MEAN STACK UNIT-02

JAVA SCRIPT

Java Script:

JavaScript is **a programming language used for creating dynamic, interactive web pages and web applications**. It is a **client-side scripting language**, which means **it runs on the user's web browser** rather than on the server.

JavaScript is commonly used for adding functionality to web pages such as form validation, creating animations, and making web pages more interactive. It is also used for developing web applications, including single-page applications (SPAs), which allow users to interact with web pages without having to reload the entire page.

JavaScript was first released in 1995 by Netscape Communications Corporation and has since become one of the most widely used programming languages on the web. It is often used alongside HTML and CSS, which are used for creating the structure and style of web pages, respectively.

JavaScript has evolved significantly over the years, with new features and updates being added to the language on a regular basis. Some of the most popular frameworks and libraries built on top of JavaScript include React, Angular, and Vue.js.

Why We Need JavaScript:

JavaScript is an essential component of modern web development, and there are several reasons why we need it:

1. **Dynamic Interactivity**: JavaScript allows web developers to create interactive and dynamic web pages that respond to user actions without needing to reload the entire page. This makes for a more engaging and user-friendly experience.
2. **Form Validation**: JavaScript can be used to validate user input on web forms before it is submitted to the server. This helps to ensure that the data submitted is accurate and meets certain criteria.
3. **Client-side Processing**: JavaScript allows web developers to perform calculations, manipulate data, and perform other operations on the client-side. This reduces the load on the server and improves the overall performance of web applications.
4. **Cross-browser Compatibility**: JavaScript is supported by all major web browsers, which means that web developers can write code once and have it run on multiple platforms without the need for additional code.
5. **Integration with other technologies**: JavaScript can be used in conjunction with other web technologies such as HTML, CSS, and AJAX to create powerful and sophisticated web applications.

Overall, JavaScript is a versatile and essential tool for modern web development, allowing developers to create interactive, engaging, and responsive web applications that can run on any platform.

What is JavaScript:

JavaScript is a **high-level**, **interpreted programming language** used to **create interactive and dynamic effects on web pages**. It is one of the three core technologies used for building web pages, along with HTML and CSS.

JavaScript is often used for front-end development, which involves creating the user interface and user experience of a website or web application. It is also used for back-end development, which involves processing data and performing other server-side operations.

JavaScript code is executed by a web browser, allowing it to interact with the Document Object Model (DOM) of a web page. This allows developers to dynamically modify the content of a web page based on user input, change the style and layout of page elements, and perform other interactive functions.

JavaScript has a large and active community of developers, and there are many frameworks and libraries available for developers to use to simplify and streamline the development process. Some popular JavaScript frameworks and libraries include React, Angular, Vue.js, and jQuery.

Overall, JavaScript is an essential tool for modern web development, allowing developers to create dynamic, engaging, and interactive web pages and web applications.

Environment Setup:

Setting up a development environment for JavaScript involves installing the necessary software and tools on your computer. Here are the basic steps to set up a development environment for JavaScript:

1. **Install a text editor**: You will need a text editor to write your JavaScript code. Some popular text editors include Sublime Text, Atom, and Visual Studio Code.
2. **Install a web browser**: You will need a web browser to test and debug your JavaScript code. Google Chrome, Mozilla Firefox, and Microsoft Edge are popular choices.
3. **Install Node.js**: Node.js is a JavaScript runtime environment that allows you to run JavaScript code outside of a web browser. It includes a package manager called npm that you can use to install libraries and frameworks. You can download and install Node.js from the official website.
4. **Install a version control system**: A version control system, such as Git, allows you to keep track of changes to your code and collaborate with others. Git is a popular choice, and you can download it from the official website.
5. **Install a JavaScript framework or library**: Depending on your project requirements, you may need to install a JavaScript framework or library. Some popular choices include React, Angular, Vue.js, and jQuery. You can use npm to install these libraries and frameworks.

Once you have set up your development environment, you can start writing and testing your JavaScript code.

Working With Identifiers:

Identifiers are names that you give to variables, functions, objects, and other elements in your JavaScript code. They are used to identify and refer to these elements in your code. Here are some rules and conventions for working with identifiers in JavaScript:

1. **Naming Conventions**: It is a common convention in JavaScript to use camelCase when naming identifiers. This means that the first word is lowercase, and subsequent words start with an uppercase letter (e.g. firstName, myVariable, calculateTotal).
2. **Keywords**: You cannot use JavaScript keywords as identifiers. Keywords are reserved words that have special meanings in the language (e.g. if, else, while, for).
3. **Special Characters**: Identifiers can only contain letters, numbers, underscores, and dollar signs. They cannot contain spaces or other special characters.
4. **Case Sensitivity**: JavaScript is case sensitive, so myVariable and myvariable are considered different identifiers.
5. **Meaningful Names**: It is important to use meaningful names for your identifiers to make your code easier to understand and maintain. For example, instead of using a generic name like x for a variable, use a more descriptive name like totalPrice.
6. **Scope**: Identifiers are only visible within the scope in which they are defined. For example, a variable defined inside a function is only visible within that function.

Overall, working with identifiers in JavaScript is a crucial part of writing clear and maintainable code. By following these conventions and rules, you can make your code more readable and easier to work with.

Types of Identifiers:

In JavaScript, there are several types of identifiers that are used to name variables, functions, objects, and other elements in your code. Here are some common types of identifiers:

1. **Variables**: Variables are used to store data values in JavaScript. They are declared using the "var", "let", or "const" keywords, followed by the identifier name. For example:
   ```
   var firstName = "John";
   let age = 30;
   const PI = 3.14;
   ```
2. **Functions**: Functions are used to perform actions and return values in JavaScript. They are declared using the "function" keyword, followed by the identifier name and a set of parentheses. For example:
   ```
   function calculateSum(a, b) {
     return a + b;
   }
   ```

3. **Objects**: Objects are used to group related data and functions in JavaScript. They are declared using curly braces, with key-value pairs separated by colons. For example:

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  getFullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

4. **Classes**: Classes are a new feature in JavaScript that allow you to create objects using a template. They are declared using the "class" keyword, followed by the identifier name. For example:

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

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

Overall, identifiers are an important part of writing clear and readable code in JavaScript. By using meaningful and descriptive names for your variables, functions, objects, and classes, you can make your code easier to understand and maintain.

Primitive & Non-Primitive Data Types:

In JavaScript, there are two main categories of data types: primitive and non-primitive (or reference) types.

