Guide to JavaScript

JavaScript is a high-level, dynamic, and interpreted programming language. It is used to add interactivity and other dynamic elements to websites.

The console in JavaScript

The console in JavaScript is a tool used for debugging and testing purposes. It allows developers to log information to the browser console for viewing. This can be useful for checking the values of variables, examining the output of functions, and tracking down errors in code.

There are several methods to log information to the console in JavaScript:

console.log(): This method logs the specified data to the console.
For example:

```
console.log("Hello World"); // Output: Hello World
```

console.error(): This method logs an error message to the console.
For example:

```
console.error("This is an error"); // Output: Error:
This is an error
```

console.warn(): This method logs a warning message to the console.
For example:

```
console.warn("This is a warning"); // Output: Warning:
This is a warning
```

These methods can be used directly in the JavaScript code and the output can be viewed in the browser console by opening the Developer Tools in most modern browsers.

Here are some basics of JavaScript with code examples:

Variables:

variables are used to store values in JavaScript. You can declare a variable with the "let" keyword, like this:

```
let x = 10;
let name = "John";
let age = 30;
let isStudent = false;
let weight = 75.5;
let address = "123 Main St.";
```

JavaScript variables are containers that store data values. There are three main types of variables in JavaScript:

var: This is the original way to declare a variable in JavaScript. It is function scoped, which means it is accessible within the function it was declared in.

For example:

```
var name = "John";
function printName() {
  console.log(name);
}
printName(); // Output: John
```

let: This type of variable was introduced in ECMAScript 6 and is block scoped. It means that the variable is only accessible within the block of code it was declared in.

For example:

```
let age = 30;
if (true) {
  let age = 40;
  console.log(age); // Output: 40
}
console.log(age); // Output: 30
```

const: This type of variable is also block scoped and was introduced in ECMAScript 6. The difference between const and let is that const cannot be reassigned after it has been declared.

For example:

```
const country = "USA";
country = "Canada"; // Throws an error
```

It is important to note that the value stored inside a variable declared with const can still be mutable, meaning its properties can be changed but it cannot be reassigned to a completely new value.

JavaScript Comments

In JavaScript, you can add comments in your code to describe what it does or to temporarily ignore parts of the code. There are two types of comments: single-line comments and multi-line comments.

Single-line comment:

```
// This is a single-line comment
Multi-line comment:
/*
  This is a multi-line
  comment
*/
```

Here's an example that shows how comments can be used in a code:

```
// This is a single-line comment
/*
  This is a multi-line
  comment
*/

const name = "John"; // This is also a single-line comment
// The code below will print the value of the variable "name"
console.log(name);
```

Data Types:

JavaScript has several data types, including numbers, strings, and booleans. Here's an example of each:

```
let num = 10;
let str = "Hello World";
let bool = true;
let num1 = 20;
let num2 = -10;
let str1 = "Hello";
let str2 = 'JavaScript';
let bool1 = true;
let bool2 = false;
```

JavaScript Data Types

In JavaScript, there are seven basic data types:

Number: Used to represent numbers. Example: const num = 42;

String: Used to represent a sequence of characters. Example: const name = "John";

Boolean: Used to represent a logical value of either true or false. Example: const isMarried = true;

Undefined: Represents a value that has not been assigned. Example: const age; console.log(age); // Output: undefined

Null: Represents a value that is explicitly null. Example: const address = null;

Symbol: Used to create unique identifiers for objects. Example: const symbol = Symbol("description");

Object: Used to represent a collection of key-value pairs. Example: const person = { name: "John", age: 30 };

In JavaScript, variables have a dynamic type and can change their type based on the value assigned to them. The typeof operator can be used to check the type of a variable.

Example:

```
const num = 42;
console.log(typeof num); // Output: "number"

const name = "John";
console.log(typeof name); // Output: "string"

const isMarried = true;
console.log(typeof isMarried); // Output: "boolean"

const age;
console.log(typeof age); // Output: "undefined"
```

const address = null;

console.log(typeof address); // Output: "object"

const symbol = Symbol("description");

console.log(typeof symbol); // Output: "symbol"

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

console.log(typeof person); // Output: "object"

Arithmetic Operations:

JavaScript supports basic arithmetic operations like addition, subtraction, multiplication, and division. Here's an example:

```
let x = 10;
let y = 5;
let sum = x + y;
let difference = x - y;
let product = x * y;
let quotient = x / y;
let x = 10;
let y = 20;
let sum = x + y;
let difference = x - y;
let product = x * y;
let quotient = x / y;
let modulo = x % y;
let increment = x++;
let decrement = y--;
```

Conditional Statements:

Conditional statements are used to execute different blocks of code based on conditions. The if-else statement is the most common type of conditional statement in JavaScript. Here's an example:

```
let x = 10;
if (x > 5) {
  console.log("x is greater than 5");
} else {
  console.log("x is not greater than 5");
}
let grade = 85;

if (grade >= 90) {
  console.log("A");
} else if (grade >= 80) {
  console.log("B");
} else if (grade >= 70) {
  console.log("C");
} else {
  console.log("F");
}

let day = "Sunday";

switch (day) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  case "Wednesday":
    console.log("Today is Wednesday");
    break;
  case "Thursday":
    console.log("Today is Thursday");
    break;
  case "Friday":
    console.log("Today is Friday");
    break;
  case "Saturday":
    console.log("Today is Saturday");
    break;
  default:
    console.log("Today is Sunday");
}
```

Functions:

Functions are blocks of code that can be executed when they are called. Here's an example:

```
function greeting() {
  console.log("Hello World");
}

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

let result = add(10, 20);
console.log(result);
```

```
function greet(name) {
  console.log("Hello " + name);
}

greet("John");

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

let area = calculateArea(10, 20);
console.log(area);

function checkOddEven(num) {
  if (num % 2 === 0) {
    return "Even";
  } else {
    return "Odd";
  }
}

let result = checkOddEven(10);
console.log(result);
```

```
function generateRandomNumber() {
  return Math.floor(Math.random() * 100);
}

let randomNumber = generateRandomNumber();
console.log(randomNumber);
```

These are just some of the basics of JavaScript. There is much more to learn, but these basics will give you a good foundation to start building your skills.

JavaScript Loops:

JavaScript has two types of loops: for loops and while loops. Here are examples of each:

For Loop:

```
for (const i = 0; i < 5; i++) {
  console.log("Iteration " + (i + 1));
}

This code will output:
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

While Loop:

```
const i = 0;
while (i < 5) {
  console.log("Iteration " + (i + 1));
  i++;
}
```

This code will output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

In addition to these two loops, there is also the do-while loop, which is similar to the while loop, but with a slight difference in the way the condition is checked. Here's an example:

```
const i = 0;
do {
  console.log("Iteration " + (i + 1));
  i++;
} while (i < 5);
```

```
This code will output:
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

These are the basics of JavaScript loops. You can use them to repeat a block of code multiple times based on conditions.

JavaScript Arrays

An array in JavaScript is a data structure that stores a collection of values. You can access individual values of an array by referring to their index number. Arrays are declared using square brackets [] and items are separated by commas.

Here's an example of an array in JavaScript:

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

JavaScript provides several built-in methods for working with arrays. Here are some commonly used methods with examples:

length property: returns the number of elements in an array.

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits.length); // Output: 3
```

push() method: adds an element to the end of an array.

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

pop() method: removes the last element from an array and returns it.

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

unshift() method: adds an element to the beginning of an array.

```
const fruits = ["apple", "banana", "cherry"];
fruits.unshift("peach");
```

```
console.log(fruits); // Output: ["peach", "apple",
"banana", "cherry"]
```

shift() method: removes the first element from an array and returns it.

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

splice() method: adds or removes elements from an array.

```
const fruits = ["apple", "banana", "cherry"];
fruits.splice(1, 0, "lemon", "lime");
console.log(fruits); // Output: ["apple", "lemon",
"lime", "banana", "cherry"]
```

These are some of the most commonly used array methods in JavaScript. You can use these methods to manipulate arrays and perform various operations on them.

JavaScript Objects

In JavaScript, an object is a collection of key-value pairs that store data. Objects are declared using curly braces {} and the keys and values are separated by colons.

Here's an example of an object in JavaScript:

```
const person = {
  name: "John",
  age: 30,
  location: "San Francisco"
};
```

You can access the values of an object using the dot notation or square bracket notation.

Here's an example of how to access the values of an object using the dot notation:

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

Here's an example of how to access the values of an object using the square bracket notation:

```
const person = {
  name: "John",
  age: 30,
  location: "San Francisco"
};

const nameKey = "name";
const ageKey = "age";
const locationKey = "location";
console.log(person[nameKey]); // Output: "John"
console.log(person[ageKey]); // Output: 30
console.log(person[locationKey]); // Output: "San
Francisco"
```

You can also add new properties or change the values of existing properties in an object.
Here's an example of how to add a new property to an object:

```
const person = {
  name: "John",
  age: 30,
  location: "San Francisco"
};

person.email = "john@example.com";
```

```
console.log(person); // Output: { name: "John", age:
30, location: "San Francisco", email:
"john@example.com" }
```

Here's an example of how to change the value of an existing property in an object:

```
const person = {
  name: "John",
  age: 30,
  location: "San Francisco"
};
person.age = 35;
console.log(person); // Output: { name: "John", age:
35, location: "San Francisco" }
```

Objects are widely used in JavaScript to store data and represent real-world objects. You can use objects to create more complex data structures and manage your data more effectively.

How to output a table into the console

console.table(): This method logs the data in a table format. For example:

```
console.table([{a:1, b:2}, {a:3, b:4}]);
// Output:
// ┌─────────┬───┬───┐
// │ (index) │ a │ b │
// ├─────────┼───┼───┤
// │    0    │ 1 │ 2 │
// │    1    │ 3 │ 4 │
// └─────────┴───┴───┘
```

JavaScript String Methods

JavaScript provides several built-in methods for manipulating strings, some of the commonly used ones are:

length: Returns the length of the string.
Example: var name = "John"; console.log(name.length); // Output: 4

concat: Joins two or more strings together.
Example: var firstName = "John"; var lastName = "Doe"; console.log(firstName.concat(" ", lastName)); // Output: "John Doe"

toUpperCase: Converts the string to uppercase.

Example: var name = "John"; console.log(name.toUpperCase()); // Output: "JOHN"

toLowerCase: Converts the string to lowercase.
Example: var name = "John"; console.log(name.toLowerCase()); // Output: "john"

charAt: Returns the character at the specified index.
Example: var name = "John"; console.log(name.charAt(0)); // Output: "J"

indexOf: Returns the index of the first occurrence of the specified value, or -1 if it is not found.
Example: var name = "John"; console.log(name.indexOf("o")); // Output: 1

slice: Extracts a part of the string and returns it as a new string.
Example: var name = "John"; console.log(name.slice(0, 2)); // Output: "Jo"

replace: Replaces the first occurrence of the specified value with a new value.
Example: var name = "John"; console.log(name.replace("J", "j")); // Output: "john"

trim: Removes whitespaces from both ends of the string. Example: var name = " John "; console.log(name.trim()); // Output: "John"

These are just a few of the many string methods available in JavaScript, each with its own specific use case. Understanding and utilizing these methods can greatly simplify string manipulation tasks in your code.

JavaScript Number Methods

JavaScript provides several built-in methods for working with numbers. Here are some common ones:

Number.isInteger(): This method returns true if the argument is an integer and false otherwise.

For example:

```
console.log(Number.isInteger(3)); // Output: true
console.log(Number.isInteger(3.14)); // Output: false
```

Number.parseFloat(): This method parses a string argument and returns a floating-point number.

For example:

