, position) {
  Person.call(this, firstName, lastName);       // Inherit properties from Person
  this.position = position;
}

// Set up inheritance with Person's prototype
Employee.prototype = Object.create(Person.prototype);

// Set the constructor property back to Employee
Employee.prototype.constructor = Employee;

// Add a method to Employee prototype
Employee.prototype.getEmployeeProfile = function() {
  return `${this.getFullName()} - ${this.position}`;
};
let employee = new Employee('John', 'Doe', 'Developer');

console.log(employee.getFullName());              // Output: John Doe
console.log(employee.getEmployeeProfile());       // Output: John Doe - Developer
```

In this example, the **Employee** function inherits from the **Person** function through prototypal inheritance, and we use prototypal delegation to call the **getFullName** method defined on the **Person** prototype from the **Employee** prototype. This demonstrates how you can create a chain of linked objects, each delegating behavior to its prototype, forming the basis of object-oriented programming in JavaScript.

ES6 Classes:

ES6 introduced classes to JavaScript, providing a much cleaner and more concise way to create objects and deal with inheritance. Even though it might seem like a different paradigm, it's important to note that ES6 classes are essentially syntactic sugar over the existing prototype-based inheritance in JavaScript.

Class Declaration: Classes are declared using the class keyword followed by the class name.

Constructor: The constructor method is a special method for creating and initializing an object created from a class. It runs automatically when a new instance of the class is created.

Methods: Methods are defined inside the class body and represent the behaviors that instances of the class can perform.

Inheritance: Classes can inherit from other classes using the extends keyword. The super keyword is used to call the constructor of the parent class.

Encapsulation: Classes allow you to bundle properties and methods together as a single unit, and control the accessibility of the members using access modifiers like public, private, and protected (note that private fields were introduced in later versions, not in the initial ES6 release).

Classes Example

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  greet() {
    console.log(`Hello, my name is ${this.getFullName()}`);
  }
}

class Employee extends Person {
  constructor(firstName, lastName, position) {
    super(firstName, lastName); // Call the parent class constructor
    this.position = position;
  }

  getEmployeeProfile() {
    return `${this.getFullName()} - ${this.position}`;
  }
}

// Creating an instance of the Person class
const person = new Person('John', 'Doe');
person.greet();                              // Output: Hello, my name is John Doe

// Creating an instance of the Employee class
const employee = new Employee('Jane', 'Smith', 'Developer');
console.log(employee.getEmployeeProfile());     // Output: Jane Smith - Developer
```

In this example:

- We define a **Person** class with a constructor and two methods: **getFullName** and **greet**.
- We then define an **Employee** class that extends the **Person** class, inheriting its properties and methods, and adding a new property (**position**) and a new method (**getEmployeeProfile**).
- We create instances of both classes and call their methods to demonstrate how they work.

Getters and Setters:

In JavaScript, getters and setters are special kinds of methods in objects and classes that allow you to define how to get and set values of object properties. They provide a way to control the access and modification of the properties, and can be used to add additional logic or validations when getting or setting property values.

Getters

- **Definition:** Getters are defined using the **get** keyword followed by a function. This function is called when the property is accessed.
- **Usage:** Getters are used to access the value of an object's property. You can also use them to compute a value based on other properties of the object.
- **Syntax:** You can define a getter like this: **get propertyName()** { ... }.

Setters

- **Definition:** Setters are defined using the **set** keyword followed by a function. This function is called when the property is set to a new value.
- **Usage:** Setters are used to set the value of an object's property. You can also use them to validate or transform the value before setting it.
- **Syntax:** You can define a setter like this: **set propertyName(value) { ... }.**