1. Primitive Data Types: **Primitive data types** are **simple**, **immutable data types** that are **stored directly in memory**. They are also passed by value, meaning that when you pass a primitive data type as an argument to a function or assign it to a variable, a copy of the value is created. The following are the six primitive data types in JavaScript:
   - **Number**: Represents numerical values, including integers and floating-point numbers. For example: **var x = 3; var y = 3.14;**
   - **String**: Represents textual data, enclosed in single or double quotes. For example: **var name = "John"; var message = 'Hello, world!';**
   - **Boolean**: Represents a logical value, either true or false. For example: **var isStudent = true; var isAdmin = false;**
   - **Null**: Represents the intentional absence of any object value. For example: **var value = null;**

- **Undefined**: Represents a declared variable that has not been assigned a value. For example:
  var value;
- **Symbol**: Represents a unique, immutable value that is used as an identifier for object properties. Symbols were added in ECMAScript 6. For example:
  var id = Symbol("id");

2. <u>**Non-Primitive (Reference) Data Types**</u>: **Non-primitive data types** are **more complex data types** that are **not stored directly in memory**, but rather as a reference to a location in memory where the data is stored. When you pass a non-primitive data type as an argument to a function or assign it to a variable, you are passing a reference to that data, rather than a copy of the data itself. The following are examples of non-primitive data types in JavaScript:
- **Object**: Represents a collection of properties and values. For example:
  var person = { name: "John", age: 30 };
- **Array**: Represents a collection of values, stored in an ordered list. For example:
  var numbers = [1, 2, 3, 4, 5];
- **Function**: Represents a reusable block of code that performs a specific task. For example:
  function greet(name) { console.log("Hello, " + name + "!"); }

Overall, understanding the difference between primitive and non-primitive data types in JavaScript is important for writing efficient and effective code.

Operators & Type of Operators:

In JavaScript, operators are used to perform various types of operations on values, such as mathematical operations, logical operations, and assignment operations. Here are some common types of operators in JavaScript:

1. <u>**Arithmetic Operators**</u>: Arithmetic operators are used to perform mathematical operations on numerical values. The following are arithmetic operators in JavaScript:

Addition: +

Subtraction: -

Multiplication: *

Division: /

Modulus: %

Increment: ++

Decrement: --

2. **Comparison Operators**: Comparison operators are used to compare two values and return a Boolean value (true or false). The following are comparison operators in JavaScript:

   Equal to: ==

   Not equal to: !=

   Strict equal to: ===

   Strict not equal to: !==

   Greater than: >

   Less than: <

   Greater than or equal to: >=

   Less than or equal to: <=

3. **Logical Operators**: Logical operators are used to perform logical operations on Boolean values. The following are logical operators in JavaScript:

   Logical AND: &&

   Logical OR: ||

   Logical NOT: !

4. **Assignment Operators**: Assignment operators are used to assign values to variables. The following are assignment operators in JavaScript:

   Assignment: =

   Addition assignment: +=

   Subtraction assignment: -=

   Multiplication assignment: *=

   Division assignment: /=

   Modulus assignment: %=

5. **Bitwise Operators**: Bitwise operators are used to perform bitwise operations on binary values. The following are bitwise operators in JavaScript:

   Bitwise AND: &

   Bitwise OR: |

   Bitwise XOR: ^

   Bitwise NOT: ~

   Left shift: <<

   Right shift: >>

   Zero-fill right shift: >>>

Overall, understanding the different types of operators in JavaScript is important for writing complex and efficient code. By using operators effectively, you can perform a wide range of operations on values and create complex logic in your code.

Types of Statements:

In JavaScript, there are several types of statements that can be used to control the flow of the program, define variables, and execute code. Here are the most common types of statements in JavaScript:

1. **Expression Statements**: Expression statements are used to evaluate an expression and potentially update a variable. For example:

```
var x = 5;
x = x + 1;
```

2. **Conditional Statements**: Conditional statements are used to control the flow of the program based on certain conditions. The following are conditional statements in JavaScript:
   - **If statement**: Executes a block of code if a specified condition is true. For example:

```
if (x > 5) {

  console.log("x is greater than 5");

}
```

   - **If-else statement**: Executes one block of code if a specified condition is true and another block of code if the condition is false. For example:

```
if (x > 5) {
  console.log("x is greater than 5");
} else {
  console.log("x is less than or equal to 5");
}
```

- **Switch statement**: Evaluates an expression and executes a block of code based on the value of the expression. For example:

```
switch (dayOfWeek) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  default:
    console.log("Today is not Monday or Tuesday");
}
```

3. **Looping Statements**: Looping statements are used to execute a block of code repeatedly. The following are looping statements in JavaScript:

- **For loop**: Executes a block of code a specified number of times. For example:

```
for (var i = 0; i < 10; i++) {
  console.log(i);
}
```

- **While loop**: Executes a block of code as long as a specified condition is true. For example:

```
while (x < 10) {
  console.log(x);
  x++;
}
```

- **Do-while loop**: Executes a block of code at least once and then continues to execute the block of code as long as a specified condition is true. For example:

```
do {
  console.log(x);
  x++;
} while (x < 10);
```

4. **Jump Statements**: Jump statements are used to transfer control to another part of the program. The following are jump statements in JavaScript:

- **Break statement**: Terminates the current loop or switch statement. For example:

```
for (var i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
```

- **Continue statement**: Skips the current iteration of a loop and continues with the next iteration. For example:

```
for (var i = 0; i < 10; i++) {
  if (i === 5) {
    continue;
  }
  console.log(i);
}
```

- **Return statement**: Terminates the execution of a function and returns a value. For example:

```
function add(x, y) {
  return x + y;
}
```

Overall, understanding the different types of statements in JavaScript is important for writing complex and effective code. By using statements effectively, you can control the flow of the program, define variables, and execute code in a way that meets your specific needs.

Non-Conditional Statements:

Non-conditional statements in JavaScript are statements that do not involve any decision making based on a condition. Here are some of the most common non-conditional statements in JavaScript:

1. **Variable Declaration Statement**: A variable declaration statement is used to declare a variable and optionally assign it an initial value. For example:

```
var x;
var y = 5;
```

2. **Assignment Statement**: An assignment statement is used to assign a value to a variable. For example:

```
x = 5;
```

3. **Function Declaration Statement**: A function declaration statement is used to define a function. For example:

```
function add(x, y) {
  return x + y;
}
```

4. **Function Call Statement**: A function call statement is used to call a function and optionally pass in arguments. For example:

```
var result = add(2, 3);
```

5. **Object Declaration Statement**: An object declaration statement is used to define an object. For example:

```
var person = {
  name: "John",
  age: 30,
  gender: "male"
};
```

6. **Object Property Assignment Statement**: An object property assignment statement is used to assign a value to an object property. For example:

```
person.age = 31;
```

7. **Array Declaration Statement**: An array declaration statement is used to define an array. For example:

```
var fruits = ["apple", "banana", "orange"];
```

8. **Array Element Assignment Statement**: An array element assignment statement is used to assign a value to an array element. For example:

```
fruits[1] = "kiwi";
```

These non-conditional statements are essential building blocks for writing JavaScript code. By understanding how to use them effectively, you can create variables, functions, objects, and arrays that can be used to perform a wide range of tasks in your JavaScript programs.