```
console.log(Number.parseFloat("3.14")); // Output: 3.14
```

Number.parseInt(): This method parses a string argument and returns an integer.

For example:

```
console.log(Number.parseInt("3.14")); // Output: 3
```

Number.toFixed(): This method returns a string representation of a number with a specified number of decimal places.

For example:

```
console.log((3.14).toFixed(2)); // Output: 3.14
```

Number.toPrecision(): This method returns a string representation of a number with a specified number of significant digits.

For example:

```
console.log((3.14).toPrecision(2)); // Output: 3.1
```

Math.abs(): This method returns the absolute value of a number.

For example:

```
console.log(Math.abs(-3.14)); // Output: 3.14
```

Math.ceil(): This method returns the smallest integer greater than or equal to a number.

For example:

```
console.log(Math.ceil(3.14)); // Output: 4
```

Math.floor(): This method returns the largest integer less than or equal to a number.

For example:

```
console.log(Math.floor(3.14)); // Output: 3
```

These methods can be used to perform various operations on numbers in JavaScript.

JavaScript Math

JavaScript provides several built-in methods for working with mathematical operations. Here are some common ones:

Math.abs(): This method returns the absolute value of a number.

For example:

```
console.log(Math.abs(-3.14)); // Output: 3.14
```

Math.ceil(): This method returns the smallest integer greater than or equal to a number.

For example:

```
console.log(Math.ceil(3.14)); // Output: 4
```

Math.floor(): This method returns the largest integer less than or equal to a number.

For example:

```
console.log(Math.floor(3.14)); // Output: 3
```

Math.max(): This method returns the largest of zero or more numbers.

For example:

```
console.log(Math.max(3, 7, 4)); // Output: 7
```

Math.min(): This method returns the smallest of zero or more numbers.

For example:

```
console.log(Math.min(3, 7, 4)); // Output: 3
```

Math.pow(): This method returns the value of a number raised to the specified power.

For example:

```
console.log(Math.pow(3, 2)); // Output: 9
```

Math.random(): This method returns a random number between 0 (inclusive) and 1 (exclusive).

For example:

```
console.log(Math.random()); // Output: a random number
between 0 and 1
```

Math.round(): This method returns the value of a number rounded to the nearest integer.

For example:

```
console.log(Math.round(3.14)); // Output: 3
```

These methods can be used to perform various mathematical operations in JavaScript.

Variables:

```
let name = "John Doe";
let age = 30;
let isStudent = false;
console.log(name);
console.log(age);
console.log(isStudent);
```

Explanation: In this code, we are declaring 3 variables using the let keyword. The let keyword allows us to declare variables in JavaScript. The first variable name is assigned a string value of "John Doe". The second variable age is assigned a number value of 30. The third variable isStudent is assigned a boolean value of false. Finally, we log the values of these variables to the console using the console.log method.

Arrays:

```
let names = ["John", "Jane", "Jim"];
console.log(names[0]);
console.log(names.length);
names.push("Jake");
console.log(names);
```

Explanation: In this code, we are declaring an array named names that contains three string values "John", "Jane", and "Jim". The first console.log statement logs the first element of the array to the console, which is "John". The second console.log statement logs the length of the array to the console, which is 3. The push method is then used to add another element "Jake" to the end of the array. The final console.log statement logs the updated array to the console.

Example : Array

```
let fruits = ["apple", "banana", "cherry"];
for (let i = 0; i < fruits.length; i++) {
  console.log("I like " + fruits[i]);
}
```

Explanation: This example demonstrates the use of an array. We create an array fruits containing three strings "apple", "banana", and "cherry". We then use a for loop to iterate over each element in the array and log a message to the console using console.log(). This will log the message "I like apple", "I like banana", and "I like cherry" to the console.

Example: Object

```
let car = {
  make: "Toyota",
  model: "Camry",
  year: 2022,
};

console.log("I drive a " + car.year + " " + car.make +
" " + car.model);
```

Explanation: This example demonstrates the use of an object. We create an object car with three properties make, model, and year. We then use dot notation

Objects:

```
let person = {
  name: "John Doe",
  age: 30,
  isStudent: false
};
console.log(person.name);
console.log(person.age);
console.log(person.isStudent);
```

Explanation: In this code, we are declaring an object named person with properties name, age, and isStudent. The properties of the object store values "John Doe", 30, and false respectively. The console.log statements log the values of the properties to the console.

Conditional Statements:

```
let grade = 75;
if (grade >= 60) {
  console.log("Passed");
} else {
  console.log("Failed");
}
```

Explanation: In this code, we are declaring a variable grade with a value of 75. We then use an if...else statement to determine whether the student has passed or failed based on their grade. If the value of grade is greater than or equal to 60, the code inside the first block (i.e., console.log("Passed")) will be executed, and the message "Passed" will be logged to the console. If the value of grade is less than 60, the code inside the second block (i.e., console.log("Failed")) will be executed, and the message "Failed" will be logged to the console.

Functions:

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

Explanation: In this code, we are defining a function named greet that takes a parameter name. The function logs a greeting message to the console by concatenating the string "Hello, " with the value of the name parameter. The function is then called by passing a string argument "John" to it, which results in the message "Hello, John" being logged to the console.

Example: Simple Function

```
function greet(name) {
  console.log("Hello " + name);
}

greet("John");
```

Explanation: In this example, we have defined a simple function greet that takes a single argument name and logs a greeting to the console using console.log(). To call the function, we pass in an argument, such as "John", to the function and invoke it. This will log the message "Hello John" to the console.

Example: Conditional Statement

```
let num = 5;
if (num > 0) {
  console.log(num + " is a positive number");
} else {
  console.log(num + " is a negative number");
}
```

Explanation: This example demonstrates the use of a conditional statement using an if statement. The code checks if the value of num is greater than 0. If it is, the code inside the first set of curly braces is executed, and the message "5 is a positive number" is logged to the console. If the value of num is not greater than 0, the code inside the else block is executed, and the message "5 is a negative number" is logged to the console.

String Methods:

```
let message = "Hello World";
console.log(message.toUpperCase());
console.log(message.includes("Hello"));
```

Explanation: In this code, we are declaring a variable message with a string value of "Hello World". The first console.log statement logs the result of calling the toUpperCase method on the message string, which returns an uppercase version of the string: "HELLO WORLD". The second console.log statement logs the result of calling the includes method on the message string, which checks if the string "Hello" is a part of the message string. Since it is, the method returns true, which is then logged to the console.

Ternary Operators:

```
let grade = 75;
let result = (grade >= 60) ? "Passed" : "Failed";
console.log(result);
```

Explanation: In this code, we are declaring a variable grade with a value of 75 and using a ternary operator to assign a value to the result variable based on the value of grade. The expression (grade >= 60) ? "Passed" : "Failed" says that if grade is greater than or equal to 60, the value of result should be "Passed", otherwise it should be "Failed". The final console.log statement logs the value of result to the console.

Using if statements:

```
let age = 25;
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are not an adult.");
}
```

Explanation: In this code, we are declaring a variable age with the value 25. We are then using an if statement to determine if the value of age is greater than or equal to 18. If it is, the first console.log statement is executed, logging "You are an adult." to the console. If not, the else block is executed, logging "You are not an adult." to the console.

Using the switch statement:

```
let day = 2;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

Explanation: In this code, we are using a switch statement to control the flow of the program based on the value of a variable. The switch statement takes an expression as an argument and compares it to the case labels. If a match is found, the code inside the corresponding case block is executed. If no match is found, the code inside the default block is executed. In this case, the value of day is 2, so the code inside the case 2 block is executed and the result "Tuesday" is logged to the console.

Example of switch statements:

```
let grade = "A";

switch (grade) {

  case "A":

    console.log("Excellent");

    break;

  case "B":

    console.log("Good");

    break;

  case "C":

    console.log("Average");

    break;

  default:

    console.log("Invalid grade");

    break;

}
```

Explanation: In this code, we are declaring a variable grade with the value "A". We are then using a switch statement to match the value of grade with different cases. If the value of grade is "A", the first console.log statement is executed, logging "Excellent" to the console. If the value of grade is "B", the second console.log statement is executed, logging "Good" to the console. If the value of grade is "C", the third console.log statement is executed, logging "Average" to the console. If the value of grade does not match any of the cases, the default block is executed, logging "Invalid grade" to the console.

Using an object to store data:

```
let person = {
  name: "John",
  age: 30,
  occupation: "Teacher"
};
console.log(person.name);
```

Explanation: In this code, we are using an object to store data about a person. An object is a collection of key-value pairs, and in this case, the keys are name, age, and occupation. The values are the corresponding data for each key. To access the data stored in an object, we use dot notation, such as person.name to access the value of the name key. The value of person.name is logged to the console using the console.log() method.

Using a function to define a reusable piece of code:

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

Explanation: In this code, we are using a function to define a reusable piece of code. A function is a block of code that can be executed repeatedly with different arguments. In this case, the function sayHello takes an argument name and logs a greeting to the console using the console.log() method. To call the function, we use its name followed by a set of parentheses, such as sayHello("John"). This causes the function to run and the greeting "Hello, John!" is logged to the console.

Anonymous Functions:

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

Explanation: In this code, we are declaring an anonymous function and assigning it to the variable greet. The function takes a parameter name and logs a greeting message to the console by concatenating the string "Hello, " with the value of the name parameter. The function is then called by passing a string argument "John" to it, which results in the message "Hello, John" being logged to the console.

Arrow Functions:

```
let add = (a, b) => a + b;
console.log(add(2, 3));
```

Explanation: In this code, we are declaring an arrow function named add that takes two parameters, a and b, and returns their sum. The final console.log statement calls the add function, passing the values 2 and 3 as arguments, and logs the result to the console, which is 5.

Example: For Loop

```
for (let i = 0; i < 5; i++) {
  console.log("The value of i is: " + i);
}
```

Explanation: This example demonstrates the use of a for loop. The loop initializes the variable i to 0 and runs the loop as long as i is less than 5. The loop increments the value of i by 1 with each iteration. This will log the message "The value of i is: 0", "The value of i is: 1", "The value of i is: 2", "The value of i is: 3", and "The value of i is: 4" to the console.

While Loops:

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

Explanation: In this code, we are using a while loop to log the values 0 to 4 to the console. The loop initializes a variable i with a value of 0. The condition in the loop checks if i is less than 5. If the condition is true, the code inside the loop will be executed, and the value of i will be logged to the console.

Using a for loop to iterate through an array:

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

Explanation: In this code, we are using a for loop to iterate through an array of numbers. The loop uses the variable i as a counter, and it continues to run as long as i is less than the length of the numbers array. On each iteration of the loop, the current value of numbers[i] is logged to the console using the console.log() method.

Using a while loop to calculate the factorial of a number:

```
let number = 5;
let factorial = 1;
while (number > 1) {
  factorial *= number;
  number--;
}
console.log(factorial);
```

Explanation: In this code, we are using a while loop to calculate the factorial of a number. The while loop continues to run as long as the value of number is greater than 1. On each iteration of the loop, the value of factorial is multiplied by number, and then number is decremented by 1. The final value of factorial is the factorial of the original number, and it is logged to the console using the console.log() method.

Loops:

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

Explanation: In this code, we are using a for loop to log the values 0 to 4 to the console. The loop initializes a variable i with a value of 0. The condition in the loop checks if i is less than 5. If the condition is true, the code inside the loop will be executed, and the value of i will be logged to the console. The final statement in the loop increments the value of i by 1. This process continues until i is no longer less than 5, at which point the loop terminates. The result is that the values 0 to 4 are logged to the console.

