JavaScript Fundamentals - Organized Notes

COURSE INFORMATION

RQF LEVEL 3
SWDJF301 SOFTWARE DEVELOPMENT
JavaScript Fundamentals
TRAINEE'S MANUAL
October, 2024

MODULE CODE AND TITLE: SWDJF301 JAVASCRIPT FUNDAMENTALS

Learning Outcome 1: Apply JavaScript Basic Concepts

JAVASCRIPT OVERVIEW

Applications of JavaScript

Web Development

JavaScript is a scripting language used to develop web pages. Some famous websites built by the use of JavaScript are Google, YouTube, Facebook, Wikipedia, Yahoo, Amazon, eBay, Twitter, and LinkedIn, to name a few.

Presentations

A very popular application of JavaScript is to create interactive presentations as websites. The RevealJs and BespokeJs libraries can be used to generate web-based slide decks using HTML.

Server Applications

JavaScript is also used to write server-side software through Node.js open-source runtime environment. Developers can write, test and debug code for fast and scalable network applications.

Web Servers

Node.js is a powerful JavaScript runtime that enables developers to build scalable and efficient web servers.

Games

Creating games on the web is another important one among applications of JavaScript. The combination of JavaScript and HTML5 plays a major role in games development using JS. The EaselJS library provides rich graphics for games.

Art

A recent feature of HTML5 in JavaScript is the canvas element, which allows drawing 2D and 3D graphics easily on a web page.

Smartwatch Apps

Pebble.js is a JavaScript framework by Pebble, allowing developers to create applications for Pebble watches using JavaScript. Create a smartwatch app with simple JavaScript code.

Mobile Apps

One of the most powerful applications of JavaScript is to create apps for non-web contexts, meaning for things, not on the Internet. With the use of mobile devices at an all-time high, JavaScript frameworks have been designed to facilitate mobile app development across various platforms like IOS, Android, and Windows.

React Native framework allows cross-platform mobile app building, where developers can use a universal front end for Android and IOS platforms.

Flying Robots

Last but not least, you can use JavaScript to program a flying robot. With the Node.js ecosystem, users can control numerous small robots, creative maker projects, and IoT devices.

DEVELOPMENT TOOLS

VS Code & node.js

Visual Studio Code is a free, lightweight but powerful source code editor that runs on your desktop and on the web and is available for Windows, macOS, Linux, and Raspberry Pi OS.

Visual Studio Code(VS Code) has support for many languages, including Python, Java, C++, JavaScript, and more.

Node.js (Node) is an open source, cross-platform runtime environment for executing JavaScript code.

Node is used extensively for server-side programming, making it possible for developers to use JavaScript for client-side and server-side code without needing to learn an additional language.

JAVASCRIPT KEY CONCEPTS

Variable

A JavaScript variable is simply a name of a storage location.

NOTE: Variables are classified into Global variables and Local variables based on their scope.

The main difference between Global and local variables is that global variables can be accessed globally in the entire program, whereas local variables can be accessed only within the function or block in which they are defined.

JavaScript Identifiers

An identifier is a sequence of characters in the code that identifies a variable, function, or property.

An identifier is simply a name. In JavaScript, identifiers are used to name constants, variables, properties, functions, and classes and to provide labels for certain loops in JavaScript code.

Data Types

Data types describe the different types or kinds of data that we're going to be working with and storing in variables.

JavaScript provides different data types to hold different types of values. There are two types of data types in JavaScript and these are Primitive data type and Non-primitive (reference) data type.

Primitive data types

The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types. There are five types of primitive data types in JavaScript. They are as follows:

DATA TYPE	DESCRIPTION
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all

The following examples illustrate the use primitive data types in JavaScript:

Number: Number data type in JavaScript can be used to hold decimal values as well as values without decimals. Examples: a) let x = 250; b) let y = 40.5;

String: The string data type in JavaScript represents a sequence of characters that are surrounded by single or double quotes. Example: let str = 'Hello';

Undefined: The meaning of undefined is 'value is not assigned'.

Boolean: The Boolean data type can accept only two values i.e. true and false.

Null: This data type can hold only one possible value that is null. Example: let x = null;

Non-primitive data types

These data types that are derived from primitive data types of the JavaScript language. There are also known as derived data types or reference data types. The non-primitive data types are as follows:

DATA TYPE	DESCRIPTION
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

Examples below explain the use of non-primitive data types in program:

Object: Object in JavaScript is an entity having properties and methods. Everything is an object in JavaScript.

How to create an object in JavaScript:

Using Constructor Function to define an object:

```
// Create an empty generic object
var obj = new Object();
// Create a user defined object
var mycar = new Car();
```

Using Literal notations to define an object:

```
// An empty object
var squar\\\\\\\\\\\\\\\\\\\\\\e = {};
// Here a and b are keys and 20 and 30 are values
var circle = {a: 20, b: 30};
```

Array: With the help of an array, we can store more than one element under a single name.

Ways to declare a single dimensional array:

```
// Call it with no arguments
var a = new Array();
// Call it with single numeric argument
var b = new Array(10);
// Explicitly specify two or more array elements
var d = new Array(1, 2, 3, "Hello");
```

Values

A value is the representation of some entity that can be manipulated by a program. The members of a type are the values of that type. The "value of a variable" is given by the corresponding mapping in the environment.

JavaScript values are the values that comprise values like Booleans, Strings, arrays, Numbers, etc.

Operators

In JavaScript, an operator is a special symbol used to perform operations on operands (values and variables). For example, 2 + 3.

Expressions

JavaScript's expression is a valid set of literals, variables, operators, and expressions that evaluate to a single value.

Keywords

In JavaScript, keywords are reserved words that have a specific purpose (meaning) and are already defined in the language.

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

| Abstract | Else | Instanceof | Switch |

| Boolean | Enum | Int | Synchronised |

| Break | Export | Interface | This |

| Byte | Extends | Long | Throw |

| Case | False | Native | Throws |

| Catch | Final | New | Transient |

| Char | Finally | Null | True |

| Class | Float | Package | Try |

| Const | For | Private | Typeof |

| Continue | Function | Protected | Var |

| Debugger | Goto | Public | Void |

| Default | If | Return | Volatile |

| Delete | Implements | Short | While |

| Do | Import | Static | With |

| Double | In | Super | |

Comments

The JavaScript comments can be used to explain JavaScript code, and to make it more readable.

It is used to add information about the code, warnings or suggestions so that end users can easily interpret the code. The comments are ignored by the JavaScript engine.

Advantages of JavaScript comments

There are mainly two advantages of JavaScript comments.

1. To make code easy to understand It can be used to elaborate the code so that end users can easily understand the code.
2. To avoid the unnecessary code It can also be used to avoid the code being executed. Sometimes, we add the code to perform some action. But after sometime, there may be a need to disable the code. In such cases, it is better to use comments.

Types of JavaScript Comments

There are two types of comments in JavaScript.

- Single-line Comment
- Multi-line Comment

Single Line Comments

Single line comments start with //. Any text between // and the end of the line will be ignored by JavaScript (will not be executed). Example:

```
//Declaration of a variable x
var x;
```

Multi-line Comments

Multi-line comments start with /* and end with /. *Any text between* / and */ will be ignored by JavaScript. Example:

```
result=6*5;
/* multiply two numbers and store the 30 output in a variable called result*/
```

JAVASCRIPT LIBRARIES AND FRAMEWORKS

JavaScript libraries

JavaScript libraries is a file that contains a set of prewritten functions or codes that you can repeatedly use while executing JavaScript tasks.

We have 3 types of JavaScript libraries to discuss on:

React Javascript

React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components.

JQuery

It's a library of JavaScript functions that make it easy for web page developers to do common tasks like manipulating the webpage, responding to user events, getting data from their servers, building effects and animations, and much more.

Three JavaScript

Three.js is a cross-browser JavaScript library and application programming interface (API) used to create and display animated 3D computer graphics in a web.

JavaScript frameworks

A JavaScript framework is a set of JavaScript code libraries that provide pre-written code for everyday programming tasks to a web developer.

Vue JavaScript

Vue.js is an open-source progressive JavaScript framework used to develop interactive web user interfaces and single-page applications (SPAs).

Angular JavaScript

AngularJS is a client-side JavaScript MVC framework to develop a dynamic web application.

Express JavaScript

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

JAVASCRIPT RUNTIME ENVIRONMENT

The JavaScript runtime environment provides access to built-in libraries and objects that are available to a program so that it can interact with the outside world and make the code work.

Node JavaScript

is an open-source, server-side JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser. It provides a platform for building scalable, networked applications and is widely used for server-side and command-line scripting.

V8 Engine

is an open-source JavaScript engine developed by Google. It is primarily used to execute JavaScript code in web browsers, but it is also the underlying runtime for the Node.js server-side JavaScript environment.

The V8 engine is written in C++ and is designed to optimize the execution of JavaScript code for performance and efficiency.

Simply, A JavaScript engine is a software component that executes JavaScript code.

JAVASCRIPT VERSIONS

JavaScript was invented by Brendan Eich in 1995 and became an ECMA standard in 1997.

ECMAScript is the official name of the language.

ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6. Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

PRACTICAL ACTIVITIES

Practical Activity 1.1.2: Installation of node.js and VS Code

After that, click on the install button

Finally, after installation completes, click on the finish button, and the visual studio code will open.

Select "Next"

The installer may prompt you to "install tools for native modules".

Do not close or cancel the installer until the install is complete

Complete the Node.js Setup Wizard. By Clicking on "Finish"

Verify that Node.js was properly installed or not.

JAVASCRIPT INTEGRATION WITH HTML

1.2.1: Description of JavaScript integration to HTML

When working with files for the web, JavaScript needs to be loaded and run alongside HTML markup. This can be done either inline within an HTML document or in a separate file that the browser will download alongside the HTML document.