Types of Conditional Statements:

Conditional statements in JavaScript are used to make decisions based on a certain condition or set of conditions. There are three types of conditional statements in JavaScript:

1. **if statement**: The if statement is the most basic type of conditional statement in JavaScript. It is used to execute a block of code if a certain condition is true. The syntax for the if statement is as follows:

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

2. **if...else statement**: The if...else statement is used when you want to execute one block of code if a condition is true, and a different block of code if the condition is false. The syntax for the if...else statement is as follows:

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

3. **switch statement**: The switch statement is used when you want to execute different blocks of code depending on the value of an expression. The switch statement evaluates the expression and executes the block of code that corresponds to the value of the expression. The syntax for the switch statement is as follows:

```
switch (expression) {
  case value1:
    // code to execute if expression is equal to value1
    break;
  case value2:
    // code to execute if expression is equal to value2
    break;
  ...
  default:
    // code to execute if expression doesn't match any of the cases
    break;
}
```

These conditional statements are essential for creating flexible and dynamic JavaScript programs. By using them, you can make your code respond to different situations and conditions, and perform different actions based on the input or data that it receives.

If & Switch Statements:

In JavaScript, **if** and **switch statements** are conditional statements that allow you to execute different blocks of code based on certain conditions. Here is an overview of how these statements work:

if Statement:

The **if** statement is used to execute a block of code if a certain condition is true. Here is the basic syntax for the **if** statement:

```
if (condition) {

  // code to execute if the condition is true

}
```

You can also use an **else** clause to execute a different block of code if the condition is false. Here is an example:

```
if (condition) {

  // code to execute if the condition is true

} else {

  // code to execute if the condition is false

}
```

You can also use multiple **else if** clauses to test for multiple conditions. Here is an example:

```
if (condition1) {
  // code to execute if condition1 is true
} else if (condition2) {
  // code to execute if condition2 is true
} else {
  // code to execute if neither condition1 nor condition2 is true
}
```

Switch Statement:

The switch statement is used to execute different blocks of code based on the value of an expression. Here is the basic syntax for the switch statement:

```
switch (expression) {
  case value1:
    // code to execute if expression is equal to value1
    break;
  case value2:
    // code to execute if expression is equal to value2
    break;
  default:
    // code to execute if expression doesn't match any of the cases
    break;
}
```

The **expression** is evaluated and compared to each of the **case** values until a match is found. If a match is found, the corresponding block of code is executed. If no match is found, the code in the **default** block is executed. You can have any number of **case** clauses and only one **default** clause in a **switch** statement.

Both **if** and **switch** statements are important tools for controlling the flow of your JavaScript code. By using these statements effectively, you can create flexible and dynamic programs that can handle a wide range of input and conditions.

Types of Loops:

In JavaScript, there are three types of loops that allow you to execute a block of code repeatedly:

for, **while**, and **do**...**while** loops.

For Loop:

The **for** loop is used when you know the number of times you want to execute a block of code. The basic syntax for the **for** loop is as follows:

```
for (initialization; condition; increment) {

  // code to execute

}
```

Here's what each part of the **for** loop syntax does:

- **initialization**: sets the starting value for the loop counter
- **condition**: specifies the condition that must be true for the loop to continue
- **increment**: increases the loop counter after each iteration

Here's an example of how to use a **for** loop to print the numbers 1 through 5:

```
for (var i = 1; i <= 5; i++) {

  console.log(i);

}
```

While Loop:

The **while** loop is used when you don't know how many times you need to execute a block of code. The loop will continue to execute as long as the condition is true. The basic syntax for the **while** loop is as follows:

```
while (condition) {

  // code to execute
}
```

Here's an example of how to use a while loop to print the numbers 1 through 5:

```
var i = 1;


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

<u>**do...while Loop**</u>:

The **do...while** loop is similar to the **while** loop, but it will always execute the block of code at least once, even if the condition is false. The basic syntax for the **do...while** loop is as follows:

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

Here's an example of how to use a **do...while** loop to print the numbers 1 through 5:

```
var i = 1;
do {
  console.log(i);
  i++;
} while (i <= 5);
```

All three types of loops are important tools for creating flexible and dynamic JavaScript programs that can handle a wide range of input and conditions.

Types of Functions:

In JavaScript, there are several types of functions that you can use to organize your code and make it more reusable. Here are the main types of functions in JavaScript:

1. **Function Declarations**:

A function declaration defines a named function with a specified set of parameters. You can call the function by using its name and passing in the required arguments. Here's the basic syntax for a function declaration:

```
function functionName(parameters) {

  // code to execute

}
```

Here's an example of how to define and call a function:

```
function sayHello(name) {

  console.log("Hello, " + name + "!");

}

sayHello("John"); // Output: Hello, John!
```

2. Function Expressions:

A function expression defines a function as part of an expression, rather than as a statement. This allows you to define and call functions dynamically at runtime. Here's the basic syntax for a function expression:

```
var functionName = function(parameters) {

  // code to execute

};
```

Here's an example of how to define and call a function expression:

```
var sayHello = function(name) {

  console.log("Hello, " + name + "!");

};

sayHello("John"); // Output: Hello, John!
```

3. **Arrow Functions**:

Arrow functions provide a more concise syntax for defining functions. They are similar to function expressions, but with a simpler syntax and an implicit return value. Here's the basic syntax for an arrow function:

```
var functionName = (parameters) => {

  // code to execute

};
```

Here's an example of how to define and call an arrow function:

```
var sayHello = (name) => {

  console.log("Hello, " + name + "!");

};

sayHello("John"); // Output: Hello, John!
```

4. **Anonymous Functions**:

An anonymous function is a function without a name. Anonymous functions are often used as callback functions, or as functions that are immediately invoked. Here's an example of how to define and call an anonymous function:

```
var addNumbers = function(a, b) {

  return a + b;

}
```

5. **Named Functions**:

A named function is a function that has a name and can be called using that name. Here is an example of a named function:

```
function addNumbers(a, b) {

  return a + b;

}
```

You can call this function by using its name, like this:

```
var result = addNumbers(2, 3);

console.log(result); // Output: 5
```

6. **<u>IIFE (Immediately Invoked Function Expressions)</u>**:

An IIFE is a function that is executed immediately after it is created. This can be useful for creating private scopes or for executing code that needs to run once. Here is an example of an IIFE:

```
(function() {

  // code to execute

})();
```

The parentheses around the function expression are necessary to tell JavaScript that this is a function expression and not a function declaration.

7. **<u>Recursive Functions</u>**:

A recursive function is a function that calls itself. Recursive functions can be used to solve problems that can be broken down into smaller sub-problems. Here is an example of a recursive function:

```
function factorial(n) {

  if (n === 0) {

    return 1;

  } else {

    return n * factorial(n - 1);

  }

}
```

This function calculates the factorial of a number by calling itself with a smaller value until it reaches the base case (n = 0).

These are some of the most common types of functions in JavaScript. By using these functions effectively, you can create powerful and flexible programs that can handle a wide range of tasks.

```
}
```

Declaring & Invoking Function:

- **Declaring a Function in JavaScript**:

To declare a function in JavaScript, we use the **function** keyword followed by the function name and parentheses. Inside the parentheses, we can list the parameters that the function takes. Then, we use curly braces to define the body of the function where we write the code that the function executes.

Here's an example of declaring a function that takes two parameters and returns their sum:

```
function addNumbers(a, b) {
  return a + b;
}
```

- **Invoking a Function in JavaScript**:

To invoke or call a function in JavaScript, we use the function name followed by parentheses. Inside the parentheses, we can pass arguments that the function takes.

Here's an example of invoking the **addNumbers** function that we declared above:

```
var result = addNumbers(2, 3);

console.log(result); // Output: 5
```

In both of these examples, we declare a function using a different syntax, but we can still invoke the function in the same way using the function name and parentheses with arguments inside.

Arrow Function:

Arrow functions are a shorthand syntax for writing function expressions in JavaScript. They were introduced in ECMAScript 6 and provide a more concise way of writing functions.

Here's an example of an arrow function that takes two parameters and returns their sum:

```
const addNumbers = (a, b) => {
  return a + b;
}
```

Here's another example of an arrow function that takes an array of numbers and returns the sum of all the numbers:

```
const sumArray = (arr) => {
  let sum = 0;
  arr.forEach((num) => sum += num);
  return sum;
}
```

Arrow functions are a powerful feature in JavaScript that can make your code more concise and easier to read. They're especially useful when working with arrays and higher-order functions like **map**, **filter**, and **reduce**.

Function Parameters:

In JavaScript, functions can take parameters, which are inputs that are passed to the function when it is called. These parameters allow the function to be more versatile and handle a wider range of input values.

The syntax for declaring function parameters is to list them inside the parentheses after the function name, separated by commas. For example:

```
function myFunction(param1, param2, param3) {
  // function body
}
```

In this example, **myFunction** takes three parameters: **param1**, **param2**, and **param3**. These parameters can then be used inside the function to perform some action or computation. For example:

```
function addNumbers(num1, num2) {
  return num1 + num2;
}
console.log(addNumbers(5, 7)); // Output: 12
```

In this example, **addNumbers** takes two parameters **num1** and **num2**, and returns their sum. When the function is called with the arguments **5** and **7**, it returns **12**.

It is worth noting that JavaScript does not enforce strict typing for function parameters, which means that parameters can be of any data type. It is the responsibility of the function developer to ensure that the function behaves correctly when passed different data types.

Nested Functions in JavaScript:

In JavaScript, a nested function is a function that is defined inside another function. The nested function can be accessed only within the scope of the outer function.

Here's an example of a nested function:

```
function outerFunction() {
  console.log('Outer function is called.');
  function innerFunction() {
    console.log('Inner function is called.');
  }
  innerFunction();
}
outerFunction();
```

In this example, **innerFunction** is defined inside **outerFunction**. When **outerFunction** is called, it logs a message to the console and then calls **innerFunction**, which logs another message to the console.

Nested functions can be useful in JavaScript because they can be used to encapsulate functionality and keep it private to the outer function. This can be useful for creating modular code and avoiding naming conflicts.

One common use case for nested functions is when creating closures. A closure is a function that has access to variables in its lexical scope, even after the outer function has returned. Here's an example:

```
function outerFunction() {
  let counter = 0;
  function incrementCounter() {
    counter++;
    console.log('Counter value:', counter);
  }
  return incrementCounter;
}
const myCounter = outerFunction();
myCounter(); // Output: Counter value: 1
myCounter(); // Output: Counter value: 2
myCounter(); // Output: Counter value: 3
```

Built-in-Functions:

JavaScript has a wide range of built-in functions, also known as global functions, that are available for use without needing to define them first. These functions are built into the JavaScript language and are available globally to all scripts.

Here are some commonly used built-in functions in JavaScript:

- **console.log()**: This function is used to output data to the console for debugging purposes. It takes any number of arguments, which are printed to the console as separate messages.
- **parseInt()**: This function converts a string to an integer. It takes a string as an argument and returns an integer.
- **parseFloat()**: This function converts a string to a floating-point number. It takes a string as an argument and returns a number.
- **alert()**: This function creates a pop-up window with a message. It takes a string as an argument and displays it in the pop-up window.
- **confirm()**: This function creates a pop-up window with a message and two buttons, "OK" and "Cancel". It returns true if the user clicks "OK" and false if the user clicks "Cancel".
- **prompt()**: This function creates a pop-up window with a message and an input field for the user to enter data. It returns the value entered by the user as a string.
- **setTimeout()**: This function executes a function after a specified amount of time. It takes two arguments: the function to execute and the time delay in milliseconds.
- **setInterval()**: This function executes a function repeatedly at a specified interval. It takes two arguments: the function to execute and the time interval in milliseconds.

These are just a few of the built-in functions available in JavaScript. There are many more functions that provide powerful functionality for manipulating data, working with the DOM, handling events, and more.

Variable Scope Functions:

In JavaScript, variable scope determines the visibility and accessibility of a variable within a program. In particular, variable scope in functions refers to the visibility and accessibility of variables defined inside and outside a function.

In JavaScript, there are two types of variable scope: global scope and local scope. Global scope refers to variables that are defined outside of a function and are accessible everywhere in the program. Local scope refers to variables that are defined inside a function and are only accessible within that function.

Here's an example to illustrate variable scope in functions:

```
let globalVariable = 'I am global';

function myFunction() {

  let localVariable = 'I am local';

  console.log(globalVariable);

  console.log(localVariable);

}

myFunction(); // Output: I am global, I am local

console.log(globalVariable); // Output: I am global

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

When **myFunction** is called, it logs the value of **globalVariable** and **localVariable** to the console. When **globalVariable** is accessed inside the function, it is found because it has global scope. When **localVariable** is accessed outside the function, it results in an error because it has local scope and is not accessible outside the function.