Using the for loop:

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

Explanation: In this code, we are using a for loop to iterate over a range of values. The loop starts by declaring a variable i with the value 0. The loop will then continue to execute as long as i is less than 10. At the end of each iteration, the value of i is incremented by 1. This means that on each iteration, the value of i is logged to the console using the console.log() method. The result will be the numbers 0 through 9 logged to the console.

Using the while loop:

```
let i = 0;
while (i < 10) {
  console.log(i);
  i++;
}
```

Explanation: In this code, we are using a while loop to iterate over a range of values. The loop starts by declaring a variable i with the value 0. The loop will then continue to execute as long as i is less than 10. At the end of each iteration, the value of i is incremented by 1. This means that on each iteration, the value of i is logged to the console using the console.log() method. The result will be the numbers 0 through 9 logged to the console.

For-of Loop:

```
let numbers = [1, 2, 3, 4, 5];
for (let number of numbers) {
  console.log(number);
}
```

Explanation: In this code, we are declaring an array named numbers with 5 elements. We are then using a for-of loop to log each element of the numbers array to the console. The for-of loop creates a variable named number that takes on the value of each element in the numbers array one at a time, and the code inside the loop logs the value of number to the console.

Array Methods:

```
let numbers = [1, 2, 3, 4, 5];
console.log(numbers.length);
console.log(numbers.slice(1, 3));
```

Explanation: In this code, we are declaring an array named numbers with 5 elements. The first console.log statement logs the length of the array, which is 5. The second console.log statement logs the result of calling the slice method on the numbers array, which returns a new array containing elements from the original array at indices 1 to 2 (the second and third elements).

Object Destructuring:

```
let person = {
  name: "John Doe",
  age: 30,
  isStudent: false
};
let { name, age } = person;
console.log(name);
console.log(age);
```

Explanation: In this code, we are declaring an object named person with properties name, age, and isStudent. Then, using object destructuring, we are extracting the name and age properties from the person object and assigning them to separate variables with the same names. The final console.log statements log the values of the name and age variables to the console.

Spread Operator:

```
let a = [1, 2, 3];
let b = [4, 5, 6];
let numbers = [...a, ...b];
console.log(numbers);
```

Explanation: In this code, we are declaring two arrays a and b with 3 elements each. We are then using the spread operator to concatenate the two arrays into a new array named numbers. The final console.log statement logs the numbers array to the console, which contains elements from both a and b.

Using the Date object:

```
let now = new Date();
console.log(now);
console.log(now.toLocaleString());
```

Explanation: In this code, we are creating a Date object with the current date and time using the new Date() constructor. The first console.log statement logs the Date object as a string. The second console.log statement logs the same Date object, but formatted as a human-readable string using the toLocaleString() method.

Example of Date object:

```
let date = new Date();
console.log(date);
```

Explanation: In this code, we are using the Date object to get the current date and time. The new Date() constructor creates a new Date object, which represents the current date and time. The result is logged to the console using the console.log() method.

Using the Math object:

```
let number = Math.round(2.5);
console.log(number);
```

Explanation: In this code, we are using the Math object to perform mathematical operations. The Math.round() method takes a number as an argument and returns the rounded value. In this case, the number 2.5 is rounded to 3. The result is logged to the console using the console.log() method.

Example of Math object:

```
console.log(Math.PI);
console.log(Math.ceil(3.14));
console.log(Math.floor(3.14));
```

Explanation: In this code, we are using the Math object to perform mathematical operations. The first console.log statement logs the value of π (Pi) from the Math object. The second console.log statement logs the smallest integer that is greater than or equal to 3.14 using the ceil() method. The third console.log statement logs the largest integer that is less than or equal to 3.14 using the floor() method.

Using try and catch statements:

```
try {
  let x = y;
  console.log(x);
} catch (error) {
  console.error(error);
}
```

Explanation: In this code, we are using a try and catch statement to handle errors. Within the try block, we are attempting to assign the value of y to a variable x. However, y has not been defined, so this operation will result in a ReferenceError. This error will be caught by the catch block, which logs the error to the console using the console.error() method. This is useful for handling unexpected errors in your code and allowing your program to continue running, rather than crashing.

Try-catch statement Example:

```
try {
  let num = Number("hello");
} catch (error) {
  console.log(error.message);
}
```

Explanation: In this code, we are using the try-catch statement to handle errors. The try block contains code that might throw an error, while the catch block contains code that is executed when an error is thrown. In this case, the Number("hello") expression tries to convert the string "hello" to a number, but since this is not possible, an error is thrown. The error is caught by the catch block and its message property is logged to the console using the console.log() method.

Using the map() method:

```
let numbers = [1, 2, 3, 4, 5];
let doubled = numbers.map(function(num) {
  return num * 2;
});
console.log(doubled);
```

Explanation: In this code, we are using the map() method to transform an array of numbers. The map() method takes a callback function as an argument and applies that function to each element in the array. In this case, the callback function doubles each element in the numbers array. The result is a new array doubled that contains the doubled values. The final console.log statement logs the doubled array to the console.

Using the filter() method:

```
let numbers = [1, 2, 3, 4, 5];
let evens = numbers.filter(function(num) {
  return num % 2 === 0;
});
console.log(evens);
```

Explanation: In this code, we are using the filter() method to select certain elements from an array based on a condition. The filter() method takes a callback function as an argument and applies that function to each element in the array. In this case, the callback function checks if each element in the numbers array is even. The result is a new array evens that contains only the even numbers from the numbers array. The final console.log statement logs the evens array to the console.

Using the reduce() method:

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce(function(accumulator,
currentValue) {
  return
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce(function(accumulator,
currentValue) {
  return accumulator + currentValue;
});
console.log(sum);
```

Explanation: In this code, we are using the reduce() method to reduce an array of numbers to a single value. The reduce()

method takes a callback function as an argument and applies that function to each element in the array. The first argument to the callback function is an accumulator, which is initialized to the first value in the array. The second argument is the current value being processed. In this case, the callback function adds the current value to the accumulator. The result is a single value sum that is the sum of all the numbers in the numbers array. The final console.log statement logs the sum to the console.

Using the Array.includes() method:

```
let numbers = [1, 2, 3, 4, 5];
let result = numbers.includes(3);
console.log(result);
```

Explanation: In this code, we are using the Array.includes() method to check if an array contains a certain value. The Array.includes() method takes a value as an argument and returns true if the array contains that value and false otherwise. In this case, the value 3 is included in the numbers array, so the result is `true

Using a class to define a blueprint for creating objects:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name} and I
am ${this.age} years old.`);
  }
}
let john = new Person("John", 30);
john.sayHello();
```

Explanation: In this code, we are using a class to define a blueprint for creating objects. A class is a blueprint for creating objects with similar properties and methods. In this case, the Person class has a constructor that takes two arguments, name and age, and sets them as properties of the object being created. The class also has a method sayHello that logs a message to the console using the console.log() method. To create an object using a class, we use the new keyword, such as let john = new Person("John", 30). This creates a new Person object with the properties name set to "John" and age set to 30. To call the sayHello method on the object, we use dot notation, such as john.sayHello(). This causes the message "Hello, my name is John and I am 30 years old." to be logged to the console.

Example: Higher-Order Function

```
function multiplyBy(factor) {
  return function (number) {
    return number * factor;
  };
}
let double = multiplyBy(2);
console.log(double(5)); // 10
```

Explanation: This example demonstrates the use of a higher-order function. A higher-order function is a function that returns another function. In this example, the multiplyBy function takes a factor as its argument and returns a new function that takes a number as its argument. The returned function calculates and returns the product of the number and factor. We then assign the returned function to the variable double and call it with an argument 5. This will log the result 10 to the console.

Example: Closure

```
function outerFunction() {
  let counter = 0;

  return function innerFunction() {
    counter++;
    console.log(counter);
  };
}
let counterFunction = outerFunction();
counterFunction(); // 1
counterFunction(); // 2
counterFunction(); // 3
```

Explanation: This example demonstrates the use of a closure. A closure is a function that has access to variables in its outer scope even after the outer function has returned. In this example, the outerFunction returns the innerFunction which logs the value of counter to the console. The variable counter is declared in the outer scope and is accessible to the inner function. We assign the returned function to the variable counterFunction and call it multiple times. This will log the values

1, 2, and 3 to the console, indicating that the inner function still has access to the counter variable even after the outer function has returned.

Example: Destructuring

```
let person = {
  name: "John Doe",
  age: 30,
  address: {
    street: "123 Main St",
    city: "San Francisco",
    state: "CA",
  },
};
let { name, age, address: { city } } = person;
console.log(name); // "John Doe"
console.log(age); // 30
console.log(city); // "San Francisco"
```

Explanation: This example demonstrates the use of destructuring in JavaScript. Destructuring allows you to extract values from objects and arrays and assign them to separate variables. In this example, we have an object person with three properties: name, age, and address. We use destructuring to extract the values of name, age, and city properties and assign them to separate variables. This will log the values "John Doe", 30, and "San Francisco" to the console.

Example: Promises

```
let promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve("Success!");
  }, 1000);
});
promise
  .then(function (result) {
    console.log(result); // "Success!"
  })
  .catch(function (error) {
    console.error(error);
  });
```

Explanation: This example demonstrates the use of Promises in JavaScript. A Promise is a returned object representing the eventual completion or failure of an asynchronous operation. In this example, we create a new Promise using the Promise constructor. The constructor takes a callback function with resolve and reject parameters. We use the setTimeout function to wait for 1 second and then call resolve with a string value of "Success!". The returned Promise object has a then method that takes a success callback function as its argument. This function will be called if the Promise is resolved. The Promise object also has a catch method that takes an error callback function as its argument. This function will be called if the Promise is rejected. In this example, the Promise is resolved, so the then method logs the value "Success!" to the console.