Getters and Setters

```
class Person {
  constructor(firstName, lastName) {
    this._firstName = firstName;
    this._lastName = lastName;
  }

 // Getter for the fullName property
  get fullName() {
    return `${this._firstName} ${this._lastName}`;
  }

 // Setter for the fullName property
  set fullName(value) {
    const parts = value.split(' ');
    if (parts.length === 2) {
      this._firstName = parts[0];
      this._lastName = parts[1];
    } else {
      throw new Error('Invalid name format');
    }
  }

 // Additional getters and setters for individual properties
  get firstName() {
    return this._firstName;
  }

  set firstName(value) {
    if (value) {
      this._firstName = value;
    } else {
      throw new Error('First name cannot be empty');
    }
  }
  get lastName() {
    return this._lastName;
  }

  set lastName(value) {
    if (value) {
      this._lastName = value;
    } else {
      throw new Error('Last name cannot be empty');
    }
  }
}

const person = new Person('John', 'Doe');
console.log(person.fullName); // Output: John Doe

person.fullName = 'Jane Smith';
console.log(person.fullName); // Output: Jane Smith

person.firstName = 'Emily';
console.log(person.fullName); // Output: Emily Smith

// person.fullName = 'SingleName'; // This will throw an error because the format is invalid
```

- We define a Person class with private properties (**_firstName** and **_lastName**) and public getters and setters (**firstName**, **lastName**, and **fullName**).
- The **fullName** getter computes the full name by concatenating the first and last names.
- The **fullName** setter splits the input value into first and last names and sets the individual properties. It also validates the input format.
- The **firstName** and **lastName** setters include validation to prevent setting empty values.

Static Methods:

Static methods are methods that are associated with the class itself, rather than with instances of the class.

Definition: Static methods are defined using the **static** keyword. They are called on the class itself, not on instances of the class.

Usage: Static methods are often used to create utility functions for an application, or to create factory methods which can return instances of the class with a certain setup.

Accessing: Static methods are accessed using the class name, followed by the method name (e.g., **ClassName.staticMethodName()**).

Context: Inside a static method, the **this** keyword refers to the class itself, not an instance of the class. Therefore, static methods cannot access non-static properties and methods of the class.

Static Method Example

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }

 // Static method
  static greet() {
    console.log('Hello from the Person class!');
  }

 // Static factory method
  static createAnonymousPerson() {
    return new Person('Anonymous', 'Person');
  }
}
// Calling the static method
Person.greet(); // Output: Hello from the Person class!

// Using the static factory method to create a new Person instance
const anonymousPerson = Person.createAnonymousPerson();
console.log(anonymousPerson.getFullName()); // Output: Anonymous Person
```

- We define a **Person** class with a constructor, a method (**getFullName**), and two static methods (**greet** and **createAnonymousPerson**).
- The **greet** static method is a simple utility method that logs a message to the console.
- The **createAnonymousPerson** static method is a factory method that creates and returns a new **Person** instance with a predefined setup.
- We demonstrate calling the static methods using the class name (**Person**), and show that the static factory method can create and return new instances of the class.

Object.create Method:

The **Object.create()** method is a static method in JavaScript that creates a new object, using an existing object as the prototype of the newly created object.

Syntax
`Object.create(proto, [propertiesObject])`
proto: This is the object which should be the prototype of the newly-created object. **propertiesObject (optional):** This is an object representing the properties to be added to the newly-created object. Each property is defined by a property descriptor.

Prototype Linking: The primary purpose of Object.create() is to extend the prototype chain. The new object created has its prototype set to the object passed as the first parameter.

Property Descriptors: The second parameter allows you to add additional properties to the new object, with a detailed level of configuration, including setting properties as writable, enumerable, and configurable.

Usage: It is commonly used for implementing inheritance and for creating objects with a specific prototype.

Create Method Example

```
const personProto = {
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};
// Creating a new object with personProto as its prototype
const john = Object.create(personProto, {
  name: {
    value: 'John',
    writable: true,
    enumerable: true,
    configurable: true
  },
  age: {
    value: 30,
    writable: true,
    enumerable: true,
    configurable: true
  }
});