It is important to be aware of variable scope in functions because it can affect the behavior and performance of a program. In particular, it is important to avoid naming conflicts between variables with the same name but different scopes.

Working With Classes:

In JavaScript, classes were introduced in ECMAScript 2015 as a new syntax for creating objects and defining their properties and methods. Classes are a template for creating objects, and they can be thought of as blueprints for objects. Here's an example of a class in JavaScript:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
}
const john = new Person('John', 30);
john.greet(); // Output: Hello, my name is John and I'm 30 years old.
```

Classes in JavaScript also support inheritance, which allows you to create a new class based on an existing class. Here's an example:

```
class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);
    this.grade = grade;
  }
  study() {
    console.log(`${this.name} is studying.`);
  }
}
const jane = new Student('Jane', 20, 'A');
```

```
jane.greet(); // Output: Hello, my name is Jane and I'm 20 years old.

jane.study(); // Output: Jane is studying.
```

To create an object from the **Student** class, the same **new** keyword is used along with the **Student** class name and any arguments required by the constructor method. The jane instance is then created with the **name** property set to "**Jane**", the **age** property set to **20**, and the **grade** property set to "**A**". The greet and study methods can then be called on the **jane** instance to log messages to the console.

Creating & Inheriting Classes:

In JavaScript, you can create classes using the class keyword, and you can inherit from a parent class using the extends keyword. Here is an example of creating and inheriting classes in JavaScript:

```
class Animal {

  constructor(name) {

    this.name = name;

  }

  speak() {

    console.log(`${this.name} makes a noise.`);

  }

}

class Dog extends Animal {

  constructor(name) {

    super(name);

  }

 speak() {

    console.log(`${this.name} barks.`);

  }
}

const dog = new Dog('Fido');

dog.speak(); // Output: Fido barks.
```

We then create a child class Dog that extends the Animal class using the extends keyword. The Dog class also has a constructor method that calls the super method to invoke the constructor of the parent class and set the name property. The Dog class overrides the speak method of the parent class to log a different message.

Finally, we create an instance of the Dog class called dog and call the speak method on it, which logs the message "Fido barks." to the console.

In addition to inheriting properties and methods from the parent class, a child class can also define its own properties and methods. Here is an example:

```
class Animal {

  constructor(name) {

    this.name = name;

  }

  speak() {

    console.log(`${this.name} makes a noise.`);

  }

}

class Dog extends Animal {

  constructor(name, breed) {

    super(name);

    this.breed = breed;

  }

  speak() {
```

```
  console.log(`${this.name} barks.`);
 }
 fetch() {
  console.log(`${this.name} fetches a ball.`);
 }
}
const dog = new Dog('Fido', 'Labrador');
dog.speak(); // Output: Fido barks.
dog.fetch(); // Output: Fido fetches a ball.
console.log(dog.breed); // Output: Labrador
```

In summary, you can create classes in JavaScript using the class keyword, and you can inherit from a parent class using the extends keyword. Child classes can override parent class methods and define their own properties and methods.

In-Built Events & Handlers:

In JavaScript, there are many built-in events that you can use to trigger actions in your web page or application. You can use event handlers to respond to these events and execute code when the events occur.

Here are some of the most common built-in events and their associated event handlers in JavaScript:

1. **Click Event** - This event occurs when a user clicks on an element, such as a button or a link. The event handler for this event is the **onclick** event handler.

Example:

```
<button onclick="myFunction()">Click me</button>
<script>
function myFunction() {
  alert("Hello World!");
}

</script>
```

2. **Submit Event** - This event occurs when a user submits a form. The event handler for this event is the **onsubmit** event handler.

Example:

```
<button onclick="myFunction()">Click me</button>

<script>

function myFunction() {

  alert("Hello World!");

}

</script>
```

3. **Mouseover Event** - This event occurs when a user moves their mouse over an element. The event handler for this event is the **onmouseover** event handler.

Example:

```
<div onmouseover="myFunction()">Hover over me</div>

<script>

function myFunction() {

  alert("Mouse over!");

}

</script>
```

4. **Keydown Event** - This event occurs when a user presses a key on their keyboard. The event handler for this event is the **onkeydown** event handler.

Example:

```
<input type="text" onkeydown="myFunction()">
<script>
```

```
function myFunction() {
  alert("Key down!");
}
</script>
```

5. **Load Event** - This event occurs when a web page or image has finished loading. The event handler for this event is the onload event handler.

Example:

```
<img src="myimage.jpg" onload="myFunction()">

<script>

function myFunction() {

  alert("Image loaded!");

}

</script>
```

These are just a few examples of the many built-in events and event handlers available in JavaScript. By using these events and handlers, you can create dynamic and interactive web pages and applications.

Working with Objects:

In JavaScript, objects are a fundamental data structure that allows you to store and manipulate data in a flexible way. Objects are collections of properties, where each property has a key and a value.

Here's an example of how to create a simple object in JavaScript:

```
let person = {

  name: 'John',

  age: 30,

  gender: 'male'

};
```

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

```
// Dot notation
```

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


// Bracket notation

console.log(person['age']); // Output: 30
```

You can also add or modify properties of an object:

```
person.email = 'john@example.com'; // Add a new property

person.age = 31; // Modify an existing property
```

JavaScript also has built-in functions called constructors that allow you to create objects based on a blueprint or template. Here's an example:

```
function Person(name, age, gender) {

  this.name = name;

  this.age = age;

  this.gender = gender;

}

let person1 = new Person('John', 30, 'male');

let person2 = new Person('Jane', 25, 'female');

console.log(person1.name); // Output: 'John'

console.log(person2.age); // Output: 25
```

JavaScript also has a built-in **Object** class that provides many useful methods for working with objects, such as **Object.keys()**, **Object.values()**, and **Object.entries()**. Here's an example:

```
let person = {

  name: 'John',

  age: 30,

  gender: 'male'
};

console.log(Object.keys(person)); // Output: ['name', 'age', 'gender']

console.log(Object.values(person)); // Output: ['John', 30, 'male']

console.log(Object.entries(person)); // Output: [['name', 'John'], ['age', 30], ['gender', 'male']]
```

These are just a few examples of the many ways to work with objects in JavaScript. By using objects, you can create powerful and flexible data structures that can be used in a variety of applications.