Example: Generators

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}
let iterator = generator();
console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
```

Explanation: This example demonstrates the use of generators in JavaScript. A generator is a special type of function that can be paused and resumed multiple times. In this example, we create a generator function using the function* syntax. The generator function uses the yield keyword to return a value each time it is resumed. We create an iterator using the generator function and use the next method to get the next value from the iterator. This will log the values 1, 2, and 3 to the console, indicating that the generator has been resumed and returned the values one by one.

Example: Asynchronous Iteration

```
async function asyncIteration() {
  let array = [1, 2, 3];
  for await (const value of array) {
    console.log(value);
  }
}
asyncIteration();
// Output:
// 1
// 2
// 3
```

Explanation: This example demonstrates the use of asynchronous iteration in JavaScript. Asynchronous iteration allows us to iterate over asynchronous data sources, such as an async function or a Promise, in a synchronous-like manner. In this example, we create an async function that contains a for-await-of loop. The for-await-of loop is used to iterate over the array of values. The loop logs each value to the console. The use of the async keyword ensures that the loop will wait for each iteration to complete before moving on to the next one.

Example: Map and Set

```
let map = new Map();
map.set("key1", "value1");
map.set("key2", "value2");
console.log(map.get("key1")); // "value1"
console.log(map.size); // 2
let set = new Set();
set.add("value1");
set.add("value2");
console.log(set.has("value1")); // true
console.log(set.size); // 2
```

Explanation: This example demonstrates the use of the Map and Set data structures in JavaScript. A Map is an ordered collection of key-value pairs, while a Set is an unordered collection of unique values. In this example, we create a Map and use the set method to add two key-value pairs. We use the get method to retrieve the value associated with a specific key and the size property to get the number of elements in the Map. We also create a Set and use the add method to add two values. We use the has method to check if a value exists in the Set and the size property to get the number of elements in the Set.

Example: WeakMap and WeakSet

```
let weakMap = new WeakMap();
let key = {};
weakMap.set(key, "value");
console.log(weakMap.has(key)); // true
key = null;
console.log(weakMap.has(key)); // false
let weakSet = new WeakSet();
let value = {};
weakSet.add(value);
console.log(weakSet.has(value)); // true
value = null;
```

```
console.log(weakSet.has(value)); // false
```

Explanation: This example demonstrates the use of the WeakMap and WeakSet data structures in JavaScript. A WeakMap is a collection of key-value pairs where the keys are objects and the values can be any value. A WeakSet is a collection of objects. Unlike Map and Set, the entries in a WeakMap or WeakSet do not prevent the objects from being garbage collected. In this example, we create a WeakMap and use the set method to add a key-value pair. We use the has method to check if the key exists in the WeakMap. We then set the key to null, which makes it eligible for garbage collection. We check the has method again and it returns false, as the key has been garbage collected. We also create a WeakSet and use the add method to add a value. We use the has method to check if the value exists in the WeakSet. We then set the value to null, which makes it eligible for garbage collection. We check the has method again and it returns false, as the value has been garbage collected.

Example: Object Destructuring

```
let obj = { x: 1, y: 2, z: 3 };
let { x, y, z } = obj;
console.log(x, y, z); // 1 2 3
```

Explanation: This example demonstrates the use of object destructuring in JavaScript. Object destructuring allows us to extract values from an object and assign them to variables with the same name as the object properties. In this example, we create an object with properties x, y, and z. We then use destructuring to extract the values of these properties and assign them to variables with the same name. The destructured variables x, y, and z now hold the values of the respective properties.

Example: Class and Inheritance

```
class Shape {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  getArea() {
    return this.width * this.height;
  }
}
class Rectangle extends Shape {
  constructor(width, height) {
    super(width, height);
  }
}
let rectangle = new Rectangle(10, 20);
console.log(rectangle.getArea()); // 200
```

Explanation: This example demonstrates the use of classes and inheritance in JavaScript. A class is a blueprint for creating objects with similar properties and methods. In this example, we create a Shape class with a constructor method that takes width and height as arguments and sets them as properties of the object. We also add a getArea method that returns the product of width and height. We then create a Rectangle class that extends the Shape class. The Rectangle class has its own constructor method that calls the super method to pass the width and height arguments to the Shape class. We create an instance of the Rectangle class and use the getArea method to get the area of the rectangle.

JavaScript Classes

JavaScript added class syntax with ECMAScript 6 (ES6) as a way to write reusable code and create objects. A class is a blueprint for creating objects that have similar properties and methods.

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
am ${this.age} years old.`);
  }
}
```

In the above example, the Person class has two properties: name and age, and a method greet that prints a greeting message.

To create an object from this class, you use the new operator and call the constructor function:

```
const person = new Person("John", 30);
person.greet();
// Output: Hello, my name is John and I am 30 years
old.
```

You can also extend a class to create a subclass and inherit properties and methods from the parent class:

```
class Student extends Person {
  constructor(name, age, major) {
    super(name, age);
    this.major = major;
  }

  study() {
    console.log(`I am studying ${this.major}.`);
  }
}

const student = new Student("Jane", 20, "Computer
Science");
student.greet();
// Output: Hello, my name is Jane and I am 20 years
old.
student.study();
// Output: I am studying Computer Science.
```

In the above example, the Student class extends the Person class and adds a new property major and a method study. The super keyword is used to call the constructor of the parent class and pass along the required properties.

Classes in JavaScript provide a way to write organized and reusable code, making it easier to maintain and extend your codebase.

Regular Expression (RegExp)

A Regular Expression (RegExp) is a pattern that specifies a set of strings. JavaScript provides built-in support for regular expressions with the RegExp object.

Here's an example of how to use regular expressions in JavaScript:

```
let string = "Hello, World!";
let pattern = /Hello/;
let result = pattern.test(string);
console.log(result); // Output: true
```

In this example, we define a string "Hello, World!" and a regular expression pattern /Hello/. We use the test() method of the RegExp object to test if the string matches the pattern. The result is a boolean value indicating whether the string matches the pattern.

You can also use the match() method of the String object to find all matches of a regular expression pattern in a string:

```
let string = "Hello, World! Hello, JavaScript!";
let pattern = /Hello/g;
let result = string.match(pattern);
console.log(result); // Output: [ "Hello", "Hello" ]
```

In this example, we add the g (global) flag to the pattern to find all matches in the string, instead of just the first match. The result is an array of strings containing all matches of the pattern in the string.

Regular expressions can be a powerful tool for matching and manipulating strings in JavaScript.

Best Practices JavaScript Code

Use const instead of var or let whenever possible. This helps prevent accidental modification of variables.

```
Example:
const name = "John";
name = "Jane"; // Error: "name" is read-only
```

Declare variables as close as possible to their first use. This reduces their scope and makes the code easier to understand.

```
Example:
```

```
function example() {
  let name;
  // ...
  name = "John";
  // ...
}
```

Use let instead of var for block-scoped variables. var is function-scoped, which can lead to unexpected behavior in certain cases.

```
Example:
if (true) {
  let name = "John";
}
console.log(name); // Error: "name" is not defined
```

Use arrow functions instead of traditional functions whenever possible. They provide a concise and easier-to-read syntax for anonymous functions.

```
Example:
const greet = name => console.log(`Hello, ${name}!`);
greet("John"); // Output: Hello, John!
```

Use forEach instead of for loops whenever possible. It's concise, readable, and eliminates the need for manual index increments.

```
Example:
const names = ["John", "Jane", "Jim"];
names.forEach(name => console.log(name));
// Output:
// John
// Jane
// Jim
```

Avoid using global variables whenever possible. They can easily lead to naming conflicts and hard-to-debug bugs.

```
Example:
let name = "John";
// ...
function example() {
  name = "Jane"; // Modifying global variable
}
```

Use Object.freeze to prevent accidental modification of objects.

```
Example:
const person = { name: "John", age: 30 };
Object.freeze(person);
person.name = "Jane"; // Error: "person" is read-only
```

Use try-catch blocks to handle errors. It makes it easier to debug code and provide meaningful error messages to users.

```
Example:
try {
  // Code that might throw an error
} catch (error) {
  console.error(error);
}
```

Use template literals instead of concatenation for string interpolation. They provide a more readable and easier-to-maintain syntax.

```
Example:
const name = "John";
console.log(`Hello, ${name}!`); // Output: Hello, John!
```

Use modern JavaScript features and syntax whenever possible, such as destructuring, spread operators, and async/await. They make code more concise, readable, and easier to maintain.

```
Example:
const [firstName, lastName] = ["John", "Doe"];
console.log(firstName, lastName); // Output: John Doe
```

By following these best practices, you can write efficient, maintainable, and scalable JavaScript code.

closure in JavaScript

What is closure in JavaScript and how can it be used to create private variables?
A closure is a function that has access to variables in its outer scope, even after the outer function has returned. Closures can be used to create private variables in JavaScript by returning an inner function that has access to the variables declared in the outer function. Here's an example:

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

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

const counter = createCounter();
console.log(counter());  // 1
console.log(counter());  // 2
console.log(counter());  // 3
```

In this example, the createCounter function declares a private variable count and returns an inner function that increments count every time it's called. The returned inner function has access to the count variable even after the createCounter function has returned, allowing it to maintain its value between invocations.

hoisting in JavaScript

What is hoisting in JavaScript and how does it work?
Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their scope. This means that variables can be used before they are declared, and functions can be called before they are defined. Here's an example:

```
console.log(x);  // undefined
var x = 10;
```

In this example, even though the x variable is declared after it is used, the JavaScript engine hoists the declaration to the top of the scope, so the code runs as if it were written like this:

```
var x;
console.log(x);  // undefined
```

```
x = 10;
Hoisting also applies to function declarations:

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

In this example, the function declaration for myFunction is hoisted to the top of the scope, so it can be called before it is defined.

difference between null and undefined in JavaScript

What is the difference between null and undefined in JavaScript?
In JavaScript, null and undefined are both values that represent the absence of a value. However, there is a subtle difference between the two.

undefined is a value that is automatically assigned to a variable when it is declared but not assigned a value:

```
let x;
```

```
console.log(x);  // undefined
null, on the other hand, is a value that must be
explicitly assigned to a variable:

let y = null;
console.log(y);  // null
```

In other words, undefined is a default value assigned to a variable when no value is explicitly assigned, while null is a value that can be explicitly assigned to represent the absence of a value.

difference between a for loop and forEach in JavaScript

What is the difference between a for loop and forEach in JavaScript?
In JavaScript, both for loops and the forEach method are used to iterate over arrays. However, there are some key differences between the two.

for loops allow you to access the current index and value of each element in the array, and you can also use break and continue statements to control the flow of the loop:

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

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

In this example, the for loop iterates over the numbers array, and the console.log statement logs the value of each element in the array.

The forEach method, on the other hand, is a higher-order function that is called on an array and takes a callback function as an argument. The callback function is invoked for each element in the array:

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number);
});
```

In this example, the forEach method is called on the numbers array, and the anonymous callback function logs the value of each element in the array.

The main difference between the two is that for loops offer more control over the flow of the loop, while forEach is a simpler and more concise way to iterate over an array.

difference between == and === in JavaScript

What is the difference between == and === in JavaScript?
In JavaScript, there are two comparison operators: == (loose equality) and === (strict equality).

The == operator performs type coercion before checking for equality, which means that it converts the operands to the same type before checking for equality:

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

In these examples, the == operator converts the string "1" to the number 1, and the boolean value true to the number 1, before checking for equality, so both expressions return true.

The === operator, on the other hand, does not perform type coercion and checks for equality without converting the operands to the same type:

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

In these examples, the === operator does not convert the string "1" or the boolean value true to the number 1, so both expressions return false.

It is generally recommended to use the === operator in JavaScript, as it ensures that equality is checked without any type coercion, leading to more predictable results.

Coding Function that returns a sum of the elements

Write a function that takes an array of numbers and returns the sum of its elements.

```
function sumArray(numbers) {
  let sum = 0;
  for (let i = 0; i < numbers.length; i++) {
    sum += numbers[i];
  }
  return sum;
}
```

```
const numbers = [1, 2, 3, 4, 5];
console.log(sumArray(numbers));  // 15
```

In this example, the sumArray function takes an array of numbers as an argument and uses a for loop to iterate over the array, adding each element to the sum variable. The function returns the sum of the elements in the array.

Function that takes an array of strings returns string lengths

Write a function that takes an array of strings and returns an array of the lengths of each string.

