ECMAScript and JavaScript

ECMA

- ECMA (European Computer Manufacturers Association) is an international standards organization responsible for standardizing ECMAScript, which serves as the foundation for JavaScript and other scripting languages. This note provides a comprehensive overview of ECMA, explains the difference between ECMAScript and JavaScript, and highlights the main purpose of ECMA standards for JavaScript.
- Founded in 1961, ECMA consists of member organizations representing various industry sectors, including technology companies, standards bodies, and government agencies.
- The organization focuses on developing and maintaining standards to promote interoperability and facilitate technological advancements.

ECMAScript vs. JavaScript:

ECMAScript:

- ECMAScript refers to the standardized scripting language specification governed by the ECMA organization.
- It defines the syntax, semantics, and behavior of scripting languages, including JavaScript.
- It provides a common foundation for multiple implementations, ensuring consistency and interoperability across platforms.

JavaScript:

- JavaScript is a popular programming language that serves as an implementation of the ECMAScript specification.
- It encompasses the ECMAScript rules, features, and core functionality, while also incorporating additional capabilities such as DOM manipulation, AJAX, and APIs specific to web browsers.

Purpose of ECMA Standards for JavaScript:

Interoperability

- The primary objective of ECMA standards for JavaScript is to promote interoperability across different implementations.
- By establishing a common set of rules and specifications, developers can write code that works consistently across various JavaScript engines, browsers, and platforms.

Stability

- ECMA standards provide a stable foundation for the JavaScript language. They ensure that core language features, syntax, and semantics remain consistent over time, reducing compatibility issues and enabling long-term code maintenance.

Language Evolution

- ECMA standards facilitate the evolution of the JavaScript language. Through a standardized process, proposals for new language features, enhancements, and modifications can be reviewed, refined, and incorporated into future ECMAScript versions.
- This enables JavaScript to stay relevant and adapt to the changing needs of the web development community.

Collaboration and Consensus

- The ECMA standards process encourages collaboration among developers, implementers, and other stakeholders.

- It provides a platform for the JavaScript community to actively participate in shaping the language, making decisions collectively, and ensuring that the standards reflect the needs and perspectives of a diverse range of developers.

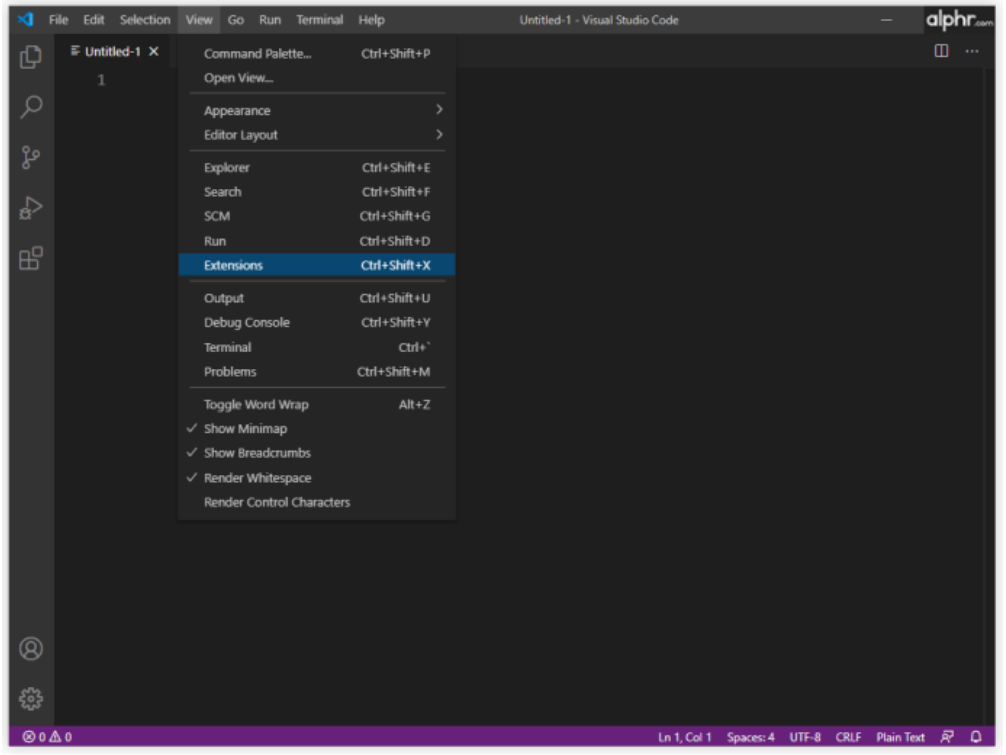
More Into VS Code

Installation of Extension in VS Code

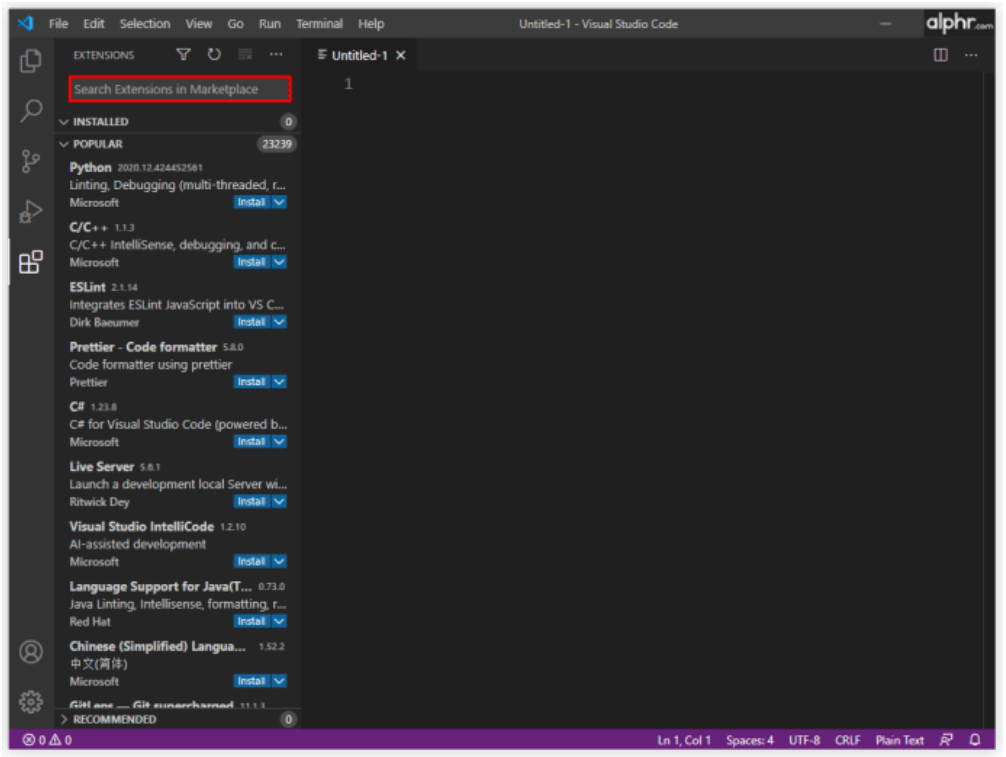
VS Code provides you with various extensions that you can use to set up your work environment and ease up your work. It lets you **add languages, debuggers, and tools to your installation to support your development workflow**.

To Download any extension follow these simple steps:

- Click on the "Extensions" button in the Activity Bar. It's located on the side of VS Code's client. Alternatively, you can use the keyboard shortcut "Ctrl+Shift+X" to open the "Extensions" screen.



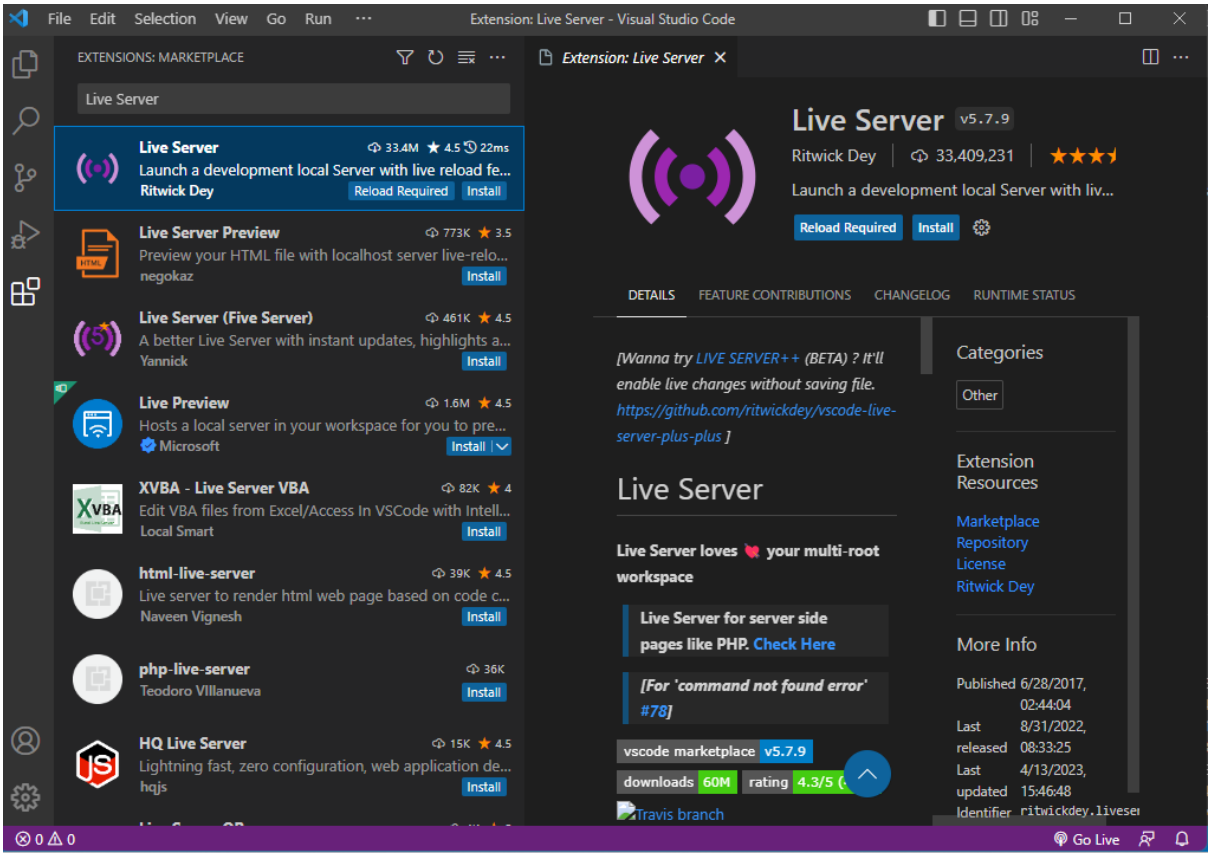
- This will bring you to the "Extensions" list. VS Code automatically sorts Extensions by popularity. You can also filter your results by using the search box at the top of the page.



- Each extension in the list will have a brief description, the download count (the number of times it has been downloaded), the publisher's name, and a rating from zero to five stars.

- Type the Extension name you want to add to your VS Code.

- You can click on the extension to see more details before committing to a download. Details include a changelog, frequently asked questions, and a list of contributions and dependencies the extension gives and requires from VS Code.



- Click on the install button to add that extension to your VS Code.
- You can also see all extensions installed in your VS Code by going to Extensions, and in the search Extension, type "@installed." This will give you all the installed extensions on your VS code.

Installation of live server

"Live server" is a popular web development tool that allows you to create a local development server on your computer. This tool is helpful for web developers who want to see changes in their code immediately without reloading the page.

Live server is a lightweight extension for popular code editors such as Visual Studio Code, Sublime Text, and Atom.

To install the live server, go to the extension and type live server, click on the install button, and the installation will begin.

Once you install the extension, you can simply open an HTML file and click the "Go Live" button to start a local development server. The server will automatically refresh the page whenever you change your code.

Installation of Prettier

Prettier is a popular code formatting tool that helps developers maintain consistent coding styles across their projects. It is available in multiple programming languages such as JavaScript, TypeScript, CSS, HTML, JSON, and more.

Prettier works by analyzing your code and automatically reformatting it to follow a set of predefined rules. These rules are based on best practices and community standards, ensuring your code is easy to read and maintain. Prettier can also detect and fix common code errors such as missing semicolons, unused variables, and trailing commas.

You can install prettier from the extension in VS Code. Once installed, you will be able to see the formatting once you save your file.

Shortcuts available in JavaScript

Visual Studio Code has a wide range of keyboard shortcuts to help developers increase their productivity. Here are some of the most commonly used keyboard shortcuts in VSCode that you may need in development:

Basic Navigation

Ctrl + P - Open file by name
Ctrl + Shift + E - Show Explorer sidebar
Ctrl + Shift + F - Search across files
Ctrl + Shift + D - Show Debug panel
Ctrl + Shift + X - Show Extensions panel

Editing

Ctrl + D - Select word or next occurrence
Ctrl + Shift + L - Select all occurrences
Ctrl + / - Toggle line comment
Ctrl + Shift + 7 - Toggle block comment
Ctrl + F - Find
Ctrl + H - Replace

Debugging

F5 - Start debugging
F9 - Toggle breakpoint
Shift + F5 - Stop debugging
F10 - Step over
F11 - Step into

Shift + F11 - Step out

Terminal

Ctrl + ` - Toggle terminal
Ctrl + Shift + ` - Create new terminal
Ctrl + Shift + T - Reopen closed terminal

These are just a few of the many keyboard shortcuts available in Visual Studio Code. You can view and customise the shortcuts by going to the "Keyboard Shortcuts" option in the "Preferences" menu.

Some References

Extension in VS Code: **link**

Getting Started with JavaScript

Why JavaScript?

Earlier, you learned how to create HTML elements and position or style them with CSS. But until now, you could not manipulate HTML elements or make the page dynamic. This is where javascript comes into play. Javascript provides you with broad functionality which you can use to create your page dynamic.
Here are a few reasons why JavaScript is so popular and what it offers over HTML and CSS:

- **Interactivity:** HTML and CSS are static languages that can create beautiful and well-structured web pages, but they are limited in terms of interactivity. JavaScript offers the ability to add interactivity to web pages, which can help engage users and create a more dynamic experience.
- **Dynamic content:** JavaScript allows for creating dynamic content on web pages, which can help make websites more engaging and responsive. For example, JavaScript can be used to create live updating news feeds, interactive maps, or real-time chat applications.
- **Event handling:** JavaScript offers powerful event handling capabilities, which allow web developers to respond to user interactions and other events on the web page. This can be used to trigger animations, update content, or perform other actions in response to user input.
- **Third-party libraries and frameworks:** JavaScript has a large and vibrant ecosystem of third-party libraries and frameworks, which can help developers build applications more efficiently. These libraries and frameworks can provide pre-built components, tools, and utilities that can save time and effort.

What is Javascript?

Javascript is a High Level, Light-weighted, Interpreted JIT compiled Multi-Paradigm Prototype Based Synchronous, Single Threaded Dynamic Language.
To understand the above definition, you should know what each term signifies within this context.
Here is a definition of each term used above:

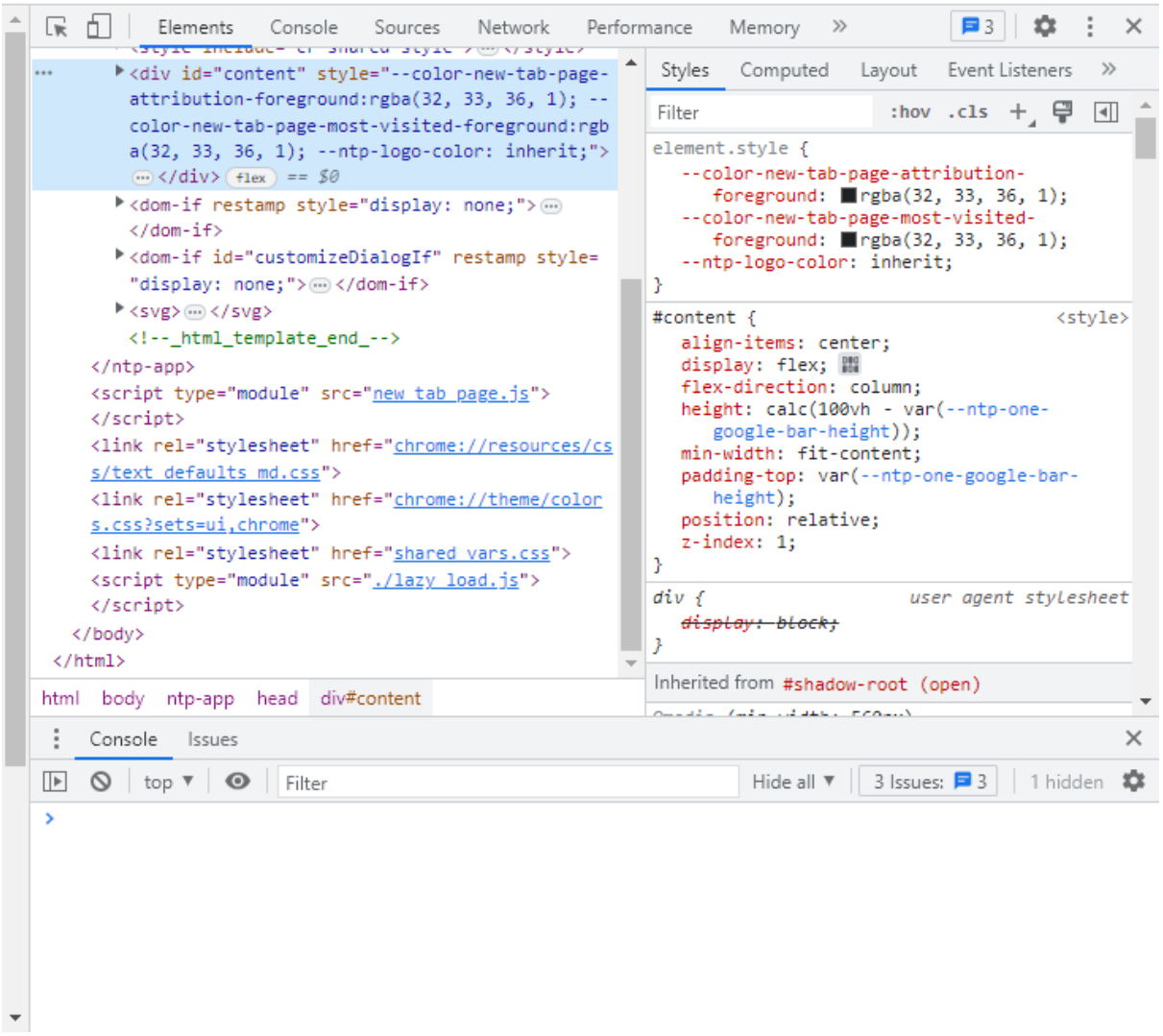
- **High Level**
  - High-level languages are those languages that are independent of system architecture.
- **Light-weighted**
  - The lightweight application is a computer program designed to have low system resource usage.
- **Interpreted or JIT-compiled**
  - An interpreter in the browser reads over the JavaScript to JS code, interprets each line, and runs it. More modern browsers use a JIT compilation which compiles JS to executable bytecode just as it is about to run.
- **Multi-Paradigm**
  - It supports different styles of writing the same code.
- **Prototype Based**
  - This type of style allows the creation of an object without first defining its class.
- **Synchronous**
  - Synchronous means the code runs in a particular sequence of instructions as given in the program.
- **Single-Threaded**
  - Single-threaded means it executes one line of code at a time. So it executes the current line of code before going to the next line.
- **Dynamic Language**
  - Dynamic refers to the value which is capable of change at run time.

History of JS

- In Dec 1991, the Internet was invented, and the first web browser created was Mosaic by Marc Andreessen and Eric Bina at the University of Illinois.
- In 1993 a new web browser called “Netscape” was co-founded by Marc Andreessen.
- But until then, only static websites were created and rendered using HTML. There was no responsiveness on the web pages.
- The developer felt the need for dynamic websites which would change with user interaction.
- In 1995 Brendan Eich was hired by Netscape to develop a scripting language for the browsers.
- Within 10 days, Brendan Eich created the first draft of Javascript and named it MOCHA.
- Later it was renamed to LiveScript and finally to Javascript over the years.
- At that time, Java was a hot language, so naming MOCHA to JavaScript was purely a marketing strategy, and JAVA is not related to JavaScript anyhow.
- At this time, to compete with Netscape, Microsoft decided to include scripting technologies in their browser. In 1996 they released Internet Explorer 3 with their own implementation of JavaScript called JScript.
- With the availability of two scripting languages for different browsers, it became a nuisance for developers to learn two languages.
- To solve this problem, Netscape submitted JavaScript to Ecma International to “standardize the syntax and semantics of a general purpose, cross-platform, vendor-neutral scripting language.”
- In 1997 the first version of ECMA 262 or ECMAScript (ES), now commonly known as JavaScript, was released.
- To this date, javaScript has been evolving, and new versions are added to it now and then.

First JavaScript Code

- To start the journey of your JavaScript, you should first understand where you can see the output of your javascript code in the browser and where you can write your first code of js.
- In every browser, you have the feature of developer tools which look something like this.



To open the DevTools or developer tools in your browser, you can follow any of these steps:

- Right-click anywhere on a page and select **Inspect**.

  or

- Click the three-dot button to the right of the address bar and select **More Tools** > **Developer Tools**.

  or

- Using Shortcut
    - For mac : Cmd + Option + **J**
    - For Windows : Ctrl + Shift + **J**

The DevTools provide many features to debug and understand programs. One of the many features of DevTools is the console window. The console displays error messages and other information related to the code running on the page.
You can write the single-line js code in the console window and perform simple operations. Just click on the console button from the devTools menu and start writing your code.

Dialog Boxes in JS

We have covered the following dialog boxes in this video:
i. Prompt box - instructs the browser to display a dialog with an optional message prompting the user to input some text and to wait until the user either submits the text or cancels the dialog.
ii. Alert box - instructs the browser to display a dialog with an optional message and to wait until the user dismisses the dialog.
iii. Confirm box - instructs the browser to display an optional message and wait until the user confirms or cancels the dialog.
These are actually web APIs and not JavaScript functions. It works on the window object, which you will learn about in the upcoming session.

Summarizing it

Let's summarize what we have learned in this Lecture:

- Why JavaScript was needed?
- What JavaScript is?
- History of Javascript.
- Where to write code in a browser
- Dialog Boxes in JS

Some References

- Definition of JavaScript: link
- Different ECMA versions available: link
- Different ways to open DevTools in Chrome: link

Fundamentals of JS-I

Code Structure

The first thing we'll study is the building blocks of code.

Statement

Statements are syntax constructs and commands that perform actions. We've already seen a statement, alert('Hello, world!'), which shows the message "Hello, world!".

We can have as many statements in our code as we want. Statements can be separated with a semicolon.

For example, here we split "Hello World" into two alerts:

```
alert('Hello'); alert('World');
```

Semicolon

A semicolon may be omitted in most cases when a line break exists.

This would also work:

```
alert('Hello')
alert('World')
```

In most cases, a newline implies a semicolon. But "in most cases" does not mean "always"!

There are cases when a newline does not mean a semicolon. For example:

```
alert(3 +
1
+ 2);
```

Comments

As time goes on, programs become more and more complex. It becomes necessary to add comments which describe what the code does and why.

Comments can be put into any place of a script. They don't affect its execution because the engine simply ignores them.

One-line comments start with two forward slash characters //.

The rest of the line is a comment. It may occupy a full line of its own or follow a statement.

```
// This comment occupies a line of its own
alert('Hello');

alert('World'); // This comment follows the statement
```

Multiline comments start with a forward slash and an asterisk `/*` and end with an asterisk and a forward slash `*/`.

Like this:

```
/* An example with two messages.
This is a multiline comment.
*/
alert('Hello');
alert('World');
```

Variables

Most of the time, a JavaScript application needs to work with information. Here are two examples.

- An online shop – the information might include goods being sold and a shopping cart.
- A chat application – the information might include users, messages, and much more. Variables are used to store this information.

A Variables

A variable is a “named storage” for data. We can use variables to store goodies, visitors, and other data.

To create a variable in JavaScript, use the let keyword

The statement below creates (in other words: declares) a variable with the name “message”:

```
let message;
```

Now we can store some information on these variables

```
let message;

message = 'Hello'; // store the string 'Hello' in the variable named message
```

Variable Naming

- Variable names must start with a letter, an underscore (_) or a dollar sign ($).
- Variable names cannot contain spaces.
- Variables cannot be the same as reserved keywords such as if or const.
- By convention, JavaScript variable names are written in camelCase.
- Variables should be given descriptive names that indicate their content and usage (e.g. sellingPrice and costPrice rather than x and y).