john.greet();                                  // Output: Hello, my name is John

const jane = Object.create(personProto);
jane.name = 'Jane';                            // Adding a property to jane
jane.greet();                                  // Output: Hello, my name is Jane
```

- We first define an object **personProto** that contains a method **greet**.
- We then use **Object.create()** to create two new objects (**john** and **jane**) with **personProto** as their prototype.
- We pass a properties object as the second parameter to **Object.create()** when creating **john** to define the **name** and **age** properties with specific property descriptors.
- We demonstrate that both **john** and **jane** have access to the **greet** method from their prototype, and that they can have their own individual properties (**name** and **age**).

Encapsulation:

Encapsulation is a core concept in object-oriented programming that restricts access to some of the object's components, which is a means to prevent unintended interference and misuse of the objects. In JavaScript, encapsulation can be achieved using various techniques including closures, module patterns, and using class syntax with private fields and methods.

Private Fields: In JavaScript, you can create private fields by prefixing the field name with a **# character**. Private fields cannot be accessed or changed from outside the class.

Private Methods: Similarly, you can create private methods using the **# prefix**. Private methods cannot be called from outside the class.

Public Methods: You can provide public methods in the class to interact with the private fields and methods, allowing controlled access to the object's properties.

Constructor: The constructor can be used to initialize both public and private properties when an object is created.

Encapsulation Example

```
class Person {
  #firstName;  // Private field
  #lastName;  // Private field

  constructor(firstName, lastName) {
    this.#firstName = firstName;
    this.#lastName = lastName;
  }
 // Public method to get the full name
  getFullName() {
    return `${this.#firstName} ${this.#lastName}`;
  }
 // Public method to set the first name
  setFirstName(firstName) {
    if (firstName) {
      this.#firstName = firstName;
    } else {
      throw new Error('First name cannot be empty');
    }
  }
 // Private method to get a formal greeting
  #getFormalGreeting() {
    return `Hello, my name is ${this.#firstName} ${this.#lastName}`;
  }
 // Public method to get a greeting
  getGreeting() {
    return this.#getFormalGreeting();
  }
}
const person = new Person('John', 'Doe');
console.log(person.getFullName());          // Output: John Doe

person.setFirstName('Jane');
console.log(person.getFullName());          // Output: Jane Doe
console.log(person.getGreeting());           // Output: Hello, my name is Jane Doe

// Trying to access private members of a class will throw errors
person.#firstName;                                          // Error
person.#getFormalGreeting();                                // Error
```

- We define a **Person** class with private fields (**#firstName** and **#lastName**) and a private method (**#getFormalGreeting**).
- We provide public methods (**getFullName**, **setFirstName**, and **getGreeting**) to interact with the private fields and methods, allowing controlled access to the object's properties.
- We demonstrate that the private fields and methods cannot be accessed directly from outside the class, enforcing encapsulation.

Section 11: Asynchronous JavaScript

Synchronous vs Asynchronous:

In JavaScript, and in programming in general, the terms "synchronous" and "asynchronous" refer to two different modes of executing code, especially when it comes to operations that take some time to complete (like network requests, file system operations, etc.).

Synchronous JavaScript

Sequential Execution: In synchronous JavaScript, operations are performed one after another. The next operation must wait for the previous operation to complete before it can begin.

Blocking: Synchronous code is blocking, meaning that if one operation takes a long time to complete, it will block the execution of the subsequent operations, which can lead to performance issues in terms of responsiveness and usability.

Simplicity: Synchronous code is generally simpler to write and understand because it follows a linear progression of steps, where each step is executed in order.

Error Handling: Error handling in synchronous code can be straightforward as you can use try-catch blocks to handle errors inline.

Synchronous Example

```
function fetchData() {
   let data = readFileSync('data.json'); //will block execution until the file is read
   console.log(data);
}
fetchData();
console.log("This will execute after data has been logged");
```

Asynchronous JavaScript

Concurrent Execution: Asynchronous JavaScript allows operations to be performed concurrently, meaning that you can start a new operation before a previous operation has completed.

Non-blocking: Asynchronous code is non-blocking, which means that long-running operations (like network requests) won't block the execution of other operations. This allows for more responsive and efficient applications.

Complexity: Asynchronous code can be more complex to write and understand due to the use of callbacks, promises, and async/await syntax, which can lead to nested or chained code structures.

Error Handling: Error handling in asynchronous code can be done using callback patterns, promise chaining with .catch() or try-catch blocks in async functions.

Event Loop: Asynchronous JavaScript leverages the event loop to manage the execution of asynchronous tasks, which can lead to more scalable and performant applications.

Asynchronous Example