```
function stringLengths(strings) {
  let lengths = [];
  strings.forEach(function(string) {
    lengths.push(string.length);
  });
  return lengths;
}

const strings = ['hello', 'world', 'foo', 'bar'];
```

```
console.log(stringLengths(strings));  // [5, 5, 3, 3]
```

In this example, the stringLengths function takes an array of strings as an argument and uses the forEach method to iterate over the array. For each string, the length is determined using the length property and pushed onto the lengths array. The function returns the array of lengths.

strict mode example

Use strict mode to enforce modern JavaScript syntax and catch errors early:

```
'use strict';
```

Use const and let

Always declare variables with const or let, rather than var:

```
// Use let
let name = 'John Doe';

// Use const
const PI = 3.14;
```

Use Arrows functions

Use arrow functions instead of function for cleaner and concise code:

```
// Function expression
const multiply = (a, b) => a * b;

// Implicit return
const square = x => x * x;
```

Use Destructuring to get values from arrays

Make use of destructuring to extract values from arrays and objects into variables:

```
// Destructuring arrays
const colors = ['red', 'green', 'blue'];
const [first, second, third] = colors;

// Destructuring objects
const person = {
  name: 'John Doe',
  age: 30,
  job: 'Software Engineer'
};
const { name, age, job } = person;
```

Use template literals

Use template literals for string concatenation and embedding expressions:

```
const name = 'John Doe';
const message = `Hello, ${name}!`;
```

Use forEach over for loop

Prefer forEach over for loop for simple iterations:

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(number => console.log(number));
```

use of higher-order functions

Make use of higher-order functions like map, filter, and reduce to process arrays:

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

```
// Use map
const double = numbers.map(number => number * 2);

// Use filter
const even = numbers.filter(number => number % 2 ===
0);

// Use reduce
const sum = numbers.reduce((acc, number) => acc +
number, 0);
```

Avoid Global Variables

Avoid using global variables and always use const or let to scope variables:

```
// Global variable (not recommended)
let name = 'John Doe';

// Scoped variable (recommended)
function sayHello() {
  const name = 'John Doe';
  console.log(`Hello, ${name}!`);
}
```

Avoid Naming Collisions

Use modules to organize your code and avoid naming collisions:

```
// math.js
export const PI = 3.14;
export const add = (a, b) => a + b;

// main.js
import { PI, add } from './math.js';
console.log(PI); // 3.14
console.log(add(1, 2)); // 3
```

Initialize variables with default values

Always initialize variables with default values to avoid undefined values:

```
// Default value
let name = 'John Doe';

// Default value with destructuring
```

```
const person = {
  name = 'John Doe',
  age: 30
};
const { name, age = 0 } = person;
```

Use spread operator

Use spread operator to combine arrays:

```
const a = [1, 2, 3];
const b = [4, 5, 6];
const c = [...a, ...b]; // [1, 2, 3, 4, 5, 6]
```

Use spread operator to combine arrays: The spread operator (...) can be used to combine arrays into a new array. The operator "spreads" the elements of the original arrays into a new array. For example, in the following code, two arrays a and b are combined into a new array c:

Use of default parameters

Make use of default parameters to handle missing arguments:

```
const greet = (name = 'stranger') =>
console.log(`Hello, ${name}!`);
greet(); // Hello, stranger!
greet('John'); // Hello, John!
```

Make use of default parameters to handle missing arguments: Default parameters allow you to provide a default value for a function argument in case the argument is not passed when the function is called. This can be useful for handling missing arguments and preventing errors.

Use rest operator

Use rest operator to pass multiple arguments as an array:

```
const add = (...numbers) => numbers.reduce((a, b) => a
+ b, 0);
console.log(add(1, 2, 3, 4, 5)); // 15
```

Use rest operator to pass multiple arguments as an array: The rest operator (...) can be used to gather all remaining arguments into an array. This is useful for passing a variable number of arguments to a function.

Use object literals

Use object literals to create objects:

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

Use object literals to create objects: Object literals are a concise and convenient way to create objects in JavaScript. The syntax is similar to an array literal, but with curly braces ({}) instead of square brackets ([]). You can also use property value shorthand to create properties using the same name as the variable.

Use destructuring with rest operator

Use destructuring with rest operator to extract remaining values:

```
const colors = ['red', 'green', 'blue', 'yellow',
'orange'];
const [first, second, ...rest] = colors;
console.log(first); // red
console.log(second); // green
console.log(rest); // [blue, yellow, orange]
```

Use destructuring with rest operator to extract remaining values: Destructuring is a powerful feature in JavaScript that allows you to extract values from arrays and objects and assign them to variables. You can use the rest operator (...) in combination with destructuring to extract remaining values from an array.

Use of async/await

Make use of async/await for asynchronous programming:

```
const fetchData = async () => {
  try {
    const response = await
fetch('https://jsonplaceholder.typicode.com/posts');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
};
fetchData();
```

Make use of async/await for asynchronous programming: The async/await syntax provides a convenient way to write asynchronous code that is easier to read and debug than traditional callback-based code. An async function returns a Promise and can be awaited to pause execution until the Promise is resolved.

Use destructuring with default values

Use destructuring with default values to handle missing properties:

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

Use destructuring with default values: Destructuring can also be used to provide default values for variables in case the values are undefined. For example:

```
const getUser = ({ name = 'stranger', age = 'unknown' }
= {}) =>
  console.log(`Name: ${name}, Age: ${age}`);
getUser({ name: 'John Doe', age: 30 }); // Name: John
Doe, Age: 30
```

```
getUser(); // Name: stranger, Age: unknown
```

Use named exports

Use named exports to export multiple values from a module:

```
// utils.js
export const PI = 3.14;
export const add = (a, b) => a + b;

// main.js
import { PI, add } from './utils.js';
console.log(PI); // 3.14
console.log(add(1, 2)); // 3
```

Use object spread operator

Use object spread operator to merge objects:

```
const a = { name: 'John Doe', age: 30 };
const b = { job: 'Software Engineer' };
const c = { ...a, ...b };
console.log(c); // { name: 'John Doe', age: 30, job:
'Software Engineer' }
```

Use try/catch

Make use of try/catch blocks to handle errors: Try/catch blocks provide a convenient way to handle errors in JavaScript. A try block is used to enclose the code that might throw an error, and a catch block is used to catch the error and handle it. For example:

```
const divide = (a, b) => {
  try {
    if (b === 0) {
      throw new Error('Cannot divide by zero');
    }
    return a / b;
  } catch (error) {
    console.error(error.message);
  }
};
console.log(divide(10, 5)); // 2
console.log(divide(10, 0)); // Cannot divide by zero
```

Use ternary operator

Use ternary operator for simple conditional statements: The ternary operator (?) provides a shorthand way to write simple if/else statements. The operator takes three operands: the condition to be tested, the expression to be returned if the condition is true, and the expression to be returned if the condition is false.

```
const isPositive = (number) => (number >= 0 ?
'positive' : 'negative');
console.log(isPositive(10)); // positive
console.log(isPositive(-10)); // negative
```

Use named export/import

Use named export/import to manage modular code: Named exports and imports allow you to organize and reuse code by breaking it up into separate modules. You can use named exports to export multiple values from a module, and named imports to import specific values into another module.

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

```
// index.js
import { add, subtract } from './utils';
console.log(add(10, 5)); // 15
console.log(subtract(10, 5)); // 5
```

Closure

What is closure in JavaScript and give an example of its usage?

A closure is a function that has access to variables in its outer scope, even after the outer function has returned.

```
Example:
// outer function
function outerFunction(x) {

  // inner function
  return function innerFunction(y) {
    return x + y;
  }
}

const add5 = outerFunction(5);
```

```
console.log(add5(3)); // 8
```

Explanation: In the above example, the innerFunction has access to the x variable of the outerFunction, even after the outerFunction has returned. By assigning the return value of outerFunction(5) to the add5 constant, we are able to create a closure that adds 5 to its input.

forEach Array

How would you implement the forEach function for an array?

```
Array.prototype.myForEach = function(callback) {
  for (let i = 0; i < this.length; i++) {
    callback(this[i], i, this);
  }
};

const arr = [1, 2, 3];
arr.myForEach(function(element, index, array) {
  console.log(element, index, array);
});
```

Explanation: The myForEach function is a custom implementation of the built-in forEach method for arrays. It takes a callback function as a parameter, which it invokes for each element in the array. The callback function takes three parameters: the current element, the index of the current element, and the entire array.

JavaScript Map function

How would you implement the map function for an array?

```
Array.prototype.myMap = function(callback) {
  const result = [];
  for (let i = 0; i < this.length; i++) {
    result.push(callback(this[i], i, this));
  }
  return result;
};

const arr = [1, 2, 3];
const doubled = arr.myMap(function(element, index,
array) {
  return element * 2;
});
console.log(doubled); // [2, 4, 6]
```

Explanation: The myMap function is a custom implementation of the built-in map method for arrays. It takes a callback function as a parameter, which it invokes for each element in the array. The callback function takes three parameters: the current element, the index of the current element, and the entire array. The myMap function returns a new array containing the results of the callback function applied to each element in the original array.

JavaScript Filter

How would you implement the filter function for an array?

```
Array.prototype.myFilter = function(callback) {
  const result = [];
  for (let i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  return result;
};

const arr = [1, 2, 3, 4, 5];
const evens = arr.myFilter(function(element, index,
array) {
  return element % 2 === 0;
});
console.log(evens); // [2, 4]
```

Explanation: The `myFilter` function is a custom implementation of the built-in `filter` method for arrays. It takes a `callback` function as a parameter, which it invokes for each element in the array. The `callback` function takes three parameters: the current element, the index of the current element, and the entire array. If the `callback` function returns `true` for a given element, that element is included in the resulting array.

Check if number is an integer

How would you check if a number is an integer?

```
function isInteger(num) {
return typeof num === 'number' &&
Number.isInteger(num);
}
```

```
console.log(isInteger(5)); // true
console.log(isInteger(5.1)); // false
```

Explanation: The `isInteger` function takes a number as an argument and returns a boolean indicating whether or not it is an integer. It does so by checking the type of the input and using the built-in `Number.isInteger` method to determine if it is an integer.

Is the string a palindrome

How would you check if a given string is a palindrome?

```
function isPalindrome(str) {
return str === str.split('').reverse().join('');
}

console.log(isPalindrome('racecar')); // true
console.log(isPalindrome('hello')); // false
```

Explanation: The `isPalindrome` function takes a string as an argument and returns a boolean indicating whether or not it is a palindrome. It does so by splitting the string into an array of characters, reversing the array, and then joining the characters back into a string. If the resulting string is equal to the original input string, then it is a palindrome.

Reverse a String

How would you reverse a string in place (without creating a new string)?

```
function reverseStringInPlace(str) {
let left = 0;
let right = str.length - 1;
str = str.split('');

while (left < right) {
const temp = str[left];
str[left] = str[right];
str[right] = temp;

sql
Copy code
left++;
right--;
}

return str.join('');
}

console.log(reverseStringInPlace('hello')); // 'olleh'
```

Explanation: The `reverseStringInPlace` function takes a string as an argument and reverses it in place (without creating a new string). It does so by using two pointers (`left` and `right`) to keep track of the first and last characters of the string. It then swaps the characters at these positions and moves the pointers towards the center of the string until they meet in the middle. Finally, it returns the reversed string by joining the characters in the array back into a string.

Find the largest number in an array

How would you find the largest number in an array?

```
function findLargest(arr) {
```

```
let largest = -Infinity;
for (let i = 0; i < arr.length; i++) {
if (arr[i] > largest) {
largest = arr[i];
}
}
return largest;
}

console.log(findLargest([3, 5, 2, 8, 1])); // 8
```

Explanation: The `findLargest` function takes an array of numbers as an argument and returns the largest number in the array. It does so by initializing a variable `largest` to a very small number, and then iterating over the array using a for loop. For each iteration, it checks if the current element is greater than `largest`, and if so, updates `largest` with the current element. Finally, it returns the `largest` number.

Check Object Property

How would you check if an object has a property?

```
function hasProperty(obj, prop) {
  return obj.hasOwnProperty(prop);
}

const obj = { name: 'John', age: 30 };
console.log(hasProperty(obj, 'name')); // true
console.log(hasProperty(obj, 'email')); // false
```

Explanation: The hasProperty function takes an object and a property name as arguments and returns a boolean indicating whether the object has the property. It does so by using the built-in hasOwnProperty method on the object, which returns true if the object has the specified property, and false otherwise.