✓Referencing HTML to JavaScript

Referencing HTML elements in JavaScript refers to the process of accessing and manipulating HTML elements within JavaScript code. It allows developers to interact with the structure, content, and behavior of HTML elements dynamically.

To include an external JavaScript file, we can use the script tag with the attribute src. The src attribute spec u ifies the URL of an external script file.

If you want to run the same JavaScript on several pages in a web site, you should create an external JavaScript file, instead of writing the same script over and over again. Save the script file with a .js extension and then refer to it using the src attribute in the <script> tag. Note: The external script file cannot contain the <script> tag.

✓ <script> tag

The **<script>** is an HTML element used to embed or reference JavaScript code within an HTML document. It allows developers to include JavaScript code directly within the HTML file or link to an external JavaScript file.

The <script> tag is used to embed a client-side script (JavaScript). Example:

```
<script type="text/javascript">
document.write("PHPTPOINT is the place to study javascript easily");
</script>
```

Using the script tag signifies that we are using JavaScript

✓External JavaScript

External JavaScript refers to JavaScript code that is stored in a separate file with a **.js** extension and linked to an HTML document using the **<script>** tag's **src** attribute. Instead of embedding the JavaScript code directly within the HTML file, it is stored in an external file, which is then referenced by the HTML document.

With JavaScript, an external JavaScript file can be created and can easily be embedded into many HTML pages. Since a single JavaScript file can be used in various HTML pages hence, it provides code reusability.

In order to save a JavaScript file, .js extension must be used. To increase the speed of the webpages, it is recommended to embed the entire JavaScript file into a single file.

✓ Using external Javascript reference (CDN)

Using an external JavaScript reference through a Content Delivery Network (CDN) involves linking to a JavaScript file hosted on a remote server or network of servers. CDNs are designed to deliver content, including JavaScript files, to users efficiently and reliably. Instead

of hosting the JavaScript file on your own server, you can reference a CDN-hosted file in your HTML document.

What is a CDN? A CDN is a Content Delivery Network. These are file hosting services for multiple versions of common libraries.

You can use external JavaScript references, often referred to as Content Delivery Network (CDN) links, to include JavaScript libraries or scripts in your HTML documents. This is a common practice to save bandwidth and improve loading times. Here's how you can do it:

1. Find the CDN link for the JavaScript library you want to include. Many popular libraries like jQuery, Bootstrap, or React have CDN links available. These links are hosted on servers provided by various content delivery networks.
2. In your HTML document, add a <script> tag with the src attribute set to the CDN link.

By using CDN links, you can easily include external JavaScript libraries in your web pages without having to host the library files on your server, and your users will benefit from faster loading times since CDNs are designed for efficient content delivery.

✓ JavaScript output

JavaScript output refers to the process of displaying or presenting information, data, or results generated by JavaScript code. JavaScript provides various methods and techniques for outputting data, allowing developers to communicate with users or display information on web pages.

JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

1.2.2: Integration of JavaScript to HTML

To integrate JavaScript into HTML by referencing HTML elements in JavaScript, you can follow these steps:

1. Identify the HTML element(s): Determine which HTML element(s) you want to reference in your JavaScript code. This could be an element with a specific ID, class, tag name, or any other attribute that uniquely identifies the element(s).
2. Choose a referencing method: Select the appropriate method for referencing the HTML element(s) in JavaScript.
3. Write the JavaScript code: In your JavaScript code, use the chosen referencing method to access the desired HTML element(s).
4. Perform desired operations: Once you have referenced the HTML element(s) in JavaScript, you can perform various operations on them. This could include modifying

their content, changing their attributes, adding event listeners, manipulating their styling, or interacting with their behavior.

5. Ensure proper execution order: Make sure that your JavaScript code is executed after the HTML elements have been loaded.
6. Test and debug: Test your JavaScript code to ensure that it correctly references and interacts with the desired HTML element(s).

By following these steps, you can effectively reference HTML elements in JavaScript and integrate them into your HTML document. This allows you to manipulate and interact with the elements dynamically, creating interactive and responsive web pages.

Example: ✓ Referencing HTML to JavaScript

Open your HTML file in a text editor or an integrated development environment (IDE).

Locate the part of the HTML file where you want to include your JavaScript code. This can be within the <head> section or the <body> section, depending on your requirements.

To reference an external JavaScript file, you can use the <script> tag. Place the following code within the <head> section or at the end of the <body> section:

```
<script src="path/to/your/javascript-file.js"></script>
```

Replace **"path/to/your/javascript-file.js"** with the actual file path of your JavaScript file. Make sure to provide the correct file path relative to your HTML file.

If you want to include JavaScript code directly within your HTML file, you can use the <script> tag and place the code between the opening and closing tags. For example:

```
<script>
// Your JavaScript code goes here
</script>
```

You can also use the type attribute within the <script> tag to specify the scripting language. However, for JavaScript, this attribute is not necessary as JavaScript is the default scripting language. For example:

```
<script type="text/javascript">
// Your JavaScript code goes here
</script>
```

Save your HTML file with the changes.

By following these steps, you have successfully integrated JavaScript into your HTML file using referencing HTML tags.

The JavaScript code will be executed when the HTML file is loaded in a web browser, allowing you to add interactivity and dynamic functionality to your web page.

Example:

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

```
<body>
<h1>The script src attribute</h1>
<script src="demo_script_src.js">
</script>
</body>
</html>
```

Using <script> tags

To integrate JavaScript into HTML using <script> tags, you can follow these steps:

1. Create an HTML file: Start by creating an HTML file using a text editor or an integrated development environment (IDE).
2. Set up the HTML structure: Define the structure of your HTML document by adding HTML tags such as <html>, <head>, and <body>. This is where you'll integrate your JavaScript code.
3. Add the <script> tag: Inside the <body> or <head> section of your HTML file, add the <script> tag to include your JavaScript code. You can either write the JavaScript directly within the <script> tags or reference an external JavaScript file using the src attribute.

Inline JavaScript: To write JavaScript code directly within the **<script>** tags, place your code between the opening and closing **<script>** tags. For example:

```
<script>
// Your JavaScript code goes here
</script>
```

External JavaScript file: To reference an external JavaScript file, use the **src** attribute within the **<script>** tag and provide the path or URL to the JavaScript file. For example:

```
<script src="script.js"> </script>
```

4. Place the <script> tag appropriately: Decide where to place the <script> tag based on your requirements. Placing it in the <head> section allows the JavaScript code to load before the HTML content, while placing it at the end of the **<body>** section ensures that the HTML content is loaded before executing the JavaScript code.
5. Test and debug: Save your HTML file and open it in a web browser. Use the browser's developer tools to check for any errors in the JavaScript code and debug as needed. This will help ensure that your JavaScript code is properly integrated and functioning as expected.

By following these steps, you can integrate JavaScript into HTML using **<script>** tags and leverage its capabilities to create interactive and dynamic web pages.

Here is an example to understand this from an initial level:

JavaScript Codes between the head tag

Example1.

```
<!DOCTYPE html>
<html>
<head>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Paragraph changed.";
</script>
</head>
<body>
<h2>Demo JavaScript in Head</h2>
</body>
</html>
```

Example2.

```
<!DOCTYPE html>
<html>
<head>
<title> page title</title>
<script>
document.write("Welcome to Javatpoint");
</script>
</head>
<body>
<p>In this example we saw how to add JavaScript in the head section </p>
</body>
</html>
```

JavaScript codes between the Body Tags

Example1.

```
<html>
<body>
<script type="text/javascript">
document.write("JavaScript is a simple language for javatpoint learners");
</script>
</body>
</html>
```

Example2.

```
<!DOCTYPE html>
<html>
<head>
<title> page title</title>
</head>
<body>
<script>
document.write("Welcome to Javatpoint");
</script>
```

```
<p> In this example we saw how to add JavaScript in the body section </p>
</body>
</html>
```

Using external JavaScript

To integrate external JavaScript into HTML, you can follow these steps:

1. Choose a JavaScript file: Select the external JavaScript file that you want to integrate into your HTML document. This could be a file you have written yourself or a library/framework file from a trusted source.
2. Obtain the file's URL or path: Ensure that you have the URL or local path to the external JavaScript file. If the file is hosted on a Content Delivery Network (CDN), you can usually find the URL from the CDN provider's documentation.
3. Link to the JavaScript file: In your HTML file, add a **<script>** tag to link to the external JavaScript file. Use the src attribute and provide the URL or path to the JavaScript file. For Example:

```
<script src="path/to/external.js"> </script>
```

4. Place the **<script>** tag appropriately: Decide where to place the **<script>** tag based on your requirements. It is common to place the **<script>** tag just before the closing **</body>** tag. This ensures that the JavaScript file is loaded after the HTML content, improving page loading performance. However, you can also place it in the **<head>** section if necessary.
5. Test and debug: Save your HTML file and open it in a web browser. Use the browser's developer tools to check for any errors in the JavaScript code and debug as needed. Ensure that the external JavaScript file is being loaded correctly and that it functions as expected.
6. Utilize the JavaScript file: Once the external JavaScript file is successfully integrated, you can use its functions, methods, or variables within your HTML document. Follow the documentation or guidelines provided by the JavaScript file's source to utilize its features effectively.

By following these steps, you can integrate external JavaScript into your HTML document, allowing you to leverage the functionality and capabilities provided by the JavaScript file.