```
function fetchData() {
   fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => console.log(data))
      .catch(error => console.log('Error:', error));
}
fetchData();
console.log("This will execute before data has been logged");
```

Common Asynchronous Patterns in JavaScript

Callbacks: Earlier approach where functions are passed as arguments to asynchronous functions and are called once the asynchronous operation completes.

Promises: An improvement over callbacks, providing a cleaner and more manageable way to handle asynchronous operations.

Async/Await: A syntactic sugar over promises, which allows you to write asynchronous code in a way that looks synchronous, making it easier to write and understand.

AJAX and API calls:

AJAX

AJAX stands for Asynchronous JavaScript and XML. It is a technique used to create faster and more interactive web applications. With AJAX, you can update parts of a web page without reloading the entire page. Here's a brief explanation:

XMLHttpRequest Object: At the heart of AJAX is the **XMLHttpRequest** object, which allows you to send HTTP requests and receive HTTP responses from a web server asynchronously.

Data Formats: Though "XML" is in the name, AJAX can work with other data formats, including JSON, HTML, and plain text.

Asynchronous Nature: AJAX operations are asynchronous, allowing the rest of your code to continue executing while the AJAX request is being processed.

Making API Calls

To make API calls in JavaScript, you can use various techniques, including using the **XMLHttpRequest object**, the **fetch API**, or third-party libraries like **Axios**.

XMLHttpRequest

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data", true);
xhr.onreadystatechange = function () {
   if (xhr.readyState == 4 && xhr.status == 200)
      console.log(JSON.parse(xhr.responseText));
};
xhr.send(null);
```

Fetch API (Recommended for Modern Browsers)

```
fetch('https://api.example.com/data')
   .then(response => response.json())
   .then(data => console.log(data))
   .catch(error => console.log('Error:', error));
```

The Fetch API is a modern, promise-based API for making network requests. It is generally easier to use and more powerful than **XMLHttpRequest**.

Async/Await with Fetch

```
async function fetchData() {
   try {
      let response = await fetch('https://api.example.com/data');
      let data = await response.json();
      console.log(data);
   } catch (error) {
      console.log('Error:', error);
   }
}
fetchData();
```

You can use the async/await syntax to work with promises in a more synchronous-looking manner.

Using Axios

```
axios.get('https://api.example.com/data')
   .then(response => {
      console.log(response.data);
   })
   .catch(error => {
      console.log('Error:', error);
   });
```

Axios is a third-party library that you can use to make HTTP requests. It supports both promise-based and async/await syntax.

First, you need to include Axios in your project (either by adding it via npm or including it via a <**script**> tag)

Error Handling

When making API calls, it's important to handle errors gracefully. This can be done using **.catch()** with promises or **try-catch blocks** with async/await.

Asynchronous Callbacks:

In an asynchronous context, callbacks are often used to handle the result of an operation that takes some time to complete, like a network request or a timer.

Timer Example

```
function waitAndExecute(time, callbackFunction) {
   setTimeout(() => {
      console.log("Time has passed.");
      callbackFunction();
   }, time);
}

function callbackFunction() {
   console.log("Executing callback function.");
}

waitAndExecute(2000, callbackFunction);

// Output:
// Time has passed.
// Executing callback function.
```

In this example, **waitAndExecute** takes a time (in milliseconds) and a callback function as arguments. It then uses **setTimeout** to wait for the specified time before executing the callback function.

Callback Hell

Callbacks can lead to complex and nested structures, a phenomenon sometimes referred to as "callback hell". This can happen when you have many nested callbacks, making the code hard to read and manage.

Callback Hell Example

```
getData('someurl', function(response) {
   parseData(response, function(parsedData) {
      transformData(parsedData, function(transformedData) {
         // ... and so on
      });
   });
});
```

In this example, **waitAndExecute** takes a time (in milliseconds) and a callback function as arguments. It then uses **setTimeout** to wait for the specified time before executing the callback function.