Common Elements in two Arrays

How would you find the common elements between two arrays?

```
function findCommonElements(arr1, arr2) {
  return arr1.filter(el => arr2.includes(el));
}
```

```
console.log(findCommonElements([1, 2, 3], [2, 3, 4]));
// [2, 3]
```

Explanation: The findCommonElements function takes two arrays as arguments and returns an array of the common elements between them. It does so by using the built-in filter method on the first array, passing a callback function that uses the built-in includes method to check if the current element exists in the second array. If it does, the element is included in the resulting array.

Function takes an array and returns a new array with only even numbers

Write a function that takes an array of numbers and returns a new array with only the even numbers.

```
function getEvenNumbers(numbers) {
  let evenNumbers = [];
  numbers.forEach(function(number) {
    if (number % 2 === 0) {
      evenNumbers.push(number);
    }
  });
  return evenNumbers;
}

const numbers = [1, 2, 3, 4, 5];
console.log(getEvenNumbers(numbers));  // [2, 4]
```

In this example, the getEvenNumbers function takes an array of numbers as an argument and uses the forEach method to iterate over the array. For each number, the function checks if it is even by using the modulo operator % to check if the remainder of the division by 2 is 0. If the number is even, it is pushed onto the evenNumbers array. The function returns the array of even numbers.

Function that takes array of objects and returns specific property values

Write a function that takes an array of objects and returns an array of values of a specific property of each object.

```
function getPropertyValues(objects, property) {
  let values = [];
  objects.forEach(function(object) {
    values.push(object[property]);
  });
  return values;
}

const objects = [
  {name: 'John', age: 32},
  {name: 'Jane', age: 28},
  {name: 'Jim', age: 35}
];
console.log(getPropertyValues(objects, 'name'));  //
['John', 'Jane', 'Jim']
```

In this example, the getPropertyValues function takes an array of objects and a property name as arguments and uses the forEach method to iterate over the array of objects

Function that returns largest number from the array

Write a function that takes an array of numbers and returns the largest number.

```
function findLargestNumber(numbers) {
  let largest = numbers[0];
  for (let i = 1; i < numbers.length; i++) {
    if (numbers[i] > largest) {
      largest = numbers[i];
    }
  }
  return largest;
}

const numbers = [1, 5, 10, 3, 20];
console.log(findLargestNumber(numbers));  // 20
```

In this example, the findLargestNumber function takes an array of numbers as an argument and uses a for loop to iterate over the array. The function initializes the largest variable to the first element of the array, and then iterates over the remaining elements, checking if each element is larger than the current largest value. If an element is larger, the largest variable is updated to that value. The function returns the largest number in the array.

Function returning array of objects and unique values

Write a function that takes an array of objects and returns an object with properties that correspond to the unique values of a specific property of each object.

```
function createObjectFromArray(objects, property) {
  let uniqueValues = {};
  objects.forEach(function(object) {
    uniqueValues[object[property]] = true;
  });
  return uniqueValues;
}

const objects = [
  {name: 'John', city: 'New York'},
  {name: 'Jane', city: 'London'},
  {name: 'Jim', city: 'New York'}
];
console.log(createObjectFromArray(objects, 'city'));
```

```
// {
//   New York: true,
//   London: true
// }
```

In this example, the createObjectFromArray function takes an array of objects and a property name as arguments and uses the forEach method to iterate over the array of objects. For each object, the value of the specified property is used as a key in the uniqueValues object, and the value is set to true. The function returns the uniqueValues object, which has properties that correspond to the unique values of the specified property in the array of objects.

Function that returns squares of array numbers

Write a function that takes an array of numbers and returns an array of the squares of each number.

```
function squareNumbers(numbers) {
  let squares = [];
  numbers.forEach(function(number) {
    squares.push(number * number);
  });
  return squares;
}

const numbers = [1, 2, 3, 4, 5];
console.log(squareNumbers(numbers));  // [1, 4, 9, 16,
25]
```

In this example, the squareNumbers function takes an array of numbers as an argument and uses the forEach method to iterate over the array. For each number, the square is determined by multiplying the number by itself and the result is pushed onto the squares array. The function returns the array of squares.

Function that returns new string with specific occurrences removed

Write a function that takes a string and returns a new string with all occurrences of a specified character removed.

```
function removeCharacter(string, character) {
  let newString = '';
 for (let i = 0; i < string.length; i++) {
    if (string[i] !== character) {
      newString += string[i];
    }
  }
  return newString;
}

const originalString = 'Hello World';
console.log(removeCharacter(originalString, 'o'));  //
'Hell Wrd'
```

In this example, the removeCharacter function takes a string and a character as arguments. The function uses a for loop to iterate over the characters in the string. If the current character is not equal to the specified character, it is concatenated onto the newString. The function returns the newString without the specified character.

Function returns new array of strings with 5 characters

Write a function that takes an array of strings and returns a new array with all strings that have a length of exactly 5 characters.

```
function findStringsOfLength5(strings) {
  let stringsOfLength5 = [];
  strings.forEach(function(string) {
    if (string.length === 5) {
      stringsOfLength5.push(string);
    }
  });
  return stringsOfLength5;
}

const strings = ['Hello', 'World', 'Five', 'Length',
'Words'];
console.log(findStringsOfLength5(strings));  //
['Hello', 'Words']
```

In this example, the findStringsOfLength5 function takes an array of strings as an argument and uses the forEach method to iterate over the array. For each string, the length is checked to see if it is equal to 5. If the length is 5, the string is pushed onto the stringsOfLength5 array. The function returns the stringsOfLength5 array, which contains all strings that have a length of exactly 5 characters.

Tips for writing better JavaScript code, along with code samples and explanations:

Use let and const instead of var

Use let and const instead of var: Use let and const instead of var to declare variables in JavaScript. This helps to avoid variable hoisting and scope issues.
Example:

```
// using let
let name = 'John Doe';
console.log(name);

// using const
const PI = 3.14;
console.log(PI);
```

Use template literals

Use template literals instead of concatenation: Use template literals (` `) instead of concatenation to create strings that contain variables.
Example:

```
// using template literals
let name = 'John Doe';
console.log(`Hello, ${name}!`);

// using concatenation
let name = 'John Doe';
console.log('Hello, ' + name + '!');
```

Use arrow functions

Use arrow functions for concise syntax: Use arrow functions (=>) to create anonymous functions with a concise syntax.
Example:

```
// using arrow functions
const square = num => num * num;
console.log(square(5));

// using traditional function syntax
function square(num) {
  return num * num;
}
console.log(square(5));
```

Use destructuring

Use destructuring to extract values from objects and arrays: Use destructuring to extract values from objects and arrays and assign them to variables.

Example:

```
// using destructuring with an object
const person = { name: 'John Doe', age: 30 };
const { name, age } = person;
console.log(name, age);

// using destructuring with an array
const numbers = [1, 2, 3, 4, 5];
const [first, second, ...others] = numbers;
console.log(first, second, others);
```

Use spread operator

Use spread operator to spread arrays: Use the spread operator (...) to spread arrays into separate values or to concatenate arrays.
Example:

```
// using spread operator to spread array into separate
values
const numbers = [1, 2, 3];
const max = Math.max(...numbers);
console.log(max);

// using spread operator to concatenate arrays
const numbers1 = [1, 2, 3];
const numbers2 = [4, 5, 6];
const allNumbers = [...numbers1, ...numbers2];
console.log(allNumbers);
```

Use map, filter, and reduce

Use map, filter, and reduce to transform arrays: Use map, filter, and reduce to transform arrays and extract information from them.
Example:

```
// using map to transform an array
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers);
```

```
// using filter to extract information from an array
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 ===
0);
console.log(evenNumbers);

// using reduce to extract information from an array
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, num) => acc + num, 0
console.log(sum);
```

In this example, the reduce method takes two arguments: a callback function and an initial value. The callback function takes two arguments: an accumulator (acc) and the current value (num). The reduce method iterates through the array, updating the accumulator with each iteration, and returns the final accumulator value. In this case, the final accumulator value is the sum of all the numbers in the array.

JavaScript Closure Explained

A closure in JavaScript is a function that has access to variables in its parent scope, even after the parent function has returned.

Closures are created when a function is defined inside another function, and the inner function retains access to the variables in the outer function's scope.

Here is an example of a closure in JavaScript:

code example
```
function outerFunction(x) {
    var innerVar = 4;
    function innerFunction() {
        return x + innerVar;
    }
    return innerFunction;
}

var closure = outerFunction(2);
console.log(closure()); // Output: 6
```

In this example, the innerFunction is a closure because it has access to the variable x and innerVar from the outerFunction even after outerFunction has returned.

A closure has three scope chains:

1. It has access to its own scope (variables defined between its curly braces {}).

2. It has access to the outer function's variables.
3. It has access to the global variables.

Closures are commonly used in JavaScript for a variety of tasks, such as:

- Implementing private methods and variables.
- Creating callback functions that retain access to variables from their parent scope.
- Creating and returning an object that has access to variables from its parent scope.

JavaScript closures are an important concept and it is important to understand how closures work in JavaScript. It is also important to be aware of the scope chain, and how closures interact with the scope chain.

```
<!DOCTYPE html>
<html>
<head>
   <title>Learn JavaScript</title>
</head>
<body>
   <h1>Complete JavaScript Course </h1>
   <div class="output"></div>
   <script src="app6.js"></script>
```

```
</body>
</html>
const val1 = 10;

function outerFun(x){
   const val2 = 10;
   function innerFun(){
       return x + val2 + val1;
   }
   return innerFun;
}

const val3 = outerFun(15);
console.log(val3());

for(let x=0;x<10;x++){
   console.log(outerFun(x+2)());
}
```

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

https://youtu.be/wdoIV_09xAc

Here is an example of JSON data:

```
{
   "name": "Laurence Svekis",
   "age": 41,
   "address": {
       "street": "10 Main St",
       "city": "New York",
       "state": "NY",
       "zip": 10001
   },
   "phoneNumbers": [
       {
           "type": "home",
           "number": "212 123-1234"
```

```
        },
        {
            "type": "work",
            "number": "646 123-4567"
        }
    ]
}
```

JavaScript provides methods JSON.stringify() and JSON.parse() to convert between JSON and JavaScript objects.

Example of converting JavaScript object to JSON:

Code Example :

```
const object = { name: 'John Doe', age: 35 };
const json = JSON.stringify(object);
console.log(json);
Example of converting JSON to JavaScript object:
```

Code Example :

```
const json = '{"name":"John Doe","age":35}';
const object = JSON.parse(json);
console.log(object.name); // "John Doe"
```

In summary, JSON is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is based on a subset of JavaScript and can be used with many programming languages. JavaScript provides built-in methods for converting between JSON and JavaScript objects.

There are several ways to get JSON data with JavaScript. One common method is to use the fetch() function to make an HTTP request to a server that returns JSON data. The fetch() function returns a promise that resolves to a response object, from which the JSON data can be extracted using the json() method.