Here is a simple example: Let's include the JavaScript file into the html page. It calls the JavaScript function on button click.

index.html file

```
<html>
<head>
<script type="text/javascript" src="message.js"></script>
</head>
<body>
<p>Welcome to JavaScript</p>
```

```
<form>
<input type="button" value="click" onclick="msg()"/>
</form>
</body>
</html>
```

Using external JavaScript reference (CDN)

To integrate JavaScript into HTML using an external JavaScript reference (CDN), you can follow these steps:

1. Choose a JavaScript library or framework: Select the JavaScript library or framework that you want to integrate into your HTML document. Common examples include jQuery, React, Vue.js, or Bootstrap.
2. Find the CDN URL: Locate the CDN (Content Delivery Network) URL for the chosen JavaScript library or framework. CDN providers like Google, Microsoft, or Cloudflare often host popular JavaScript libraries and offer CDN URLs for easy integration.
3. Link to the CDN URL: In your HTML file, add a <script> tag to link to the CDN URL. Use the src attribute and provide the CDN URL for the JavaScript library or framework. For Example:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js">
</script>
```

4. Place the <script> tag appropriately: Decide where to place the **<script>** tag based on your requirements. It is common to place the <script> tag just before the closing </body> tag. This ensures that the JavaScript library is loaded after the HTML content, improving page loading performance. However, you can also place it in the <head> section if necessary.
5. Test and debug: Save your HTML file and open it in a web browser. Use the browser's developer tools to check for any errors in the JavaScript code and debug as needed. Ensure that the external JavaScript library is being loaded correctly and that it functions as expected.
6. Utilize the JavaScript library: Once the external JavaScript library is successfully integrated, you can use its functions, methods, or components within your HTML document. Follow the documentation or guidelines provided by the JavaScript library's source to utilize its features effectively.

Here is a simple example:

```
<!DOCTYPE html>
<html>
<head>
<title>Using CDN for JavaScript</title>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
</head>
<body>
<!-- Your HTML content here -->
```

```
</body>
</html>
```

✓JavaScript output

To generate JavaScript output, you can follow these steps:

1. Open a text editor or an integrated development environment (IDE) to write your JavaScript code.
2. Start by defining the desired output or the logic that generates the output. This could involve performing calculations, manipulating data, or retrieving information from user input.
3. Use JavaScript's built-in functions, operators, and syntax to write the code that generates the desired output. This may include variables, conditional statements (if/else), loops (for/while), arrays, functions, and more.
4. Within your JavaScript code, use the **console.log()** function to output text or values to the console. For example:

```
console.log("Hello, world!");
```

   This will display the text "Hello, world!" in the console when the JavaScript code is executed.

5. Alternatively, you can output the result to the HTML document itself by manipulating the HTML elements using JavaScript. Here's an example:

```
document.getElementById("output").innerHTML = "Hello, world!";
```

6. In this case, you need to have an HTML element with the **id** attribute set to "output" to display the output.
7. Open your HTML file in a web browser, or run the JavaScript file using Node.js, and check the console or the HTML output element to view the generated output.

★ Using innerHTML

To generate JavaScript output using the innerHTML property, you can follow these steps:

1. Identify the HTML element: Determine the HTML element where you want to generate the JavaScript output. This could be a ** **, ****, **<p>**, or any other element that can contain content.
2. Reference the HTML element in JavaScript: Use JavaScript to reference the desired HTML element using methods like **getElementById**, **querySelector**, or any other appropriate method. For example:

```
const outputElement = document.getElementById('output');
```

3. Generate the JavaScript output: Use the **innerHTML** property of the referenced HTML element to generate the desired JavaScript output. Assign the desired content, including HTML tags, to the **innerHTML** property. For example:

outputElement.innerHTML = 'Hello, JavaScript!';

4. Test and observe the output: Save your HTML file and open it in a web browser. Check the specified HTML element to see the generated JavaScript output. In this case, the output would be "Hello, JavaScript!" with the word "JavaScript" displayed in bold.

To access an HTML element, JavaScript can use the document.getElementById(id) method. The id attribute defines the HTML element. The innerHTML property defines the HTML content:

Example:

```
<Html>
<Head>
<Body>
<h1>My First Web Page</h1>
<p>My First Paragraph</p>
<p id="demo"></p>
<script> document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

★ Using document.write()

To generate JavaScript output using the **document.write()** method, you can follow these steps:

1. Identify the location for the JavaScript output: Determine where you want the JavaScript output to appear in your HTML document. This could be within the **<body>** section or within a specific HTML element.

2. Write the JavaScript code: Write your JavaScript code that will generate the desired output. Use the **document.write()** method to output content directly to the HTML document. For example:

```
document.write("Hello, JavaScript!");
```

3. Place the JavaScript code appropriately: Decide where to place the JavaScript code based on your requirements. You can place it directly within a <script> tag in the <head> section or at the end of the **<body>** section.

4. Test and observe the output: Save your HTML file and open it in a web browser. The JavaScript code will be executed, and the output will be displayed at the specified location. In this case, the output would be "Hello, JavaScript!"

By following these steps, you can generate JavaScript output using the **document.write()** method.

For testing purposes, it is convenient to use document.write():

Example: 1

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

```
<h1>My First Web Page</h1>
<p>My first paragraph. </p>
<script>
document.write(5 + 6);
</script>
</body>
</html>
```

Example: 2

```
<! DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<button type="button" onclick="document.write(5 + 6)">Try it</button>
</body>
</html>
```

★ Using window.alert()

To generate JavaScript output using the **window.alert()** method, you can follow these steps:

1. Determine the message to display: Decide on the message or content that you want to display as the JavaScript output using the **window.alert()** method. This could be a notification, a prompt, or any other information you want to present to the user.
2. Write the JavaScript code: Write your JavaScript code that includes the **window.alert()** method. Provide the desired message as a string parameter within the parentheses. For example:

```
window.alert("Hello, JavaScript!");
```

3. Place the JavaScript code appropriately: Decide where to place the JavaScript code based on your requirements. You can place it directly within a **<script>** tag in the **<head>** section or at the end of the **<body>** section.
4. Test and observe the output: Save your HTML file and open it in a web browser. When the JavaScript code is executed, a pop-up alert window will appear displaying the specified message. In this case, the output would be an alert window displaying "Hello, JavaScript!".
5. Customize the alert message: You can modify the message within the window.alert() method to display different content or variable values. You can concatenate strings or include variables within the message for dynamic output.

You can use an alert box to display data:

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
```

```
<p>My first paragraph.</p>
<script> alert(5 + 6);
</script>
</body>
</html>
```

★ Using console.log()

To generate JavaScript output using the **console.log()** method, you can follow these steps:

1. Determine the message to display: Decide on the message or content that you want to display as the JavaScript output using the console.log() method. This could be a string, variable values, or any other information you want to log for debugging or informational purposes.
2. Write the JavaScript code: Write your JavaScript code that includes the console.log() method. Provide the desired message or variable as a parameter within the parentheses. For example:

```
console.log("Hello, JavaScript!");
```

3. Place the JavaScript code appropriately: Decide where to place the JavaScript code based on your requirements. You can place it directly within a **<script>** tag in the **<head>** section or at the end of the **<body>** section.
4. Test and observe the output: Save your HTML file and open it in a web browser. Open the browser's developer tools and navigate to the console tab. When the JavaScript code is executed, the specified message or variable value will be logged to the console. In this case, the output would be a log entry displaying "Hello, JavaScript!".
5. Customize the log message: You can modify the message within the console.log() method to display different content or variable values. You can concatenate strings or include variables within the message for dynamic output.

By following these steps, you can generate JavaScript output using the **console.log()** method. This method is commonly used for debugging and informational purposes, allowing you to log messages and variable values to the browser's console for analysis and troubleshooting.

For debugging purposes, you can call the console.log() method in the browser to display data.
Example:

```
<! DOCTYPE html>
<html>
<body>
<script> console.log(5 + 6);
</script>
</body>
</html>
```

★ JavaScript Print

To generate JavaScript output for printing purposes, you can follow these steps:

1. Identify the content to be printed: Determine the specific content that you want to generate as output for printing. This could be text, HTML elements, or a combination of both.
2. Create a print function: Write a JavaScript function that will handle the printing process. This function will be responsible for generating the output and initiating the print dialog. For example:

```
function printContent() {
  var content = "Hello, JavaScript!";
  var printWindow = window.open('', '_blank');
  printWindow.document.write(content);
  printWindow.document.close();
  printWindow.print();
}
```

1. Call the print function: Invoke the print function when you want to generate the output for printing. You can trigger the function through a button click, a specific event, or any other appropriate mechanism.
2. Test and observe the output: Save your HTML file and open it in a web browser. Trigger the print function and observe the output. In this case, a new window will open with the content "Hello, JavaScript!" and the print dialog will appear, allowing the user to print the content.
3. Customize the content: Modify the content variable in the print function to include the specific content you want to print. You can dynamically generate the content.

Points to Remember

Integrating JavaScript with HTML allows developers to enhance the interactivity and functionality of web pages. When working with files for the web, JavaScript needs to be loaded and run alongside HTML markup. Some ways of integrating JavaScript to HTML are: Using script tag, referencing HTML, external JavaScript reference (CDN) and external JavaScript. To integrate JavaScript into HTML you have to identify the HTML element(s) and write JavaScript codes to perform desired operations.

Generating JavaScript output allows developers to display information, results, or interact with users in various ways.

To generate JavaScript output, you can follow these steps:

1. After defining the desired output
2. Use one of JavaScript's built-in functions to display output.
3. Save your JavaScript file with a **.html** extension
4. Open your HTML file in a web browser

VARIABLES IN JAVASCRIPT

1.3.1: Description of variables in JAVASCRIPT

Let keyword:

It is used to declare variable locally.

Syntax: let variableName;

Example: let age;

Const keyword:

It is used to declare variable locally.

Syntax: const variableName= initial value;

Example: const age=15;

Note: You can declare a variable with let and var without a value. In this case, the default value will be undefined.

Examples:

```
var tomato;
let potato;
console.log(tomato); // undefined
console.log(potato); // undefined
```

The above behavior is not valid for const because an initial value is required for it. If there is no initial value, the program throws a SyntaxError.