- As JavaScript variables do not have set types, it can be useful to include an indication of the type in the name (e.g. orderNumber is obviously a numeric ID, whereas order could be an object, a string or anything else).

LET KEYWORD

Let keyword is used to declare variables in javascript. They were introduced in ECMA6

Syntax:

```
let message;
message = "let are used to store variables"
```

Variables defined by let cannot be redeclared in the same scope but can be reassigned.

```
Let x= 8; x=16;
Console.log (x);
```

The output for the above code will be 16 since you are redeclaring the variable x.

VAR KEYWORD

Var keyword is used to declare variables.

Syntax:

```
Var message = "var is used to declare variables"
console.log("message")
```

Var allows readeclaration as well as reassign of variables.

CONST KEYWORD

Const keyword is used to declare variables.Variables which are defined using const can not be changed after assigning.

Syntax:

```
const message = "var is used to declare variables"
console.log("message")
```

Const does not allows readeclaration as well as reassign of variables.

Data types

A data type is a classification of the type of data that can be stored and manipulated within a program.

Each data type has its own characteristics and methods for manipulation.

Classification of data types:

Data Types are broadly classified into the following two categories:

1. **Primitive data types**: These basic data types represent a single value. JavaScript has seven primitive data types. They include:

   - **number**: represents both integer and floating-point numbers in JavaScript.
   - **string**: represents textual data in JavaScript and can be enclosed in single quotes,double quotes, or backticks. In JavaScript, there are several ways to represent strings:
     - Single quotes ('string'): Strings can be enclosed in single quotes.
     - Double quotes ("string"): Strings can also be enclosed in double-quotes.
     - Template literals/backticks (`string`): Template literals were introduced in ECMAScript 6 and allowed strings to be enclosed in backticks. They are mostly used to represent a string literal that can contain placeholders for variables,

➢ making it easier to concatenate strings and variables. In addition, the backtick notation also allows for multiline strings.

Example:

```
// using Single Quote('')
let myString='Coding Ninjas!';

// using Double Quotes("")
let my_String="Coding Ninjas!";

// using Backticks(``)
let String=`Coding Ninjas!`;
let myTemplateString=`Coding
Ninjas!`;
```

- ❖ **boolean**: represents a logical value, which can be either true or false.
- ❖ **null**: represents the intentional absence of any object value.
- ❖ **undefined**: represents a variable that has been declared but has not been assigned a value.
- ❖ **bigint**: represents integers with arbitrary precision. These values usually have 'n' at the end of the number. For example: let num = 10n; Here, num is of bigint data type
  - ❖ symbol: represents a unique value that can be used as a key for object properties.

2. **Object data types:**

- ❖ An object is a non-primitive data type representing a collection of related properties and methods. It can be thought of as a container for related data and behaviour, similar to an object in the real world.

JavaScript Primitive Wrapper Types

JavaScript provides three primitive wrapper types: Boolean, Number, and String types.

The primitive wrapper types make it easier to use primitive values including booleans, numbers, and strings.

See the following example:

```
let language = 'JavaScript';
let s = language.charAt(4);
console.log(s);  // Script
```

Code language: JavaScript (javascript)

In this example, The variable `language` holds a primitive string value. It doesn't have any method like charAt `()`. However, the above code works perfectly.

Methods in Strings

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their length, to build and concatenate them using the + and += string operators, checking for the existence or location of substrings with the indexOf() method, or extracting substrings with the substring() method.

toLowerCase()	Converts a string to lower case
toUpperCase()	Converts a string to Upper Case
includes()	Returns a boolean value if a string is contained by another string
charAt()	Returns the character at a specified inde

- length is property of strings, not a method. It returns the number of characters in a string.
- `let str = "Hello, World!"; console.log(str.length); // Outputs: 13`
- So, it is accessed as str.length without parentheses, unlike methods that are invoked with parentheses like str.toUpperCase().

Fundamentals of JS-II

Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations on numerical values.

1. Addition (+): Adds two values.
   Example:

```
let a = 5, b = 3;
console.log(a + b); // Output: 8
```

2. Subtraction (-): Subtracts the right operand from the left operand.
   Example:

```
console.log(a - b); // Output: 2
```

3. Multiplication (*): Multiplies two values.
   Example:

```
console.log(a * b); // Output: 15
```

4. Division (/): Divides the left operand by the right operand.
   Example:

```
console.log(a / b); // Output: 1.6666666666666667
```

5. Modulus (%): Returns the remainder of the division of the left operand by the right operand.
   Example:

```
console.log(a % b); // Output: 2
```

6. Exponentiation (**): Raises the left operand to the power of the right operand.
   Example:

```
console.log(a ** b); // Output: 125
```

Comparison Operators:

Comparison operators are used to compare two values and return a Boolean result.

1. Equal (==): Returns true if the values on both sides are equal.
   Example:

```
console.log(a == b); // Output: false
```

2. Not Equal (!=): Returns true if the values on both sides are not equal.
   Example:

```
console.log(a != b); // Output: true
```

3. Greater Than (>): Returns true if the left operand is greater than the right operand. Example:

```
console.log(a > b); // Output: true
```

4. Less Than (<): Returns true if the left operand is less than the right operand.
   Example:

```
console.log(a < b); // Output: false
```

5. Greater Than or Equal To (>=): Returns true if the left operand is greater than or equal to the right operand.
   Example:

```
console.log(a >= b); // Output: true
```

6. Less Than or Equal To (<=): Returns true if the left operand is less than or equal to the right operand.
   Example:

```
console.log(a <= b); // Output: false
```

Logical Operators:

Logical operators are used to perform logical operations on Boolean values.

1. AND (&&): Returns true if both the left and right operands are true.
   Example

```
let x = true, y = false;
console.log(x && y); // Output: false
```

2. OR (||): Returns true if at least one of the operands is true.
   Example:

```
console.log(x || y); // Output: true
```

3. NOT (!): Returns true if the operand is false and vice versa.
   Example:

```
console.log(!x); // Output: false
```

Type Conversion:

Type conversion is the process of converting one data type to another.

1. Implicit Type Conversion (Coercion): Automatically performed by the interpreter.
   Example:

```
let int_var = 5 + 2.0;
console.log(int_var); // Output: 7.0
```

   (Here, the integer 5 is implicitly converted to a float)

2. Explicit Type Conversion (Casting): Done by the programmer using predefined functions. Example:

```
let str_var = String(42);
console.log(str_var); // Output: "42"
```

   (Here, the integer 42 is explicitly converted to a string)

Type Coercion:

Type coercion is the automatic conversion of one data type to another.
Example of Type Coercion:

```
const value1 = "5";
const value2 = 9;
let sum = value1 + value2;
console.log(sum);
//output: "59"
```

When using the + operator between a string and a number, JavaScript coerces the number (9) into a string ("9") and then performs string concatenation.

Overall Importance of this lecture

- These concepts are foundational for building algorithms, making decisions, and performing various operations in JavaScript.
- They are essential for creating dynamic and interactive web applications.
- Understanding operator precedence helps in writing expressions that are evaluated as intended.
- Type conversion and coercion play a crucial role in managing different types of data.

In summary, mastering these concepts is fundamental for anyone working with JavaScript, whether for web development, server-side programming, or any other application where JavaScript is used. They provide the tools needed to manipulate data and control the flow of a program effectively.

Reference

- **Arithmetic Operators** - https://javascript.info/comparison
- **Comparison Operators** - https://javascript.info/comparison
- **Logical Operators** - https://javascript.info/logical-operators
- **Bitwise Operators** - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND
- **Type Conversion** - https://developer.mozilla.org/en-US/docs/Glossary/Type_Conversion
- **Type Coercion** - https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion

codingninjas

Conditional and Looping statement in javascript

If Statement:

The if statement is used for conditional execution of code. It evaluates a condition and, if the condition is true, executes a block of code.

```
if (condition) {
  // Code to be executed if the condition is true
}
```

Example:

```
let num = 10;

if (num > 0) {
   console.log("The number is positive"); // Output: The number is positive
}
```

If-Else Statement:

The if-else statement extends the if statement by providing an alternative block of code to be executed if the condition is false.

```
  if (condition) {
  // Code to be executed if the condition is true
} else {
  // Code to be executed if the condition is false
}
```

Example

```
let num = -5;

if (num > 0) {
  console.log("The number is positive");
} else {
  console.log("The number is non-positive"); // Output: The number
is non-positive
}
```

If-Else-If Statement:

The if-else-if statement allows you to check multiple conditions and execute different blocks of code based on which condition is true.

```
if (condition1) {
 // Code to be executed if condition1 is true
} else if (condition2) {
 // Code to be executed if condition2 is true
} else {
 // Code to be executed if none of the conditions are true
}
```

Example

```
let num = 0;

if (num > 0) {
  console.log("The number is positive");
} else if (num < 0) {
  console.log("The number is negative");
} else {
  console.log("The number is zero"); // Output: The number is zero
}
```

Switch Statement:

The switch statement is used to perform different actions based on different conditions. It is often more concise than a series of if-else-if statements.

```
switch (expression) {
 case value1:
 // Code to be executed if expression equals value1
 break;
 case value2:
 // Code to be executed if expression equals value2
 break;
 // Additional cases as needed
 default:
 // Code to be executed if none of the cases match
}
```

Example:

```
switch (day) {
  case "Monday":
     console.log("It's the start of the week"); // Output: It's
the start of the week
    break;
  case "Friday":
    console.log("It's almost the weekend");
    break;
  default:
    console.log("It's a regular day");
}
```

For Loop:

The for loop is used when you know the number of iterations in advance.

```
for (initialization; condition; iteration) {
 // Code to be executed in each iteration
}
```

Example

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

While Loop:

The while loop is used when the number of iterations is not known in advance, and it continues iterating as long as the specified condition is true.

```
while (condition) {
 // Code to be executed as long as the condition is true
}
```

Example

```
let i = 0;

while (i < 5) {
  console.log("Iteration " + (i + 1));
  i++;
}
// Output:
// Iteration 1
// Iteration 2
// Iteration 3
// Iteration 4
// Iteration 5
```

Do-While Loop:

Similar to the while loop, but it guarantees that the code inside the loop will be executed at least once, as the condition is checked after the execution of the loop body.

```
do {
 // Code to be executed at least once
} while (condition);
```

Example

```
                  let i = 0;

do {
  console.log("Iteration " + (i + 1));
  i++;
} while (i < 4);
// Output:
// Iteration 1
// Iteration 2
// Iteration 3
// Iteration 4
```

Nested Loops:

You can use loops inside other loops to create nested structures for more complex iterations.

```
for (let i = 0; i < 5; i++) {
 for (let j = 0; j < 3; j++) {
 // Code to be executed for each combination of i and j
 }
}
```

Example

```
for (let i = 0; i < 3; i++) {
  for (let 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
```

References :

If statement : Click here to read more

Switch statement : click here to read more

Looping : Click here to read more

Arrays, Strings and Function

Functions

A function in JavaScript is a block of code that performs a specific task. It takes inputs (parameters) and returns a value or performs an action.

Why Use Functions?

Functions are an essential part of programming in JavaScript. Here are a few reasons why we use functions:

- **Reusability:** You can reuse code by calling the same function multiple times.
- **Modularity:** Functions help in breaking down complex code into smaller, more manageable parts.
- **DRY (Don't Repeat Yourself):** Functions help you avoid writing repetitive code.

Creating a function

Creating a function in JS involves the following steps:

1) Use the `function` keyword followed by the function name.
2) Inside the parentheses, specify any parameters the function will accept (if any).
3) Define the code to be executed by the function inside curly braces.
4) Use the `return` keyword to specify the value to be returned by the function (if any).

Creating a function using parameters:

- Here's an example of creating a function with parameters:

```
function addNumbers(num1, num2) {
  console.log(num1 + num2);
}
```

  In this example, `addNumbers` is the function name and `num1` and `num2` are the parameters. The function adds the two parameters and prints the result.

- Consider another example using string literals:
  - String literals can be used in functions in JS by enclosing the string within backticks (` `) instead of single or double quotes. This allows for the use of template literals, which can contain variables and expressions within `${}` that are evaluated and concatenated with the string at runtime.
  - Using string literals in functions can make code more readable and concise by reducing the need for concatenation or escaping characters. It can also make it easier to insert dynamic data into the string output.

```
function greet(message) {
    console.log(`Hello, ${message}!`);
}
greet('Welcome to Coding Ninja');
```

Creating function using Default parameters:

In ES6, we can define default values for function parameters. Default parameters are used when a value is not passed to the function for that parameter. Here's an example:

```
function greet(name = 'World') {
    console.log(`Hello, ${name}!`);
}
greet(); // Output: Hello, World!
```

Parameters vs arguments

Parameters are the variables declared in a function's definition. Arguments are the values passed to the function when it is called.

Return statements

- Functions in JavaScript can optionally return a value using the `return` keyword. The value returned by a function can be used in other parts of the code.
- The return statement is used to stop the execution of a function and return a value to the calling code.
- A function can only have a single return statement. This means that once the return statement is executed, the function will immediately terminate and return the value specified by the statement.

- Any statements that appear after the return statement will be considered "unreachable code," meaning they will not be executed. It's important to keep this in mind when designing functions, as any code written after the return statement will not have any effect on the function's output.

Invoking a function

To call a function in JavaScript, we simply need to use the function name followed by parentheses and any arguments (if required) inside the parentheses. Here's an example:

```
function addNumbers(a, b) {
    return a + b;
}
const result = addNumbers(5, 10);
console.log(result); // Output: 15
```

In this example, 5 and 10 act as arguments, and the value of the variable `result` is equal to the function's return value.

Arrays

In JavaScript, an array is a collection of data values of any data type. Arrays are a fundamental data structure and are used to store and manipulate data in many JavaScript programs.

Defining Array in JS:

There are different ways to define an array in JavaScript. They are listed below:

1. The most common way is to use **square brackets** and list out the elements of the array separated by commas. For example:

```
const array_name = [item1, item2, item3];
const myArray = [10, 20, 30, 40, 50];
```

2. Another way to define an array is to use the **`Array() constructor`. For example:**

```
const myArray = new Array(1, 2, 3, 4, 5);
```

3. In JavaScript, we can use the **`Array.from()` method** to create a new array from an existing array-like or iterable object or from a string.

   - `Array.from()` can be used to create an array from a string, where each character in the string becomes an element in the new array, like this:

```
const myString = 'hello';
const newArray = Array.from(myString);
console.log(newArray); // ['h', 'e', 'l', 'l', 'o']
```

   - Arrays can contain any number of elements, separated by commas. The elements in an array can be of any data type, including other arrays, objects, or functions. **For Example:**

```
const myArray = [1, "two", true, ["four", "five"], { six: 6 }];
```

Accessing Array elements:

Arrays in JavaScript are zero-indexed, which means that the first element in an array is accessed using the index 0, the second element is accessed using the index 1, and so on.

1. **Accessing using square brackets:**

```
const myArray = [1, 2, 3, 4, 5];
console.log(myArray[0]); // Output: 1
console.log(myArray[2]); // Output: 3
```

2. **Accessing using `.at()`:**

```
const arr=[1,2,3,4,5];
console.log(arr.at(1));  // o/p: 2
```

Built-in array methods:

JavaScript arrays have several built-in methods that allow you to perform various operations on array elements. Here are some of the most commonly used methods:

1. **`push()`:** adds one or more elements to the end of an array.
    - Note that the `` `push()` `` method in JavaScript returns the new length of the array after adding the element(s) to the end of the array.

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.push("Lucknow", "Bangalore"); // adds Lucknow and Bangalore in array.
console.log(state.push("Punjab")); // Note: return 7
```

2. **`pop()`:** removes the last element from an array and returns it.
    - Note that the `` `pop()` `` method in JavaScript returns the value of the element that was removed from the end of the array.

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.pop();
console.log(state); // pops Chennai from the end
console.log(state.pop()); // Note: returns Kolkata
```

3. **`sort()`:** sort the elements of an array in ascending or descending order.

- The `sort()` method sorts the elements of an array as strings in alphabetical order. For example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.sort();
console.log(state); // o/p: [ 'Chennai', 'Delhi', 'Kolkata', 'Mumbai' ]
```

- The `sort()` function sorts numerical values on the basis of their Unicode. For example:

```
const num=[11,32,23,63,57];
num.sort();
console.log(num); // o/p: [ 11, 23, 32, 57, 63 ]
```

4. **`indexOf()`:** returns the index of the first occurrence of a specified element in an array.
   Example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
let index = state.indexOf("Kolkata");
console.log(index);

// Start from index 3:
const state1 = ["Mumbai", "Delhi", "Kolkata", "Chennai", "Kolkata"];
let index1 = state.indexOf("Kolkata",3);
console.log(index1); // o/p: 4
```

5. **`slice()`:** returns a new array containing a portion of the original array.
   - Negative indices can be used with the `slice()` method to extract elements from the end of an array.
   - Example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai", "Punjab", "Bangalore", "Rajasthan"];
const slicedState1 = state.slice(2);
const slicedState2 = state.slice(-2); // output: ["Banglore" "Rajasthan"]
console.log("Original: ", state);
console.log("New: ", slicedState1);
console.log("New: ", slicedState2);
```

6. **`splice()`:** update the original array content by removing or adding elements.
   - **Syntax:**

     ```
     array.splice(start, deleteCount, item1, item2,item3);
     ```

     Here,
     - `array`: The array to be modified
     - `start`: The index at which to start changing the array. If negative, it will begin that many elements from the end of the array.
     - `deleteCount`: An integer indicating the number of old array elements to remove. If set to 0, no elements are removed.
     - `item1, item2, ... item N`: The elements to add to the array, beginning from the start index. If you don't specify any elements, `splice()` will only remove elements from the array.

7. **`concat()`:** returns a new array that combines two or more arrays.
   Example:

```
const city1 = ["Jaipur", "Mumbai"];
const city2 = ["Kota", "Shimla", "Gurgaon"];
const mergeCity = city1.concat(city2);
console.log(mergeCity);
```

Iterating over Arrays:

In JavaScript, you can use loops to iterate over the elements of an array. The most common loop types for iterating over arrays are `for` loops, `for...of` loops, and `for...in` loops.

- **Using `for` loop:** The general `for` loop works iterates over an array as shown:

```
const myArray = [1, 2, 3, 4, 5];
for (let i = 0; i < myArray.length; i++) {
  console.log(myArray[i]);
}
```

  In this example, the loop iterates over each element of the `myArray` array and logs it to the console.

- **Using `for...of` loop:** A more concise way to iterate over an array is by using this loop shown as below:

```
const myArray = [1, 2, 3, 4, 5];
for (const element of myArray) {
  console.log(element);
}
```

  In this example, the `for...of` loop iterates over each element of the `myArray` array and logs it to the console.

  - You can also use other loop types, like `while` and `do...while`, to iterate over arrays, but `for` and `for...of` are the most commonly used.

  - When iterating over arrays using loops, it's essential to remember the array index and length to avoid errors like going out of bounds or skipping elements.

- **Using `for...in` loop:** In JavaScript, the `for...in` loop is used to iterate over the properties of an object (to be covered later), but it can also be used to iterate over the elements of an array. Here's a short example of how to use the `for...in` loop to iterate over an array:

```
let fruits = ['apple', 'banana', 'orange'];

for (let index in fruits) {
  console.log(fruits[index]);
}
```

  In this example, we have an array called `fruits` that contains three elements. We use the `for...in` loop to iterate over the indices of the fruits array, and then use the current index to access the corresponding element of the array using `fruits[index]`.

Note that while it's possible to use the `for...in` loop to iterate over an array, it's generally recommended to use the `for...of` loop instead, as it's specifically designed for iterating over iterable objects like arrays and avoids potential issues with inherited properties and unexpected behavior.

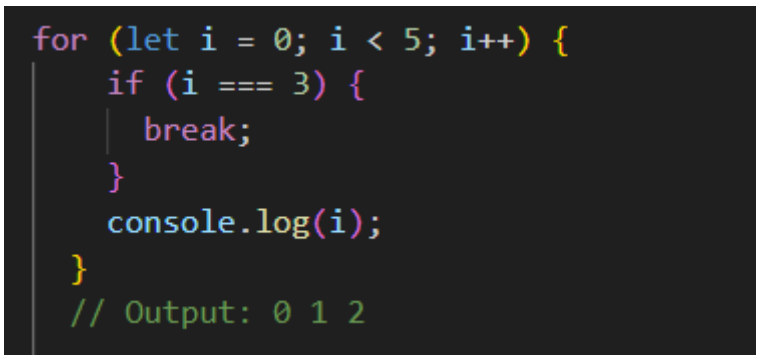
Difference between `for`, `for...of`, and `for...in` loops:

Loop Type	Syntax	Iterates Over	Returns
`for`	for (initialization; condition; iteration) { ... }	Numeric indices of an array or loop counter	Nothing
`for...in`	for (variable in object) { ... }	Property names of an object	Property values
`for...of`	for (variable of iterable) { ... }	Values of an iterable object (e.g. array, string, etc.)	Value

Break and Continue

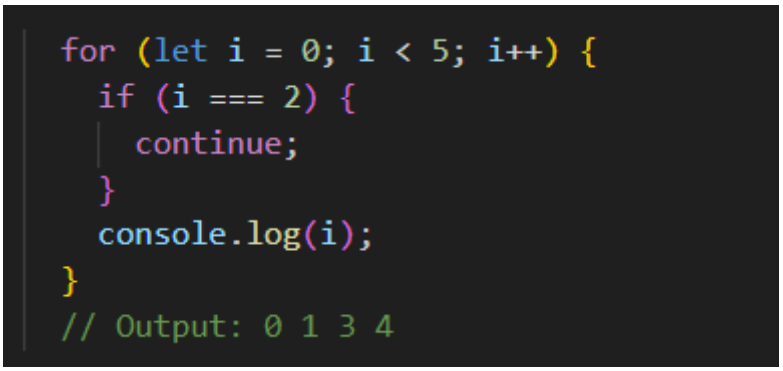
In JavaScript, `break` and `continue` are keywords used inside loops to control the flow of the iteration.

- `break` is used to exit out of a loop entirely when a certain condition is met. Once the `break` statement is executed, the loop will stop and control will move to the next line of code after the loop.
  For example:



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

- `continue` is used to skip the current iteration of a loop and move on to the next iteration. Once the `continue` statement is executed, any code that comes after it within the current iteration will be skipped.
  For example:



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

Spread and Rest Operator

Rest and Spread operators are useful tools in JavaScript that provide flexible ways to work with function arguments and arrays.

1. The `rest` **operator** `...` is used to collect multiple arguments into a single array in a function definition. It allows for a variable number of arguments to be passed to a function and accessed as an array. For example:

```
function sum(...numbers) {
    let total = 0;
    for (let number of numbers) {
      total += number;
    }
    return total;
  }

  console.log(sum(1, 2, 3)); // Output: 6
  console.log(sum(4, 5, 6, 7)); // Output: 22
```

   - In this example, the ...numbers syntax represents the `rest` parameter, which allows us to pass an arbitrary number of arguments to the `sum` function.
   - Inside the function, we use a `for` loop to iterate over the numbers array and add up all the values to get the total sum.
   - The `rest` operator is used here to collect the arguments passed to the function into an array, which we can then work with inside the function.

2. The `spread` **operator** `...` is used to spread the elements of an array into a new array or function arguments. It is commonly used to concatenate arrays or to pass an array of arguments to a function. For example:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const arr3 = [...arr1, ...arr2];

console.log(arr3); // [1, 2, 3, 4, 5, 6]
```

- In this example, we have two arrays `arr1` and `arr2`. We want to create a new array that contains all the elements from both arrays. We can use the `spread` operator to do this.

- By using `...arr1` and `...arr2`, we are spreading out the elements of the arrays and adding them to the new array `arr3`.
- Finally, we log the new array `arr3` to the console, which contains all the elements from `arr1` and `arr2`.