Avoiding Callback Hell

To avoid callback hell, you can:

- Use **Modularization** to break the code into smaller, reusable functions.
- Use **Promises** or **async/await**, which are higher-level abstractions for dealing with asynchronous operations, offering a cleaner and more manageable syntax.

Promises:

A **Promise** in JavaScript is a built-in object that represents the eventual completion or failure of an asynchronous operation. It allows you to write asynchronous code in a more readable and maintainable way.

Creating A Promise

```
let myPromise = new Promise((resolve, reject) => {
    let success = true;

    if(success) {
        resolve("The operation was successful.");
    } else {
        reject("The operation failed.");
    }
});
```

A promise can be created using the **Promise** constructor. It takes a function as an argument, which receives two parameters: **resolve** and **reject**.

Consuming a Promise

```
myPromise
    .then((message) => {
        console.log(message); // Output: The operation was successful.
    })
    .catch((message) => {
        console.log(message); // This will not be executed as the promise was resolved.
    });
```

Once a promise is created, it can be consumed using **.then()** method to handle the resolved value and **.catch()** method to handle the rejection.

Chaining Promises

```
function firstOperation() {
    return new Promise((resolve, reject) => {
        resolve("First operation completed");
    });
}

function secondOperation(message) {
    return new Promise((resolve, reject) => {
        resolve(`${message}, Second operation completed`);
    });
}

firstOperation()
    .then((result) => secondOperation(result))
    .then((finalResult) => console.log(finalResult)) // Output: First
operation completed, Second operation completed
    .catch((error) => console.log(error));
```

Promises can be chained to perform multiple asynchronous operations sequentially.

Error Handling

```
myPromise
    .then((message) => {
        throw new Error("This is an error!");
    })
    .catch((error) => {
        console.log(error.message); // Output: This is an error!
    });
```

Errors can be handled using **.catch()** method, which catches any error that occurs in the promise chain.

Promise Methods

Promise.all

```
Promise.all([promise1, promise2])
    .then((values) => console.log(values))
    .catch((error) => console.log(error));
```

Takes an iterable of promises and resolves when all of them are resolved or rejects as soon as one of them rejects.

Promise.race

```
Promise.race([promise1, promise2])
    .then((value) => console.log(value))
    .catch((error) => console.log(error));
```

Takes an iterable of promises and resolves or rejects as soon as one of the promises resolves or rejects.

Async/Await

```
async function asyncFunction() {
    try {
        let result = await myPromise;
        console.log(result); // Output: The operation was successful.
    } catch (error) {
        console.log(error.message);
    }
}

asyncFunction();
```

Promises are often used with the **async/await** syntax, which allows you to work with promises in a more synchronous-looking manner.

Async and Await:

async and **await** are extensions of Promises and provide a way to write asynchronous code in a cleaner and more readable way.

Async Function:
An async function is a function declared with the async keyword, which ensures that the function returns a promise. Even if you return a value directly from an async function, it will be wrapped in a resolved promise.

Async Example

```
async function myAsyncFunction() {
    return "Hello, World!";
}

myAsyncFunction().then(value => console.log(value)); // Output: Hello, World!
```

In the above example, even though we return a string, it gets wrapped into a resolved promise, and we can use **.then()** method to access the value.

Await Keyword:
Within an async function, you can use the await keyword to pause the execution of the function until a promise is resolved or rejected. It makes asynchronous code look and behave a little more like synchronous code, which can make it easier to write and understand.

Await Example

```
async function fetchUserData() {
    try {
        let response = await fetch('https://api.example.com/user');
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log('Error:', error);
    }
}

fetchUserData();
```

In this example, **fetchUserData** is an async function that uses **await** to pause its execution until **fetch** resolves, and then again until **response.json()** resolves.

Error Handling:
To handle errors in an async function, you can use a try-catch block. If an error occurs in any of the await expressions, it will be caught by the catch block.