```
const avocado; // SyntaxError
```

Declare a variable without using any keyword.

Syntax: variableName; **Example:** age;

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _.
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Declaring variables is a fundamental aspect of programming in JavaScript. Here are some key points to remember about variable declaration:

- **Declaration:** Variables are declared using the var, let, or const keyword, followed by the variable name.
- **Initialization:** Variables can be declared and initialized (assigned a value) at the same time using the assignment operator (=).

- **Scope:** Variables have a scope, which determines their accessibility within a program. It's important to declare variables in the appropriate scope to ensure proper usage and avoid conflicts.
- **Hoisting:** Variable declarations are hoisted to the top of their respective scopes during the compilation phase. However, the values for variables will be undefined until assigned.
- **Type Inference:** JavaScript is a dynamically typed language, meaning variables can hold different types of data. The type of a variable is determined by the value assigned to it.
- **Constants:** Variables declared with the const keyword are constants and cannot be reassigned after initialization. Use const for values that should remain constant throughout the program.
- **Naming Conventions:** Choose meaningful and descriptive names for variables to improve code readability and maintainability. Follow naming conventions.

Variable initialization

After the declaration, the variable has no value (technically it is undefined).

To assign a value to the variable, use the "= "sign The syntax of variable initialization is as follows: variable_name = variable_value;

Example:

```
carName = "Volvo";
```

You can also assign a value to the variable when you declare it:

```
let carName = "Volvo";
```

Initialization: Variables can be declared and initialized (assigned a value) at the same time. For example:

```
var myVariable = 10;
let myVariable = "Hello";
const myVariable = true;
```

Javascript variable scope (types)

Scope determines the accessibility (visibility) of variables.

Variable scope defines where in your code variables are accessible The variable scope may be local or global.

★ Javascript local variables

A JavaScript local variable is declared inside a block or function. It is accessible within the function or block only.

For example, variables declared inside a { } block cannot be accessed from outside the block:

```
{
  let x= 2;
}
// x can NOT be used here
```

★ JavaScript global variables

A JavaScript global variable is a variable declared outside a function or a block. This variable has a Global Scope.

Global variables can be accessed from anywhere in a JavaScript program.

Example

```
let carName = "Volvo";
// code here can use carName
function myFunction()
{
  // code here can also use carName
}
```

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable.

This code example will declare a global variable carName, even if the value is assigned inside a function.

Example:

```
myFunction();
// code here can use carName
function myFunction()
{
  carName = "Volvo";
}
```

Redeclaration of variable

To redeclare a variable simply means to declare an already declared variable or identifier regardless of whether in the same block or outer scope.

You could declare and initialise a variable using the const keyword. Example:

```
//declared and initialized the variable (firstName) and assigned a value of 'Patrick'.
const firstname='patrick';
console.log(firstname); //Patrick
```

You may have tried to redeclare the variable or even try to change its value still using the const keyword.

```
const firstname='patrick';
const firstname='john';
//Uncaught SyntaxError: Identifier 'firstName' has already been declared.
```

It signals a syntax error because const never allows us to tamper with its identifier. With these experiments, we could agree that a variable declared using the 'const' keyword cannot be redeclared and its value cannot be reassigned.

Redeclaring Variables using the let keyword

You can declare and initialize a variable using the let keyword.

```
//declared and initialized the variable and assigned a value of 'Okafor'.
let lastname='okafor';
console.log(lastname); //okafor
```

Let's try to redeclare the identifier and change its value.

```
let lastname='okafor';
let lastname='okafor';
//Uncaught SyntaxError: Identifier 'lastName' has already been declared
```

It logs out an error signalling that the identifier has already been declared therefore we must use another identifier.

Now with these experiments, you can see also that; a variable that is declared using the 'let' keyword cannot be redeclared again and its value cannot be changed or reassigned.

Note: We could declare a variable once and refer to it again without using the let keyword. For example

```
let lastname='okafor';
lastname='doe';
console.log(lastname); //doe
```

This can run successfully because we declared it with the let keyword. The const keyword would output an error.

```
const lastname='okafor';
lastname='doe';
console.log(lastname); //Error
```

Redeclaring Variables using the var keyword

Just like using the const and let keywords, You can also declare and initialise a variable using the var keyword.

```
//declared and initialized the variable and assigned a value of 'Cat'.
var mypet='cat';
console.log(mypet); //cat
```

Let's try to redeclare the variable and also change its value. We start with redeclaring the variable

```
var mypet='cat';
var mypet='cat'; //cat
```

This time around, it doesn't display any errors. It logs out the value. Now, let's change the value;

```
var mypet='cat';
```

var mypet='dog'; //dog

Variable initialisation

1.3.2: Use variables in JAVASCRIPT Task:

- Use meaningful variable names: Choose descriptive names for your variables that accurately represent their purpose. Follow naming conventions, such as starting with a letter and using camel case.
- Initialize variables when needed: Variables can be declared and initialized at the same time using the assignment operator (=). This assigns an initial value to the variable.
- Understand variable scope: Variables have scope, which determines their accessibility within a program. **var** has function scope, while **let** and **const** have block scope. Understand how variables are affected by scope and use them accordingly.
- Consider hoisting: Variable declarations are hoisted to the top of their respective scopes during the compilation phase. This allows you to use variables before they are declared, but their values will be **undefined** until assigned.
- Be aware of variable reassignment: Variables declared with var or let can be reassigned with a new value using the assignment operator (**=**). However, variables declared with const cannot be reassigned after initialization.
- Understand type inference: JavaScript is a dynamically typed language, meaning variables can hold different types of data. The type of a variable is determined by the value assigned to it.
- Follow best practices: Write clean and readable code by following best practices for variable declaration, such as declaring variables at the beginning of a scope and using the most appropriate keyword for the situation.

By remembering these key steps, you can effectively declare variables in JavaScript and ensure their proper usage within your programs.

Here's an example of variable declaration in JavaScript:

```
// Using var keyword
var myVariable;
myVariable = 10;
// Using let keyword
let anotherVariable = "Hello";
// Using const keyword
const PI = 3.14;

// Variable reassignment
myVariable = 20;
anotherVariable = "World";
// Outputting variable values
console.log(myVariable); // Output: 20
```

```
console.log(anotherVariable); // Output: "World"
console.log(PI); // Output: 3.14
```

In this example, we declare three variables: myVariable, anotherVariable, and PI. We initialize myVariable with the value 10, anotherVariable with the string "Hello", and PI with the value 3.14. Later in the code, we reassign myVariable to 20 and anotherVariable to "World". Finally, we output the values of the variables using console.log().

Variable initialization

Certainly! Here are the key steps to remember about variable initialization in JavaScript:

1. Declare the variable: Use the **var**, **let**, or **const** keyword to declare the variable. For Example:

```
var myVariable;
let myVariable;
const myVariable;
```

2. Assign an initial value: Initialize the variable by assigning an initial value using the assignment operator (=). For example:

```
myVariable = 10;
myVariable = "Hello";
myVariable = true;
```

3. Default initialization: If you don't assign an initial value during declaration, the variable will be automatically initialized with the value undefined.
4. Dynamic typing: JavaScript is a dynamically typed language, meaning variables can hold different types of data. The type of a variable is determined by the value assigned to it during initialization.
5. Reassignment: After initialization, variables can be reassigned with a new value using the assignment operator (=). For example: myVariable = 20;
6. Constants: If you want a variable to be a constant, use the **const** keyword during declaration and assign an initial value that cannot be changed later.
7. Initialization order: JavaScript hoists variable declarations to the top of their respective scopes during the compilation phase. This allows you to use variables before they are declared, but their values will be **undefined** until assigned.

By remembering these key steps, you can effectively initialize variables in JavaScript, assign appropriate values, and ensure they are ready for use in your code.

In the example below, we create a variable called carName and assign the value "Volvo" to it. Then we "output" the value inside an HTML paragraph with id="demo":

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Variables</h1>
<p>Create a variable, assign a value to it, and display it:</p>
```

```
<p id="demo"></p>
<script>
let carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
</script>
</body>
</html>
```

Output

JavaScript Variables
Create a variable, assign a value to it, and display it: Volvo

Example2:

```
<script>
var x = 10;
var y = 20;
var z=x+y;
```

Points to Remember

Creating a variable in JavaScript is called "declaring" a variable. Once a variable is created and get assigns with a value this action is known as variable initialization. Variables declared with the const keyword are constants and cannot be reassigned after initialization. There are four ways to Declare a JavaScript Variable are: Using var, Using let, using const and using nothing. Variable re-declaration means to declare an already declared variable.

To initialize a variable, we use the assignment operator (=) followed by value. For example: var salary =12000;

DATA TYPES IN JAVASCRIPT

1.4.1: Description of data types in JavaScript

The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types. There are five types of primitive data types in JavaScript. They are as follows:

DATA TYPE	DESCRIPTION
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all

Non-primitive (Reference) Data Types

These data types that are derived from primitive data types of the JavaScript language. There are also known as derived data types or reference data types. The non-primitive data types are as follows:

DATA TYPE	DESCRIPTION
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

Detailed Examples of Data Types

Primitive Data Types

String:

- Represents textual data enclosed in single or double quotes
- Example: *let name = "John Doe";*

Number:

- Represents both integer and floating-point numbers
- Example: *let age = 30; let price = 19.99;*

Boolean:

- Represents logical entities: true or false
- Example: *let isStudent = true;*

Undefined:

- Represents a variable that has been declared but not assigned a value
- Example: *let city; console.log(city); // undefined*

Null:

- Represents the intentional absence of any object value
- Example: *let car = null;*

Non-Primitive Data Types

Object:

- Collection of key-value pairs

Example:

```
let person = {
  firstName: "John",
  lastName: "Doe",
  age: 30
};
```

Array:

- Ordered collection of values

- Example:

let colors = ["Red", "Green", "Blue"];

RegExp:

- Represents regular expressions for pattern matching
- Example:

let pattern = /ab+c/;

Type Conversion in JavaScript

JavaScript is a dynamically typed language, which means variables can change type. JavaScript provides several ways to convert between types:

1. **String Conversion:**
    - **String(value)** or **value.toString()**
    - Example: ***String(123)*** *returns "123"*
2. **Numeric Conversion:**
    - **Number(value)**
    - Example: ***Number("123")*** *returns 123*
3. **Boolean Conversion:**
    - **Boolean(value)**
    - Example: ***Boolean(1)*** *returns true*

Special Data Values

1. **NaN (Not a Number):**
    - Represents a value that is not a legal number
    - Example: ***Number("hello")*** *returns NaN*
2. **Infinity:**
    - Represents mathematical infinity
    - Example: ***1 / 0*** *returns Infinity*
3. **-Infinity:**
    - Represents negative infinity
    - Example: ***-1 / 0*** *returns -Infinity*

Type Checking

To check the data type of a variable, use the **typeof** operator:

```
let name = "John";
console.log(typeof name); // "string"

let age = 30;
console.log(typeof age); // "number"

let isStudent = true;
console.log(typeof isStudent); // "boolean"

let car = null;
console.log(typeof car); // "object" (special case)

let city;
console.log(typeof city); // "undefined"
```

OPERATORS IN JAVASCRIPT

Types of Operators

1. **Arithmetic Operators:**
    - **+** (Addition)
    - **-** (Subtraction)
    - * (Multiplication)
    - / (Division)
    - % (Modulus)
    - **++** (Increment)
    - **--** (Decrement)

Example:

```
let a = 10;
let b = 3;
console.log(a + b); // 13
console.log(a - b); // 7
console.log(a * b); // 30
console.log(a / b); // 3.333...
console.log(a % b); // 1
console.log(++a);   // 11
console.log(--b);   // 2
```

2. **Assignment Operators:**

- **=** (Assignment)
- **+=** (Add and assign)

- **-=** (Subtract and assign)
- ***=** (Multiply and assign)
- **/=** (Divide and assign)
- **%=** (Modulus and assign)

Example:

```
let x = 10;
x += 5; // x = x + 5 → 15
x -= 3; // x = x - 3 → 12
x *= 2; // x = x * 2 → 24
x /= 4; // x = x / 4 → 6
x %= 5; // x = x % 5 → 1
```

3. **Comparison Operators:**

- **==** (Equal to)
- **===** (Equal value and equal type)
- **!=** (Not equal)
- **!==** (Not equal value or not equal type)
- **>** (Greater than)
- **<** (Less than)
- **>=** (Greater than or equal to)
- **<=** (Less than or equal to)

Example:

```
let a = 5;
let b = "5";
console.log(a == b);  // true (value equal)
console.log(a === b); // false (value equal but type different)
console.log(a != b);  // false
console.log(a !== b); // true
console.log(a > 3);   // true
console.log(a < 10);  // true
console.log(a >= 5);  // true
console.log(a <= 5);  // true
```

4. **Logical Operators:**

- **&&** (Logical AND)
- **||** (Logical OR)
- **!** (Logical NOT)

Example:

```
let x = 5;
let y = 10;
console.log(x > 3 && y < 15); // true
console.log(x > 10 || y < 15); // true
console.log(!(x > 3));        // false
```

5. **Bitwise Operators:**
    - **&** (Bitwise AND)
    - **|** (Bitwise OR)
    - **^** (Bitwise XOR)
    - **~** (Bitwise NOT)
    - **<<** (Left shift)
    - **>>** (Right shift)
    - **>>>** (Zero fill right shift)
6. **Ternary Operator:**
    - **condition ? value1 : value2**

Example:

```
let age = 18;
let canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // "Yes"
```

EXPRESSIONS IN JAVASCRIPT

An expression is a valid set of literals, variables, operators, and expressions that evaluate to a single value. Expressions can be:

1. **Primary Expressions:**
    - Basic keywords and general expressions in JavaScript
    - Example: **this, null, true, false, 42,** "Hello world"
2. **Object Initializer Expressions:**
    - Create objects
    - Example: **{a: 1, b: 2}**
3. **Array Initializer Expressions:**
    - Create arrays
    - Example: **[1, 2, 3]**
4. **Function Definition Expressions:**

- Define functions
- Example: **function(x) { return x * x; }**

5. **Property Access Expressions:**
    - Access properties or methods
    - Example: **object.property, object["property"]**
6. **Invocation Expressions:**
    - Call functions or methods
    - Example: **func(), obj.method()**
7. **Arithmetic Expressions:**
    - Evaluate to a number
    - Example: **3 + 4, 5 * 6**
8. **Relational Expressions:**
    - Evaluate to a boolean
    - Example: **a > b, x === y**
9. **Logical Expressions:**
    - Evaluate to a boolean
    - Example: **a && b, !c**
10. **Assignment Expressions:**
    - Assign a value to a variable
    - Example: **x = 5, y += 10**

Learning outcome 2: Manipulate data with JavaScript

2.1: Using String in JavaScript

✓ String Declaration

Strings are used to store text. You can declare them in three ways:

```
// 1. Single quotes
let str1 = 'Hello World';

// 2. Double quotes
let str2 = "Hello World";

// 3. Backticks (Template Literals - covered later)
let str3 = `Hello World`;
```

Key Notes:

- All three methods create the same type of string.
- Choose quotes that avoid escaping conflicts (e.g., use single quotes if the string contains double quotes).

✓ Escape Characters

Use backslash (\) to include special characters in strings:

```
let text = 'It\'s a beautiful day';  // Escapes single quote
let path = "C:\\Users\\John";        // Escapes backslashes
let newLine = "Line1\nLine2";        // Newline character
let tab = "Column1\tColumn2";        // Tab character
```

Common Escape Sequences:

CODE	RESULT
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\n`	Newline
`\t`	Tab

✓ String Concatenation

Combine strings using + or +=:

```
let firstName = "John";
```

```
let lastName = "Doe";

// Using +
let fullName = firstName + " " + lastName;  // "John Doe"

// Using +=
fullName += " Jr.";  // "John Doe Jr."
```

Modern Alternative: Template Literals (see below) are preferred for readability.

✓ String Methods

Built-in functions to manipulate strings:

```
let str = "Hello World";

// Length
console.log(str.length);  // 11

// Case conversion
console.log(str.toUpperCase());  // "HELLO WORLD"
console.log(str.toLowerCase());  // "hello world"

// Extracting parts
console.log(str.slice(0, 5));    // "Hello" (start index, end index)
console.log(str.substring(6));   // "World" (start index to end)
console.log(str.substr(6, 5));   // "World" (start index, length)

// Trimming whitespace
let messy = "  text  ";
console.log(messy.trim());  // "text"

// Replacing content
console.log(str.replace("World", "JavaScript"));  // "Hello JavaScript"

// Splitting into array
console.log(str.split(" "));  // ["Hello", "World"]
```

✓ String Search Methods

Find substrings or patterns:

```
let sentence = "The quick brown fox jumps over the lazy dog";

// indexOf() - Returns first match index (-1 if not found)
console.log(sentence.indexOf("fox"));      // 16
console.log(sentence.indexOf("cat"));      // -1

// includes() - Returns true/false
console.log(sentence.includes("fox"));     // true
```

```
console.log(sentence.includes("cat"));     // false

// startsWith() / endsWith()
console.log(sentence.startsWith("The"));   // true
console.log(sentence.endsWith("dog"));     // true

// search() - Same as indexOf() but supports regex
console.log(sentence.search(/fox/));       // 16
```

✓ String Template Literals

Modern syntax using backticks (`) for:

1. **Multi-line strings**
2. **Variable interpolation**
3. **Expression embedding**

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

// Multi-line
let poem = `Roses are red
Violets are blue`;

// Variable interpolation
let greeting = `Hello, ${name}!`;  // "Hello, Alice!"

// Expressions
let birthYear = `Born in ${2023 - age}`;  // "Born in 1993"

// Complex expressions
let info = `${name} is ${age > 18 ? "an adult" : "a minor"}`;  // "Alice is an adult"
```

Advantages over concatenation:

- No escaping quotes: He said: "It's great!"
- Readable multi-line strings
- Direct variable/expression insertion

Example Combining All Concepts:

```
let user = "John";
let role = "admin";
let message = `User: ${user}\nRole: ${role.toUpperCase()}\nAccess: ${role.includes("admin") ?
"Granted" : "Denied"}`;

console.log(message);
/*
```

```
Output:
User: John
Role: ADMIN
Access: Granted
*/
```

2.2: CONTROL STRUCTURES

Conditional Statements

1. **if Statement:**

   if Statement: Executes code only when condition is true,
   No fallback for false conditions

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

Example:

```
let temperature = 25;

if (temperature > 20) {
  console.log("It's warm outside!");  // This runs because 25 > 20
}

// Example with false condition
if (temperature < 0) {
  console.log("Freezing!");  // This won't run (25 is not < 0)
}
```

2. **if-else Statement:**

   if-else Statement: Provides two paths: one for true, one for false
   Always executes exactly one block

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

Example:

```
let isLoggedIn = true;

if (isLoggedIn) {
  console.log("Welcome back!");  // Runs because isLoggedIn is true
} else {
  console.log("Please log in");  // Won't run
}

// Example with false condition
let hasPermission = false;
```

```
if (hasPermission) {
  console.log("Access granted");  // Won't run
} else {
  console.log("Access denied");   // Runs because hasPermission is false
}
```

3. **else if Statement:**

   else if Statement: Checks multiple conditions in sequence
   Executes first true condition and skips the rest
   Final else catches all remaining cases

```
if (condition1) {
  // code to execute if condition1 is true
} else if (condition2) {
  // code to execute if condition2 is true
} else {
  // code to execute if all conditions are false
}
```

Example:

```
let score = 75;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");  // This runs (75 >= 70)
} else {
  console.log("Grade: F");
}

// Example with no conditions met
let weather = "cloudy";

if (weather === "sunny") {
  console.log("Wear sunglasses");
} else if (weather === "rainy") {
  console.log("Take an umbrella");
} else {
  console.log("Check the forecast");  // Runs because weather is "cloudy"
}
```

4. **switch Statement:**

   switch Statement: Compares one value against multiple cases
   Uses strict equality (===) for matching
   break prevents "fall-through" to next case
   default handles unmatched values