JavaScript Copying: Shallow vs Deep

- In JavaScript, when using the `spread` operator to copy an object or array, there are two types of copying: shallow and deep.
    - A shallow copy creates a new object or array, but only the top-level values are copied by reference. This means that if any copied values are themselves objects or arrays, they will still be referenced to the original objects or arrays.
    - On the other hand, a deep copy creates a completely new object or array, where all the nested values are also copied by value. This means that all the properties of the original object or array are duplicated, and no references are retained.
- The `spread` operator can be used to create a shallow copy of an object or array but not to create a deep copy. Here's an example of creating a shallow copy using the `spread` operator:

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];

originalArray[0] = 0;

console.log(originalArray); // [0, 2, 3]
console.log(copiedArray); // [1, 2, 3]
```

In this example, we created a new array `copiedArray` using the `spread` operator and assigned the values of `originalArray` to it. When we later modified the first value of `originalArray`, the change was not reflected in `copiedArray`, as it is a separate object.

Objects

Objects are collections of key-value pairs, where each key is a string (also called a "property name"), and each value can be any data type, including other objects. Objects can be used to represent complex data structures, such as arrays or even other objects.

Creating and Accessing Objects

Objects can be created using object literals, constructors, or using the Object.`create()` method. Once an object is created, its properties can be accessed and modified using the dot notation or bracket notation.

Example:

Objects can be created using object literals, enclosed in curly braces {} and using a colon to separate keys from values.

```
const person = {
    name: "John",
    age: 30,
    hobbies: ["reading", "hiking", "cooking"]
  };

  console.log(person.name); // output: John
  console.log(person.hobbies[0]); // output: reading
```

- In the example above, we have an object person with three properties: `name`, `age`, and `hobbies`. The hobbies property is an array of strings.
- To access the properties of the object, we can use dot notation (e.g. `person.name`) or square bracket notation (e.g., `person['name']`).
- To access elements of the hobbies array, we use the square bracket notation with the index of the element we want to access (e.g., `person.hobbies[0]` to access the first element of the hobbies array).

Summarizing it

Let's summarize what we have learned in this Lecture:

- Different types of Loops (`for`, `while`, and `do while`)
- Function in JS
- Arrays in JS
- Break and Continue
- Spread and Rest operator
- Objects in JS

References

- In-built Array methods: **Link**

Life Cycle of Variables

Life Cycle of Variables in the TDZ: `let`, `const`, and `var`

In JavaScript, the life cycle of variables varies depending on their type and presence in the Temporal Dead Zone (TDZ is the period between the creation and declaration of a `let` or `const` variable).

A brief overview of how `let`, `const`, and `var` declarations go through different stages, including TDZ, initialization, and usability is described below:

1. `let` and `const`

- **Creation:** Variables are created during the creation phase but unlike var it is not initialized with “undefined” rather it remain uninitialized.
- **TDZ (Temporal Dead Zone):** Variables enter the TDZ until they are formally declared.
- **TDZ Reference Error:** Accessing or assigning values during the TDZ results in a `ReferenceError`.
- **Declaration:** Variables are declared, initializing them with the assigned value.
- **Usable:** Once declared, variables can be accessed and assigned new values.

2. `var`

- **Hoisting:** `var` variables are hoisted during the creation phase to the top of their scope.
- **Initialization:** `var` variables are initialized with the value `undefined` during hoisting.

- **Usable:** `var` variables can be accessed and assigned values throughout their scope, even before their actual declaration in the code.

Reference: Link

Working of JS

Execution Context in JS

- Execution context is a fundamental concept in JavaScript that describes the environment in which a piece of code is executed.
- It includes variables, functions, and other elements that are necessary for the code to run.
- In other words, an execution context in JavaScript is a container that holds information about the current state of code being executed. The concept of execution context is important in understanding how JavaScript code is executed.

Components of Execution Context

There are two important components of an execution context: the **Variable Environment** and the **Thread of Execution**.

1) Variable Environment:

a) The Variable Environment is a fundamental component that organizes and holds all the variables, functions, and parameters accessible within a given scope. However, we will delve into the concept of scope and its significance in greater detail later on.

b) It is created when a new function is executed and contains information about all the variables that are declared within that function and the values assigned to them.

c) The variable environment also includes a reference to the outer environment, which is the variable environment of the parent scope.

2) Thread of Execution:

a) The Thread of Execution is the sequence of code execution that is currently being executed.
b) It is responsible for running the code one line at a time and keeping track of where the execution is at any given moment.
c) When a new function is called, a new thread of execution is created, and the execution continues within that thread until the function returns.

Phases of Execution Context

The Execution Context goes through two phases during its lifecycle:

1. Creation Phase:

a. During the creation phase, the JavaScript engine creates a new execution context and sets up the environment for executing the code.
b. This involves establishing a fresh variable environment, configuring the scope chain, and establishing a reference to the outer environment, which will be further explored when we cover the concept of scope chain."

2. Execution Phase:

a. During the execution phase, the JavaScript engine executes the code line by line within the thread of execution.
b. It uses the variable environment to look up variables and functions as needed and updates the values of variables as they are changed in the code.

Call Stack Overview

- Before we delve into the details of execution context, it's essential to understand how a computer remembers and manages the order of execution in JavaScript. This is where the **call stack** comes into play.
- The call stack is a fundamental mechanism used by JavaScript to keep track of function calls and their corresponding execution contexts.
- It acts as a memory structure that helps the computer remember which functions are currently being executed and where to return after a function completes its execution.

- We will cover the call stack more comprehensively in the following section, exploring its inner workings and how it influences the flow of our code.

Global vs Local Execution Context

Global Execution Context:

- The global execution context is the default environment in which JavaScript code is executed.
- The global execution context is created when the JavaScript program starts running and stays in memory until the program ends.
- During the creation phase, the JavaScript engine performs several steps to set up the environment for executing the code. These steps are described below:

1. Defining Window Object: The engine defines the global window object, which serves as the outermost object in the environment.

2. Creating `this` Variable: The `this` variable is created and assigned to the window object. It represents the context in which the current code is executing.
3. Hoisting: Hoisting takes place, where variable and function declarations are moved to the top of their respective scopes. This allows you to use variables and functions before they are formally declared in the code.
   We will cover the concept of hoisting in more detail in a dedicated section later on.

4. Memory Allocation: After hoisting, the engine allocates memory for variables and functions, preparing them for later use during the execution phase. It's important to note that the way variables are allocated and their initial values can vary depending on the type of declaration.

a. `var` Variables: When a variable declared with `var` is encountered during memory allocation, it is assigned the default value of `undefined`. This means that although the variable exists in memory, it holds the value `undefined` until a value is explicitly assigned to it.

b. `let` and `const` Variables: Variables declared with let and const also go through memory allocation. However, their behavior is more nuanced and will be discussed in a later section when we cover the Temporal Dead Zone (TDZ). It's during this phase that let and const variables are assigned the initial value of `undefined` within the TDZ.

Local Execution Context:

- A local execution context is created each time a function is called. It serves as a separate environment for the function's execution, encompassing several important aspects.
- Firstly, the local execution context defines the `this` variable. The value of `this` is determined based on how the function is invoked. If the function is called in the global scope or without any specific context, `this` will be assigned the global `window` object. However, in strict mode, `this` will be `undefined` in such cases.
- Additionally, the local execution context includes the creation of the `**arguments**` object. This object is available within the function and contains a list of all the arguments passed to it.
- Furthermore, during the creation phase of the local execution context, memory allocation takes place similar to the one we discussed above.
- When a variable is referenced in a function, JavaScript first looks for it in the local execution context's variable environment. If the variable is not found there, it looks in the parent execution context (if any), and so on, until it reaches the global execution context.
- When a function completes its execution, its local execution context is removed from memory.

Consider the example given below to understand the workflow of Execution Context in JavaScript.

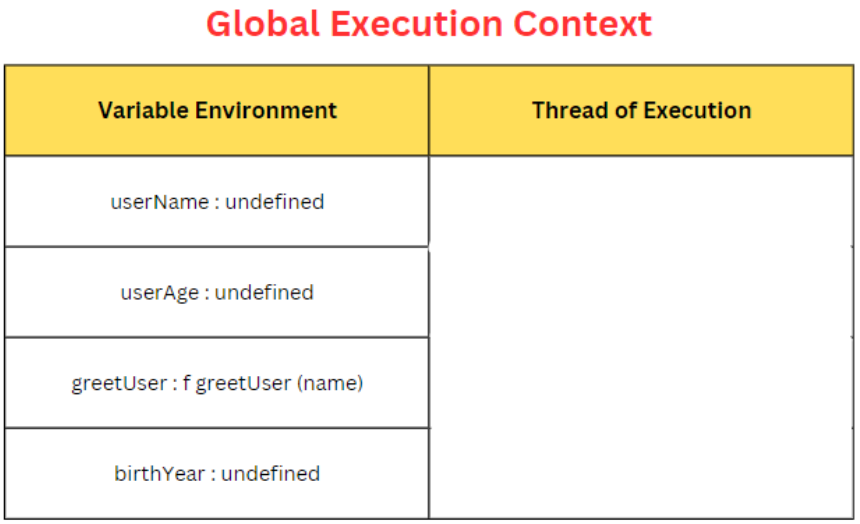
```
var userName='Tom';
var userAge=10;
console.log(`username: ${userName}`);
console.log(`userage: ${userAge}`);

function greetUser(name){
  var greet='I hope you are doing fine.';
  console.log(`hello, ${name}, ${greet}`);
  var currentYear = 2030;
  const year = currentYear - userAge;
  return `Your birthyear is ${year}`;
}
const birthYear = greetUser(userName);
console.log(birthYear);
```

The detailed explanation of the phases of Execution Context for the given code is as follows:

- The global and local execution context for the given code snippet can be described as shown below:

- The workflow for the execution context of this code snippet will be as follows:

1. **Creation Phase:**
    - During the creation of the global execution context, the JavaScript engine declares three variables (`userName, userAge`, and `birthYear`) and initializes them to undefined. It also declares a function named `greetUser`, which is stored in memory but not executed yet.
    - This work-flow is depicted in the figure below:



2. **Execution Phase:**
    a. The JavaScript engine assigns 'Tom' to the `userName` variable and 10 to the `userAge` variable. It then executes console.log() twice, outputting the values of `userName` and `userAge` to the console as "`username`: Tom" and "`userAge`: 10" as shown below:

Global Execution Context

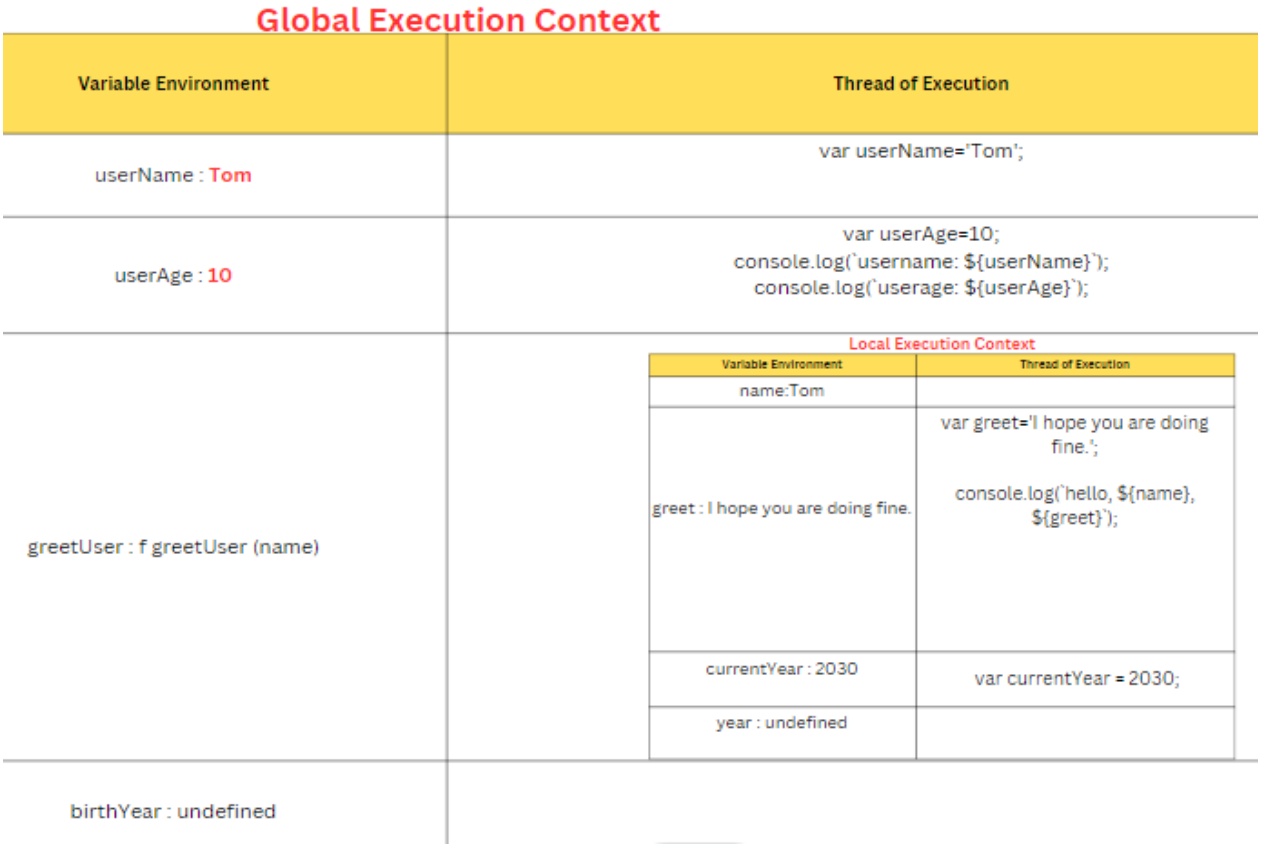
Variable Environment	Thread of Execution
userName : Tom	var userName='Tom';
userAge : 10	var userAge=10;
greetUser : f greetUser (name)	
birthYear : undefined	

b. When a function is invoked, a new Execution Context is built all together to carry out the same procedures for that function call/invoke. The JavaScript engine creates a new execution context (as shown below) for the `greetUser` function following the same procedure discussed above.

Global Execution Context

<table>
<tr><th>Variable Environment</th><th>Thread of Execution</th></tr>
<tr><td>userName : Tom</td><td>var userName='Tom';</td></tr>
<tr><td>userAge : 10</td><td>var userAge=10;
console.log(`username: ${userName}`);
console.log(`userage: ${userAge}`);</td></tr>
<tr><td>greetUser : f greetUser (name)</td><td>Local Execution Context
<table>
<tr><th>Variable Environment</th><th>Thread of Execution</th></tr>
<tr><td>name:Tom</td><td></td></tr>
<tr><td>greet : undefined</td><td></td></tr>
<tr><td>currentYear : undefined</td><td></td></tr>
<tr><td>year : undefined</td><td></td></tr>
</table></td></tr>
<tr><td>birthYear : undefined</td><td></td></tr>
</table>

- The further workflow of the Local Execution Context is described in the following figure, where the variables within the `greetUser` function are assigned values.



- Finally, The `greetUser` function returns a string containing the calculated birth year and the local execution context for the `greetUser` function is removed, and control is returned to the global execution context.

Execution Context in Dev Tools

To access and inspect the execution context in Google Chrome Dev Tools specifically, you can use the following steps:

1. Open Google Chrome browser.
2. Navigate to the webpage or web application where the JavaScript code is running.
3. Right-click on the page and select "**Inspect**" from the context menu. Alternatively, you can use the keyboard shortcut:
    a. **Windows:** Ctrl + Shift + I

b. **Mac:** Command + Option + I

Once the Dev Tools panel is open, follow these steps to access the execution context:

I. In the Dev Tools panel, locate and click on the "**Sources**" tab.
II. In the left-hand sidebar, expand the file or snippet containing the JavaScript code you want to inspect.
III. Set breakpoints at desired locations by clicking on the line number next to the code or using shortcut F9.
IV. Refresh the webpage or trigger the execution of the JavaScript code.
V. The execution will pause at the breakpoints, allowing you to inspect the execution context.
VI. Use the "**Scope**" section in the right-hand sidebar to explore the variables and their values within the execution context.
VII. You can also use the "**Call Stack**" section to see the sequence of function calls that led to the current execution context.

Hoisting in JS

Hoisting is a behavior in JavaScript where variable and function declarations are relocated to the beginning of their code blocks during the compilation phase, no matter where they are actually written in the code. This means that they can be accessed before they are declared.

- For example, the following code will work without any errors:

```
x = 5;
console.log(x);
var x;
```

This is because the `var x;` declaration is hoisted to the top of its code block, which in this case is the global block, and is executed before the `x = 5;` assignment.

- Function declarations are also hoisted in a similar way.
  For example:

```
greet();

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

This code will also work without errors because the `greet()` function declaration is hoisted to the top of its code block (in this case, the global block) before it is called.

- However, it's important to note that **only the declarations themselves are hoisted**, not their assignments.
  For example:

```
console.log(x);

var x = 5;
```

In this code, the `var x;` declaration is hoisted to the top of its code block, but the assignment `x = 5;` is not. So when `console.log(x);` is executed, `x` is still undefined.

Understanding Hoisting in JavaScript: Variables, Functions, and Declarations:

- When using the `var` keyword, variable declarations are hoisted to the top of their code blocks, whether it's the global scope or a function scope. This allows variables to be accessed before they are formally declared in the code.
- Similarly, function declarations are hoisted, enabling functions to be called before they appear in the code.
- However, it's important to note that function expressions, where functions are assigned to variables, are not hoisted. Only the function declarations themselves are hoisted.

- Furthermore, block-scoped variables declared with `let` and `const` do not experience hoisting behavior. They are not moved to the top of their code blocks and remain in their lexical position, ensuring that they are not accessible before their actual declaration in the code.

Generally, it's best practice to always declare variables and functions before using them to avoid unexpected behavior due to hoisting.

Call Stack

- In JavaScript, the call stack is a data structure that tracks the execution of functions during the runtime of a program.
- Every time a function is called, a new frame is created on top of the call stack to hold information about the function call, such as its arguments and local variables.
- The call stack operates on a "last in, first out" (LIFO) basis, meaning that the most recent function is the first to be completed and removed from the stack.
- The call stack is essential for understanding how JavaScript executes functions, and it plays a crucial role in identifying and debugging errors that occur during program execution.

```
var userName = 'Tom';
var userAge = 10;
console.log(`username: ${userName}`);
console.log(`userAge: ${userAge}`);
console.log(this);

greetUser(userName);

function greetUser(name) {
  console.log(`*****************************`);
  var greet = 'I hope you are doing fine.';
  console.log(`hello ${name}, ${greet}`);
  var currentYear = 2030;
  const year = birthYear(currentYear, userAge);
  console.log(`*****************************`);
  return `Your birth year is ${year}`;
}

function birthYear(year, age) {
  console.log('Calculating the birth year');
  return year - age;
}

var bYear = greetUser(userName);
console.log(bYear);

```

Here is how the call stack builds up for the given code:

1. The global execution context (here referred to as General Execution Context in the figures) is created in the call stack, as shown in figure-3(a).

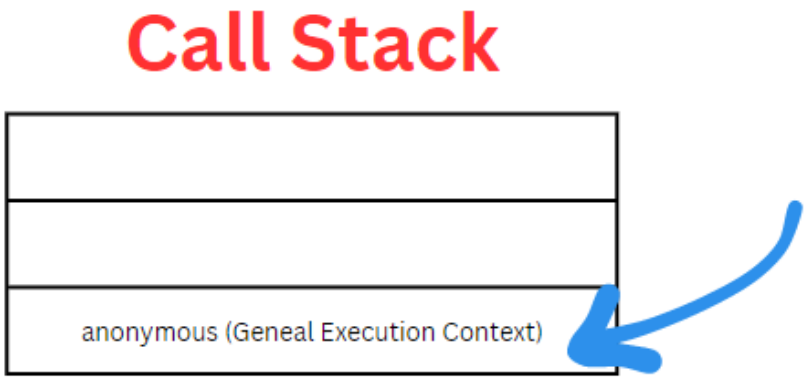


figure- 3(a)

2. The `greetUser()` function is called with `userName` as an argument, and a new execution context is created as shown in figure-3(b)

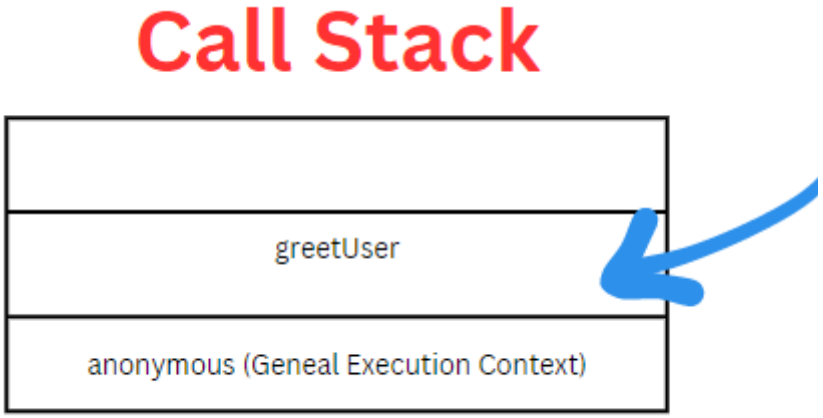


figure-3(b)

3. The `birthYear()` function is called with `currentYear` and userAge as arguments, and a new execution context is created on top of the current execution context for the `greetUser()` function as depicted in the figure-3(c)

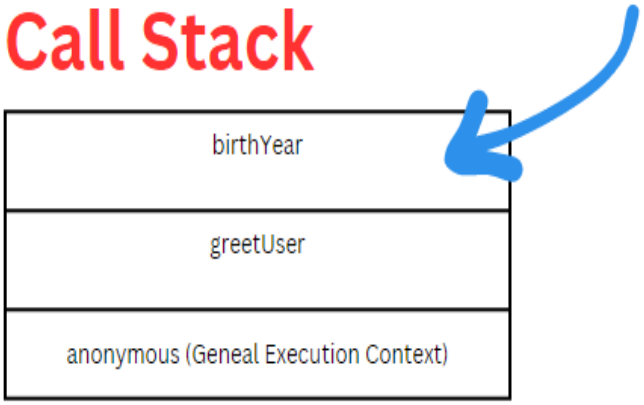


figure-3(c)

4. Once the operations of the `birthYear` function have been completed, it will be popped out of the stack as shown in figure-3(d).

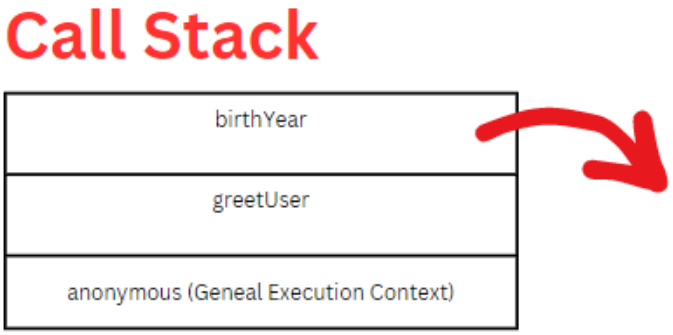


figure-3 (d)

5. Similarly, the `greetUser` function is popped out after its operation has been performed. This operation is depicted in figure-3(e).

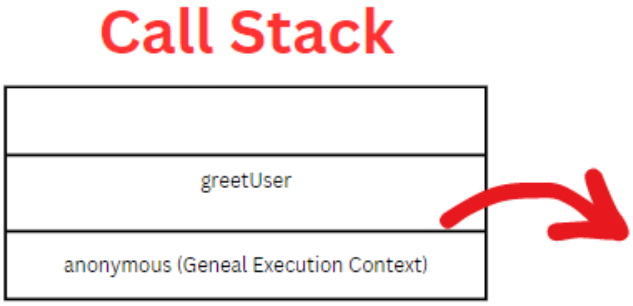


figure-3 (e)

6. Finally, in the end, the Global Execution Context (General Execution Context) is popped out of the Call Stack as depicted in figure-3(f).

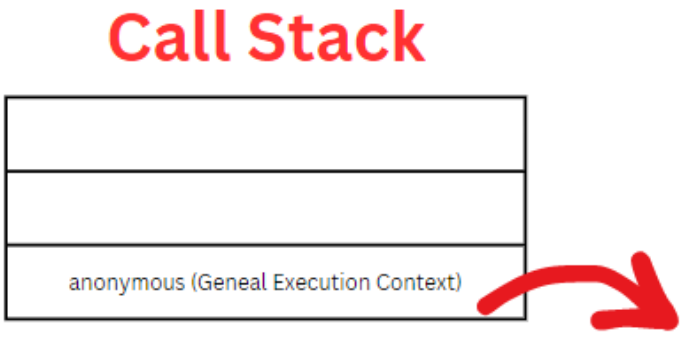


figure-3 (f)

Scope in JS

- In JavaScript, scope refers to the accessibility of variables and functions in your code. Understanding scope is crucial for writing clean and efficient code.
- There are three types of scopes: global scope, functional/local scope, and block scope.

1. Global Scope:

Variables declared outside of any function or block are in the global scope. They can be accessed from anywhere in the code, including inside functions or blocks.