Error Handling Example

```
async function fetchUserData() {
    try {
        let response = await fetch('https://api.nonexistent.com/user');
        if (!response.ok) {
            throw new Error('Network response was not ok ' +
response.statusText);
        }
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log('Error:', error);
    }
}

fetchUserData();
```

In this example, if the fetch request fails or the response is not ok, it will throw an error, which will be caught by the catch block.

Advantages:

- **Readable Code: async/await** makes it easier to write and read asynchronous code.

- **Error Handling:** Simplified error handling with try-catch blocks, making it similar to synchronous code.

- **Avoiding Callback Hell:** Helps to avoid "callback hell" and the need for chaining many **.then()** and **.catch()** methods.

Things to Remember:

- **await** can only be used inside an async function.
- **async** functions always return a promise, whether you use **await** inside them or not.
- If an error occurs in an await expression, it will reject the returned promise, which can be caught with **.catch()** or a try-catch block.

Practical Example

```
function simulateAsyncOperation(data, time) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(data);
        }, time);
    });
}

async function performOperations() {
    try {
   let result1 = await simulateAsyncOperation('First operation', 1000);
        console.log(result1);

   let result2 = await simulateAsyncOperation('Second operation', 500);
        console.log(result2);

   let result3 = await simulateAsyncOperation('Third operation', 200);
        console.log(result3);
    } catch (error) {
        console.log('Error:', error);
    }
}

performOperations();
```

Here's an example where we are simulating asynchronous operations with **setTimeout** and using **async/await** to handle them.

Extra Section 1: Project Planning

Project Planning:

1. **User Stores**
2. **Features**
3. **Flowchart**
4. **Architecture**
5. **Development**

Project planning is a very important part of software development and makes creating the actual code much easier. Most teams will start with a **Project Planning Phase** before moving to a **Development Phase.**

User Stories: A description of the applications functionality from the user's perspective. All user stories put together describe the entire application.

Features: putting User Stories together will allow developers to come up with the different features that allow the user to interact with the software

Flowchart: The features are then put into a flowchart format to see how each module of the software interacts with each other and how the "program flows" from start to finish.

Architecture: describes how we will organize the code and what JavaScript (or other language) features will be included (eg. frameworks, APIs, external resources, etc.)