```
switch (expression) {
```

```
  case value1:
    // code to execute when expression equals value1
    break;
  case value2:
    // code to execute when expression equals value2
    break;
  default:
    // code to execute when expression doesn't match any case
}
```

Example:

```
let day = "Wednesday";

switch (day) {
  case "Monday":
    console.log("Work day");
    break;
  case "Tuesday":
    console.log("Team meeting");
    break;
  case "Wednesday":
    console.log("Midweek review");  // This runs
    break;
  case "Thursday":
    console.log("Project deadline");
    break;
  case "Friday":
    console.log("Almost weekend!");
    break;
  default:
    console.log("Weekend!");
}

// Example with default case
let fruit = "Mango";

switch (fruit) {
  case "Apple":
    console.log("Red fruit");
    break;
  case "Banana":
    console.log("Yellow fruit");
    break;
  default:
    console.log("Unknown fruit");  // Runs because fruit is "Mango"
}
```

Real-World Scenario Example:

```
// User authentication system
let userRole = "editor";
let isActive = true;

// 1. Simple if
if (isActive) {
  console.log("Account is active");
}

// 2. if-else
if (userRole === "admin") {
  console.log("Full access");
} else {
  console.log("Limited access");  // Runs because userRole is "editor"
}

// 3. else if chain
if (userRole === "admin") {
  console.log("Can delete content");
} else if (userRole === "editor") {
  console.log("Can edit content");  // Runs
} else if (userRole === "viewer") {
  console.log("Can view content");
} else {
  console.log("No permissions");
}

// 4. switch for role-based dashboard
switch (userRole) {
  case "admin":
    console.log("Admin dashboard");
    break;
  case "editor":
    console.log("Editor dashboard");  // Runs
    break;
  case "viewer":
    console.log("Viewer dashboard");
    break;
  default:
    console.log("Access denied");
}
```

Output:

```
Account is active
Limited access
```

Can edit content
Editor dashboard

How to run the JavaScript codes:

Method 1: Browser Console (Easiest)

1. Open your web browser (Chrome, Firefox, Edge, etc.)
2. Right-click anywhere on a webpage and select "Inspect" or "Inspect Element"
3. Click on the "Console" tab in the developer tools
4. Copy and paste the entire code into the console
5. Press Enter

Method 2: HTML File

1. Create a new file name it,
2. Add JavaScript Content,
3. Save the file and open it in your browser
4. Open the developer tools (F12) and check the Console tab

Method 3: Node.js

If you have Node.js installed:

1. Create a file named **example.js**
2. Paste the JavaScript code into it
3. Open your terminal/command prompt
4. Navigate to the file's directory
5. Run: **node example.js**

The output will appear in your terminal

2.3: Loops

1. **for Loop:**

```
for (initialization; condition; increment) {
  // code to execute repeatedly
}
```

Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

2. **while Loop:**

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

Example:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

3. **do-while Loop:**

```
do {
  // code to execute at least once
} while (condition);
```

Example:

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

4. **for-in Loop:**

- Iterates over the properties of an object

```
for (variable in object) {
  // code to execute for each property
}
```

Example:

```
const person = {name: "John", age: 30, city: "New York"};
for (let prop in person) {
  console.log(prop + ": " + person[prop]);
}
```

5. **for-of Loop:**

- Iterates over iterable objects (like arrays, strings)

```
for (variable of iterable) {
  // code to execute for each value
}
```

Example:

```
const colors = ["Red", "Green", "Blue"];
for (let color of colors) {
  console.log(color);
}
```

Jump Statements

1. **break Statement:**
    - Exits the current loop or switch statement

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // exits the loop when i is 5
  }
  console.log(i);
}
```

2. **continue Statement:**
    - Skips the current iteration and continues with the next

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    continue; // skips iteration when i is 5
  }
  console.log(i);
}
```

3. **return Statement:**
    - *Exits a function and optionally returns a value*

```
function add(a, b) {
  return a + b; // returns the sum of a and b
}
```

2.4: FUNCTIONS IN JAVASCRIPT

✓ Function Definition

Definition: A reusable block of code that performs a specific task. Defined once but executed multiple times.

Syntax:

```
function functionName(parameters) {
  // code to execute
  return value; // optional
}
```

Example:

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

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

✓ Function Parameters

Definition: Input values passed to a function. Parameters are variables listed in the function definition.

Syntax:

```
function funcName(param1, param2, ..., paramN) {
  // use parameters
}
```

Example:

```
function calculateArea(width, height) {
  return width * height;
}

console.log(calculateArea(5, 3)); // Output: 15
```

✓ Arrow Functions

Definition: Concise function syntax introduced in ES6. Always anonymous (no name) and lexically bind this.

Syntax:

```
(param1, param2) => { statements }
// or for single expression:
(param1, param2) => expression
```

Example:

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

// Arrow function equivalent
const add = (a, b) => a + b;

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

// With no parameters
const getRandom = () => Math.random();

// With single parameter (parentheses optional)
const square = x => x * x;
```

✓ Built-in Functions

Definition: Predefined functions provided by JavaScript (no need to define them).

Examples:

```
// Math functions
console.log(Math.max(1, 5, 3));   // Output: 5
console.log(Math.floor(3.9));     // Output: 3

// String functions
console.log("hello".toUpperCase()); // Output: "HELLO"

// Array functions
const numbers = [1, 2, 3];
console.log(numbers.join("-"));   // Output: "1-2-3"

// Global functions
console.log(parseInt("42"));      // Output: 42
```

```
console.log(isNaN("test"));       // Output: true
```

✓ Function Call

Definition: Executing a function by using its name followed by parentheses ().

Syntax:

```
functionName(argument1, argument2);
```

Example:

```
function multiply(x, y) {
  return x * y;
}

// Function call
const result = multiply(4, 5);
console.log(result); // Output: 20
```

✓ Function Apply

Definition: Calls a function with a given ***this*** value and arguments provided as an array.

Syntax:

```
functionName.apply(thisArg, [argsArray]);
```

Example:

```
function introduce(greeting, punctuation) {
  console.log(`${greeting}, I'm ${this.name}${punctuation}`);
}

const person = { name: "John" };

// Using apply
introduce.apply(person, ["Hi", "!"]);
// Output: "Hi, I'm John!"
```

✓ Function Bind

Definition: Creates a new function with a specific ***this*** value and initial arguments.

Syntax:

```
const boundFunction = functionName.bind(thisArg, arg1, arg2, ...);
```

Example:

```
function calculateTax(rate) {
  return this.price * (rate / 100);
}

const product = { price: 100 };

// Bind rate to 20%
const calculateVAT = calculateTax.bind(product, 20);

console.log(calculateVAT()); // Output: 20 (100 * 0.20)
```

✓ **Function Closure**

Definition: A function that retains access to variables from its outer (enclosing) scope even after the outer function has finished executing.

Example:

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

  return function() {
    count++;
    return count;
  };
}

const counter = createCounter();

console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

✓ **Asynchronous Functions**

Definition: Functions that operate asynchronously via the event loop, using callbacks, promises, or async/await.

Example (Callback):

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}

fetchData((data) => {
  console.log(data); // Output after 1 sec: "Data received"
});
```

✓ **Promise Functions**

Definition: Objects representing eventual completion/failure of async operations. Use ***.then()*** for success and ***.catch()*** for errors.

Syntax:

```
new Promise((resolve, reject) => {
  // async operation
});
```

Example:

```
function fetchUser() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
    const success = true; // Simulate success/failure
    success
      ? resolve({ name: "Alice", id: 1 })
      : reject("Network error");
  }, 1000);
 });
}

fetchUser()
 .then(user => console.log(user))  // Output: {name: "Alice", id: 1}
 .catch(error => console.error(error));
```

✓ Async/Await Function

Definition: Syntactic sugar over promises. *async* functions return promises, *await* pauses execution until promises resolve.

Syntax:

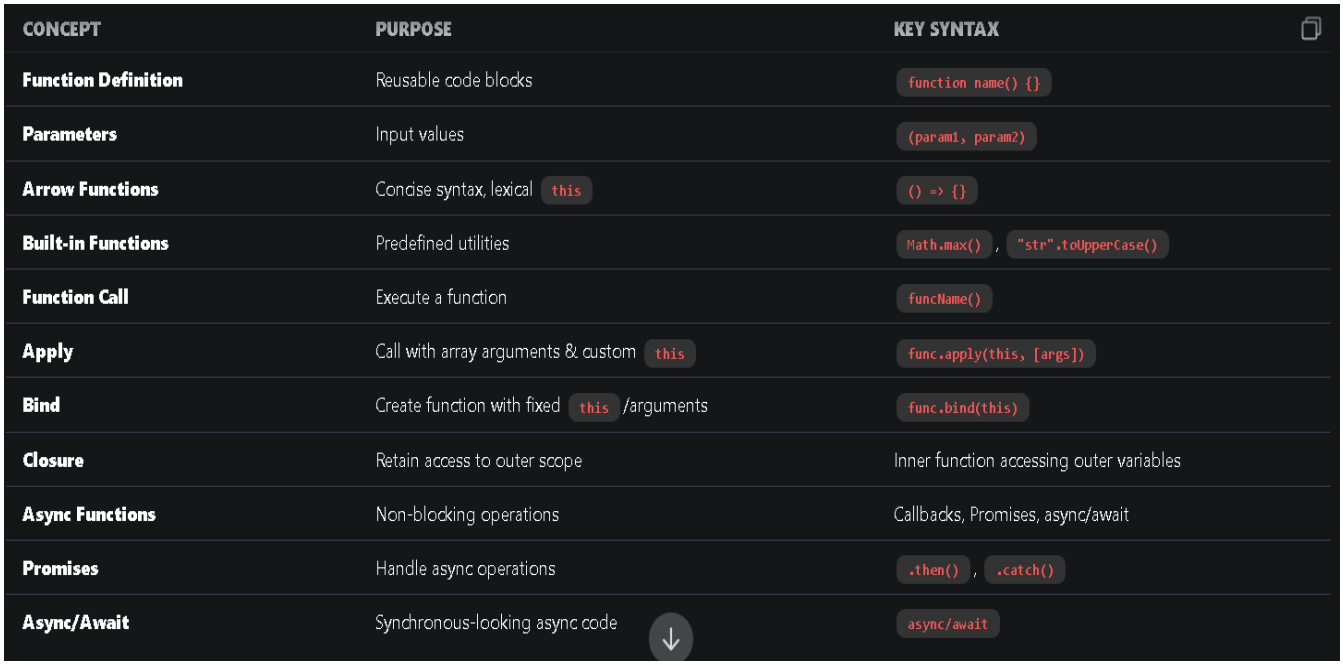
```
async function functionName() {
 const result = await promise;
}
```

Example:

```
async function getUserData() {
 try {
  const response = await fetch('https://api.example.com/user');
  const user = await response.json();
  console.log(user);
 } catch (error) {
  console.error("Failed:", error);
 }
}

getUserData();
```

Key Takeaways



CONCEPT	PURPOSE	KEY SYNTAX
Function Definition	Reusable code blocks	`function name() {}`
Parameters	Input values	`(param1, param2)`
Arrow Functions	Concise syntax, lexical `this`	`() => {}`
Built-in Functions	Predefined utilities	`Math.max()` , `"str".toUpperCase()`
Function Call	Execute a function	`funcName()`
Apply	Call with array arguments & custom `this`	`func.apply(this, [args])`
Bind	Create function with fixed `this` /arguments	`func.bind(this)`
Closure	Retain access to outer scope	Inner function accessing outer variables
Async Functions	Non-blocking operations	Callbacks, Promises, async/await
Promises	Handle async operations	`.then()` , `.catch()`
Async/Await	Synchronous-looking async code	`async/await`

Higher-Order Functions

Functions that can take other functions as arguments or return functions as results.

Example:

```
function applyOperation(a, b, operation) {
  return operation(a, b);
}

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

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

console.log(applyOperation(5, 3, add)); // 8
console.log(applyOperation(5, 3, multiply)); // 15
```

Function Scope

1. Global Scope

Definition:

Variables declared outside any function or block. Accessible from **anywhere** in the code.

Syntax:

```
// Global variable declaration
var globalVar = "I'm global";
let globalLet = "Global let";
```

```
const globalConst = "Global const";
```

Key Characteristics:

- Accessible in all functions and blocks
- Created in the global object (**window** in browsers)
- Persists until the page/application closes
- Risk of name collisions and pollution

Examples:

```
// Global declaration
const API_KEY = "abc123";

function fetchData() {
  console.log(API_KEY); // ☑ Accessible inside function
}

if (true) {
  console.log(API_KEY); // ☑ Accessible inside block
}

fetchData(); // "abc123"
console.log(API_KEY); // "abc123"

// Global variables become properties of window (browsers)
console.log(window.API_KEY); // "abc123"
```

2. Local Scope (Function Scope)

Definition:

Variables declared inside a function. Only accessible **within that function** and its nested functions (closures).

Syntax:

```
function myFunction() {
  // Local variable declarations
  var localVar = "I'm local";
  let localLet = "Local let";
  const localConst = "Local const";
}
```

Key Characteristics:

- Created when function is called, destroyed when it completes
- Not accessible outside the function
- **var, let,** and **const** all have function scope
- Enables data encapsulation

Examples:

```
function calculateTax() {
  const taxRate = 0.15; // Local scope

  return function(price) {
    return price * taxRate; // Closure access
  };
}

const taxCalculator = calculateTax();
console.log(taxRate); // ✗ ReferenceError (not accessible outside)
console.log(taxCalculator(100)); // ☑ 15 (via closure)

// Nested function access
function outer() {
  let secret = "123";

  function inner() {
    console.log(secret); // ☑ Accessible (closure)
  }

  inner(); // "123"
}
```

3. Block Scope

Definition:
Variables declared within a block (**{...}**) - e.g., **if** statements, loops, or standalone blocks. Only accessible **within that specific block.**

Syntax:

```
{
  // Block-scoped declarations
  let blockLet = "I'm block-scoped";
  const blockConst = "Block const";
}

// Note: var does NOT have block scope
```

Key Characteristics:

- Only applies to **let** and **const** (not **var**)
- Created when block executes, destroyed when block completes
- Prevents variable leakage from loops/conditionals
- Essential for modern JavaScript safety

Examples:

```
// if block
if (true) {
  let message = "Inside if";
  console.log(message); // ☑ "Inside if"
}
console.log(message); // ✗ ReferenceError

// for loop block
for (let i = 0; i < 3; i++) {
  let loopValue = i * 2;
  console.log(loopValue); // 0, 2, 4
}
console.log(i); // ✗ ReferenceError
console.log(loopValue); // ✗ ReferenceError

// Standalone block
{
  const temp = "Temporary";
  console.log(temp); // ☑ "Temporary"
}
console.log(temp); // ✗ ReferenceError

// var vs. let in blocks
function testVar() {
  if (true) {
    var x = 1;      // Function-scoped (leaks)
    let y = 2;      // Block-scoped
  }
  console.log(x);   // ☑ 1 (accessible)
  console.log(y);   // ✗ ReferenceError
}
```

Scope Comparison Table

Scope Comparison Table

FEATURE	GLOBAL SCOPE	LOCAL (FUNCTION) SCOPE	BLOCK SCOPE
Accessibility	Everywhere	Within function + closures	Within `{...}` block
Declaration	Outside all functions	Inside function	Inside `{...}`
Lifetime	Until app closes	Until function exits	Until block exits
Keywords	`var` , `let` , `const`	`var` , `let` , `const`	Only `let` , `const`
Hoisting	Yes (all)	Yes (all)	No (TDZ for `let` / `const`)
Pollution Risk	High	Low	None
Example	`const global = "x";`	`function() { let x; }`	`{ let x; }`

Real-World Scope Examples

1. Global Scope Pitfall

a likely mistake or problem in a situation

```
// BAD: Global pollution
let counter = 0;

function increment() {
  counter++; // Modifies global variable
}

increment();
console.log(counter); // 1 (accessible globally)

// BETTER: Encapsulate in function scope
function createCounter() {
  let count = 0; // Local scope

  return {
    increment: () => ++count,
    getCount: () => count
  };
}

const counter = createCounter();
counter.increment();
console.log(counter.getCount()); // 1
```

```
console.log(count); // ✗ ReferenceError (protected)
```

The Correct way should be these

```
// BAD: Global pollution
let count = 0;

function increment() {
  count++; // Modifies global variable
}

increment();
console.log(count); // 1 (accessible globally)

// BETTER: Encapsulate in function scope
function createCounter() {
  let count = 0; // Local scope

  return {
    increment: () => ++count,
    getCount: () => count
  };
}

const counter = createCounter();
counter.increment();
console.log(counter.getCount()); // 1
console.log(count); // ✗ ReferenceError (protected)
```

2. Block Scope in Loops

```
const global = "GLOBAL"; // Global scope

function outer() {
  const outerVar = "OUTER"; // Function scope

  function inner() {
    const innerVar = "INNER"; // Function scope

    console.log(global);   // ☑ "GLOBAL" (global)
    console.log(outerVar); // ☑ "OUTER" (outer function)
    console.log(innerVar); // ☑ "INNER" (current function)
  }
```

```
  inner();
  // console.log(innerVar); // ✗ ReferenceError
}

outer();
// console.log(outerVar); // ✗ ReferenceError
```

Key Takeaways

1. **Global Scope:**
    - Use sparingly (mainly for constants/configurations)
    - Avoid for mutable data to prevent pollution
2. **Local (Function) Scope:**
    - Foundation of encapsulation in JavaScript
    - Enables closures and private data patterns
3. **Block Scope:**
    - Modern safety feature (ES6+)
    - Always prefer **let**/**const** over **var**
    - Critical for loops and conditional blocks

2.5: **Using Objects in JavaScript**

✓ Definition

Object
An object is a collection of key-value pairs where keys are strings (or Symbols) and values can be any data type (including other objects). Objects are used to store complex data and represent real-world entities.

Key Characteristics:

- **Unordered**: Key-value pairs have no guaranteed order (though modern JS engines maintain insertion order for string keys).
- **Dynamic**: Properties can be added, modified, or deleted at runtime.
- **Reference Type**: Objects are copied by reference (not value).

✓ Syntax

1. Object Literal (Most Common)

```
const person = {
  name: "Alice",
  age: 30,
  isStudent: false,
```

```
  greet: function() {
    return `Hello, I'm ${this.name}`;
  }
};
```

2. **new Object() Constructor**
```
const car = new Object();
car.make = "Toyota";
car.model = "Camry";
car.year = 2022;
```
3. **ES6 Computed Property Names**
```
const propName = "dynamicKey";
const obj = {
  [propName]: "Dynamic value",
  [`key_${1 + 2}`]: "Computed"
};
```

✓ Accessing Object Methods and Properties

1. **Dot Notation**
```
console.log(person.name);      // "Alice"
console.log(person.greet());   // "Hello, I'm Alice"
```
2. **Bracket Notation**
```
console.log(person["age"]);    // 30
console.log(person["is" + "Student"]); // false (dynamic keys)
```
3. **Modifying/Adding Properties**
```
person.age = 31;               // Update
person.country = "USA";        // Add new property
```
4. **Deleting Properties**
```
delete person.isStudent;
```
5. **Checking Property Existence**
```
console.log("name" in person);    // true
console.log(person.hasOwnProperty("age")); // true
```

✓ Object Constructors

Definition:
Blueprints for creating multiple objects with the same structure. Use the **new** keyword to instantiate.

1. **Constructor Function**
```
function User(name, role) {
  this.name = name;
  this.role = role;
  this.login = function() {
    return `${this.name} logged in`;
  };
}
```

```
const admin = new User("Bob", "Admin");
console.log(admin.login()); // "Bob logged in"
```

2. **ES6 Class (Syntactic Sugar)**

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  getDiscount() {
    return this.price * 0.9;
  }
}

const laptop = new Product("MacBook", 2000);
console.log(laptop.getDiscount()); // 1800
```

✓ Object Sets (JavaScript Set Objects)

Definition:
A **Set** is a collection of **unique values** (no duplicates). Unlike objects, keys are not stored—only values.