For example:

```
// 1. Global scope
var globalVar = 'I am in the global scope';

function position() {
  console.log(globalVar); // Output: I am in the global scope
}
position();
```

2. Functional/Local Scope:

Variables declared inside a function or block are in the functional/local scope. They can only be accessed from within that function or block.

For example:

```
// 2. Local Scope
function position() {
  var localVar = 'I am in the functional scope';

  console.log(localVar); // Output: I am in the functional scope
}

position();
console.log(localVar); // Throws an error: localVar is not defined
```

3. Block Scope:

Variables declared using `let` or `const` inside a block (e.g., inside a `for` loop or `if` statement) are in the block scope. They can only be accessed from within that block.

For example:

```
// 3. block scope
function foo() {
  if (true) {
    let blockVar = 'I am in the block scope';
    console.log(blockVar); // Output: I am in the block scope
  }

  console.log(blockVar); // Throws an error: blockVar is not defined
}
```

Difference between `let`, `var`, and `const`

In JavaScript, `let`, `var`, and `const` are used to declare variables, but they differ in terms of scoping and mutability.

1. **let:**

- `let` is used to declare block-scoped variables.
- It allows you to declare a variable inside a block and use it only within that block.
- `let` variables can be re-assigned but not re-declared within the same scope.
- `let` is more strict than `var` and helps avoid bugs caused by variable hoisting.

2. **var:**

- `var` is used to declare function-scoped variables.
- It can be declared and re-declared multiple times within the same scope.
- `var` declarations are hoisted to the top of the function or global scope, which means that they are processed before any code is executed.
- Because of hoisting, `var` can lead to bugs in code if not used properly.

- However, it's important to note that `var` has two scopes: global scope and functional scope. Variables declared with `var` in the global scope are accessible throughout the entire program, while variables declared with `var` inside a function are only accessible within that function.

3. **const:**

- `const` is used to declare read-only variables.
- It can be used to declare a variable once and cannot be re-assigned within the same scope.
- `const` variables are also block-scoped.
- `const` is useful for declaring constants that should not be changed throughout the program.

Scope Chaining

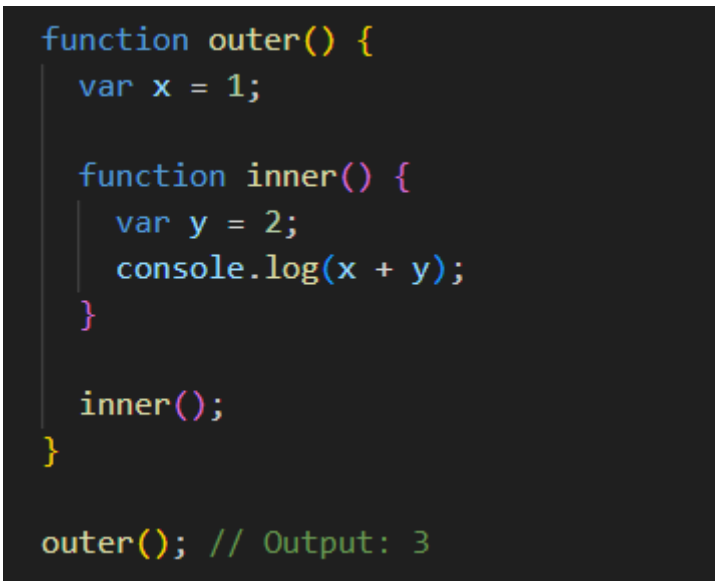
Lexical environment

- Lexical environment is a fundamental concept in JavaScript that refers to the specific context in which code is executed.
- It encompasses variables, functions, and objects that are accessible and in scope at a particular point during the execution of code.
- A fresh lexical environment is generated whenever a function is called or invoked in JavaScript. This lexical environment encompasses all the variables and functions that are in scope and can be accessed within that particular function call.

Scope chaining

- Scope chaining, also known as lexical scoping, is a mechanism in JavaScript that allows a function to access variables from its outer (enclosing) lexical environment as well as from the global scope.
- This means that functions can access variables defined in their parent functions, grandparent functions, and so on, all the way up to the global scope.

Here's an example to illustrate how scope chaining works in JavaScript:



```
function outer() {
  var x = 1;

  function inner() {
    var y = 2;
    console.log(x + y);
  }

  inner();
}

outer(); // Output: 3
```

- In this example, `inner()` is nested inside `outer()`, so it has access to `x`, which is defined in the lexical environment of `outer()`.
- When `outer()` is called, a new lexical environment is created that contains the variable `x`, and when `inner()` is called, a new lexical environment is created that contains both `x` and `y`.
- Therefore, `inner()` can access `x` from its outer lexical environment and `y` from its own lexical environment, and the result of x + y is 3.

It's important to note that **scope chaining only works in one direction, from inner to outer, and not the other way around.** That means variables defined in an inner scope cannot be accessed from an outer scope.

Undefined vs Not Defined

In JavaScript, undefined and not-defined are two different concepts that often confuse developers who are new to the language. Understanding the difference between them is crucial in writing bug-free and robust code.

- `undefined` is a value that indicates the absence of a value. It is a primitive type and is assigned to a variable when it is declared but not initialized with a value or when a function does not return anything. For example:

```
let x; // x is declared but not initialized, so its value is undefined
console.log(x); // output: undefined

function func() {
  // no return statement, so the function returns undefined
}
console.log(func()); // output: undefined
```

In this example, `x` and `func()` are defined variables but have not been initialized with a value. Therefore, their value is undefined.

- **not-defined** is a state of a variable that has not been declared at all. If you try to access such a variable, JavaScript throws a `ReferenceError` exception. For example:

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

In this example, `y` is not defined anywhere in the code. Therefore, JavaScript throws a `ReferenceError` exception.

Strict mode

- Strict mode in JavaScript is a feature introduced in ECMAScript 5 (ES5) that allows developers to opt into a stricter set of rules and best practices for writing JavaScript code.
- When strict mode is enabled, the JavaScript interpreter enforces a more stringent set of rules, catches common mistakes, and disables certain silent errors.

Here are some key points about strict mode:

1. **Activation:** Strict mode can be enabled for an entire script or a specific function. To enable strict mode for an entire script, add the following line at the beginning of your script:

```
'use strict';
```

To enable strict mode for a specific function, add the same line at the beginning of the function's body.

2. **Benefits:** Strict mode helps developers write more reliable and maintainable code by addressing some of the quirks and pitfalls of JavaScript. It prevents the use of undeclared variables, eliminates the automatic creation of global variables, prohibits the use of duplicate function parameter names, and disallows certain unsafe language features.
3. **Variable Declarations:** In strict mode, all variables must be explicitly declared with var, let, or const keywords. Implicitly creating global variables by omitting the var keyword is disallowed. This catches accidental global variable declarations, reducing the risk of naming conflicts.
4. **Undeclared Variables:** Accessing undeclared variables in strict mode throws a ReferenceError. In non-strict mode, accessing an undeclared variable creates an implicit global variable, which can lead to subtle bugs and unintended consequences.

Enabling strict mode is highly recommended for modern JavaScript development as it promotes better coding practices and helps catch potential errors early on. By adhering to strict mode guidelines, developers can write more robust and predictable code.

Temporal Dead Zone

- In JavaScript, the Temporal Dead Zone (TDZ) is a behavior that occurs during the variable creation phase of the execution context.
- It refers to the period between the creation of a variable and its initialization, during which the variable cannot be accessed.
- Attempting to access a variable during the TDZ will result in a `ReferenceError`.

Let's take a look at an example to illustrate the concept of TDZ:

```
console.log(a); // ReferenceError: a is not defined
let a = 10;
```

In this example, we try to access the variable `'a'` before it is declared. This results in a `ReferenceError` because the variable is in the TDZ and cannot be accessed until it has been declared.

- It occurs only for variables declared with `let` and `const` keywords and not for variables declared with `var`. This is because `var` is function-scoped and hoisted to the top of the function, making it accessible throughout the function, even before it is declared.

  ```
  console.log(b); // undefined
  var b = 20;
  ```

  In this example, we try to access the variable `'b'` before it is declared, but we get `undefined` instead of a `ReferenceError`. This is because `var` variables are hoisted to the top of their function and initialized with undefined by default.

- To avoid TDZ errors, it's best practice to declare variables at the beginning of their scope, before they are accessed. This will ensure that they are not in the TDZ and can be accessed without throwing a `ReferenceError`.

Closures in JS

- In JavaScript, a closure is created when a function accesses variables outside of its immediate lexical scope.
- The closure retains a reference to the environment in which it was created, allowing the function to access and manipulate variables in that environment, even after the outer function has returned.
    a. Here's an example of a closure in JavaScript:

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

  function inner() {
    count++;
    console.log(count);
  }

  return inner;
}

const counter = outer();
counter(); // 1
counter(); // 2
counter(); // 3
```

- In this example, the `outer` function creates a variable `count` and a nested function `inner` that increments the `count` variable and logs it to the console. The `outer` function then returns the `inner` function.
- We then assign the returned `inner` function to the counter variable. We can now call the counter function multiple times, and each time it will increment and log the `count` variable.
- The important thing to note here is that the `inner` function retains a reference to the `count` variable in the environment where it was created (inside the `outer` function). This is possible because JavaScript functions are **closures**, and they **maintain a reference to the environment in which they were created.**

b. Here's another example to illustrate the concept of closures:

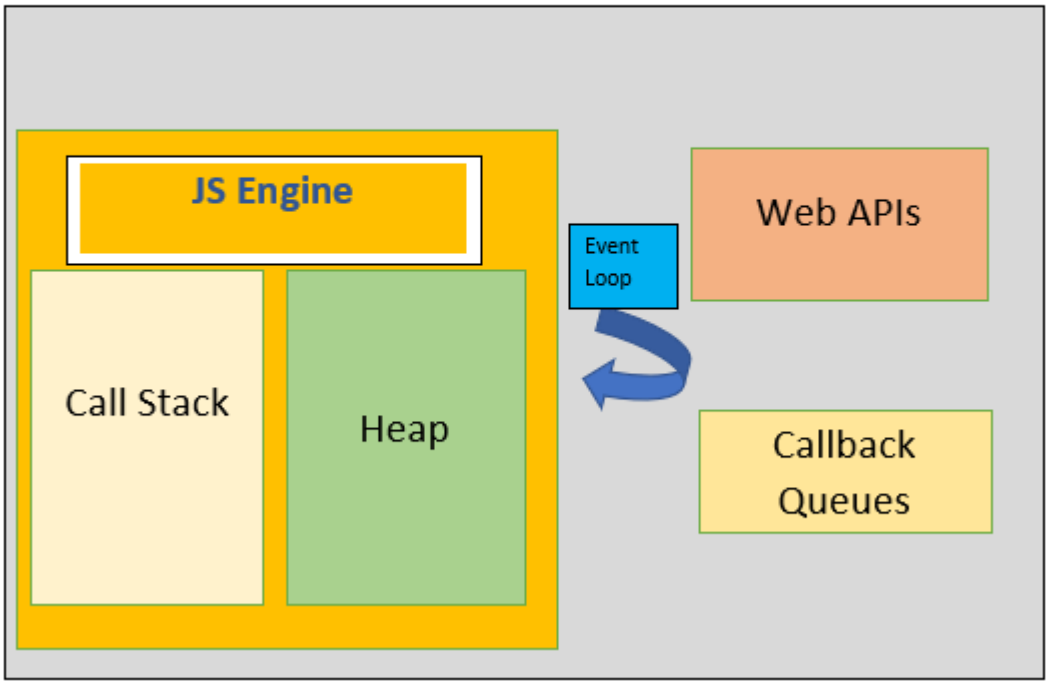
```
function createCounter() {
  let count = 0;

  function increment() {
    count++;
    console.log(count);
  }

  function decrement() {
    count--;
    console.log(count);
  }

  return { increment, decrement };
}

const counter = createCounter();
counter.increment(); // 1
counter.increment(); // 2
counter.decrement(); // 1
```

- In this example, the `createCounter` function returns an object with two methods: `increment` and `decrement`. Both methods have access to the `count` variable, which is created in the `createCounter` function's environment.
- We then assign the returned object to the `counter` variable, and we can call the `increment` and `decrement` methods on it. Each time we call a method, it will update and log the `count` variable.

JavaScript Runtime Environment

- JavaScript runtime environment (JRE) is a term used to describe the environment in which JavaScript code is executed. It includes the JavaScript engine, call stack, heap, web APIs, and callback queues.

JavaScript Runtime Environment (JRE)



- The JavaScript engine is the component responsible for interpreting and executing JavaScript code. Popular engines include Google's V8 engine used in Chrome and Node.js, Mozilla's SpiderMonkey, and Apple's JavaScriptCore.
- The heap is the memory space the JavaScript engine uses to store objects and values created by the code.
- Web APIs are interfaces provided by the browser environment that allows JavaScript to interact with browser features. These APIs include the `alert()`, `confirm()`, `prompt()`, and `setTimeout()` and `setInterval()` methods.
- Callback queues and the event loop are used to manage asynchronous code execution in JavaScript. We will be covering it in the upcoming lectures.

The JavaScript runtime environment is the foundation for executing JavaScript code in the browser environment. It provides a set of tools and interfaces that allow developers to create complex and interactive web applications.

Summarizing it

Let's summarize what we have learned in this Lecture:

- Execution Context in JS
- Hoisting in JS
- Call Stack
- Scope and Scope Chaining
- Strict modes and Temporal Dead Zone
- Closures in JS
- JavaScript Runtime Environment

References

- Closures in JS: **Link**

Functions in JS

Function Declaration:

- In JavaScript, a function declaration is a way to define a function using the "`function`" keyword followed by the function name and a block of code.
- Function declarations are hoisted, meaning they are moved to the top of the current scope during the compilation phase.
- Having thoroughly covered the fundamental functionality, uses, and features of functions in Lecture-3, we will now delve into an advanced exploration, focusing on different types of functions.
- Understanding the different types of functions in JavaScript empowers developers to choose the appropriate approach for various programming scenarios, leading to cleaner and more efficient code.

In the following section, we will provide a comprehensive discussion on the various types of functions that exist in JavaScript.

Function Expression

- A function expression involves assigning a function to a variable or a property of an object.
- Function expressions are not hoisted and must be defined before they are called.
- They can be anonymous (without a name) or named.
- Function expressions are often used to create functions on the fly, as arguments to other functions, or to encapsulate code within a specific scope.
- Examples of function expressions include anonymous functions, arrow functions, and immediately invoked function expressions (IIFE).

Anonymous Function:

- An anonymous function, also known as a nameless function or function expression, is a function that is defined without a specified name. Instead, it is assigned to a variable or used as an argument directly within the code.
- Anonymous functions are useful when you need to define a small function for a specific task without the need for a named function declaration.

Example:

```
// Using an anonymous function to calculate the square of a number

const square = function(number) {

  return number * number;

};

console.log(square(5)); // Output: 25
```

Immediately Invoked Function Expression (IIFE):

- An Immediately Invoked Function Expression (IIFE) is a JavaScript design pattern that allows you to execute a function immediately after its declaration. It provides a way to create a private scope for variables and functions, preventing them from polluting the global scope.
- **Syntax:** The basic syntax of an IIFE is as follows:

```
(function() {

  // Function body

})();
```

The function is enclosed in parentheses to indicate that it is a function expression, followed by an additional set of parentheses to invoke it immediately.

- Benefits of using IIFE:

a. Encapsulation: IIFEs create a separate scope for variables and functions, avoiding global scope pollution. This helps prevent naming conflicts and provides a way to encapsulate code and data.

b. Privacy: Variables and functions declared inside an IIFE are not accessible outside of the function, creating privacy and protecting them from external interference.

c. Modularization: IIFEs can be used to create self-contained modules, allowing you to define and expose only specific properties or methods to the outer world.

d. Isolation: IIFEs provide a level of isolation, allowing you to define temporary variables and execute code without affecting the global state or interfering with other parts of the application.

Example:

```
(function() {
  var message = 'Hello, world!';
  function displayMessage() {
    console.log(message);
  }
  displayMessage();
})();
```

Arrow Function:

- An arrow function is a concise syntax introduced in ES6 for defining functions. It offers a more compact and expressive way to write functions, particularly for one-liners.
- It's important to note that arrow functions have a lexically bound "`this`" context, which differs from regular functions. However, since we haven't covered "`this`" in the earlier lectures, we will explore it in detail shortly.

Here are some important points to understand about arrow functions:

1. **Syntax:** The basic syntax of an arrow function is as follows:

```
const functionName = (parameters) => {
  // Function body
};
```

Arrow functions are defined using the => (fat arrow) notation, followed by the function body enclosed in curly braces {}.

2. **Implicit return:** Arrow functions have an implicit return feature. The return statement is not required if the function body consists of a single expression. The result of the expression is automatically returned. For example:

```
const double = (number) => number * 2;

console.log(double(5)); // Output: 10
```

3. **Handling parameters:** Arrow functions can have zero or more parameters. When there is only one parameter, the parentheses around the parameter can be omitted. For multiple parameters, parentheses are required. For example

```
const greet = name => `Hello, ${name}!`;

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

const addNumbers = (x, y) => x + y;

console.log(addNumbers(2, 3)); // Output: 5
```

Callback Function

In JavaScript, a callback function is a function that is passed as an argument to another function and is executed later in response to an event or an asynchronous operation. Callback functions provide a way to ensure that certain code is executed only when a specific task is completed or an event occurs.

Here are some important points to understand about callback functions:

1. **Asynchronous programming:** Callback functions are commonly used in asynchronous programming to handle tasks that may take some time to complete, such as reading data from a file, making an HTTP request, or processing a database query.
2. **Event handling:** Callback functions are used to respond to events in web development, such as button clicks, form submissions, or user interactions. The callback function is triggered when the event occurs.
3. **Execution order:** In JavaScript, functions are first-class objects, meaning they can be assigned to variables, passed as arguments, or returned from other functions. A function is not executed immediately when it is passed as a callback. Instead, it is stored and executed later when the conditions are met.

Example:

```
function greet(name, callback) {
  const message = "Hello, " + name + "!";
  callback(message);
}
function displayMessage(message) {
  console.log(message);
}
greet("John", displayMessage); // Output: Hello, John!
```

In this example, the `greet` function takes a `name` and a `callback` function callback as arguments. It constructs a greeting `message` using the `name` parameter and then invokes the callback function with the message.

Pure & Impure Function

Impure Function

- A pure function is a function that always produces the same output for the same inputs and does not cause any side effects.
- It relies only on its arguments and does not modify any external state. Pure functions are predictable, easier to test, and promote code reusability.
- Pure functions are a fundamental concept in functional programming and play a crucial role in building reliable and maintainable code. By following the principles of pure functions, you can minimize side effects, improve code predictability, and make your programs easier to understand and reason about.

Example:

```
// Pure function example
function add(a, b) {
  return a + b;
}
const result = add(3, 4); // Output: 7
```

Impure Function

- An impure function is a function that can produce different outputs for the same inputs or cause side effects by modifying external state.
- It may rely on variables outside its scope or perform actions like modifying global variables or making network requests. Impure functions can be harder to reason about and test.

Example:

```
// Impure function example
let counter = 0;

function incrementCounter() {
  counter++;
  console.log("Counter incremented.");
}
incrementCounter(); // Output: Counter incremented.
```

```
console.log(counter); // Output: 1
```

In this example, the `incrementCounter` function is an impure function that modifies the external variable `counter` and logs a message to the console. Each time `incrementCounter` is called, the `counter` variable is incremented, and the message is logged.

Higher-Order Function

In JavaScript, a higher-order function is a function that can accept other functions as arguments or return a function as its result. Higher-order functions provide a powerful and flexible way to work with functions and enable functional programming paradigms.

Here are some important points to understand about higher-order functions:

- **Function as First-Class Citizens:** In JavaScript, functions are treated as first-class citizens, which means they can be assigned to variables, passed as arguments, and returned from other functions. This allows higher-order functions to be defined and used effectively.
- **Accepting Functions as Arguments:** Higher-order functions can accept other functions as arguments. These functions are often referred to as callback functions or function parameters. The higher-order function can then invoke the callback function at a specific time or condition.
- **Returning Functions:** Higher-order functions can also return functions as their result. This is particularly useful for creating specialized functions or building function factories.
- **Abstraction and Code Reusability:** Higher-order functions promote abstraction and code reusability by encapsulating common functionality in a higher-level function. This can reduce code duplication and make the codebase more modular and maintainable.

Higher Order methods in Array

- Common examples of higher-order functions in JavaScript include **`map()`**, **`filter()`**, **`reduce()`**, and **`forEach()`**, which accept a callback function as an argument. These functions encapsulate common operations on arrays and can be customized by providing different callback functions.

1. `map()`:

- The **`map()`** method is a higher-order function that operates on arrays in JavaScript. It creates a new array by applying a provided callback function to each original array element.
- The callback function is executed for every element, and the return value of each function call is added to the new array.
- The resulting array has the same length as the original array, with each element transformed based on the logic defined in the callback function.

Example:

```
// map() higher-order function example

const names = ["Alice", "Bob", "Charlie", "Dave"];

const nameLengths = names.map(function (name) {

  return name.length;

});

console.log(nameLengths); // Output: [5, 3, 7, 4]
```

In this example, the `map()` function is used to create a new array `nameLengths` that contains the lengths of each name in the `names` array.

2. `filter()`:

- The **`filter()`** method is another higher-order function that works with arrays. It creates a new array containing elements from the original array that satisfy a specified condition.

- The callback function provided to **`filter()`** is executed for each element, and if the return value is true, the element is included in the resulting array. If the return value is false, the element is excluded.

Example:

```
// filter() higher-order function example

const words = ["apple", "banana", "grape", "orange", "kiwi"];

const filteredWords = words.filter(function (word) {

  return word.length > 5;

});

console.log(filteredWords); // Output: ["banana", "orange"]
```

In this example, the `filter()` function is used to create a new array `filteredWords` that contains only the words with a length greater than 5 characters.

3. `reduce()`:

- The **`reduce()`** method is a higher-order function that allows for the accumulation of a single value by iterating over the elements of an array.
- It executes a reducer function on each element and maintains an **accumulator** that stores the accumulated value.
- The reducer function takes two arguments: the **accumulator** and the **current element**. The accumulator is updated based on the logic defined in the reducer function, and the final accumulated value is returned.

Example:

```
// reduce() higher-order function example

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

```
const product = numbers.reduce(function (accumulator, currentValue) {
  return accumulator * currentValue;
}, 1);
console.log(product); // Output: 120
```

In this example, the reduce() function is used to calculate the product of all the numbers in the numbers array.

4. find() and findIndex() functions

The find() and findIndex() methods are array methods introduced in ECMAScript 2015 (ES6) that allow you to search for elements within an array based on a specified condition. Here's an overview of each method and examples of their usage:

find():

- The `find()` method returns the first element in an array that satisfies the provided testing function.
- It executes the callback function once for each element in the array until it finds a match, and then stops the iteration.
- If a matching element is found, it is returned; otherwise, `undefined` is returned.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const foundNumber = numbers.find(function (number) {
  return number > 3;
});
console.log(foundNumber); // Output: 4
```

In this example, the `find()` method is used to search for the first element in the numbers array that is greater than 3. The callback function takes a `number` as an argument and returns `true` if the condition is satisfied. The `find()` method stops iterating as soon as it finds the first match, and returns that element (4 in this case).

findIndex():

- The `findIndex()` method returns the index of the first element in an array that satisfies the provided testing function.
- It executes the callback function once for each element in the array until it finds a match, and then stops the iteration.
- If a matching element is found, its index is returned; otherwise, `` `-1` `` is returned.

Example:

```
const fruits = ["apple", "banana", "grape", "orange"];

const index = fruits.findIndex(function (fruit) {

  return fruit === "grape";

});

console.log(index); // Output: 2
```

In this example, the `findIndex()` method is used to search for the index of the first occurrence of the string "`grape`" in the `fruits` array. The callback function takes a `fruit` as an argument and returns `true` if the condition is satisfied. The `findIndex()` method stops iterating as soon as it finds the first match, and returns the index of that element (`` `2` `` in this case).

Other functions in JavaScript

- There are various functions in-built functions in JavaScript that make it easy to use.
- The array functions given below are powerful tools in JavaScript for manipulating and working with arrays. They provide convenient ways to iterate, search, and modify arrays based on specific conditions or operations.

1. every:

The `every` function tests whether all elements in an array pass a specific condition defined by a provided function. It returns a boolean value indicating if all elements satisfy the condition.

Example:

```
const numbers = [2, 4, 6, 8];

const allEven = numbers.every(function(number) {

  return number % 2 === 0;

});

console.log(allEven); // Outputs: true
```

2. fill:

The `fill` function changes all elements in an array with a static value, starting from a specified start index and ending at a specified end index.

Example:

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

  numbers.fill(0, 2, 4);

  console.log(numbers); // Outputs: [1, 2, 0, 0, 5]
```

3. findLast:

The `findLast` function returns the last element in an array that satisfies a given condition defined by a provided function. It searches the array from right to left.