User Story

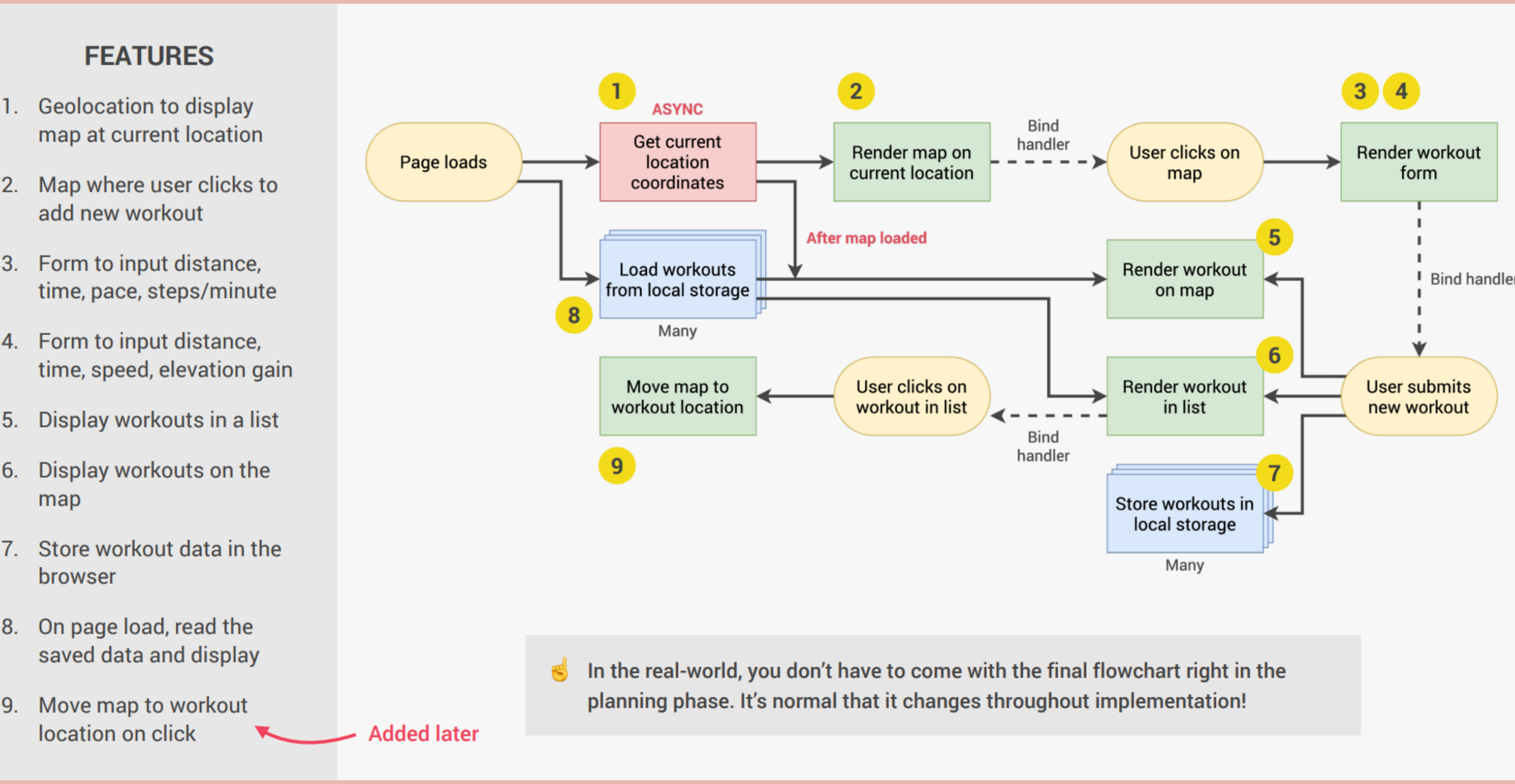
Common Format: As a **[type of user]**, I want **[an action]** so that **[a benefit].**

User Story Examples (Mapty App)
As an (App)user, I want to log my running workouts with location, distance, time, pace and steps/minutes, so I can keep a log of all my running.
As an (App)user, I want to log my cycling workouts with location, distance, time, and elevation gain, so I can keep a log of all my cycling.
As an (App)user, I want to see all my workouts at a glance, so I can easily track my progress.
As an (App)user, I want to see my workouts on a map, so I can easily check where I workout the most.
As an (App)user, I want to see all my workouts when I leave the app and come back later, so I can keep using the app over time.

Features

Features from User Stories	
As an (App)user, I want to log my running workouts with location, distance, time, pace and steps/minutes, so I can keep a log of all my running.	Map where user clicks to add new workout geolocation to display map at current location form to input distance, time, pace, steps/min
As an (App)user, I want to log my cycling workouts with location, distance, time, and elevation gain, so I can keep a log of all my cycling.	form to input distance,time, speed, elevation gain
As an (App)user, I want to see all my workouts at a glance, so I can easily track my progress.	display all workouts in a list
As an (App)user, I want to see my workouts on a map, so I can easily check where I workout the most.	display all workouts on the map
As an (App)user, I want to see all my workouts when I leave the app and come back later, so I can keep using the app over time.	Store workout data in the browser using local storage API

Flowchart



FEATURES

1. Geolocation to display map at current location
2. Map where user clicks to add new workout
3. Form to input distance, time, pace, steps/minute
4. Form to input distance, time, speed, elevation gain
5. Display workouts in a list
6. Display workouts on the map
7. Store workout data in the browser
8. On page load, read the saved data and display
9. Move map to workout location on click

☝️ In the real-world, you don't have to come with the final flowchart right in the planning phase. It's normal that it changes throughout implementation!