Types of Objects:

In JavaScript, there are several types of objects that you can work with. Here are some of the most common types of objects:

1. **Built-in Objects** - These are objects that are built into the JavaScript language, such as **Object**, **Array**, **String**, and **Number**.

Example:

```
let person = {

  name: 'John',

  age: 30,

  gender: 'male'

};

let names = ['John', 'Jane', 'Joe'];

let message = 'Hello, world!';

let number = 42;
```

2. **Custom Objects** - These are objects that you create yourself using constructors or classes.

Example:

```
function Person(name, age, gender) {

  this.name = name;
  this.age = age;

  this.gender = gender;

}

let person1 = new Person('John', 30, 'male');

class Animal {

  constructor(name, species) {

    this.name = name;

    this.species = species;

  }

}

let animal1 = new Animal('Fluffy', 'cat');
```

3. **Document Objects** - These are objects that represent elements in an HTML document, such as **document**, **window**, and **HTMLElement**.

Example:

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

window.alert('Hello, world!');

document.title = 'My Website';
```

4. **Host Objects** - These are objects that are provided by the environment in which JavaScript is running, such as the **XMLHttpRequest** object for making HTTP requests in a web browser.

Example:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', 'https://api.example.com/data');

xhr.onload = function() {

  console.log(xhr.response);
};

xhr.send();
```

These are just a few examples of the many types of objects that you can work with in JavaScript. By understanding the different types of objects and how they work, you can create powerful and flexible applications that take advantage of JavaScript's object-oriented features.

Creating Objects:

In JavaScript, you can create objects in several ways:

1. **Object Literal Syntax**:

This is the simplest way to create an object. You can define an object using a pair of curly braces, and then specify key-value pairs separated by commas.

Example:

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

2. **Constructor Function**:

You can create an object using a constructor function. This involves creating a function that serves as a blueprint for the object, and then using the "new" keyword to create a new instance of the object.

Example:

```
function Person(name, age, hobbies) {
  this.name = name;
  this.age = age;
  this.hobbies = hobbies;
}

const person = new Person("John", 30, ["reading", "running"]);
```

3. **Object.create() method**:

This method allows you to create a new object with a specified prototype. You can specify the prototype using an existing object or null.

Example:

```
const personProto = {

  greet() {

    console.log("Hello, my name is " + this.name);

  }

};

const person = Object.create(personProto);

person.name = "John";

person.age = 30;

person.hobbies = ["reading", "running"];
```

4. **ES6 Class Syntax**:

You can create an object using the ES6 class syntax, which is essentially a syntactic sugar over the constructor function approach.

Example:

```
class Person {

  constructor(name, age, hobbies) {

    this.name = name;

    this.age = age;

    this.hobbies = hobbies;
  }

}

const person = new Person("John", 30, ["reading", "running"]);
```

Combining & Cloning Objects Using Spread Operator:

In JavaScript, you can use the spread operator (...) to combine and clone objects. The spread operator can be used to spread the properties of one object into another object, and also to create a new object with the properties of an existing object.

1. **Combining objects using the spread operator**:

You can combine objects by spreading their properties into a new object. This can be useful when you want to merge two or more objects into a single object.

Example:

```
const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3, d: 4 };

const combinedObj = { ...obj1, ...obj2 };

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

2. **Cloning objects using the spread operator**:

You can clone an object by spreading its properties into a new object. This creates a new object with the same properties as the original object, but they are not reference equal.

Example:

```
const obj1 = { a: 1, b: 2 };

const clonedObj = { ...obj1 };

console.log(clonedObj); // Output: { a: 1, b: 2 }

console.log(obj1 === clonedObj); // Output: false
```

To perform a deep copy of an object, you can use a library such as lodash, or you can write your own recursive function that clones nested objects.

Destructing Objects:

In JavaScript, object destructuring is a way to extract properties from an object and assign them to variables with the same name as the property. This can make your code more concise and easier to read.

Here's an example of how to destructure an object:

```
const person = { name: 'John', age: 30, hobbies: ['reading', 'running'] };

const { name, age, hobbies } = person;

console.log(name);     // Output: 'John'

console.log(age);      // Output: 30

console.log(hobbies);  // Output: ['reading', 'running']
```

You can also use destructuring to assign default values to variables if the property does not exist on the object:

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

const { name, age, hobbies = [] } = person;

console.log(hobbies);  // Output: []
```

You can also use destructuring with nested objects:

```
const person = {

  name: 'John',

  age: 30,

  address: {

    street: '123 Main St',

    city: 'Anytown',

    state: 'CA'

  }
};

const { name, age, address: { city } } = person;

console.log(city);  // Output: 'Anytown'
```

Browser & Document Object Model:

In JavaScript, the Browser Object Model (BOM) and the Document Object Model (DOM) are two important APIs that allow JavaScript code to interact with the web browser and the document displayed in it.

The Browser Object Model (BOM) provides JavaScript access to the browser's window and its various components such as the location bar, history, and screen. The BOM is not standardized and varies between different browsers. Some commonly used BOM methods include:

- **window.alert()** - displays an alert box with a message and an OK button.
- **window.prompt()** - displays a dialog box that prompts the user for input.
- **window.open()** - opens a new browser window or tab.
- **window.close()** - closes the current browser window.

The Document Object Model (DOM) provides JavaScript access to the HTML document displayed in the browser. The DOM represents the document as a tree-like structure of nodes, and allows JavaScript code to manipulate the structure and content of the document. Some commonly used DOM methods include:

- **document.getElementById()** - returns the element with the specified ID.
- **document.createElement()** - creates a new HTML element.
- **element.appendChild()** - adds a new child node to an element.
- **element.innerHTML** - gets or sets the HTML content of an element.