Example:

```
const numbers = [10, 20, 30, 40, 50];

const foundNumber = numbers.findLast(function(number) {

  return number > 30;

});

console.log(foundNumber); // Outputs: 40
```

4. findLastIndex:

The `findLastIndex` function returns the index of the last element in an array that satisfies a given condition defined by a provided function. It searches the array from right to left.

Example:

```
const numbers = [10, 20, 30, 40, 50];

const foundIndex = numbers.findLastIndex(function(number) {

  return number > 30;

});

console.log(foundIndex); // Outputs: 3
```

5. forEach:

The `forEach` function executes a provided function once for each element in an array. It is commonly used to perform an operation on each item of the array without returning a new array.

Example:

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

numbers.forEach(function(number) {

  console.log(number * 2);

});
```

Function currying

- Function currying is a technique in JavaScript that involves transforming a function with multiple arguments into a sequence of functions, each taking a single argument.
- It allows you to create new functions by partially applying the original function with some arguments, resulting in a more specialized and reusable function.
- Function currying is a powerful technique in JavaScript that enables code reuse, modularity, and composability. It helps create more flexible and specialized functions by partially applying arguments, leading to cleaner and more expressive code.

Example of Function Currying:

```
function multiply(a, b) {

  return a * b;

}

function curriedMultiply(a) {

  return function (b) {

    return multiply(a, b);

  }

}
```

```
const multiplyByTwo = curriedMultiply(2);

console.log(multiplyByTwo(5)); // Output: 10

const multiplyByThree = curriedMultiply(3);

console.log(multiplyByThree(5)); // Output: 15
```

- In this example, the `multiply()` function is a regular function that takes two arguments `a` and `b` and returns their product.
- We define a separate function called `curriedMultiply()` which takes the first argument `a`. Inside this function, we return another function that takes the second argument `b` and calls the `multiply()` function with the provided `a` and `b` arguments.
- This approach achieves function currying by using a separate function (`curriedMultiply()`) to generate the curried functions based on the original `multiply()` function.

this keyword

The `this` keyword in JavaScript refers to the object that is currently executing the code or the object that a function is a method of. It is a special identifier that allows you to access properties and methods within the context of the object.Some of its usage are mentioned below:

1. **Implicit Binding:**

- In most cases, the this keyword is implicitly bound to the object that is left of the dot when invoking a method.
- The value of this is determined dynamically at runtime based on how a function is called.
- It allows methods to access the properties and other methods of the object they belong to.

Example:

```
const person = {
  name: "John",
  age: 25,
  greet: function() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age}
years old.`);
  }
};
person.greet(); // Output: Hello, my name is John and I'm 25 years
old.
```

In this example, the `greet()` method of the `person` object uses the `this` keyword to access the `name` and `age` properties of the same object. The `this` keyword is implicitly bound to `person` since person is left of the dot when invoking the `greet()` method.

2. **Explicit Binding:**

- In the subsequent lectures, we will delve into the concepts of `call()`, `apply()`, and `bind()`. These powerful functions provide explicit control over the binding of the this keyword to a specific object.
- By covering these functions, you will acquire the knowledge and techniques required to precisely manage the association between `` `this` `` and an object.
- This allows you to control the value of this within a function.

3. **Global Context:**

- In the global scope or when this is not inside any object or function, it refers to the global object (window in browsers, global in Node.js).
- However, in strict mode ("use strict"), the value of this is undefined in the global context.

Example:

```
console.log(this); // Output: Window (in browsers) or Global
```

The behavior of the this keyword can vary depending on how a function is called or the context in which it is used. Understanding the different binding mechanisms of this is essential for properly accessing and manipulating object properties and methods in JavaScript.

Summarizing it

In this set of notes, we covered various concepts related to functions in JavaScript.

- We started with function declaration and function expression, understanding their differences and usage scenarios.
- We explored anonymous functions, which are functions without a name, and saw how they can be used in different contexts
- We then learned about IIFE (Immediately Invoked Function Expression), a technique to create self-contained and private scopes.
- Next, we delved into arrow functions, which provide a concise syntax
- Moving on, we explored callback functions, which are functions passed as arguments to other functions.
- Furthermore, we explored higher-order functions, which are functions that operate on other functions. We covered higher-order functions like `map()`, `reduce()`, `filter()`, `find()`, `findIndex()`, and some other functions which allow for powerful array transformations and operations.
- We also briefly touched on function currying, a technique of creating new functions by partially applying arguments to an existing function and at the end we got our hands on `` `this` `` in JavaScript.

References

- `this` keyword in JavaScript: Link

OOP in JS

Objects

Brief Recap:

- In Lecture 3, we briefly introduced objects and discussed object creation using literals and accessing objects using dot and bracket notation.
- In this session, we will explore objects in more detail, focusing on their properties, methods, and additional concepts.

Introduction to Objects:

- In JavaScript, objects are composite data types used to represent real-world entities or concepts.
- Objects are containers that store related data and functions together as properties and methods.
- Objects can be instantiated using two different approaches: object literals or constructor functions. In the following sections of the notes, we will delve deeper into the concept of constructor functions and discuss their usage in creating and initializing objects.

Object Creation Using Literals:

- Object literals provide a convenient way to create and initialize objects.
- Object literals consist of key-value pairs enclosed in curly braces {}.
- Each key-value pair represents a property, where the key is the property name and the value is the property value.
- Properties can hold various data types, including strings, numbers, booleans, arrays, or even other objects.

Example:

```
// Creating an object using object literal
var person = {
```

```
    name: "Peter",
    age: 25,
    address: "123 Street",
    hobbies: ["reading", "playing guitar"]
  };
```

- In ES5 object literals, methods are defined similarly to other properties. The function expressions are used to assign values to these properties.

For instance:

```
  var obj = {
    calculateArea: function (radius) {
        return Math.PI * radius * radius;
    },
    displayMessage: function (name) {
        console.log("Hello, " + name + "!");
    }
};
```

In the code above, the `obj` object has two methods: `calculateArea` and `displayMessage`. The `calculateArea` method calculates the area of a circle based on the given radius, while the `displayMessage` method logs a personalized greeting to the console.

- In ES6, a new syntax called method definitions was introduced to make method creation more concise:

```
const obj = {
    calculateArea(radius) {
        return Math.PI * radius * radius;
    },
    displayMessage(name) {
        console.log("Hello, " + name + "!");
    }
```

```
};
```

With the ES6 method definitions, we can directly define methods within the object literal without using the `function` keyword. The functionality remains the same as in the previous example.

Accessing Objects Using Dot Notation:

- Once an object is created, we can access its properties using dot notation.
- Dot notation involves using the dot operator **(.)** followed by the property name to access the corresponding value.

Example:

```
// Accessing properties using dot notation
console.log(person.name);        // Output: Peter
console.log(person.age);         // Output: 25
console.log(person.address);     // Output: 123 Street
console.log(person.hobbies[0]);  // Output: reading
```

Accessing Objects Using Bracket Notation:

- In addition to dot notation, JavaScript also allows accessing object properties using bracket notation.
- Bracket notation involves using square brackets **[]** with the property name inside.
- Bracket notation is useful when the property name contains special characters, spaces, or is determined dynamically at runtime.

Example:

```
// Accessing properties using bracket notation
console.log(person["name"]);        // Output: Peter
console.log(person["age"]);         // Output: 25
console.log(person["address"]);     // Output: 123 Street
console.log(person["hobbies"][0]);  // Output: reading
```

Comparison between Dot Notation and Bracket Notation:

- Both dot notation and bracket notation achieve the same result of accessing object properties.
- Dot notation is commonly used when the property name is known and valid as an identifier.
- Bracket notation is more versatile as it allows using variables or property names with special characters or spaces.
- Bracket notation is also used when the property name is dynamically determined.

It's important to note that dot notation and bracket notation are interchangeable, but bracket notation provides more flexibility in certain situations.

`this` keyword

- In lecture 5, we discussed the keyword `'this'` in JavaScript and its general purpose. Now, let's delve deeper into the concept of `'this'` specifically in the context of JavaScript objects.
- In this section, we will provide a comprehensive explanation of `'this'` in object-oriented programming, highlighting its significance and providing clear examples.

I. Recap: Understanding `'this'` in JavaScript

- Previously, we learned that `'this'` refers to the current execution context in JavaScript.
- It allows us to access and manipulate object properties and methods within the scope of an object.

II. `'this'` in Object Methods:

- When used within an object method, `'this'` refers to the object itself.
- This allows seamless access to other properties and methods of the same object.

Let's illustrate this concept with an example:

```
const person = {
  name: 'Peter',
  age: 30,
  greet: function() {
    console.log('Hello, my name is ' + this.name + ' and I am ' +
this.age + ' years old.');
  }
};

person.greet(); // Output: Hello, my name is Peter and I am 30 years
old.
```

Here, `'this'` within the `'greet'` function allows for dynamic access to object properties, ensuring that the function can be applied to different objects while maintaining the correct context.

Constructor Functions

- When creating objects in JavaScript, object literals serve as a convenient and straightforward approach.
- However, there are scenarios where object literals may not be suitable or efficient for object creation. This leads us to the introduction of constructor functions, which provide a powerful alternative for creating and initializing objects.

So, why do we require the new concept of constructor functions?

- Object literals have their limitations when creating multiple objects with similar properties and behaviors. Each object created with an object literal has its own properties and methods, resulting in duplicated code and inefficiency.
- Constructor functions address this limitation by providing a blueprint for creating multiple objects with shared properties and methods.

- By defining a constructor function, we can instantiate new objects that inherit properties and methods from the constructor's prototype.
- This approach promotes code reusability, reduces redundancy, and enables efficient object creation.

Syntax:

1. Constructor functions are defined using function declarations or expressions.
2. By convention, the function name starts with a capital letter to indicate that it is a constructor.

Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    console.log('Hello, my name is ' + this.name + ' and I am ' +
this.age + ' years old.');
  };
}
```

The Role of `'this'` Keyword in Constructor Functions:

A. Context and Ownership:

1. Inside the constructor function, the `'this'` keyword refers to the object being created.
2. `'this'` allows access to and assignment of object properties within the function.

B. Assigning Properties:

1. Using `'this.propertyName'`, properties can be assigned to the object being created.
2. In the example above, `'this.name'` and `'this.age'` assign the `'name'` and `'age'` properties to the created object.

C. Defining Shared Methods:

1. Constructor functions can define methods that will be shared among all instances of the object.
2. The `'greet'` method in the example is assigned to `'this.greet'`, making it accessible to all instances.

Invoking Constructor Functions with the 'new' Keyword:

1. To create an object instance, the constructor function is invoked using the **`'new'`** keyword.
2. The **`'new'`** keyword creates a new object and binds '`this`' to that object inside the constructor function.

Example:

```
const person1 = new Person('Tommy', 30);
const person2 = new Person('Arthur', 25);

person1.greet(); // Output: Hello, my name is Tommy and I am 30 years old.
person2.greet(); // Output: Hello, my name is Arthur and I am 25 years old.
```

Understanding constructor functions allows for efficient code organization and the creation of reusable object templates.

Prototype

Object Prototype in JavaScript using Constructor Functions

- In JavaScript, constructor functions are used to create objects. When you define a constructor function, you can add properties and methods to its prototype object. Any objects created with that constructor will then inherit those properties and methods.

- The `**prototype**` property of a constructor function is not the same thing as an instance's prototype - rather, it's used to add properties and methods that will be inherited by instances created from the constructor.

Accessing an Object's Prototype

The property of an object that points to its prototype is not called "prototype". Its name is not standard across browsers; however,proto can be used in practice for most browsers. The recommended way to access an object's prototype is through the `**Object.getPrototypeOf()**` method.

Example:

```
function Person(name) {
  this.name = name;
}
var bob = new Person('Bob');
console.log(Object.getPrototypeOf(bob)); // returns Person {}
```

Prototypal Inheritance

Prototypal inheritance refers to how objects inherit from their prototypes. When you look up a property on an object (e.g., `bob.name`), if it doesn't exist on the object itself, JavaScript looks at its prototype chain until it finds what it needs.

Example:

```
function Animal() {}
Animal.prototype.move = function() { console.log('moving') };

function Dog() {}
Dog.prototype.bark = function() { console.log('woof!') };
Dog.prototype.__proto__ = Animal.prototype;

const fido = new Dog();
fido.move(); // logs 'moving'
fido.bark(); // logs 'woof!'
```

- Here we have set up prototypal inheritance between our `Animal` and `Dog` constructors so that dogs inherit both their own unique behaviors (`bark`) and those shared with animals (`move`).

- This works because when we call `.move()` on our instance of `Dog`, which doesn't actually have its own `.move()` method defined, JavaScript follows the chain up through `Dog`'s parent (`Animal`) until it finds one that does.

Prototype in Array

Arrays in JavaScript are also objects, and therefore have a prototype. The **`Array.prototype`** object contains properties and methods that are available to all arrays created from the **`Array`** constructor function.

Example:

```
const myArr = [1, 2, 3];
console.log(myArr.map(x => x * 2)); // returns [2,4,6]
```

Here we've used the **`.map()`** method of our array instance to return a new array with each element doubled. This method is actually defined on the **`Array.prototype`**, so when we call it on our specific instance (`myArr`), JS looks up its prototype chain to find where `.map()` is defined (in this case: `Array.prototype`).

Object.create

- In JavaScript, the `Object.create()` method is used to link objects together and establish prototype chains. It allows us to create a new object with a specified prototype object.
- The `Object.create()` method creates a new object and sets the specified object as its prototype.
- It provides a way to create objects that inherit properties and methods from a prototype object.

Example:

```
const personPrototype = {
    greet: function() {
      console.log(`Hello, my name is ${this.name}.`);
    }
  };
  const john = Object.create(personPrototype);
  john.name = "John";
  john.greet(); // Output: Hello, my name is John.
```

call/apply/bind methods

- In JavaScript, the **`call()`**, **`apply()`**, and **`bind()`** methods are used to manipulate the execution context of functions and explicitly set the value of `this`. These methods provide flexibility and control over how functions are invoked.

1. **The `call()` Method:**

- The **`call()`** method is used to invoke a function and explicitly set the value of **`this`**.
- It accepts arguments individually, separated by commas.

Example:

```
  function greet(name) {
    console.log(`Hello, ${name}! I'm ${this.title}.`);
  }
  const person = {
    title: "Mr."
  };
  greet.call(person, "John"); // Output: Hello, John! I'm Mr.
```

2. **The `apply()` Method:**

- The **`apply()`** method is similar to **`call()`** but accepts arguments as an array or an array-like object.
- It invokes the function and sets the value of **`this`**.

Example:

```
function greet(name) {
  console.log(`Hello, ${name}! I'm ${this.title}.`);
}
const person = {
  title: "Mr."
};
greet.apply(person, ["John"]); // Output: Hello, John! I'm Mr.
```

3. **The `bind()` Method:**

- The **`bind()`** method creates a new function that has a specified **`this`** value and, optionally, initial arguments.
- It does not immediately invoke the function but instead returns a new function that can be called later.

Example:

```
function greet(name) {
  console.log(`Hello, ${name}! I'm ${this.title}.`);
}
const person = {
  title: "Mr."
};
const greetPerson = greet.bind(person);
greetPerson("John"); // Output: Hello, John! I'm Mr.
```

Object and Array Destructuring

Object Destructuring:

In JavaScript, object destructuring is a convenient way to extract values from an object and assign them to variables. It allows you to access and unpack properties from an object in a concise and structured manner.

Syntax:

```
const { prop1, prop2, ... } = object;
```

- **Some of the key points include the following:**

1. **Variable Assignment:** Object destructuring enables you to assign object properties to variables with the same name. For example:

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

2. **Renaming Variables:** You can assign object properties to variables with different names using the colon (:) syntax. This can be useful when the variable name conflicts with another identifier.

```
const { name: fullName, age: years } = person;
console.log(fullName); // Output: John
console.log(years); // Output: 30
```

3. **Default Values:** Object destructuring allows you to assign default values to variables if the corresponding object property is undefined.

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

- **Property value shorthand:**

  It allows you to omit the property key in an object literal if the variable name matches the key. This shorthand is also applicable during object destructuring:

  Example 1:

```
const { x, y } = { x: 11, y: 8 }; // x = 11; y = 8
```

  This shorthand syntax is equivalent to:

```
const { x: x, y: y } = { x: 11, y: 8 };
```

  Example 2:

  You can combine property value shorthand with default values:

```
const { x, y = 1 } = {}; // x = undefined; y = 1
```

Array Destructuring

Similarly, array destructuring in JavaScript provides a concise way to extract values from an array and assign them to variables.

Syntax:

```
const [item1, item2, ...] = array;
```

Some of the key points include the following:

1. **Variable Assignment:** Array destructuring allows you to assign array elements to variables based on their order.

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;
console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

2. **Ignoring Elements:** You can skip elements in the array by leaving empty slots (commas) in the destructuring pattern.

```
const [x, , z] = numbers;
```

```
console.log(x); // Output: 1
console.log(z); // Output: 3
```

3. **Rest Syntax:** The rest syntax (...) allows you to capture the remaining elements of an array into a new array.

```
const [first, ...rest] = numbers;
console.log(first); // Output: 1
console.log(rest); // Output: [2, 3]
```

4. **Default Values:** Array destructuring supports default values as well, which are assigned if an array element is undefined.

```
const [p = 0, q = 0, r = 0] = numbers;
console.log(p); // Output: 1
console.log(q); // Output: 2
console.log(r); // Output: 3
```

Object and array destructuring are powerful features in JavaScript that simplify the extraction of values from objects and arrays, providing cleaner and more readable code.

Summarizing it

In this lecture, we have covered the following topics:

- Objects, Object Creation and Accessing: We explored the concept of objects in JavaScript. We discussed object creation using object literals, which provide a convenient way to create and initialize objects using key-value pairs. We also learned about accessing object properties using dot notation and bracket notation.
- The "this" Keyword in Objects: We learned that "this" refers to the object itself when used within an object method, allowing seamless access to other properties and methods of the same object.
- Constructor Functions: We introduced constructor functions as an alternative approach for creating and initializing objects.

- Object Prototype: We learned that properties and methods can be added to the prototype object of a constructor function, and any objects created with that constructor will inherit those properties and methods.
- Object.create: We briefly discussed the `Object.create()` method, which is used to link objects together and establish prototype chains. It allows for the creation of new objects that inherit properties and methods from a prototype object.
- Call/Apply/Bind Methods: We covered the call(), apply(), and bind() methods, which are used to manipulate the execution context of functions and explicitly set the value of "this".
- Object and Array Destructuring in JavaScript

References

- Object Prototypes: **Link**
- Array Prototype: **Link**

Encapsulation in JavaScript

Encapsulation

Encapsulation is the concept of bundling the data (state) and methods (functions) that operate on the data into a single unit known as a class. It restricts access to some of the object's components, providing a way to control the data's visibility and behavior.

Benefits of Encapsulation

- **Data Protection:** It helps protect the integrity of an object's data by controlling how it can be accessed and modified.
- **Abstraction:** Encapsulation hides the internal implementation details of a class, allowing you to interact with objects using a simple interface, which is useful for managing complexity.
- **Maintainability:** It makes code easier to maintain and refactor because changes to the internal implementation of a class do not affect the code that uses the class.
- **Reduced Bugs:** By controlling access to data and enforcing constraints through setter methods, you can reduce the chances of introducing bugs and invalid states.

Defining Private Properties and Methods:

To define a private property or method, use the # symbol followed by the property or method name within a class. Private properties are typically used to store internal state, while private methods encapsulate logic.

```
class Person {

    #name; // Private property

    #age; // Private property

    constructor(name, age) {

      this.#name = name;

      this.#age = age;

    }

    #incrementAge() { // Private method

      this.#age++;

    }

    sayHello() {

      console.log(`Hello, my name is ${this.#name} and I am ${this.#age}
years old.`);

    }

  }
```

Accessing Private Properties and Methods Inside the Class:

Inside the class, private properties and methods can be accessed directly like any other property or method. They are in the same scope as other class members.

```
class Person {
    #name;
    #age;
    constructor(name, age) {
      this.#name = name;
      this.#age = age;
    }
    greet() {
      console.log(`Hello, my name is ${this.#name}`);
      this.#incrementAge(); // Accessing private method
    }
    #incrementAge() {
      this.#age++;
    }
  }
```

Accessing Private Properties and Methods Outside the Class:

Private properties and methods **cannot be accessed directly from outside the class**. Attempting to do so will result in a TypeError. You can only access them through public methods like getter and setter.

Getter- A method that returns the property of a class.

Setter - A method that is used to manipulate the property of the class.

Getter Methods for Private Properties:

To provide controlled access to private properties, you can define public-getter methods within the class. These getter methods allow you to retrieve the values of private properties.

```
class Person {
    #name;
    #age;
    constructor(name, age) {
      this.#name = name;
      this.#age = age;
    }
    getName() {
      return this.#name; // Getter method for private property
    }
```

```
}

const person = new Person('Alice', 30);

console.log(person.getName()); // Outputs: Alice
```

This way, you can retrieve the value of the private property #name without exposing it directly.

Setter Methods for Private Properties:

To modify the values of private properties, you can define public setter methods within the class. These setter methods allow you to update the values of private properties with validation logic if needed.

```
class Person {

    #name;

    constructor(name) {

      this.#name = name;

    }

    setName(newName) {

      if (newName.length >= 3) {

        this.#name = newName; // Setter method for private property

      }

    }

}

const person = new Person('Alice');

person.setName('Bob');

console.log(person.getName()); // Outputs: Bob
```

The setName method checks the length of the new name and only updates the private property if the condition is met.

Summary:

In summary, encapsulation in JavaScript using ES6 classes involves defining classes, using naming conventions for property and method visibility, and employing getter and setter methods to control access to class properties. It helps in creating well-structured and maintainable code by encapsulating data and behavior within classes.

RegEx Object

- Regular Expressions, commonly known as RegEx, are powerful patterns used to match and manipulate strings in JavaScript and many other programming languages.
- They provide a concise and flexible way to search, validate, and modify text data. In JavaScript, RegEx is supported through the `RegExp` object and a set of methods for working with patterns.

Creating a RegEx object:

To create a RegEx object in JavaScript, you can use the `RegExp` constructor or use the literal syntax, which involves enclosing the pattern within forward slashes (/pattern/). For example:

```
// Using the RegExp constructor
var regex = new RegExp("pattern");


// Using the literal syntax
var regex = /pattern/;
```

Common RegEx methods in JavaScript:

- test(): This method checks if a pattern exists within a string and returns a boolean value.

```
var regex = /pattern/;
var text = "This is a pattern example.";
var isMatch = regex.test(text);
console.log(isMatch);  // Output: true
```

- exec(): This method searches for a pattern match within a string and returns an array containing the matched result. It also provides additional information about the match, such as the index and input string.

```
var regex = /pattern/;

var text = "This is a pattern example.";

var result = regex.exec(text);

console.log(result);  // Output: ["pattern", index: 10, input: "This is a pattern example."]
```

- match(): This method searches for a pattern match within a string and returns an array of all matched occurrences.

```
var regex = /pattern/g;

var text = "This is a pattern example. Another pattern example.";

var matches = text.match(regex);

console.log(matches);  // Output: ["pattern", "pattern"]
```

- search(): It searches for a pattern match within a string and returns the index of the first occurrence. If no match is found, it returns -1.

```
var regex = /pattern/;

var text = "This is a pattern example.";

var index = text.search(regex);

console.log(index);  // Output: 10
```

- replace(): This method searches for a pattern match within a string and replaces it with a specified replacement value.

```
var regex = /pattern/;

var text = "This is a pattern example.";
```

```
var modifiedText = text.replace(regex, "new pattern");

console.log(modifiedText);  // Output: "This is a new
pattern example."
```

- split(): This method splits a string into an array of substrings based on a specified pattern match.

```
var regex = /[, ]+/;

var text = "Apple, Banana, Orange";

var fruits = text.split(regex);

console.log(fruits);  // Output: ["Apple", "Banana",
"Orange"]
```

These are just a few of the many methods available for working with RegEx in JavaScript. Regular expressions offer a wide range of pattern-matching capabilities, including character classes, quantifiers, anchors, capturing groups, and more. They are a powerful tool for string manipulation and text processing.