Architecture

Class Layout

Class Workout

- id
- distance
- duration
- coords
- date
- constructor()

Child Class Running

- name
- cadence
- pace
- constructor()

Child Class Cycling

- name
- elevationGain
- speed
- constructor()

Class App

- workouts[]
- map
- constructor()
- _getPosition()
- _loadMap(position)
- _showForm()
- _toggleElevationField()
- _newWorkout()

Extra Section 2: Modern JavaScript Development

Modules:

In JavaScript, modules refer to small units of independent, reusable code. They are a fundamental aspect that helps in structuring JavaScript programs, especially when developing complex applications. Modules allow you to split your code into separate files, which can then be imported and used in other files, fostering better organization, and maintainability.

ES6+ Modules (ECMAScript Modules)

With the release of ECMAScript 6 (ES6), a native module system was introduced to JavaScript.

Exporting Modules

```
// module1.js
export const sayHello = (name) => {
  console.log(`Hello, ${name}!`);
};

export default sayHello;
```

To make functions, objects, or primitives available to other modules, you can export them using the export keyword.

Importing Modules

```
// app.js
import { sayHello } from './module1.js';

sayHello('World'); // Output: Hello, World!
```

You can import functions, objects, or primitives from other modules using the import keyword.

Default Export

// app.js
```
import sayHello from './module1.js';

sayHello('World');
```
// Output: Hello, World!

Each module can have one **default** export, which can be imported without using curly braces

CommonJS (Used in Node.js)

CommonJS is another module system, predominantly used in Node.js environments.

CommonJS Exporting Modules

// module1.js
```
module.exports.sayHello = (name) => {
  console.log(`Hello, ${name}!`);
};
```

To make functions, objects, or primitives available to other modules, you can export them using the export keyword.

CommonJS Importing Modules

// app.js
```
const { sayHello } = require('./module1.js');

sayHello('World');
```
// Output: Hello, World!

You can import functions, objects, or primitives from other modules using the import keyword.

AMD (Asynchronous Module Definition)

AMD is a module system designed for the browser, allowing modules to be loaded asynchronously.

AMD Example

```
// Defining a module
define('module1', [], function() {
  return {
    sayHello: function(name) {
      console.log(`Hello, ${name}!`);
    }
  };
});

// Loading a module
require(['module1'], function(module1) {
  module1.sayHello('World'); // Output: Hello, World!
});
```

Module Scope

Each module in JavaScript has its own scope, meaning that variables, functions, and classes defined in a module are not accessible from outside the module, unless explicitly exported.

Module Loaders and Bundlers

In modern JavaScript development, tools like Webpack and Parcel are often used to bundle modules for browser compatibility, since not all browsers support ES6 modules natively.

Using Parcel:

How to Use Parcel v2 in Modern Web Development Workflow:

Step 1: Set Up Your Project

- **Initialize a New Project:** Start by setting up a new project using **npm init** or **yarn init**.
- **Install Parcel:** Install Parcel globally or as a dev dependency in your project using **npm install -g parcel** or **npm install --save-dev parcel**, respectively.

Step 2: Develop Your Application

- **Creating Files:** Create your HTML, CSS, and JavaScript files as per your project needs.
- **Importing Assets:** You can easily import assets such as images, CSS in your JavaScript files, and Parcel will handle the dependencies and bundling.

Step 3: Building and Running Your Application

- **Development Server:** Use the command **parcel serve <your_entry_file.html>** to start a development server with hot module replacement.
- **Building the Project:** To build your project for production, use the command **parcel build <your_entry_file.html>**. It will create optimized and minified bundles in the dist directory by default.
- **Scripts in package.json:** Add scripts like **"start": "parcel serve index.html"** and **"build": "parcel build index.html"** to your **package.json** for easier management.

Step 4: Configuration and Optimization (If Necessary)

- **Configuration Files:** If needed, add configuration files like **.browserslistrc, .postcssrc, etc.,** to tailor the build process to your needs.
- **Using Plugins:** Install and use plugins to add more functionalities or support for other file types.

Step 5: Deployment

- **Deploying Your Application:** Once built, your application is ready to be deployed. You can deploy the contents of the **dist directory** to your preferred hosting service.