Syntax & Methods:

```
// Create a Set
const roles = new Set();

// Add values
roles.add("Admin");
roles.add("Editor");
roles.add("Admin"); // Ignored (duplicate)

console.log(roles); // Set(2) {"Admin", "Editor"}

// Check size
console.log(roles.size); // 2

// Delete value
roles.delete("Editor");

// Check existence
console.log(roles.has("Admin")); // true

// Iterate
roles.forEach(role => console.log(role)); // "Admin"

// Convert to Array
const roleArray = [...roles]; // ["Admin"]
```

Use Cases:

- Removing duplicates from an array
- Tracking unique items (e.g., tags, categories)

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

const uniqueNumbers = [...new Set(numbers)]; // [1, 2, 3, 4, 5]
```

✓ Object Maps (JavaScript Map Objects)

Definition:
A **Map** is a collection of key-value pairs where **keys can be any data type** (including objects, functions, primitives). Unlike objects, keys maintain insertion order.

Syntax & Methods:

```
// Create a Map
const userRoles = new Map();

// Add key-value pairs
userRoles.set("alice", "Admin");
userRoles.set(123, "Guest");
userRoles.set({ id: 1 }, "Editor");

// Get value by key
console.log(userRoles.get("alice")); // "Admin"

// Check size
console.log(userRoles.size); // 3

// Check key existence
console.log(userRoles.has(123)); // true

// Delete entry
userRoles.delete(123);

// Iterate
userRoles.forEach((role, key) => {
  console.log(`${key}: ${role}`);
});
// "alice: Admin"
// "[object Object]: Editor"

// Convert to Array
const entries = [...userRoles];
// [["alice", "Admin"], [{id:1}, "Editor"]]
```

Use Cases:

- When keys are not strings (e.g., objects as keys)
- Preserving insertion order
- Frequent additions/removals of key-value pairs

Key Differences: Objects vs. Sets vs. Maps

FEATURE	OBJECT	SET	MAP
Purpose	Key-value pairs (string keys)	Unique values	Key-value pairs (any key)
Key Types	Strings/Symbols only	N/A (values only)	Any data type
Order	Unordered (but insertion order preserved in modern JS)	Insertion order	Insertion order guaranteed
Duplicates	Keys must be unique	Values must be unique	Keys must be unique
Size	Manual calculation (`Object.keys()`)	`.size` property	`.size` property
Performance	Not optimized for frequent add/remove	Optimized for uniqueness	Optimized for frequent add/remove

Practical Examples

1. **Object for User Profile**

```
const userProfile = {
  id: 101,
  name: "Alice",
  preferences: {
    theme: "dark",
    notifications: true
  },
  updateTheme(newTheme) {
    this.preferences.theme = newTheme;
  }
};

userProfile.updateTheme("light");
console.log(userProfile.preferences.theme); // "light"
```

2. **Set for Unique Tags**

```
const tags = ["tech", "js", "web", "tech", "js"];
const uniqueTags = new Set(tags);
console.log(uniqueTags); // Set(3) {"tech", "js", "web"}
```

3. **Map for Metadata Tracking**

```
const fileMetadata = new Map();
const file1 = { name: "report.pdf" };
const file2 = { name: "image.png" };

fileMetadata.set(file1, { size: "2MB", modified: "2023-10-01" });
fileMetadata.set(file2, { size: "500KB", modified: "2023-10-02" });

console.log(fileMetadata.get(file1).size); // "2MB"
```

When to Use:

- **Use Objects:**
  When you need simple key-value pairs with string keys (e.g., configuration, API responses).
- **Use Sets:**
  When you need to store unique values (e.g., removing duplicates, tracking active users).
- **Use Maps:**
  When keys are non-string types (e.g., objects as keys), or you need frequent additions/deletions with guaranteed order.

2.6: Using Array in JavaScript

✓ Syntax

Definition:
Arrays are ordered collections of values (elements) that can be of any data type. They are zero-indexed and dynamic (size can change).

Declaration Syntax:

```
// 1. Array Literal (Most Common)
const fruits = ["Apple", "Banana", "Cherry"];

// 2. Array Constructor
const numbers = new Array(1, 2, 3, 4);

// 3. Empty Array with Predefined Size
const empty = new Array(5); // Creates array with 5 empty slots
```

Key Notes:

- Use [] instead of new Array() for simplicity
- Arrays can hold mixed types:

  ```
  const mixed = [1, "Hello", true, { name: "Alice" }, null];
  ```

✓ Types

JavaScript arrays are technically objects, but we categorize them by usage:

1. **Indexed Arrays**
   Standard arrays with numeric indices:
   ```
   const colors = ["Red", "Green", "Blue"];
   console.log(colors[0]); // "Red"
   ```
2. **Associative Arrays (Not Recommended)**
   Using non-numeric indices (treated as object properties):

```
const user = [];
user["name"] = "Bob"; // Avoid this! Use objects instead.
```

3. **Multidimensional Arrays**

   Arrays containing other arrays:

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
console.log(matrix[1][2]); // 6
```

✓ Methods

1. **Adding/Removing Elements**

```
const arr = ["A", "B", "C"];

// Add to end
arr.push("D");     // ["A", "B", "C", "D"]

// Remove from end
arr.pop();         // ["A", "B", "C"]

// Add to start
arr.unshift("X");  // ["X", "A", "B", "C"]

// Remove from start
arr.shift();       // ["A", "B", "C"]
```

2. **Splicing (Modify Anywhere)**

```
const months = ["Jan", "Mar", "Apr"];
// Add at index 1
months.splice(1, 0, "Feb"); // ["Jan", "Feb", "Mar", "Apr"]
// Remove 2 elements starting at index 2
months.splice(2, 2);       // ["Jan", "Feb"]
```

3. **Slicing (Extract Subarray)**

```
const letters = ["A", "B", "C", "D"];
const subset = letters.slice(1, 3); // ["B", "C"] (doesn't modify original)
```

4. **Searching**

```
const nums = [1, 2, 3, 4, 5];
console.log(nums.indexOf(3));     // 2
console.log(nums.includes(5));    // true
console.log(nums.find(x => x > 3)); // 4
```

5. **Transformation**

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

// Map: Transform each element
const doubled = numbers.map(x => x * 2); // [2, 4, 6]

// Filter: Select elements
```

```
const evens = numbers.filter(x => x % 2 === 0); // [2]

// Reduce: Compute single value
const sum = numbers.reduce((acc, x) => acc + x, 0); // 6
```

6. **Sorting**

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

const nums = [3, 1, 10, 2];
nums.sort((a, b) => a - b); // [1, 2, 3, 10] (ascending)
```

7. **Utility Methods**

```
const arr = [1, 2, 3];
console.log(arr.join("-"));    // "1-2-3"
console.log(arr.reverse());    // [3, 2, 1]
console.log(arr.concat([4, 5])); // [1, 2, 3, 4, 5]
```

✓ Array Iterations

1. **Classic for Loop**

```
const colors = ["Red", "Green", "Blue"];
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

2. **for...of Loop (ES6)**

```
for (const color of colors) {
  console.log(color);
}
```

3. **forEach() Method**

```
colors.forEach((color, index) => {
  console.log(`${index}: ${color}`);
});
```

4. **Iteration with Methods**

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

// Map (transform)
const squared = numbers.map(n => n ** 2); // [1, 4, 9, 16]

// Filter (select)
const bigNumbers = numbers.filter(n => n > 2); // [3, 4]

// Reduce (aggregate)
const product = numbers.reduce((acc, n) => acc * n, 1); // 24
```

5. **while Loop**

```
let i = 0;
while (i < colors.length) {
  console.log(colors[i]);
```

```
  i++;
}
```

6. **for...in Loop (Avoid for Arrays)**

```
// Works but not recommended (iterates over enumerable properties)
for (const index in colors) {
  console.log(colors[index]);
}
```

Key Differences: Iteration Methods

METHOD	RETURNS	PURPOSE	MUTATES ORIGINAL?
`forEach()`	`undefined`	Execute function per element	❌ No
`map()`	New array	Transform elements	❌ No
`filter()`	New array	Select elements	❌ No
`reduce()`	Single value	Aggregate values	❌ No
`find()`	First match	Find element	❌ No
`some()`	`true/false`	Test if any element passes	❌ No
`every()`	`true/false`	Test if all elements pass	❌ No

Practical Examples

1. **Data Processing Pipeline**

```
const users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 17 },
  { name: "Charlie", age: 30 }
];

// Get names of adults
const adultNames = users
  .filter(user => user.age >= 18)
  .map(user => user.name);

console.log(adultNames); // ["Alice", "Charlie"]
```

2. **Flattening Multidimensional Array**

```
const nested = [[1, 2], [3, 4], [5]];
const flat = nested.flat(); // [1, 2, 3, 4, 5]
```

3. **Removing Duplicates**

```
const duplicates = [1, 2, 2, 3, 4, 4, 5];
const unique = [...new Set(duplicates)]; // [1, 2, 3, 4, 5]
```

4. **Chaining Methods**

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

// Sum of absolute values
const result = numbers
  .map(Math.abs)        // [1, 2, 3, 4, 5]
  .filter(n => n > 1)   // [2, 3, 4, 5]
  .reduce((a, b) => a + b, 0); // 14
```

2.7: Using JavaScript in HTML