Attribute in RegEx

In regular expressions (RegEx), there are several attributes or flags that can be used to modify the behavior of the pattern matching. These attributes are represented by single characters and are added to the end of the RegEx pattern. Here are some commonly used attributes in JavaScript:

- g (global): This attribute performs a global search, meaning it searches for all occurrences of the pattern in the input string, rather than stopping after the first match.

```
var regex = /pattern/g;
```

- i (ignore case): This attribute performs a case-insensitive search, ignoring the difference between uppercase and lowercase characters.

```
var regex = /pattern/i;
```

- m (multiline): This attribute enables multiline mode, which changes the behavior of the `^` and `$` anchors to match the beginning and end of each line, rather than the entire string.

```
var regex = /^pattern/m;
```

- s (dotAll): This attribute allows the dot (.) metacharacter to match newline characters **(\n)**, which is normally not the default behavior.

```
var regex = /pattern/s;
```

- u (unicode): This attribute enables full Unicode matching, including support for Unicode code point escape sequences **(\u{...}).**

```
var regex = /pattern/u;
```

- y (sticky): This attribute enables sticky mode, which restricts the search to match only at the current position in the input string, indicated by the lastIndex property of the RegEx object.

```
var regex = /pattern/y;
```

These attributes can be used individually or in combination. For example, **/pattern/gi** creates a case-insensitive global search. It's important to note that these attributes affect the behavior of the RegEx pattern and how it interacts with the input string during matching.

- To add multiple attributes, you simply append them together to the end of the RegEx pattern. For example: **/pattern/gim**
- To access or modify the attributes of a RegEx object in JavaScript, you can use the **flags** property. For example:

```
var regex = /pattern/gi;
```

```
console.log(regex.flags);  // Output: "gi"
```

These attributes provide flexibility and control over how RegEx patterns are matched against strings in JavaScript, allowing you to tailor the matching behavior to your specific needs.

ES6 Classes

Classes in JS

Classes in JavaScript provide a convenient and object-oriented way to define and create objects with similar behavior and properties.

Why use classes?

- While constructor functions have been used to create objects in JavaScript for a long time, they're not as intuitive or easy to read as classes. Classes also provide a more concise and consistent syntax for working with inheritance.
- Classes are just an alternative way of creating objects and dealing with object-oriented programming in JavaScript - but one that many developers find more convenient.
- However, the existence of constructor functions in JavaScript prior to the introduction of classes makes them still relevant. They are compatible with older codebases and are still widely used in various scenarios.

Class Declaration:

A class declaration defines a new class using the `class` keyword. It consists of a class name and body containing constructor and method definitions.

Syntax:

```
class ClassName {
    constructor(parameters) {
      // Initialize object properties
    }
    method1() {
      // Method definition
    }
```

```
    method2() {
      // Method definition
    }
    // Additional methods...
  }
```

Insights into the syntax of class declaration in JavaScript.

- **Constructor:** The constructor method is a special method used for initializing class instances. It is invoked automatically when a new object is created from the class. It is defined using the `` `constructor` `` keyword.
- **Methods:** Methods are functions defined within a class. They are shared by all instances of the class and can be accessed through the instance objects. Methods do not require the `` `function` `` keyword.

Class Expression:

Similar to function expressions, classes can also be defined using class expressions. Class expressions can be named or unnamed.

Syntax:

1. **Unnamed class expression**

```
const ClassName = class {
    constructor(parameters) {
      // Initialize object properties
    }
    method1() {
      // Method definition
    }
    method2() {
      // Method definition
    }
    // Additional methods...
```

```
};
```

2. Named class expression

```
const ClassName = class ClassName {
  constructor(parameters) {
    // Initialize object properties
  }
  method1() {
    // Method definition
  }
  method2() {
    // Method definition
  }
  // Additional methods...
};
```

Hoisting and Execution of Strict Mode:

- Unlike function declarations, class declarations are not hoisted. They need to be declared before they are used. Class expressions, on the other hand, behave like variables and can be hoisted.
- When using classes, the **`use strict`** directive is implicitly applied, ensuring a stricter mode of JavaScript execution within the class.

Encapsulation

- Encapsulation is a fundamental principle of object-oriented programming that promotes data hiding and ensures that the internal implementation details of an object are inaccessible from outside the object.
- JavaScript, being a flexible and dynamic language, traditionally lacks built-in support for private properties and methods.

- However, with the introduction of new versions of JavaScript, specifically ECMAScript 2019 (ES10) and later, private properties and methods can be achieved using the `#` symbol.

Private Properties with the `#` Symbol:

- Encapsulation using private properties and methods with the `#` symbol provides a more robust and secure approach to object-oriented programming in JavaScript, enabling better data protection, code organization, and controlled access to internal functionality.
- The `#` symbol before a property name in a class or object denotes that the property is private and cannot be accessed or modified from outside the class or object. It restricts direct access to the property, providing encapsulation.

Example:

```
class MyClass {
    #privateProperty = 42;
    #privateMethod() {
      return 'This is a private method';
    }
    publicMethod() {
      console.log(this.#privateProperty); // Accessing private
property
      console.log(this.#privateMethod()); // Invoking private method
    }
  }
  const myObj = new MyClass();
  myObj.publicMethod(); // Output: 42, "This is a private method"
```

```
console.log(myObj.#privateProperty); // Error: SyntaxError

console.log(myObj.#privateMethod()); // Error: SyntaxError
```

In the example above, `#privateProperty` and `#privateMethod()` are private within the `MyClass` class. They can only be accessed and invoked from within the class itself, not from outside.

- It's important to note that private properties and methods are unique to each instance of the class. Each instance has its own private property values and private methods.
- The use of private properties and methods helps in achieving better encapsulation by hiding implementation details and preventing accidental modification or access from external code.

Benefits of Encapsulation and Private Properties:

1. **Information Hiding:** Private properties and methods encapsulate implementation details, hiding them from the outside world. This protects sensitive data and prevents direct manipulation, reducing the risk of unintended side effects.
2. **Controlled Access:** Encapsulation allows developers to control access to internal state and behavior. Only the necessary methods and properties are exposed as part of the public interface, ensuring more predictable and maintainable code.
3. **Code Organization**: Encapsulation promotes modular and organized code. Private properties and methods keep implementation-specific details hidden, allowing developers to focus on the public interface and higher-level functionality.

Inheritance

- Inheritance is a key concept in object-oriented programming that allows objects to inherit properties and methods from a parent class. It enables code

reuse, promotes code organization, and facilitates the creation of specialized subclasses.

- In other words, Inheritance is the practice of creating new objects based on existing ones, inheriting properties and methods from their parent objects.
- In JavaScript, inheritance can be achieved using the `super` keyword, the `extends` keyword, and function overriding.

The `extends` and `super` Keywords:

- The `extends` keyword is used to create a child class that inherits from a parent class in JavaScript. It establishes a hierarchical relationship between classes, allowing the child class to inherit the properties and methods of the parent class.
- The `super` keyword is primarily used within the constructor of a child class to call the constructor of the parent class. It enables the child class to access and initialize properties defined in the parent class.
- However, it's worth noting that the `super` keyword can also be used to call other methods defined in the parent class, not just the constructor. This allows the child class to leverage and extend the functionality provided by the parent class.

Example:

```
class Animal {

  constructor(name) {

    this.name = name;

  }

}

class Dog extends Animal {
```

```
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
}
const myDog = new Dog('Max', 'Labrador');
console.log(myDog);
```

- In the example above, the `Animal` class serves as the parent class, and the `Dog` class extends it using the **`extends`** keyword.
- The **`super`** keyword is used in the `Dog` class's constructor to call the `Animal` class's constructor and pass the `name` parameter.
- This allows the `Dog` class to inherit the `name` property from the `Animal` class.
- The final line outputs the `myDog` object, allowing you to see the values of the `name` and `breed` properties.

Function Overriding

- Function overriding is the ability of a child class to provide a different implementation for a method that is already defined in its parent class.
- When a child class overrides a method, the implementation in the child class is executed instead of the parent class's implementation.

Example:

```
class Shape {
  draw() {
    console.log('Drawing a shape.');
```

```
  }
}
class Circle extends Shape {
  draw() {
    console.log('Drawing a circle.');
  }
}
const myShape = new Shape();
myShape.draw(); // Output: Drawing a shape.
const myCircle = new Circle();
myCircle.draw(); // Output: Drawing a circle.
```

In the example above, the `Circle` class extends the `Shape` class and overrides the `draw()` method. When `draw()` is called on a `Circle` instance, the overridden implementation in the `Circle` class is executed instead of the one in the `Shape` class.

Inheritance in Constructor Functions

- In JavaScript, inheritance can also be achieved using constructor functions and the prototype chain. Constructor functions serve as blueprints for creating objects and can be used to establish inheritance relationships between classes.
- To implement inheritance in constructor functions, the `**prototype**` property is used to define shared properties and methods that will be inherited by all instances of the class. The `**prototype**` object of the parent constructor function is assigned to the `**prototype**` property of the child constructor function, creating a prototype chain.

Example:

```
function Vehicle(make) {
  this.make = make;
}

Vehicle.prototype.start = function() {
  console.log(`Starting the ${this.make} vehicle.`);
};

function Car(make, model) {
  Vehicle.call(this, make);
  this.model = model;
}
Car.prototype = Object.create(Vehicle.prototype);
const myCar = new Car('Toyota', 'Camry');
myCar.start(); // Output: Starting the Toyota vehicle.
```

- In this example, we have a `Vehicle` constructor function that sets the `make` property and defines a `start` method on its prototype.
- The `Car` constructor function extends `Vehicle` by calling `Vehicle.call(this, make)` in its own constructor. This ensures that the `make` property is properly set for each `Car` instance.
- To establish the inheritance relationship, we use `Object.create(Vehicle.prototype)` to create a new object with `Vehicle.prototype` as its prototype. This new object is assigned to `Car.prototype`, allowing `Car` instances to inherit properties and methods from `Vehicle`.
- Finally, we create an instance of `Car` named `myCar` with the make 'Toyota' and model 'Camry'. We can call the `start` method on `myCar`, which invokes the inherited `start` method from `Vehicle`, resulting in the output "Starting the Toyota vehicle."

Static keyword in JS

- The `static` keyword is used to define static properties and methods within a class. Static members are associated with the class itself rather than its instances.
- They are accessed and invoked directly on the class, without the need to create an object. They are accessed using the class name followed by the dot notation.
- Here's an example that demonstrates the usage of static properties and the `static` keyword:

```
class Circle {
  static PI = 3.14159;

  constructor(radius) {
    this.radius = radius;
  }

  calculateArea() {
    return Circle.PI * this.radius * this.radius;
  }

  static formatNumber(number) {
    return number.toFixed(2);
  }
}

const myCircle = new Circle(5);

console.log(myCircle.calculateArea()); // Output: 78.53975
console.log(Circle.formatNumber(5)); // Output: 5.00
console.log(myCircle.PI); // Output: undefined
console.log(Circle.PI); // Output: 3.14159
```

- In the example, we have a `Circle` class with a static property `PI` representing the mathematical constant π.
- The `calculateArea` method uses this static property to calculate the area of the circle.
- The `formatNumber` method is also declared as static and can be directly accessed on the class. It formats a given number to two decimal places.
- When we create an instance of the `Circle` class (`myCircle`) and call the `calculateArea` method, it correctly calculates the area using the static property `PI`.
- However, when we try to access the static property `PI` through the instance `myCircle`, it returns undefined. Static properties are not accessible through instances; they are only accessible through the class itself.
- To access the static property `PI`, we use the class name `Circle` followed by the dot notation (`Circle.PI`). This returns the value of the static property, which is 3.14159.

Getter and Setter

- Getters and Setters are special methods that allow controlled access to object properties. They provide a way to define custom behavior when getting or setting the values of properties.
- Getters are used to retrieve the value of a property, while setters are used to set the value of a property.
- They provide control over property access and manipulation, allowing for more robust and controlled data handling within objects.

Key Points about Getters and Setters:

1. **Getter:**

a. A getter is a method that is used to retrieve the value of a property.

b. It is defined using the `get` keyword followed by the method name.

c. Getters do not accept any parameters.

d. Getters are accessed like regular properties, without using parentheses.

e. Getters can be useful for performing computations or returning modified values based on existing properties.

2. **Setter:**

a. A setter is a method that is used to set the value of a property.

b. It is defined using the `set` keyword followed by the method name.

c. Setters accept a single parameter, representing the value to be set.

d. Setters are accessed like regular properties, using the assignment operator (=).

e. Setters can be useful for performing validation or implementing side effects when assigning values to properties.

3. **Syntax:**

- **Getter syntax:**

```
`get propertyName() { /* code to return value */ }`
```

- **Setter syntax:**

```
`set propertyName(value) { /* code to set value */ }`
```

4. **Usage and Benefits:**

a. Getters and setters provide an interface to access and modify object properties in a controlled manner.

b. They allow encapsulation of data, enabling validation or manipulation of property values before setting or retrieving them.

c. Getters and setters can be used to implement computed properties, where the value is derived from other properties or calculations.

d. They provide a way to handle edge cases and ensure consistency in data manipulation within an object.

Example:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  get uppercaseName() {
    return this.name.toUpperCase();
  }
  set setName(newName) {
    this.name = newName;
  }
}
const alice = new Person("Alice");
console.log(alice.uppercaseName); // logs "ALICE"
alice.setName = "Bob"; // sets the name property to "Bob"
```

```
console.log(alice.uppercaseName); // logs "BOB"
```

- The example defines a `Person` class with a `name` property. It has a getter method `uppercaseName` that returns the name property in uppercase.
- The setter method `setName` allows updating the `name` property. An instance of `Person` named `alice` is created with the name "Alice".
- By accessing `alice.uppercaseName`, it returns the uppercase name "ALICE". Setting `alice.setName` to "Bob" updates the name property.
- Accessing `alice.uppercaseName` again returns the updated uppercase name "BOB".

Built-in Objects in JS

Built-in objects in JavaScript provide a set of predefined functionalities and properties that can be used in your code without the need for additional declarations or external libraries. These objects offer a wide range of capabilities to work with various data types, perform mathematical operations, manipulate dates, and more. Let's explore a few important built-in objects:

1. **Date Object:**
    a. The **`Date`** object is used for working with dates and times in JavaScript.
    b. It provides methods to create, manipulate, and format dates.
    c. You can create a new instance of **`Date`** using the **`new Date()`** constructor.

    ❖ **Date Formats:**

    ○ The **`Date`** object supports various date formats, such as:

- ISO 8601 format: "YYYY-MM-DDTHH:mm:ss.sssZ"
- Short date format: "MM/DD/YYYY"
- Long date format: "Month DD, YYYY"
- Custom formats using the `toLocaleDateString()` method.

❖ **Common Methods:**

- `getFullYear()`: Returns the year (4 digits) of the date.
- `getMonth()`: Returns the month (0-11) of the date.
- `getDate()`: Returns the day of the month (1-31).
- `getDay()`: Returns the day of the week (0-6), starting from Sunday.
- `getHours()`, `getMinutes()`, `getSeconds()`: Returns the respective time components.
- `toString()`: Returns a string representation of the date.

Example:

```
const currentDate = new Date();
console.log(currentDate.toString()); // Output: Wed Jun 14 2023
12:34:56 GMT+0530 (India Standard Time)
```

2. **Math Object:**
   a. The `Math` object provides mathematical operations and constants.
   b. It contains various methods for common mathematical calculations, such as `Math.sqrt()`, `Math.pow()`, `Math.abs()`, `Math.sin()`, `Math.cos()`, etc.
   c. Constants like `Math.PI` and `Math.E` are also available.

Example:

```
  console.log(Math.sqrt(16)); // Output: 4
  console.log(Math.PI); // Output: 3.141592653589793
```

3. Random Number Generation:

a. JavaScript provides the Math.random() method to generate random numbers between 0 and 1 (exclusive).
b. You can multiply it by a desired range and apply appropriate rounding or truncation to get random integers within a specific range.

Example:

```
const randomNumber = Math.floor(Math.random() * 10); // Generates a random integer between 0 and 9
console.log(randomNumber);
```

JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format widely used in web development for data transmission between client and server, storing configuration settings, and exchanging data with APIs.
- It provides a simple and readable format for representing structured data.
- Let's explore JSON and its related methods in more detail:

1. JSON Format:

- JSON represents data using a combination of object notation and key-value pairs. It resembles the syntax of JavaScript objects and arrays.
- Object Notation:
    - Data is structured as a collection of key-value pairs, enclosed in curly braces {}. Key-value pairs are separated by a colon (:), and each pair is separated by a comma (,).
    - JSON values can be strings, numbers, booleans, arrays, objects, or null.

Example:

```
const person = {
  "name": "Chris Evans",
  "age": 30,
  "isStudent": false
};
```

- Array Notation:
    - Data can also be represented as an ordered list using square brackets `[]`.

Example:

```
const people = [
  {
    "name": "Peter Parker",
    "age": 30,
    "isStudent": false
  },
  {
    "name": "Jane Smith",
    "age": 25,
    "isStudent": true
  }
];
```

2. **File Extension:**

- JSON files typically have the `.json` extension.

- They are commonly used for data storage and exchange between applications.

In JavaScript, JSON is a built-in object that offers methods for parsing and stringifying JSON data.

1. **JSON.parse():**

- The `**JSON.parse()**` method converts a JSON string into a JavaScript object.
- It takes a valid JSON string as input and returns a corresponding JavaScript object.

Example:

```
const jsonString = '{"name":"John","age":30,"city":"New York"}';

const person = JSON.parse(jsonString);

console.log(person.name); // Output: "John"
```

2. **JSON.stringify():**

- The `**JSON.stringify()**` method converts a JavaScript object into a JSON string.
- It takes an object as input and returns a string representation of the object in JSON format.

Example:

```
const person = {

  name: "John",

  age: 30,

  city: "New York"

};
```

```
  const jsonString = JSON.stringify(person);

  console.log(jsonString); // Output:
'{"name":"John","age":30,"city":"New York"}'
```

Shallow and Deep Copy in JS

- In JavaScript, when working with objects and arrays, it's important to understand the concepts of shallow copy and deep copy.
- It's important to choose the appropriate copy method based on the specific requirements of your use case.
- Shallow copy is suitable when you want to create a new object or array with the same references to nested objects or arrays.
- Deep copy, while not always straightforward, is necessary when you need to create a completely independent copy that does not affect the original object or array.

Let's explore what they mean and how they can be achieved using different methods:

Shallow Copy

A shallow copy creates a new object or array and copies the references of the original values. This means that if the original object contains nested objects or arrays, the shallow copy will still reference the same nested objects or arrays.

Methods for Shallow Copy:

1. **Spread Operator:**

- The spread operator (`...`) can be used to create a shallow copy of an array or object.
- Shallow copy only affects the first layer of the object or array. If there are nested objects or arrays, they will still be referenced, and modifying them in the shallow copy will affect the original object or array.

Example:

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

const shallowCopyArray = [...originalArray];

const originalObject = { name: 'John', age: 30 };

const shallowCopyObject = { ...originalObject };
```

2. **Object.assign():**

- The `Object.assign()` method can be used to create a shallow copy of an object.

Example:

```
const originalObject = { name: 'John', age: 30 };

const shallowCopyObject = Object.assign({}, originalObject);
```

Deep Copy

A deep copy creates a completely new object or array with its own set of values. Any changes made to the deep copy will not affect the original object or array, and vice versa.

Methods for Deep Copy:

1. **JSON.stringify() and JSON.parse():**

- The combination of `JSON.stringify()` and `JSON.parse()` can be used to create a deep copy of an object or array.

Example:

```
const originalObject = { name: 'John', age: 30 };

const deepCopyObject = JSON.parse(JSON.stringify(originalObject));
```

Note: Although `JSON.stringify()` and `JSON.parse()` are commonly used for deep copying, they have limitations. They cannot handle functions, undefined values, or circular references. Additionally, if the object or array contains complex data types

like dates or regular expressions, they may be converted to strings during the stringification process and lose their original type.

Summarizing it

Let's summarize what we have learned in this Lecture:

- ES6 classes: Introduced for class-based object creation and inheritance.
- Encapsulation: Concept of bundling data and methods within a class.
- Inheritance: Mechanism for inheriting properties and methods from parent classes.
- Static keyword: Used to define static properties and methods in classes.
- Getter and Setter: Special methods for property retrieval and assignment.
- Built-in objects: JavaScript provides objects like Date, Math, and Array for common operations.
- JSON: Lightweight format for data interchange and storage.
- Deep and Shallow Copy: Techniques for creating copies of objects and arrays with varying levels of independence.

References

- Standard built-in objects: **Link**
- Working with JSON: **Link**

setAttribute Method

setAttribute is a method in JavaScript used to modify or set the value of an attribute for a specified HTML element. It is particularly useful when you want to dynamically change or add attributes to elements on the page.

Syntax

```
element.setAttribute(attributeName, attributeValue);
```

- **element:** The reference to the HTML element on which you want to set the attribute.
- **attributeName:** The name of the attribute you want to modify or add.
- **attributeValue:** The value you want to assign to the specified attribute.

Modifying Existing Attributes:

You can use setAttribute to modify the value of an existing attribute. For example, to change the src attribute of an image element:

html

```
<img id="myImage" src="image.jpg" alt="My Image">
```

javascript

```
const imageElement = document.getElementById('myImage');

imageElement.setAttribute('src', 'new-image.jpg');
```

Now, the src attribute will be changed to "new-image.jpg".

Adding New Attributes:

If you want to add a new attribute to an element, setAttribute can be used for that as well. For instance, you can add a data attribute to a div element:

html

```
<div id="myDiv">Hello</div>
```

Javascript

```
const divElement = document.getElementById('myDiv');

divElement.setAttribute('data-info', 'Some additional info');
```

This will add a data-info attribute with the value "Some additional info" to the div element.

Special Considerations:

When using setAttribute, be mindful of the attribute names and their case sensitivity. Attribute names are generally case-insensitive in HTML, but it's recommended to use lowercase for consistency.

If an attribute with the given name already exists, setAttribute will update its value. If it doesn't exist, the method will create a new attribute.

For certain attributes like class, style, and onclick, using setAttribute might not work as expected. It's better to use specific property assignments in those cases.

Removal of Attributes:

To remove an attribute from an element, you can set its value to null or use the removeAttribute method:

Javascript

```
const myElement = document.getElementById('myElement');

// Remove an attribute using setAttribute

myElement.setAttribute('data-info', null);
```

setAttribute provides a flexible way to manipulate attributes in the DOM, making it an essential tool for dynamic web development and interactive user interfaces. However, it's worth noting that when dealing with simple attributes like src, href, and others that have corresponding properties, directly setting the properties (e.g., element.src = 'new-image.jpg') is generally more straightforward and efficient.

DOM Content Manipulation

Understanding the concept of JavaScript properties, such as `**.innerHTML**`, and `**.textContent**` is vital for effectively manipulating and accessing content within HTML elements.

.innerHTML:

- Usage: Used to get or set the HTML content within an element.
- Returns: A string representing the HTML content, including tags and markup.
- Effect: Can be used to dynamically update or retrieve the markup and text within an element.
- Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <div id="myElement">Initial content</div>

    <script>
      var element = document.getElementById("myElement");
      var htmlContent = element.innerHTML;
      console.log( htmlContent);
      // Output:Initial content
```

```
      element.innerHTML = "<strong>New content</strong>";

      var updatedHtmlContent = element.innerHTML;

      console.log("Updated HTML content:", updatedHtmlContent);

      // Output:  <strong>New content</strong>

    </script>

  </body>

</html>
```

.textContent:

- Usage: Used to get or set the text content of an element, excluding any HTML tags or markup.
- Returns: A string representing the plain text content without any HTML formatting.
- Effect: Allows manipulation of the text content specifically.
- Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <div id="myElement">Initial text content</div>


    <script>
      var element = document.getElementById("myElement");
```

```
    var textContent = element.textContent;

    console.log("Initial text content:", textContent);

    // Output: Initial text content: Initial text content


    element.textContent = "New text content";

    var updatedTextContent = element.textContent;

    console.log("Updated text content:", updatedTextContent);

    // Output: Updated text content: New text content

  </script>

 </body>

</html>
```

- **Note:** **`.textContent`** provides a safe way to modify the textual content without the risk of executing potential malicious code.