Here's an example of how to use the DOM to manipulate the content of an HTML page:

```
<!DOCTYPE html>

<html>

<head>

  <title>My Page</title>

</head>

<body>

  <h1 id="myHeading">Hello World!</h1>
```

```
 <button id="myButton">Click me</button>
 <script>
   const heading = document.getElementById('myHeading');
   const button = document.getElementById('myButton');
   button.addEventListener('click', function() {
     heading.innerHTML = 'Hello JavaScript!';
   });
 </script>
</body>
</html>
```

Creating Arrays:

In JavaScript, arrays are a type of object that can store multiple values in a single variable. Here are some ways to create arrays in JavaScript:

1. **Literal notation**: You can create an array using literal notation by enclosing a comma-separated list of values in square brackets []:
   ```
   const fruits = ['apple', 'banana', 'orange'];
   ```
2. **Constructor notation**: You can also create an array using the Array constructor and passing in the initial values as arguments:
   ```
   const numbers = new Array(1, 2, 3, 4, 5);
   ```
3. **Empty array**: You can create an empty array by using literal notation with no values between the brackets or by using the Array constructor with no arguments:
   ```
   const emptyArray1 = [];
   const emptyArray2 = new Array();
   ```

Once you have created an array, you can access its elements using bracket notation and the index of the element, which starts at 0:

```
const fruits = ['apple', 'banana'];
console.log(fruits[0]);  // Output: 'apple'
console.log(fruits[1]);  // Output: 'banana'
```

You can also modify the values of individual elements or add new elements to the array:

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

fruits[1] = 'kiwi';

fruits.push('grape');

console.log(fruits);  // Output: ['apple', 'kiwi', 'orange', 'grape']
```

Destructing Arrays:

In JavaScript, destructuring is a way to extract individual values from arrays and objects and assign them to variables using a concise syntax. Here's an example of how to use array destructuring:

```
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
```

Array destructuring can also be used to assign default values to variables in case the array does not have enough values:

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

const [a = 0, b = 0, c = 0] = numbers;

console.log(a); // Output: 1

console.log(b); // Output: 2

console.log(c); // Output: 0
```

You can also use array destructuring in function parameters to extract values from arrays passed as arguments:

```
function sum([a, b, c]) {

  return a + b + c;
}

const numbers = [1, 2, 3];

console.log(sum(numbers)); // Output: 6
```

Accessing Arrays:

In JavaScript, you can access individual elements of an array using their index. Array indices start at 0, so the first element of an array has an index of 0, the second element has an index of 1, and so on. Here are some examples of how to access elements in an array:

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

console.log(numbers[0]); // Output: 1

console.log(numbers[2]); // Output: 3

console.log(numbers[numbers.length - 1]); // Output: 5
```

You can also use array methods to access and manipulate arrays. Here are some examples:

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

console.log(fruits.indexOf('banana')); // Output: 1

fruits.push('grape');

console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']

fruits.pop();

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

In the first example, we declare an array fruits and use the indexOf() method to find the index of the string 'banana' in the array. The output is 1.

In the second example, we use the push() method to add the string 'grape' to the end of the fruits array. The output is ['apple', 'banana', 'orange', 'grape'].

In the third example, we use the pop() method to remove the last element of the fruits array. The output is ['apple', 'banana', 'orange'].

Array Methods:

In JavaScript, arrays come with many built-in methods that make it easy to perform common tasks such as adding, removing, and sorting elements in an array. Here are some examples of commonly used array methods:

1. **push()** - adds one or more elements to the end of an array.

```
const fruits = ['apple', 'banana'];
fruits.push('orange', 'grape');
console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']
```

2. **pop()** - removes the last element from an array and returns it.

```
const fruits = ['apple', 'banana', 'orange'];
const lastFruit = fruits.pop();
console.log(lastFruit); // Output: 'orange'
console.log(fruits); // Output: ['apple', 'banana']
```

3. **shift()** - removes the first element from an array and returns it.

```
const fruits = ['apple', 'banana', 'orange'];
const firstFruit = fruits.shift();
console.log(firstFruit); // Output: 'apple'
console.log(fruits); // Output: ['banana', 'orange']
```

4. **unshift()** - adds one or more elements to the beginning of an array.

```
const fruits = ['apple', 'banana'];
fruits.unshift('orange', 'grape');
console.log(fruits); // Output: ['orange', 'grape', 'apple', 'banana']
```

5. **slice()** - returns a new array containing a portion of an existing array.

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

const citrus = fruits.slice(2);

console.log(citrus); // Output: ['orange', 'grape']
```

6. **splice()** - adds or removes elements from an array at a specific position.

```
const fruits = ['apple', 'banana', 'orange', 'grape'];
fruits.splice(2, 1, 'pear');
console.log(fruits); // Output: ['apple', 'banana', 'pear', 'grape']
```

7. **concat()** - returns a new array by combining two or more arrays.

```
const fruits1 = ['apple', 'banana'];
const fruits2 = ['orange', 'grape'];
const allFruits = fruits1.concat(fruits2);
console.log(allFruits); // Output: ['apple', 'banana', 'orange', 'grape']
```

8. **sort()** - sorts the elements of an array.

```
const fruits = ['orange', 'apple', 'banana'];
fruits.sort();
```

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

9. **reverse()** - reverses the order of the elements in an array.

```
const fruits = ['apple', 'banana', 'orange'];
fruits.reverse();
console.log(fruits); // Output: ['orange', 'banana', 'apple']
```

10. **join()** - converts all the elements in an array into a string and concatenates them.

```
const fruits = ['apple', 'banana', 'orange'];
const fruitString = fruits.join(', ');
console.log(fruitString); // Output: 'apple, banana, orange'
```

These are just a few examples of the many array methods available in JavaScript. Understanding and using these methods can make working with arrays much easier and more efficient.

Introduction to Asynchronous Programming:

Asynchronous programming in JavaScript is a programming model that allows the program to continue running while a task is being executed in the background. This allows the program to perform other tasks while waiting for a response from a long-running process, such as making a network request.

Traditionally, JavaScript is a single-threaded language, meaning that it can only execute one task at a time. However, asynchronous programming allows us to write code that can perform multiple tasks at the same time.

Asynchronous programming in JavaScript is often achieved using callbacks, promises, and async/await.

Callbacks are functions that are passed as arguments to other functions, and are executed when the function is complete. For example, when making a network request, we might pass a callback function that is executed when the request is complete.

Promises are a newer feature in JavaScript that provide a cleaner way of handling asynchronous code. A promise is an object that represents a value that may not be available yet. It has three states: pending, fulfilled, and rejected. A promise is fulfilled when the task is complete, and rejected if there was an error.

Async/await is a newer syntax in JavaScript that makes asynchronous programming even easier. The async keyword is used to define a function as asynchronous, and the await keyword is used to pause the function until the asynchronous task is complete.