.innerHTML vs .textContent

a. **`.innerHTML`** is primarily used when you need to work with the HTML content of an element, including tags and markup. It allows you to retrieve or modify both the text and HTML structure within an element.

b. This property is useful when you want to update or extract HTML content, such as inserting new elements, modifying attributes, or applying formatting.

c. For instance, if you need to dynamically insert new elements or modify the existing structure, .innerHTML is the appropriate choice.

d. On the other hand, **`.textContent`** is specifically designed to work with the text content of an element, excluding any HTML tags or markup.

e. It provides a straightforward way to manipulate or extract the plain text within an element, regardless of any HTML formatting. Use .textContent when you need to perform operations solely on the text, such as extracting data or updating the textual content without affecting the HTML structure or any child elements within the element.

f. Unlike `.innerHTML`, `.textContent` treats the assigned value purely as text, ensuring that it is not interpreted as HTML or executed as code, making it a safer option.

append vs appendChild

`appendChild()` and `append()`

When working with JavaScript to manipulate the DOM, it's essential to understand the distinctions between **`appendChild()`** and **`append()`** methods, as they have different behaviors and use cases for adding elements.

- **`appendChild()`** is a method used to append a single element as a child to another element. It takes an element node as its parameter and adds it as the last child of the specified parent element.
- Example:

```
// Creating new elements
var parentElement = document.getElementById("parent");
var childElement = document.createElement("div");

// Appending the child element to the parent element
parentElement.appendChild(childElement);
```

  In this example, the **`appendChild()`** method is used to add the **`childElement`** as the last child of the **`parentElement`**. This method is useful when you want to add a single element to another element.

- **`append()`**, on the other hand, is a more versatile method that allows you to append multiple elements or text to another element. It accepts a variable number of arguments, which can be elements, text strings, or a combination of both.

- Example:

```
// Creating new elements

var parentElement = document.getElementById("parent");

var childElement = document.createElement("div");

var textNode = document.createTextNode("Hello, World!");


// Appending multiple elements to the parent element

parentElement.append(childElement, textNode);
```

In this example, the **`append()`** method is used to add both the **`childElement`** and **`textNode`** to the **`parentElement`**. It can accept any number of elements or text strings as arguments, allowing you to append multiple elements at once.

Usage of Backticks (``) to Add Elements in JavaScript

In JavaScript, backticks (``) are used to create template literals, which provide a concise and powerful way to create strings that can include variables, expressions, and even HTML elements. This functionality proves especially useful when dynamically adding elements or groups of elements to the DOM.

Adding an Element using Backticks:

- Backticks allow you to define an HTML string with embedded variables or expressions. This approach simplifies the process of creating elements and their associated attributes.

- Example:

```
// Creating a new element using backticks
var className = "my-class";
var content = "This is a new element";
```

```
var element = `<div class="${className}">${content}</div>`;
```

In this example, backticks are used to create an HTML string assigned to the **`element`** variable. The string includes the **`className`** variable to dynamically set the class attribute and the **`content`** variable to insert the desired content. This HTML string can then be inserted into the DOM using methods like **`innerHTML`** or by appending it as a child to another element.

Adding a Group of Elements using Backticks:

- Backticks also enable the creation of a group of elements within a single template literal. This technique allows for cleaner and more readable code when adding multiple elements at once.
- Example:

```
// Creating a group of elements using backticks
var elements = `
  <div class="element">Element 1</div>
  <div class="element">Element 2</div>
  <div class="element">Element 3</div>
`;
```

In this example, backticks are used to define a multi-line string representing a group of elements. Each element is enclosed within **`<div>`** tags and has a shared class name. The resulting string, assigned to the **`elements`** variable, can be inserted into the DOM using appropriate methods like **`innerHTML`** or appended to an existing element using **`appendChild()`** or **`append()`**.

Different Ways to add elements in DOM

1. **appendChild() and append() method:**
   The appendChild() and append() method is used to add a new child element to the end of the specified parent element.

2. **insertAdjacentHTML() method:**

   The insertAdjacentHTML() method allows you to insert HTML content at a specific position relative to the element.

3. **innerHTML property:**

   The innerHTML property allows you to get or set the HTML content of an element. You can add new elements as HTML strings and set them as the innerHTML of the parent element.

4. **insertBefore() method:**

   The insertBefore() method can be used to insert an element before a specified reference element.

DOM Manipulation

Application Programming Interface

- An Application Programming Interface (API) is a set of rules and protocols that allows different software applications to communicate and interact with each other.
- Web APIs specifically refer to the APIs provide a standardized way for different software components to exchange data and perform specific functions.

Web APIs

As we have seen in Lecture 4, web APIs are an integral part of web development. They allow developers to access and interact with web-based services, retrieve data, and perform various operations.

Types of Web APIs:

Web APIs can be broadly categorized into two types:

a. **Browser APIs:**

- These APIs are built into web browsers and provide capabilities for accessing and manipulating various browser-related features.
- Browser APIs are provided by web browsers and enable developers to interact with browser features and functionalities.
- Some common browser APIs are accessible through the `**window**` object, which is a global object in the browser environment.
- Examples of browser APIs include:

- **Console API:** The Console API, accessible through the **`window.console`** property, provides methods to output messages and debug information to the browser console. It is commonly used for logging and debugging purposes.
- **Alert API:** The Alert API, accessible through the **`window.alert`** method, displays a simple dialog box with a message to the user. It is often used to show important notifications or prompts.
- **DOM API (Document Object Model):**

  The DOM API allows developers to access and manipulate the elements of an HTML or XML document. It represents the document as a tree-like structure, where each element is a node.

  The Document object, accessible through the **`window.document`** property, represents the web page loaded in the browser and provides methods and properties to interact with the document's content.

b. Third-Party APIs:

- These APIs are developed by external providers and allow developers to access their services and integrate their functionalities into their applications.
- Examples include:
  - **Google Maps API:** Google Maps API provides developers with the ability to embed and interact with maps within their web applications. It offers various services like geocoding, directions, and map customization.

Document Object Model

- The Document Object Model (DOM) is a programming interface for HTML and XML documents, representing them as a structured tree-like model.
- The DOM allows developers to access, manipulate, and navigate the elements and content of a web page dynamically.

Representation of the DOM:

- The DOM represents an HTML or XML document as a hierarchical structure known as the DOM tree.
- The DOM tree consists of various nodes, where each node represents an element, attribute, or text within the document.
- The relationship between nodes is defined by parent-child relationships, with each node having zero or more child nodes.
- **DOM Tree Representation:** It represents the structure of an HTML or XML document. Here's an example of a simplified DOM tree for an HTML document:

- The topmost node in the DOM tree is the `**document**` object, representing the entire HTML document.
- The `**documentElement**` node represents the `**<html>**` element, which serves as the root of the DOM tree.
- The `**documentElement**` node has child nodes, such as the `**head**` and `**body**` elements.
- The `**head**` element contains child nodes, including the `**title**` element, which represents the title of the document.
- The `**body**` element represents the document's body and can contain various elements, such as headings (`**h1**`, `**h2**`, etc.), paragraphs (`**p**`), images (`**img**`), and more.
- The DOM tree continues to branch out, with each element having its own child nodes, forming a hierarchical structure.

Node Access using the Document Object

The Document object provides various methods and properties to access different nodes within the DOM tree using the dot operator.

a. `document.body`

- Returns the `**<body>**` element of the document.
- Example:

```
var bodyElement = document.body;
```

b. `firstElementChild`

- Returns the first immediate child element of the document.
- Example:

```
var firstChild = document.firstElementChild;
```

c. `lastElementChild`

- Returns the last immediate child element of the document.
- Example:

```
var lastChild = document.lastElementChild;
```

d. `children`

- Returns a collection of immediate child elements of the document.
- Example:

```
var children = document.children;
```

These are just a few examples of the many methods available for the Document object. Each method provides different ways to interact with and manipulate the elements and content within the DOM tree.

Selectors in JavaScript

- Selectors in JavaScript are used to fetch data or access particular nodes within the Document Object Model (DOM) of an HTML document.
- Selectors allow developers to target specific elements, classes, or IDs to perform operations on them dynamically.

Commonly Used Selectors:

- **`document.querySelector(selector)`**
  - Returns the first element that matches the specified CSS selector.
  - Example:

```
var element = document.querySelector(".myClass");
```

- **`document.querySelectorAll(selector)`**
  - Returns a collection of elements that match the specified CSS selector.
  - Example:

```
var elements = document.querySelectorAll("div");
```

- **`document.getElementById(id)`**
    - Returns the element with the specified ID attribute.
    - Example:
      ```
      var element = document.getElementById("myElement");
      ```
- **`document.getElementsByClassName(className)`**
    - Returns a collection of elements with the specified class name.
    - Example:
      ```
      var elements =
      document.getElementsByClassName("myClass");
      ```
- **`document.getElementsByTagName(tagName)`**
    - Returns a collection of elements with the specified tag name.
    - Example:
      ```
      var elements = document.getElementsByTagName("div");
      ```

Best Practices for Placement of Script Tag

When including JavaScript code in an HTML document, it's important to consider the placement of the **`<script>`** tag for optimal performance and functionality.

- **Placing `<script>` in the `<head>`:**
    - Placing the **`<script>`** tag in the **`<head>`** section allows the script to be loaded and executed before the HTML content is parsed and rendered.
    - This approach is suitable when the script needs to be executed early or when it requires elements in the **`<head>`** section (e.g., modifying the document's metadata).

- Example:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js"></script>
  </head>
  <body>
    <!-- HTML content -->
  </body>
</html>
```

- **Placing `<script>` before the closing `</body>` tag:**
  - Placing the **`<script>`** tag just before the closing **`</body>`** tag allows the HTML content to load and render before the script is executed.
  - This approach improves the perceived page load time since the user can see and interact with the page while the script is being fetched and executed.
  - Example:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- HTML content -->
  </head>
  <body>
    <!-- HTML content -->
    <script src="script.js"></script>
  </body>
</html>
```

It's generally recommended to place scripts just before the closing </body> tag to ensure faster initial page load and better user experience.

Styling Fetched Data from Selectors

- Using JavaScript, we can dynamically fetch elements with selectors and apply custom styles to enhance their appearance or modify their properties based on specific conditions.
- Here's an example of how you can add styling to fetched data using selectors:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .highlight {
        background-color: yellow;
      }

      .bold {
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <h1 class="highlight">Welcome to My Website</h1>
    <p>This is a paragraph with some content.</p>
    <ul>
      <li class="bold">Item 1</li>
      <li>Item 2</li>
      <li class="highlight">Item 3</li>
    </ul>

    <script>
```

```
      // Fetching and styling elements
      var highlightedElements =
document.getElementsByClassName("highlight");
      for (var i = 0; i < highlightedElements.length; i++) {
        highlightedElements[i].style.color = "red";
      }

      var boldElement = document.querySelector("li.bold");
      boldElement.style.fontSize = "20px";
    </script>
  </body>
</html>
```

- In the above example:
    - The CSS styles defined within the `**<style>**` tag specify a yellow background color for elements with the class `**highlight**` and bold font weight for elements with the class `**bold**`.
    - Inside the `**<body>**` section, there are heading, paragraph, and list elements with various classes applied.
    - The JavaScript code within the `**<script>**` tag selects elements using selectors and applies additional styling to them.
    - The `**highlightedElements**` variable fetches all elements with the class `**highlight**`, and a loop sets their color to red.
    - The `**boldElement**` variable uses the selector `**li.bold**` to fetch the specific list item and modifies its font size.

JavaScript Element Manipulation

1. Creating Elements:

In JavaScript, you can dynamically create new HTML elements using the `document.createElement(tagName)` method. Here's an example:

```
// Creating a new paragraph element
var paragraph = document.createElement("p");

// Adding text content to the paragraph
paragraph.textContent = "This is a dynamically created paragraph.";

// Adding a CSS class to the paragraph
paragraph.classList.add("highlight");

// Appending the paragraph to the body element
document.body.appendChild(paragraph);
```

- In the above example:
    - The `document.createElement("p")` method creates a new `<p>` element.
    - The `textContent` property sets the text content of the paragraph.
    - The `classList.add("highlight")` adds the CSS class "highlight" to the paragraph.
    - The `appendChild` method appends the paragraph as a child of the `document.body` element.

2. Appending Elements:

To append an existing element as a child to another element, you can use the `**appendChild(childElement)**` method. Here's an example:

```
// Creating a new <li> element
var listItem = document.createElement("li");
listItem.textContent = "New Item";

// Selecting the parent <ul> element
var parentList = document.getElementById("myList");

// Appending the new <li> element to the parent <ul>
parentList.appendChild(listItem);
```

- In the above example:
    - The `**document.createElement("li")**` method creates a new `**<li>**` element.
    - The `**textContent**` property sets the text content of the list item.
    - The `**getElementById("myList")**` method selects the parent `**<ul>**` element.
    - The `**appendChild(listItem)**` method appends the list item as a child to the parent `**<ul>**` element.

3. Removing Elements:

To remove an element from the DOM, you can use the remove() method or manipulate its parent element using the `**removeChild(childElement)**` method. Here's an example:

```
// Selecting the element to remove

var elementToRemove = document.getElementById("myElement");
```

```
// Removing the element using the remove() method
elementToRemove.remove();


// Alternatively, removing the element using the parent's removeChild() method
var parentElement = document.getElementById("parentElement");
parentElement.removeChild(elementToRemove);
```

- In the above example:
    - The `**`getElementById("myElement")`**` method selects the element to be removed.
    - The `**`remove()`**` method removes the element directly from the DOM.
    - Alternatively, the `**`getElementById("parentElement")`**` method selects the parent element, and the `**`removeChild(elementToRemove)`**` method removes the specified child element.

Event Listeners

- Event listeners in JavaScript allow you to respond to various actions or events triggered by user interactions or system events.
- By attaching event listeners to elements, you can execute specific code or functions in response to those events.

Using addEventListener():

- The `addEventListener()` method is commonly used to attach event listeners to DOM elements. It allows you to specify the event type and the function to be executed when the event occurs.
- The syntax for `addEventListener()` is as follows:

```
element.addEventListener(eventType, callbackFunction);
```

  - **element:** The DOM element to which the event listener is attached.
  - **eventType:** The type of event to listen for (e.g., "click", "mouseover", "keydown").
  - **callbackFunction:** The function that will be executed when the event occurs.

Some examples that demonstrate the usage of addEventListener() for different event types:

a) Handling a Click Event:

```
var button = document.getElementById("myButton");

button.addEventListener("click", function() {
  console.log("Button clicked!");
});
```

In this example, when the button with the ID "myButton" is clicked, the anonymous function is executed, which logs a message to the console.

b) Handling a Mouseover Event:

```
var image = document.getElementById("myImage");

image.addEventListener("mouseover", function() {
  image.src = "hover-image.jpg";
});
```

Here, when the mouse cursor hovers over the image with the ID "myImage," the anonymous function is executed, changing the image source to "hover-image.jpg."

c) Handling a Keydown Event:

```
var inputField = document.getElementById("myInput");

inputField.addEventListener("keydown", function(event) {
  console.log("Key pressed: " + event.key);
});
```

In this example, as a key is pressed within the input field with the ID "myInput," the anonymous function is executed, logging the pressed key to the console.

Note: The event object (e.g., event in the examples) can be accessed within the callback function, providing additional information about the event that occurred.

Event Propagation

- Event propagation refers to the process by which events are handled and propagated through the DOM tree.
- Understanding event propagation is essential for managing event flow and handling events effectively.

Event Phases:

Event propagation occurs in two phases: capturing phase and bubbling phase.

Capturing Phase:

During the capturing phase, the event starts at the root of the DOM tree and traverses through each parent element down to the target element.

Bubbling Phase:

In the bubbling phase, after reaching the target element, the event propagates back up the DOM tree, triggering event handlers on each ancestor element.

Example:

Let's consider the following HTML structure for the examples:

```
<div id="outer">
    <div id="middle">
      <div id="inner">
        Click Me
      </div>
    </div>
  </div>
```

a) Bubbling Example:

When a click event occurs on the innermost element, the event bubbles up through the parent elements. You can listen to the event at different levels of the DOM tree:

```
var outer = document.getElementById("outer");
var middle = document.getElementById("middle");
var inner = document.getElementById("inner");

outer.addEventListener("click", function() {
  console.log("Outer div clicked!");
});
```

```
middle.addEventListener("click", function() {
  console.log("Middle div clicked!");
});

inner.addEventListener("click", function() {
  console.log("Inner div clicked!");
});
```

If you click the "Click Me" text, the following output will be logged:

```
Inner div clicked!
Middle div clicked!
Outer div clicked!
```

b) Capturing Example:

You can also listen to events during the capturing phase by setting the third parameter of `addEventListener()` to `true`:

```
outer.addEventListener("click", function() {
  console.log("Outer div clicked during capturing phase!");
}, true);

middle.addEventListener("click", function() {
  console.log("Middle div clicked during capturing phase!");
}, true);
```

```
inner.addEventListener("click", function() {

  console.log("Inner div clicked during capturing phase!");

}, true);
```

If you click the "Click Me" text, the following output will be logged:

```
Outer div clicked during capturing phase!
Middle div clicked during capturing phase!
Inner div clicked during capturing phase!
```

Stop Propagation:

To prevent event propagation to further elements, you can use the stopPropagation() method. Here's an example:

```
inner.addEventListener("click", function(event) {
  event.stopPropagation();
  console.log("Inner div clicked! Event propagation stopped.");
});
```

With the `**stopPropagation()**` method, only the innermost element's event handler will execute. The output will be:

```
Inner div clicked! Event propagation stopped.
```

By utilizing capturing and bubbling phases and using the `**stopPropagation()**` method, you can control event flow and create more robust event handling mechanisms in your JavaScript applications.

Summarizing it

In this lecture, we have covered the following topics:

- API: We have covered Web API (accessing external services), including Browser API (built-in browser functionality) and third-party API.
- DOM: We have explored DOM representation (hierarchical tree structure) and the concept of the DOM tree.
- Selectors: We have learned about selectors and how to access specific elements using criteria such as ID, class, or tag name.
- DOM Manipulation: We have discussed creating, appending, and removing elements dynamically in the DOM.
- Event Listeners: We have covered event listeners and how to handle user/system events like clicks, mouse movements, or key presses.
- Event Propagation: We have learned about event propagation, including the capturing and bubbling phases in the DOM tree.

References

- Document Object Model: **Link**
- Events in JS: **Link**

Asynchronous JavaScript - I

Introduction

- JavaScript (JS) is a programming language that is known for being synchronous and single-threaded, which means it executes code line by line in a sequential manner.
- This synchronous nature can sometimes lead to blocking behavior, where subsequent lines of code have to wait for the completion of previous operations before executing.
- An example illustrating the blocking nature of JS is as follows:

```
console.log('Message before alert');
alert('blocking JS');
console.log('Message after alert');
```

In this example, when executed in a browser environment, the first `**console.log**` statement will be displayed in the console. However, once **`alert('blocking JS')`** is encountered, it blocks further execution until the user closes the alert dialog. As a result, the last `**console.log**` statement will not be executed until after closing the alert dialog.

Asynchronous Nature

- Although JavaScript is primarily synchronous and single-threaded, it also supports asynchronous behavior through mechanisms like the event loop.
- The event loop allows non-blocking execution by handling tasks asynchronously without interrupting other operations.

- Several APIs leverage this asynchronous capability in JavaScript:
    - `setTimeout`: Allows you to schedule a function to run after a specified delay.
    - `setInterval`: Executes a function repeatedly at defined intervals.
    - `fetch`: Enables making HTTP requests asynchronously.
- These APIs make use of callback functions or Promises/async-await syntax to handle asynchronous operations effectively while allowing other code to continue executing concurrently.
- The event loop ensures that such asynchronous tasks are queued and processed separately from regular synchronous execution. It monitors task completion and triggers their associated callbacks when ready.

As we delve deeper into concepts like Promises and async-await later in these notes, we'll explore how they further enhance JavaScript's ability to handle asynchronous programming efficiently.

setTimeout API

- The `setTimeout` API in JavaScript allows you to schedule the execution of a function after a specified delay.
- It takes two parameters: a callback function and the delay time in milliseconds. Additionally, it can accept extra arguments that will be passed to the callback function when it is invoked.
- **Syntax:**

```
setTimeout(callback, delay, arg1, arg2, ...);
```

    - **callback:** A function to be executed after the specified delay.
    - **delay:** The amount of time (in milliseconds) to wait before executing the callback function.

- **arg1, arg2, ...:** Extra arguments (optional) that will be passed as parameters to the callback function.

- **Example:**

```
function greet(name) {
  console.log(`Hello ${name}!`);
}

// Execute greet() with an argument after 2 seconds (2000 milliseconds)
const timeoutId = setTimeout(greet, 2000, "John");

// Cancel timeout before it executes
clearTimeout(timeoutId);
```

In this example, we have defined a **`greet`** function that takes a parameter **`name`**. By passing **`"John"`** as an additional argument in **`setTimeout`**, it gets passed into the **`greet`** function when it is called asynchronously after a delay of 2 seconds. However, if we want to cancel or stop this timeout from executing before its completion, we can use **`clearTimeout`** by passing in its timer ID (**`timeoutId`**) obtained from **`setTimeout`**.

setInterval API

- The **`setInterval`** API is similar to **`setTimeout`**, but instead of executing a callback once after a specific interval, it repeatedly executes the callback at defined intervals until cleared or stopped.

- Like **`setTimeout`**, it can also accept extra arguments for passing values into the callback.

- **Syntax:**

```
setInterval(callback, interval[, arg1[, arg2[, ...]]]);
```

  - **callback:** A function to be executed repeatedly at each interval.
  - **interval:** The time duration (in milliseconds) between each invocation of the callback.
  - **arg1, arg2, …:** Optional additional arguments that will be passed to the callback function.

- **Example:**

```
function greet(name) {
  console.log(`Hello ${name}!`);
}

// Greet with a name every second (1000 milliseconds)
const intervalId = setInterval(greet, 1000, "Alice");

// Stop greeting after 5 seconds
setTimeout(() => {
   clearInterval(intervalId); // Clearing interval using
clearInterval()
}, 5000);
```

In this example, we define a `greet` function that takes a `name` parameter. Using `setInterval`, we repeatedly call the `greet` function every second while passing `"Alice"` as an extra argument. After 5 seconds, we clear the interval using the timer ID obtained from `setInterval`. This prevents further execution of the scheduled callbacks.

Additional Methods: clearInterval and clearTimeout

To stop or cancel timeouts and intervals before they are executed or completed, JavaScript provides two methods:

- **clearTimeout:** This method cancels a timeout previously set by calling `setTimeout`. It takes one parameter, which is the timer ID returned by `setTimeout`.
- **clearInterval:** This method clears an interval set by calling `setInterval`. It also accepts one parameter - the timer ID returned by `setInterval`.

Asynchronous HTTP Requests

- In JavaScript, performing HTTP requests is a common requirement for interacting with web servers and retrieving data. With the introduction of asynchronous programming, developers can make HTTP requests in a non-blocking manner, allowing the rest of the program to continue executing while waiting for the response.
- There are several ways to accomplish this, including the older XMLHttpRequest (XHR) approach using callback functions, the modern Promise-based approach using the fetch API, and the more recent async-await syntax.

XMLHttpRequest (XHR)

The XMLHttpRequest (XHR) object is an older approach for making asynchronous HTTP requests from a web browser. It utilizes callback functions to handle different phases of the request-response cycle.

Note: Please remember that callbacks and their usage have been covered in Lecture 5. If you need further information on callbacks, please refer back to that lecture material.

XHR Phases:

1. Initiating an HTTP Request:

To initiate an HTTP request using XHR, follow these steps:

A. Create an instance of `XMLHttpRequest` using the `new` keyword:

```
const xhr = new XMLHttpRequest();
```

B. Open a connection with the server by specifying the method (`GET`, `POST`, etc.) and URL:

```
xhr.open('GET', 'https://api.example.com/data');
```

C. Set up a callback function to handle the response when it's received:

```
xhr.onload = function() {
  // Handle successful response here
  console.log(xhr.responseText);
};

xhr.onerror = function() {
  // Handle error here
  console.error('An error occurred.');
};

xhr.onprogress = function(event) {
   // Track progress of request if needed
   console.log(`Loaded ${event.loaded} bytes`);
 };

 // ... Other event handlers can be set as well ...
```

D. Send the request to the server:

```
xhr.send();
```

2. Handling Response Data:

When the response is received successfully, or if there's any error during transmission or processing, corresponding callback functions are invoked.

- The `onload` callback handles a successful response.
- The `onerror` callback handles errors such as network issues or failed connections.
- The `onprogress` callback can be used to track the progress of a request, like showing download progress.

Within these callbacks, you can access and process the response data using `xhr.responseText`.

Callback Hell

- Dealing with callbacks in XHR can lead to **"callback hell"** or **"pyramid of doom,"** where nested callbacks become difficult to manage and maintain code readability. This issue arises when multiple asynchronous operations are involved or when error handling becomes complex.
- To address this problem, JavaScript introduced Promises. Promises provide a cleaner way to handle asynchronous operations by utilizing chaining and avoiding excessive nesting of callbacks. Promises allow for better organization of code and more readable syntax.
- By using promises with modern approaches like fetch API or async-await syntax, developers can avoid callback hell and write more maintainable code that is easier to understand.

Summarizing it

In this lecture, we have covered the following topics:

- Introduction to synchronous and single-threaded JS: JavaScript is a synchronous and single-threaded language, meaning it processes tasks one at a time in a sequential manner.
- Asynchronous nature of JS: JavaScript also supports asynchronous behavior, allowing tasks to run independently without blocking the main thread.
- APIs like setTimeout and setInterval: These APIs are used to introduce delays and execute functions at specified time intervals, enabling asynchronous behavior in JavaScript programs.
- Asynchronous HTTP requests: JavaScript provides different methods for performing asynchronous HTTP requests, allowing communication with web servers and retrieval of data without blocking the execution.
- XMLHttpRequest (XHR) and its various phases: XHR is an older method for making HTTP requests in JavaScript, involving various phases like initiating the request, handling responses, and error handling.
- Callback Hell: Callback Hell refers to the issue of nesting multiple callback functions within one another, making the code complex and challenging to read and maintain.

References

- Asynchronous JavaScript: **Link**
- XMLHttpRequest: **Link**

Asynchronous JavaScript - II

As discussed earlier, the use of repetitive callbacks can lead to a phenomenon known as "**callback hell**" or the "**pyramid of doom.**" To overcome this issue, modern programming languages and frameworks introduced a powerful concept called Promises. In this section, we will delve into Promises and understand their purpose, states, and how they are created and utilized.

Promises

- Promises are objects that represent the eventual completion or failure of an asynchronous operation.
- They provide a cleaner and more organized way to handle asynchronous code, making it easier to read, write, and maintain.
- They represent the eventual completion or failure of an asynchronous operation and allow you to attach callbacks that will be executed when the operation is resolved or rejected.

States of Promises

Promises can be in one of three states:

- **Pending:** The initial state when a Promise is created, and its outcome is yet to be determined.
- **Fulfilled:** The Promise has successfully completed, and the associated value (result) is available.
- **Rejected:** The Promise encountered an error or failure, and the reason for the failure is available.

Promise Constructor

- To create a new Promise, we use the Promise constructor. The Promise constructor takes a single argument, which is a callback function with two parameters: **resolve** and **reject**.
    - The resolve function is used to fulfill the Promise and pass the resolved value.
    - The reject function is used to reject the Promise and pass the reason for rejection.

- Example:

```
const myPromise = new Promise((resolve, reject) => {
    // Perform an asynchronous task

    if (/* Task completed successfully */) {
      resolve('Task completed.'); // Resolve with a value
    } else {
      reject('Task failed.'); // Reject with a reason/error
    }
  });
```

In the example above, inside the callback function, you perform your desired asynchronous task. If it completes successfully, you call **resolve()** passing any relevant data as an argument (e.g., **`'Task completed.'`**). If there's an error or failure during execution, you call **reject()** passing an appropriate reason or error message (e.g., **`'Task failed.'`**).

Working with Promises:

Once created, promises expose several methods for handling their resolution and rejection states:

- **.then():** Attaches a callback function to be executed when the promise is fulfilled.
- **.catch():** Attaches a callback function to be executed when the promise is rejected.
- **.finally():** Attaches a callback function to be executed regardless of the promise's state (whether fulfilled or rejected).
- Example:

```
myPromise
  .then((result) => {
    console.log(result); // Handle successful fulfillment
  })
  .catch((error) => {
    console.error(error); // Handle rejection/error
  })
  .finally(() => {
    console.log('Task completed, regardless of outcome.'); //
Perform cleanup tasks if needed
});
```

In the example above, we use `.then()` to attach a success callback that will receive the resolved value as its argument. If there's an error during execution, `.catch()` handles it by receiving and processing the rejected reason. Finally, `.finally()` allows you to execute code that needs to run regardless of whether the promise was fulfilled or rejected.

Promises with fetch API

Promises are commonly used in conjunction with the Fetch API to handle asynchronous network requests and process their responses. The Fetch API provides a modern, promise-based approach for making HTTP requests.

Using Promises with Fetch:

- To make an HTTP request using the Fetch API, we create a Promise that resolves when the response is received successfully. We can then use `.then()` to handle the response data or perform further operations.
- Example:

```
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error(error);
});
```

In this example, `fetch()` sends an HTTP GET request to `'https://api.example.com/data'`. Inside the first .then(), we check if the server responded successfully (`response.ok`). If not, we throw an error. Otherwise, we parse the JSON response by calling `response.json()` and pass it along to our next `.then()`. Finally, any errors during execution are caught and handled by `.catch()`.

Promise Chaining

- Promises can be chained together by returning another promise within each `` `.then()` `` block. This allows us to perform sequential operations or transform data based on previous results.
- By chaining promises together, you can create more complex flows for handling asynchronous operations and avoid nested callbacks or "callback hell."
- Example

```
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    return response.json();
  })
  .then((data) => {
    // Perform additional processing or transformation of data
    const transformedData = processData(data);

    // Make another fetch request based on transformed data
    return fetch('https://api.example.com/another-endpoint', {
      method: 'POST',
      body: JSON.stringify(transformedData),
      headers: {
        'Content-Type': 'application/json'
      }
    });
  })
  .then((response) => {
     if (!response.ok) {
       throw new Error('Network response was not okay.');
```

```
      }

      console.log('Request completed successfully.');
    })
    .catch((error) => {
      console.error(error);
});
```

In this example, after the initial fetch request and parsing of the JSON response, we perform additional processing on the data. Then, within our second `.then()`, we make another fetch request to a different endpoint using `fetch()` again. We pass along the transformed data as the request payload in this case.

Async/Await

- Despite the introduction of Promises as a powerful mechanism to handle asynchronous operations in JavaScript, the code can still become complex and difficult to read when dealing with multiple asynchronous tasks.
- To address this, the Async/Await syntax was introduced, providing a more concise and synchronous-like way to write asynchronous code.
- Despite the existence of Promises, async/await has become increasingly popular due to its simplicity and improved code readability.

Necessity of Async/Await

- While Promises offer an elegant way to handle asynchronous operations, they can sometimes lead to complex chaining or nesting when multiple asynchronous tasks need to be executed sequentially. This can result in **"callback hell"** or hard-to-read code structures.
- Async/await addresses this issue by allowing developers to write asynchronous code using a more synchronous style, making it easier to understand and maintain.

Error Handling with Async/Await in JavaScript

- To use async/await, we mark a function as `async`, which automatically makes it return a Promise.
- Within an async function, we use the `await` keyword before calling any Promise-based functions or expressions.
- The `await` keyword pauses the execution of the function until the awaited Promise is resolved or rejected.
- Example:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');

    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    const data = await response.json();

    console.log(data);
  } catch (error) {
     console.error(error);
  }
}
```

In this example, we define an async function called `` `fetchData()` ``. Inside this function:

- We use **try-catch** blocks for proper error handling.
- We call `fetch()` with await keyword before it so that execution pauses until the HTTP request's response is received.

- If there are errors during fetching (e.g., network error), an exception will be thrown and caught by the surrounding catch block.
- Using the if statement, we check if the response received is not okay (e.g., a non-successful HTTP status code). If it's not okay, we throw an Error using the `throw` keyword. Throwing an Error in an async function will cause the Promise to be rejected.
- If the response is successful, we parse it using `` `response.json()` `` with await keyword before it to pause execution until the JSON data is extracted.

By using async/await, our code structure becomes much cleaner compared to traditional promise chains with `.then()` callbacks. It allows us to write asynchronous code in a more linear and sequential manner, resembling synchronous code.

Advantages of Async/Await over Promises:

1. **Readability:** Async/await provides a more intuitive and readable syntax for handling asynchronous operations, making the code easier to understand and maintain.
2. **Error Handling:** With async/await, error handling is simplified by using try-catch blocks instead of separate `.catch()` handlers for each Promise.
3. **Sequential Execution:** Async/await allows us to write asynchronous code that executes sequentially without deeply nested or chained callbacks, resulting in cleaner and more manageable code structures.
4. **Debugging:** Debugging async/await-based code is simpler since it closely resembles synchronous programming flow with step-by-step execution.

Event Loop

The event loop is a crucial component of JavaScript's concurrency model. It manages the execution order of asynchronous operations, ensuring that they are executed in an efficient and non-blocking manner.

Working of the JavaScript Program:

- **Initial Execution:**

  When a JavaScript program starts running, the main script is executed synchronously, line by line, pushing functions and their local variables onto the Call Stack.

- **Web API Interaction:**

  If an asynchronous task, like an API request, is encountered, it is handed over to the appropriate Web API. The JavaScript runtime continues executing subsequent tasks without waiting for the result.

- **Event Loop Process:**

  The Event Loop continuously checks the Call Stack. If it is empty, it looks into the Microtask Queue first, executing any pending microtasks until it is empty. This ensures timely execution of high-priority tasks.

- **Callback Queue Execution:**

  After processing the Microtask Queue, the Event Loop moves on to the Callback Queue. It takes the oldest task from the Callback Queue and pushes it onto the Call Stack for execution. This process repeats until the Callback Queue is empty.

To better understand the event loop, let's break it down into its key components as mentioned below:

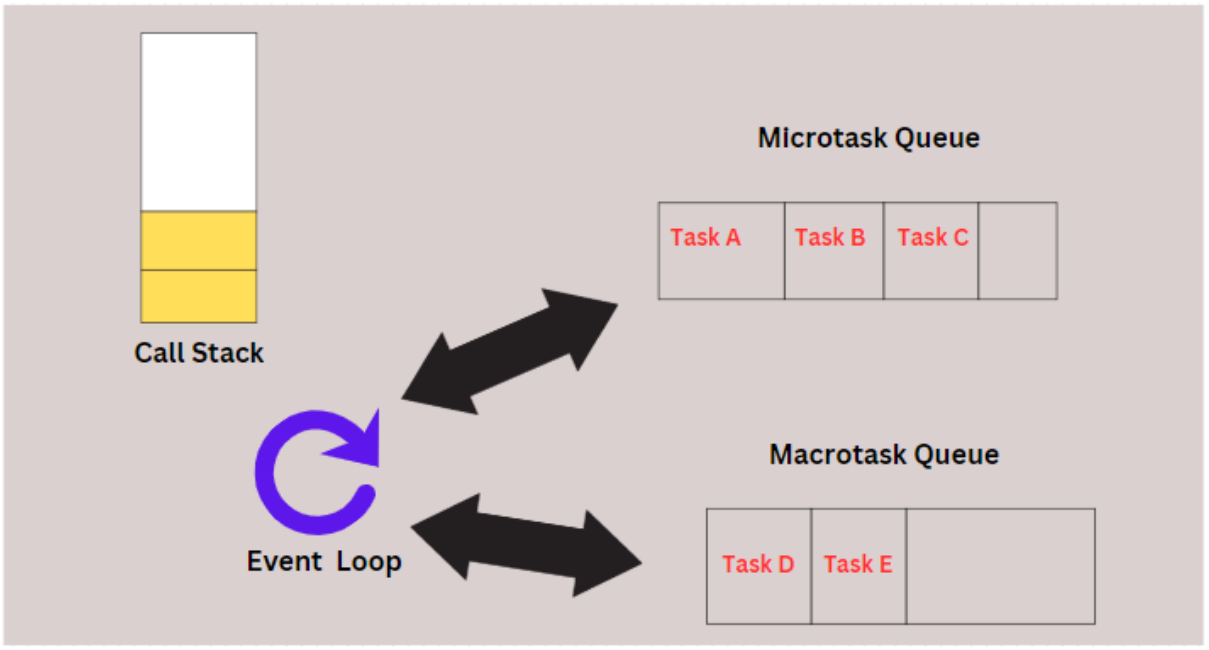


Figure:1

1. Call Stack:

The call stack is responsible for keeping track of function calls during program execution. When a function is called, it gets pushed onto the call stack, and when it returns, it gets popped off.

2. Web APIs:

Web APIs are provided by web browsers and allow us to perform various asynchronous tasks such as making HTTP requests (`fetch()`), setting timers (`setTimeout()`), or handling user events (`addEventListener()`). These tasks are performed outside the JavaScript runtime environment.

3. Callback Queue:

When an asynchronous task completes (e.g., timer expires or network request finishes), its associated callback function is placed in the callback queue.

4. Microtask Queue:

Microtasks have higher priority than regular callbacks and are used for tasks like Promise resolutions via `resolve()`, `reject()`, or using `async/await`. They get executed before regular callbacks from the callback queue.

Summarizing it

In this lecture, we have covered the following topics:

- Promises:
    - Covers the states of Promises (pending, fulfilled, rejected) and the Promise constructor.
    - Explains how to create Promises and introduces methods like resolve, reject, `.then()`, `.catch()`, and `.finally()`.
- Promises with Fetch API:
    - Demonstrates using Promises with the Fetch API to make asynchronous data requests.
    - Highlights how Promises simplify handling the response and error scenarios.
- Promise Chaining:
    - Explores Promise chaining, where multiple Promises can be sequenced and their results handled in a linear manner.
    - Illustrates how Promise chaining improves code organization and readability.
- Async/Await:
    - Introduces Async/Await, a more concise and synchronous-like syntax for handling asynchronous operations.
    - Describes how to mark functions as async and use the `await` keyword to pause execution until Promises resolve.
- Error Handling using Try/Catch:
    - Discusses error handling in Promises and Async/Await using the try/catch block.

- Event Loop and Queues:
    - Mentions the Event Loop's role in managing asynchronous tasks in JavaScript.
    - Describes the Callback Queue and Microtask Queue and the preference of Microtasks over Callbacks for priority execution.

References

- Promise: Link
- Async/await: Link

Modules in JS

Modules are a fundamental concept in modern JavaScript development, allowing developers to organize code into separate files or modules. Each module encapsulates its variables, functions, and classes, avoiding naming conflicts and polluting the global scope. This modularity promotes code reusability, maintainability, and scalability in large-scale projects.

Benefits of Using Modules

- **Encapsulation:** Modules enable encapsulation by providing a private scope for variables and functions. This prevents accidental variable overwriting and conflicts with other parts of the application.

- **Code Reusability:** Modules can be easily imported and used in different parts of the application, promoting code reusability and reducing duplication.

- **Dependency Management:** By explicitly defining dependencies between modules, developers can easily manage the order of execution and ensure the correct loading of files.

- **Readability and Maintainability:** Separating code into modules enhances readability and makes it easier to maintain and debug the application.

The problem in adding more than one JS file in HTML:

When working on large JavaScript projects, it's common to split the code into multiple files for better organization and maintainability. However, directly adding multiple JS files to an HTML document can cause issues like:

- **Global Scope Pollution:** All variables and functions declared in the files become part of the global scope, leading to potential naming conflicts and accidental overwriting of variables.

- **Order Dependencies:** If the files depend on each other, the order of inclusion becomes crucial, and it can be challenging to manage the correct sequence.

- **Script Loading and Performance:** Having many separate script tags can slow down the page loading process as each file requires a separate HTTP request.

IIFE (Immediately Invoked Function Expression) for Encapsulation:

IIFE is a design pattern used to encapsulate code and prevent it from polluting the global scope. It involves defining an anonymous function and immediately invoking it. The code inside the IIFE is contained within its own scope, ensuring that variables and functions declared inside the IIFE do not clash with global ones.

Named Exports:

Named exports allow you to selectively export multiple functions, variables, or classes from a module by giving each export a name. When importing named exports, you need to use the same name you specified during export.

Named Export Syntax:

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

Importing Named Exports:

```
// app.js
import { add, subtract } from './math.js';

console.log(add(5, 3)); // Output: 8
console.log(subtract(10, 4)); // Output: 6
```

Default Export:

Default exports allow you to export a single value or functionality as the "default" export from a module. Unlike named exports, you can choose any name while importing a default export.

Default Export Syntax:

```
// utility.js
const greet = (name) => `Hello, ${name}!`;
export default greet;
```

Importing Default Export:

```
// app.js
import greeting from './utility.js';

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

Combining Named and Default Exports:

A module can have both named exports and a default export together. In such cases, you can use a combination of named and default imports in the consuming file.

Example of Combined Exports:

```
// utility.js
const greet = (name) => `Hello, ${name}!`;
const farewell = (name) => `Goodbye, ${name}!`;

export default greet;
export { farewell };
```

Importing Combined Exports:

```
// app.js
import defaultGreeting, { farewell } from './utility.js';

console.log(defaultGreeting('Alice')); // Output: "Hello, Alice!"
console.log(farewell('Bob')); // Output: "Goodbye, Bob!"
```

Renaming Exports and Imports:

You can also rename imports and exports using the as keyword to use a different name locally while maintaining the original name from the module.

Example of Renaming Exports and Imports:

```
// math.js
const multiply = (a, b) => a * b;
```

```
const divide = (a, b) => a / b;

export { multiply as times, divide as quotient };
```

Importing with Renamed Exports:

```
// app.js
import { times, quotient } from './math.js';

console.log(times(5, 3)); // Output: 15
console.log(quotient(10, 2)); // Output: 5
```

Benefits of Using Named and Default Exports:

- Named exports provide a clear way to selectively import specific functionalities from a module, making the code more readable and maintainable.
- Default exports allow a single "main" value or functionality to be easily imported without the need for curly braces.
- Combining named and default exports provides flexibility when creating modules, allowing you to offer a primary functionality as a default export while providing additional utilities as named exports.
- Renaming exports and imports allows you to use more descriptive names or avoid naming conflicts in the consuming codebase.
- Importing everything with a namespace helps avoid potential naming conflicts when dealing with multiple modules.

References

- More about module Link

Topic : JQuery

1. JQUERY

JQuery is a lightweight JavaScript library. It contains a lot of methods that can do tasks requiring many lines of JavaScript code, in just a single line of code. Examples of such tasks are - removing element from HTML, adding element to HTML, etc.

The benefits of using jQuery can be summarized below -

- Accelerates the speed of development
- More work in less code
- Writing code is simpler and easy
- Provides a lot of methods and features
- Works around multiple browsers

Using jQuery does not mean that JavaScript is not required. JQuery is just a library build using JavaScript that provide functions for easy use. You still need to use JavaScript to create logic and to use it with these functions.

1.1. Fetch Elements

In JavaScript, we have different methods to fetch elements using an id or by class name. But JQuery makes fetching of elements very easy by using CSS selectors (just like the way we use in CSS).

The syntax for using jQuery is - `$(selector).action()`

$ - defines a ***function to access jQuery***
selector - ***CSS selector*** to query HTML elements
action - ***jQuery function*** to be executed on selected elements

We have some examples of fetching different elements -

- ➔ `$("p")` - this will return all the ***p tags***
- ➔ `$(".prim")` - this will return all the ***elements with class 'prim'***
- ➔ `$("#cringe")` - this will return the ***element with id 'cringe'***
- ➔ `$("span.light")` - this will return all the ***span elements with class 'light'***
- ➔ `$("form#subscribe")` - this will return the ***form element with id 'subscribe'***
- ➔ `$(".hide .del")` - this will return all the ***elements with class 'hide' and 'del'***

NOTE: *The above examples will fetch the jQuery object and not the DOM object.*

2. SOME PROPERTIES OF JQUERY

In this section, we will learn about some of the methods in jQuery. But let's first discuss about one very important jQuery function - **ready()**.

This method is used to ***prevent any code from running before the document is finished loading***. Any method that is written inside this method, then starts executing only after the document has finished loading. The way to use this method is -

```
$(document).ready(function() {
  // jQuery methods go here...
});
```

It is a good practice to wait for the document to be fully loaded before working on it. This makes it possible to use script tag in the head section of the HTML document. Another method for document ready event is -

```
$(function() {
  // jQuery methods go here...
});
```

Alternatively, you can also include Javascript code at the end of body tag. Basically when all your html content is done.

2.1. Hide Elements

The **hide()** method of jQuery is used to ***hide elements from the HTML page*****.** It actually makes the ***visibility of the element hidden***.

The syntax is - `$(selector).hide(speed, callback)`

speed - ***optional parameter*** to set speed of hiding having values - 'slow', 'fast' or milliseconds.
callback - ***optional parameter*** that runs a callback function after the hide() method completes.

Eg., the below code will hide all the p tags with class 'hide' in 1 second -

```
$("p.hide").hide(1000);
```

2.2. Remove Elements

The **remove()** method is used to ***remove elements from the HTML page***. It deletes the selected element and its child elements from the web page.

The syntax is - `$(selector).remove()`

Eg., the below code will remove all the p tags with class 'del' -

```
$("p.del").remove();
```

2.3. Get/Set Element Content

The **html()** method is used to ***set or return the content (i.e. innerHTML) of the selected elements***.

When this method is used to **return** content, it returns ***the content of the first matched element***. The syntax is - `$(selector).html()`

When this method is used to **set** content, it ***overwrites the content of all matched elements***. The syntax is - `$(selector).html(content)`

Eg., the below code will set the content of all the p tags with 'Hello world!' -

```
$("p").html("Hello <b>world</b>!");
```

2.4. Get/Set Element Text

The **text()** method is used to ***set or return the text content of the selected elements***.

When this method is used to **return** content, it returns ***the text content of all matched elements*** and the child HTML element will not be returned.

When this method is used to **set** content, it ***overwrites the content of all matched elements***. This will ***remove the child HTML elements*** of the selected elements.

Eg., the below code will set the text content of all the p tags with 'Hello world!' and remove any of its child elements-

```
$("p").text("Hello world!");
```

3. MODIFY CSS

Modifying the CSS has become very easy by using the jQuery. It provides a very easy method to add one or more styles to the selected elements.

JQuery provides **css()** method to ***change the CSS properties of element***.

To ***set a specific property*** use syntax - `css("propertyname","value")`

To ***set multiple properties*** use syntax -
`css({"propertyname":"value","propertyname":"value",...})`

Examples to modify the css properties are -

- ➔ `$("p").css("background-color", "red");`
- ➔ `$("p").css("backgroundColor", "red");`
- ➔ `$("p").css({"background-color": "red", "font-size": "30px"});`
- ➔ `$("p").css({"backgroundColor": "red", "fontSize": "30px"});`

JQuery allows to have ***both type of format for multiple-word properties***. It sets the same property for both of these multi-word format.

3.1. addClass() method

The **addClass()** method is used to ***add class attributes to the selected elements***. It allows you to add multiple classes to the elements as well.

Examples are -

- ➔ `$("span, b, u").addClass("blue");` - it will add class 'blue' to all the selected elements
- ➔ `$("img").addClass("collage");` - it will add class 'collage' to all the image elements
- ➔ `$("#imp").addClass("important blue");` - it will add 'important' and 'blue' classes to the element with id 'imp'

3.2. removeClass() method

The **removeClass()** method is used to ***remove one or more classes from the selected elements***.

Examples are -

- ➔ `$("span, b, u").removeClass("blue");` - it will remove class 'blue' to all the selected elements
- ➔ `$("img").removeClass("collage");` - it will remove class 'collage' to all the image elements
- ➔ `$("#imp").removeClass("important blue");` - it will remove 'important' and 'blue' classes to the element with id 'imp'

3.3. toggleClass() method

The **toggleClass()** method is used to ***toggle between adding and removing classes from the selected elements***. If the class is not present it will be added first and if it is present then it will be removed first.

Examples are -

- ➔ `$("span, b, u").toggleClass("blue");` - it will toggle the class 'blue' to all the selected elements
- ➔ `$("img").toggleClass("collage");` - it will toggle the class 'collage' to all the image elements
- ➔ `$("#imp").toggleClass("important blue");` - it will toggle 'important' and 'blue' classes separately for the element with id 'imp'

4. EVENT HANDLING

JQuery has also made adding events to the elements easy. The syntax for jQuery event methods are somewhat similar to those of JavaScript.

All the JavaScript event methods are also written in jQuery. Some of them are -

- ***click()***
- ***mouseenter()***
- ***mousedown()***
- ***hover()***
- ***keyup()***

The syntax to add events to the element(s) is -

```
$(selector).event(function(e) {
    // Write something here
});
```

Ex., the below code will ***add a click event to a button with id 'submit-page'*** -

```
$("#submit-page").click(function() {
    alert("Form has been submitted");
});
```

Just like JavaScript, there is also another method to add events to the elements, i.e. by using the 'on' method. The syntax for it is -

```
$(selector).on(event, function(e) {
    // Write something here
});
```

Eg., you can modify the above code as -

```
$("#submit-page").on("click", function() {
    alert("Form has been submitted");
});
```