Here's an example of using async/await to make an asynchronous network request:

```
async function getData() {

  try {
```

```
  const response = await fetch('https://example.com/data.json');

  const data = await response.json();

  console.log(data);

 } catch (error) {

  console.error(error);

 }

}

getData();
```

In this example, the getData function is defined as asynchronous using the async keyword. The await keyword is used to pause the function until the fetch and json tasks are complete. The try/catch block is used to handle any errors that may occur during the asynchronous tasks.

Asynchronous programming is an important concept in modern JavaScript development, and it's essential for creating responsive and efficient applications. Understanding how to use callbacks, promises, and async/await can greatly improve the performance and user experience of your JavaScript applications.

Callbacks:

Callbacks in JavaScript are functions that are passed as arguments to other functions and are executed when certain events occur. Callbacks are often used in asynchronous programming, where a task may take some time to complete, and we want to execute some code when the task is done.

Here's a simple example of using a callback to handle an event:

```
function buttonClicked(callback) {

  // Do some work...

  console.log('Button clicked!');

  // Call the callback function

  callback();

}
```

```
buttonClicked(function() {
  console.log('Callback function executed!');
});
```

Callbacks can also be used with asynchronous tasks, such as making an HTTP request. Here's an example of using a callback to handle the response from an HTTP request:

```
function makeRequest(url, callback) {
  let xhr = new XMLHttpRequest();
  xhr.open("GET", url);
  xhr.onload = function() {
    if (xhr.status === 200) {
      callback(xhr.response);
    } else {
      callback('Error');
    }
  };
  xhr.send();
}
makeRequest('https://jsonplaceholder.typicode.com/posts/1', function(data) {
  console.log(data);
});
```

Promises:

Promises in JavaScript are a way to handle asynchronous operations that return a value or throw an error. They are an alternative to using callbacks, and provide a cleaner and more structured way of handling asynchronous code.

A Promise is an object that represents a value that may not be available yet, but will be at some point in the future. It has three states:

- **Pending**: The initial state, before the operation is completed.
- **Fulfilled**: The operation completed successfully, and the promised value is available.
- **Rejected**: The operation failed, and an error was thrown.

A Promise can be created using the Promise constructor, which takes a single function as an argument. This function has two parameters: resolve and reject. The resolve function is called when the operation completes successfully, and the reject function is called when the operation fails.

Here's an example of creating a Promise that resolves after a short delay:

```
const promise = new Promise((resolve, reject) => {

  setTimeout(() => {

    resolve('Hello, world!');

  }, 1000);

});

promise.then((result) => {

  console.log(result);

});
```

Promises can also be used to handle errors. Here's an example of creating a Promise that rejects with an error after a short delay:

```
const promise = new Promise((resolve, reject) => {

  setTimeout(() => {

    reject(new Error('Something went wrong!'));

  }, 1000);
});

promise.catch((error) => {

  console.log(error.message);

});
```

Async & Await:

Async/Await is a newer way of handling asynchronous operations in JavaScript, which provides a more readable and synchronous-like way of writing asynchronous code. async/await is built on top of Promises and is available in modern versions of JavaScript.

The async keyword is used to define an asynchronous function, which returns a Promise that will be resolved with the value returned by the function or rejected with an error thrown by the function. The await keyword is used inside an async function to pause the execution of the function until a Promise is resolved. When a Promise is resolved, await returns the resolved value.

Here's an example of using async/await to fetch data from an API:

```
async function getData() {

  const response = await fetch('https://jsonplaceholder.typicode.com/users/1');

  const data = await response.json();

  return data;

}

getData()

  .then((data) => console.log(data))

  .catch((error) => console.error(error));
```

async/await can also be used to chain asynchronous operations together. Here's an example:

```
async function getUserData() {

  const response = await fetch('https://jsonplaceholder.typicode.com/users/1');

  const userData = await response.json();
  const response2 = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userData.id}`);

  const userPosts = await response2.json();

  return { userData, userPosts };

}

getUserData()

  .then((data) => console.log(data))

  .catch((error) => console.error(error));
```

In this example, we define an async function called getUserData that fetches data from two different endpoints and returns an object containing the user data and posts. We then call the getUserData function and log the returned data to the console.

Executing Network Requests using Fetch API:

The Fetch API is a modern way of making network requests in JavaScript. It provides a simple and standardized way of fetching resources from the network, such as JSON data or images, and can be used in both the browser and Node.js environments.

To make a basic GET request using the Fetch API, you can use the fetch() function. Here's an example:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')

  .then((response) => response.json())

  .then((data) => console.log(data))

  .catch((error) => console.error(error));
```

You can also include additional options, such as headers or a request method other than GET, by passing a configuration object as the second argument to the fetch() function. Here's an example:

```
fetch('https://jsonplaceholder.typicode.com/todos', {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json'
```

```
 },
 body: JSON.stringify({
   title: 'My todo',
   completed: false,
   userId: 1
 })
})
 .then((response) => response.json())
 .then((data) => console.log(data))
 .catch((error) => console.error(error));
```

The Fetch API can also be used to handle errors, such as a network failure or an invalid response from the server. You can use the ok property of the Response object to check whether the response was successful, and the status and statusText properties to get more information about the response. Here's an example:

```
fetch('https://jsonplaceholder.typicode.com/todos/123456789')
 .then((response) => {
   if (!response.ok) {
     throw new Error(`HTTP error! status: ${response.status} ${response.statusText}`);
   }
   return response.json();
 })
 .then((data) => console.log(data))
 .catch((error) => console.error(error));
```

Creating & Consuming Modules:

In JavaScript, modules are a way to organize and encapsulate code. They allow you to split your code into separate files and reuse code across different parts of your application. Modules also help to avoid naming conflicts and keep your code clean and maintainable.

Creating a module in JavaScript involves exporting functions, classes, or other objects from one file and importing them into another file. Here's an example:

```
// in math.js file

export function add(a, b) {

  return a + b;

}

export function subtract(a, b) {

  return a - b;

}

// in app.js file

import { add, subtract } from './math.js';

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

console.log(subtract(5, 2)); // prints 3
```

You can also export a default value from a module using the export default syntax. Here's an example:

```
// in math.js file

export default function multiply(a, b) {

  return a * b;

}

// in app.js file

import multiply from './math.js';

console.log(multiply(2, 3)); // prints 6
```

When working with modules in the browser, you need to use a module bundler like Webpack or Rollup to bundle your code and handle dependencies. Alternatively, you can use the type="module" attribute in your HTML script tags to load ES modules directly in the browser.

THANK YOU

HAPPY LEARNING