Here is an example of how to get JSON data from a remote server:

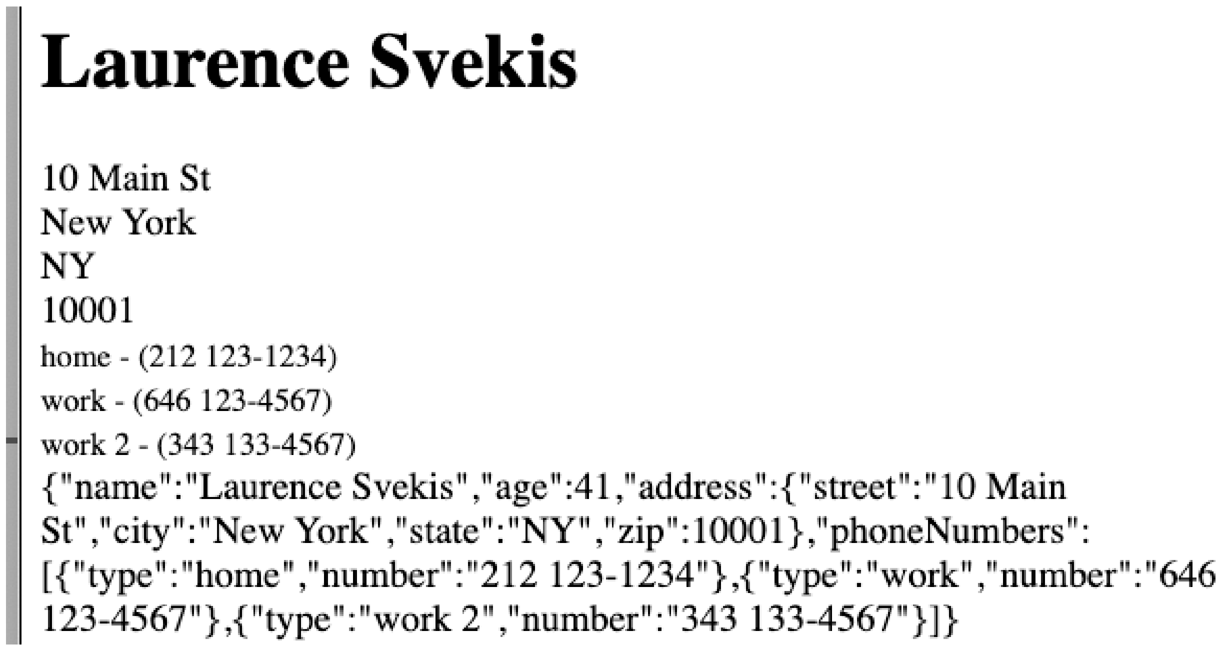
```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Another way to get JSON data is to load it from a local file using the XMLHttpRequest object or the fetch() function.

Here is an example of how to get JSON data from a local file:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'data.json', true);
xhr.responseType = 'json';
xhr.onload = function() {
  if (xhr.status === 200) {
    console.log(xhr.response);
  }
};
xhr.send();
```

In summary, there are several ways to get JSON data with JavaScript, including using the fetch() function to make an HTTP request to a server that returns JSON data or by loading JSON data from a local file using the XMLHttpRequest object or the fetch() function. Once you have the data you can use json() to access the data.



```
Laurence Svekis

10 Main St
New York
NY
10001
home - (212 123-1234)
work - (646 123-4567)
work 2 - (343 133-4567)
{"name":"Laurence Svekis","age":41,"address":{"street":"10 Main
St","city":"New York","state":"NY","zip":10001},"phoneNumbers":
[{"type":"home","number":"212 123-1234"},{"type":"work","number":"646
123-4567"},{"type":"work 2","number":"343 133-4567"}]}
```

```
<!DOCTYPE html>

<html>

<head>

   <title>Learn JavaScript</title>

</head>

<body>

   <h1>Complete JavaScript Course </h1>

   <div class="output">Data</div>

   <script src="app7.js"></script>

</body>

</html>


const url = 'my1.json';

const output = document.querySelector('.output');
const dataSt = '{"name":"Laurence
Svekis","age":41,"address":{"street":"10 Main
St","city":"New
York","state":"NY","zip":10001},"phoneNumbers":[{"type"
:"home","number":"212
123-1234"},{"type":"work","number":"646
123-4567"},{"type":"work 2","number":"343
133-4567"}]}';
console.log(dataSt);
const dataObj = JSON.parse(dataSt);
console.log(dataObj);

output.addEventListener('click',getJsonData);

function getJsonData(){
   output.textContent = 'loading.....';
   fetch(url)
   .then(response => response.json())
   .then(data => {
       myOutput(data);
   })
   .catch(error => {
       console.error('Error:',error);
   })
```

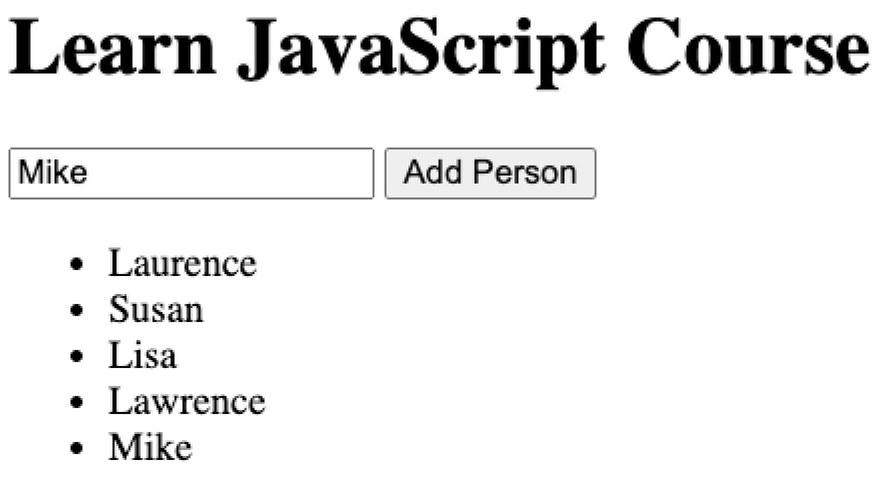
```
}

function myOutput(data){
   let html = `<h1>${data.name}</h1>
<div>${data.address.street}</div>
<div>${data.address.city}</div>
<div>${data.address.state}</div>
<div>${data.address.zip}</div>
`;
data.phoneNumbers.forEach(el =>{
   html += `<small>${el.type} -
(${el.number})</small><br>`;
})
html += JSON.stringify(data);
   output.innerHTML = html;
}


{
   "name": "Laurence Svekis",
   "age": 41,
   "address": {
       "street": "10 Main St",
       "city": "New York",
       "state": "NY",
       "zip": 10001
   },
   "phoneNumbers": [
       {
           "type": "home",
           "number": "212 123-1234"
       },
       {
           "type": "work",
           "number": "646 123-4567"
       },
       {
           "type": "work 2",
           "number": "343 133-4567"
       }
   ]
}
```

JavaScript Create Element List

https://youtu.be/oBuBoCrLRWg

The document.createElement() method in JavaScript is used to create a new HTML element with a specified tag name. The method takes a single argument, which is the tag name of the element to be created. For example, document.createElement("div") creates a new div element. The newly created element can be accessed and modified through the DOM API, such as adding content, attributes, and styles to the element. It can also be added to the document by using methods such as appendChild() or insertAdjacentHTML().

In the below example we will be creating a dynamic list, all the elements are created using JavaScript, adding the button for interaction when the user wants to add new people to the list.

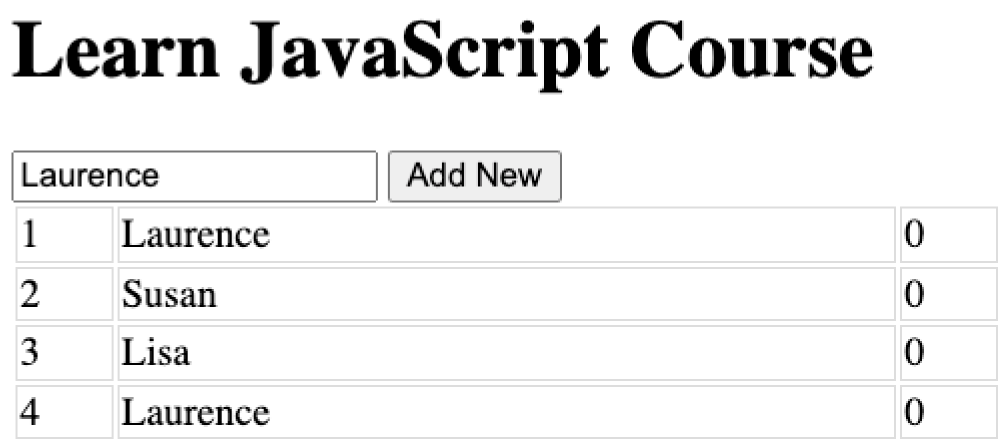

Learn JavaScript Course

- Laurence
- Susan
- Lisa
- Lawrence
- Mike

```
const myArr = ['Laurence','Susan','Lisa'];
const output = document.querySelector('.output');
const btn = document.createElement('button');
btn.textContent = 'Add Person';
output.append(btn);

const myInput = document.createElement('input');
myInput.setAttribute('type','text');
myInput.value = 'Lawrence';
output.prepend(myInput);

const ul = document.createElement('ul');
output.append(ul);
build();

btn.addEventListener('click',addPerson);

function addPerson(){
   const newPerson = myInput.value;
   myArr.push(newPerson);
   adder(newPerson);
   console.log(myArr);
}

function adder(person){
   const li = document.createElement('li');
   li.textContent = person;
   ul.append(li);
}

function build(){
   myArr.forEach(ele => {
       adder(ele);
   })
}
```

Create an interactive table list of item object values from a JavaScript array.

https://youtu.be/4Pvz_ILMEdE



1	Laurence	0
2	Susan	0
3	Lisa	0
4	Laurence	0

Create a list of items within a table using JavaScript. Data is contained within an array with object values.

```
<!DOCTYPE html>
<html>
<head>
   <title>Learn JavaScript</title>
   <style>
       table{
           width:100%;
       }
       td:first-child{
           width:10%;
       }
       td:last-child{
           width:10%;
       }
       td{
           border: 1px solid #ddd;
       }
   </style>
</head>
<body>
   <h1>Learn JavaScript Course </h1>
   <div>
       <input type="text" id="addFriend" >
       <input type="button" id="addNew" value="Add
New">
       <div class="output"></div>
   </div>
   <script src="app10.js"></script>
</body>
</html>

const myArr = [
   {name:'Laurence',score:0,id:1} ,
   {name:'Susan',score:0,id:2} ,
   {name:'Lisa',score:0,id:3}
];
const output = document.querySelector('.output');
const btn = document.querySelector('#addNew');
const addFriend = document.querySelector('#addFriend');
const tblOutput = document.createElement('table');
output.append(tblOutput);
addFriend.value = 'Laurence';
build();

btn.addEventListener('click',()=>{
   const myObj =
{name:addFriend.value,score:0,id:myArr.length+1} ;
   myArr.push(myObj );
   console.log(myArr);
   build();
})

function build(){
   tblOutput.innerHTML = '';
   myArr.forEach((ele,ind) =>{
       const tr = document.createElement('tr');
       tblOutput.append(tr);
       const td1 = document.createElement('td');
       td1.textContent = ele.id;
       tr.append(td1);
       const td2 = document.createElement('td');
       td2.textContent = ele.name;
       tr.append(td2);
       const td3 = document.createElement('td');
       td3.textContent = ele.score;
       tr.append(td3);
       tr.addEventListener('click',()=>{
           ele.score++;
           td3.textContent = ele.score;
```

```
        })
    })

}
```

How to Create Page Elements with JavaScript

Create Page Elements with JavaScript

https://youtu.be/x8STY2Bat-Y

How to Create Page Elements and make them Interactive with Event LIsteners

There are several ways to create page elements with JavaScript, including:

Using the document.createElement() method, which creates a new element with the specified tag name. For example, the following code creates a new div element:

```
let newDiv = document.createElement("div");
```

Using the innerHTML property to add HTML content to an existing element. For example, the following code adds a new p element to an existing div element with an id of "container":

```
let container = document.getElementById("container");
container.innerHTML += "<p>Hello World</p>";
```

Using the appendChild() method to add a new element as a child of an existing element. For example, the following code adds a new p element as a child of an existing div element with an id of "container":

```
let container = document.getElementById("container");
let newP = document.createElement("p");
newP.innerHTML = "Hello World";
container.appendChild(newP);
```

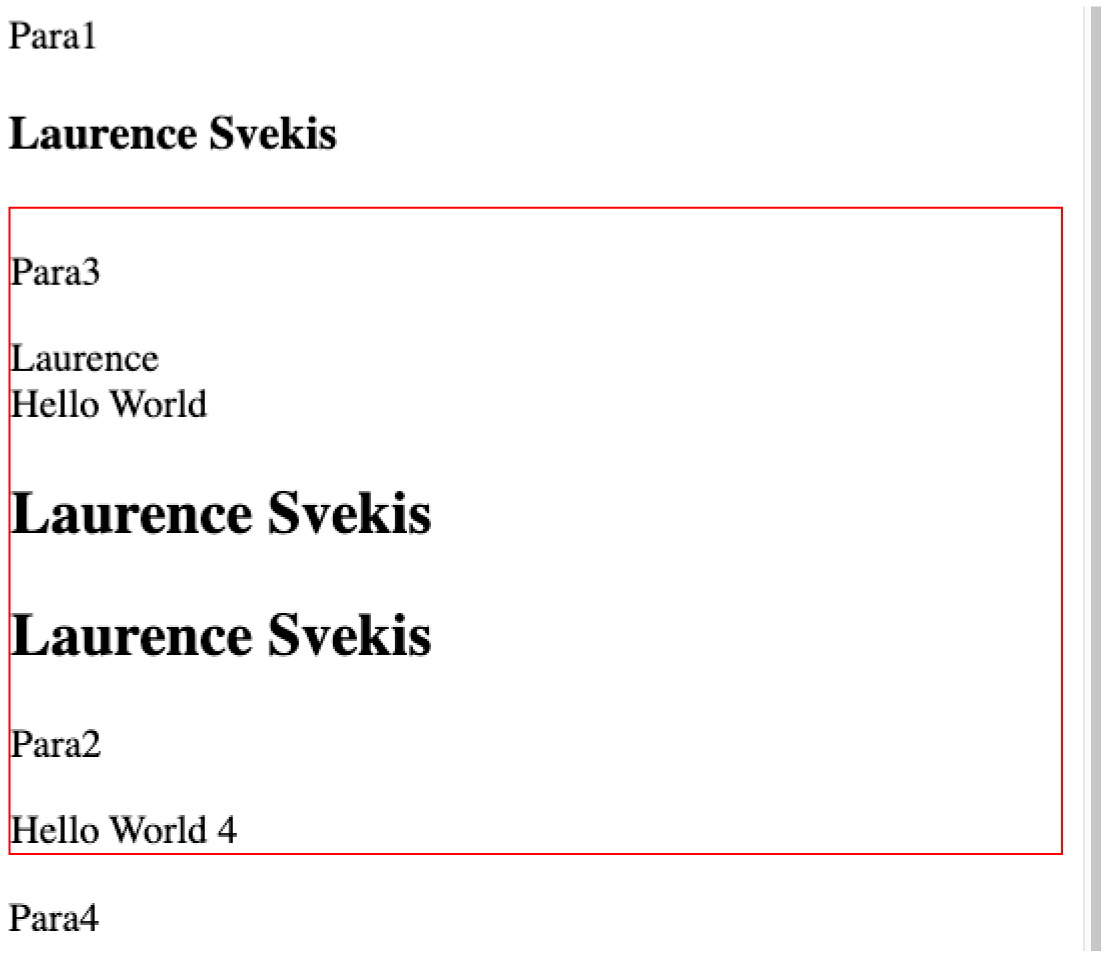
Using the insertAdjacentHTML() method to insert HTML content at a specific position relative to an existing element. For example, the following code adds a new p element before an existing div element with an id of "container":

```
let container = document.getElementById("container");
container.insertAdjacentHTML("beforebegin", "<p>Hello
World</p>");
```

You can also use any of the above methods to add CSS styles, classes and attributes to the newly created elements.

Coding Example of how to insert page content , html elements into your DOM page.

Coding Exercise to demo how to insert HTML page elements into the page with JavaScript


Para1

Laurence Svekis

Para3

Laurence
Hello World

Laurence Svekis

Laurence Svekis

Para2

Hello World 4

Para4

```
const ele1 = document.createElement('div');
ele1.textContent = 'My new element';
document.body.prepend(ele1);

const output = document.querySelector('.output');
output.innerHTML += '<div>Laurence</div>';
output.innerHTML += '<div>Hello World</div>';
output.style.border = '1px solid red';

const ele2 = document.createElement('h2');
```

```
ele2.innerHTML = 'Laurence Svekis';
const el = output.appendChild(ele2);
console.log(el);

const ele3 = document.createElement('h2');
ele3.innerHTML = 'Laurence Svekis';
const el2 = output.append(ele3);
console.log(el2);

output.insertAdjacentHTML('beforebegin','<p>Para1</p>')
;
output.insertAdjacentHTML('beforeend','<p>Para2</p>');
output.insertAdjacentHTML('afterbegin','<p>Para3</p>');
output.insertAdjacentHTML('afterend','<p>Para4</p>');

const ele4 = document.createElement('h3');
ele4.textContent = 'Laurence Svekis';
output.insertAdjacentElement('beforebegin',ele4);
output.insertAdjacentText('beforeend','Hello World 4');
```

JavaScript Async Code Examples

JavaScript is a single-threaded language, which means that it can only execute one piece of code at a time. This can make it challenging to perform long-running tasks, such as network requests, without freezing up the UI.

To solve this issue, JavaScript provides the concept of asynchronous programming, which allows you to run tasks in the background while the main thread continues to execute.

There are several ways to perform asynchronous operations in JavaScript:

Callbacks: A callback is a function that gets executed after another function has finished executing.

```
Example:
function fetchData(callback) {
  setTimeout(() => {
    callback({ data: "Example data" });
  }, 1000);
}
fetchData(data => console.log(data));
// Output (after 1 second): { data: "Example data" }
```

Promises: A Promise represents the eventual result of an asynchronous operation. A Promise can be in one of three states: pending, fulfilled, or rejected.

Example:

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve({ data: "Example data" });
  }, 1000);
});
fetchData.then(data => console.log(data));
// Output (after 1 second): { data: "Example data" }
```

async/await: async/await is a more concise and readable way to write asynchronous code, built on top of Promises. The async keyword is used to declare an asynchronous function, and the await keyword is used to wait for a Promise to be resolved.

Example:

```
async function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({ data: "Example data" });
    }, 1000);
  });
}
```

```
async function main() {
  const data = await fetchData();
  console.log(data);
}
main();
// Output (after 1 second): { data: "Example data" }
```

By using these asynchronous programming techniques, you can run long-running tasks in the background, without blocking the main thread and freezing the UI.

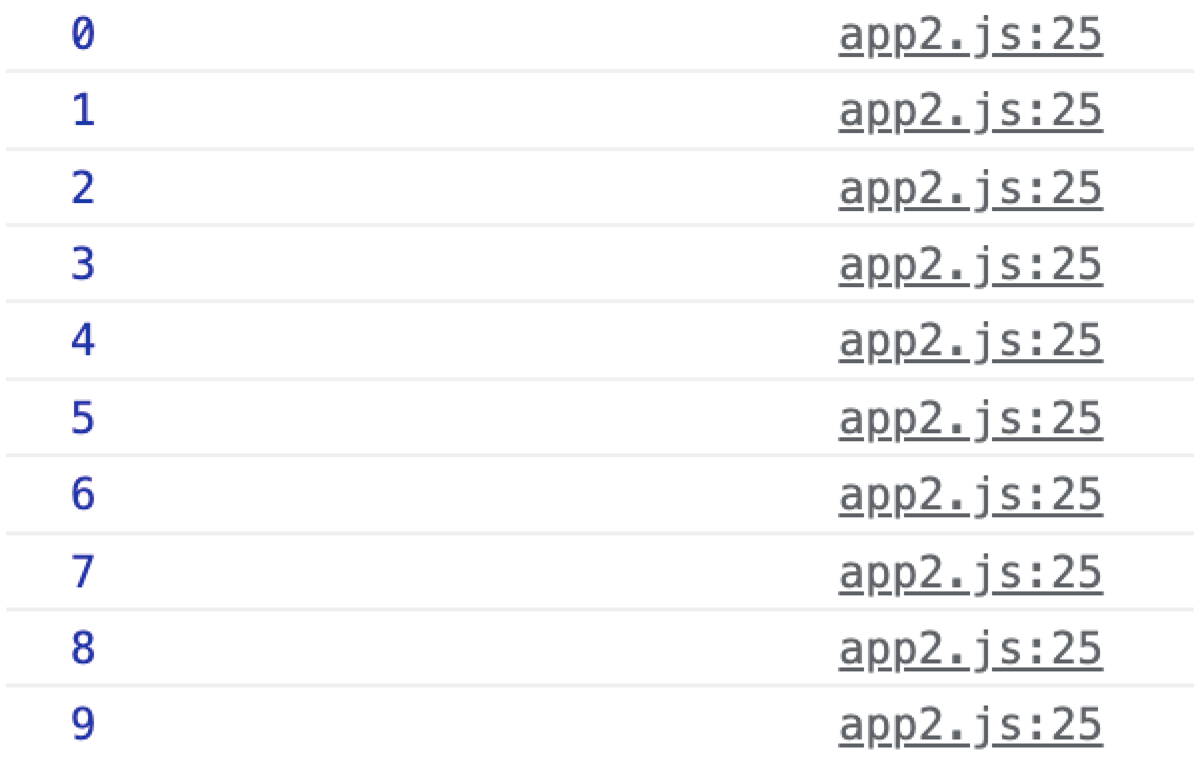
JavaScript Closure

A closure in JavaScript is a function that has access to the variables in its parent scope, even after the parent function has completed execution. This allows for data to be "closed over" or remembered by the inner function, even after the outer function has returned.

https://youtu.be/AyQRYwV69cc

```
For example:
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  }
}

let counter = makeCounter();
console.log(counter()); // outputs 0
console.log(counter()); // outputs 1
console.log(counter()); // outputs 2
```

Here, the makeCounter function returns an inner function that has access to the count variable declared in its parent scope, and can "remember" the current count value even after the makeCounter function has completed execution. Each time the inner function is called, it returns the current value of count and increments it by 1.



```
0                                   app2.js:25
1                                   app2.js:25
2                                   app2.js:25
3                                   app2.js:25
4                                   app2.js:25
5                                   app2.js:25
6                                   app2.js:25
7                                   app2.js:25
8                                   app2.js:25
9                                   app2.js:25
```

```
const a = 'hello';
console.log(a);
abc();

function abc(){
   //const a = 'world';
   console.log(a);
}

function myCount(){
   let count = 0;
   return function(){
        return count++;
    }
}
function myCount2(){
    let count = 0 ;
    return count++;
}

let cnt = myCount();
let cnt2 = myCount2;

for(let x=0;x<10;x++){
    console.log(cnt());
    console.log(cnt2());
}
```

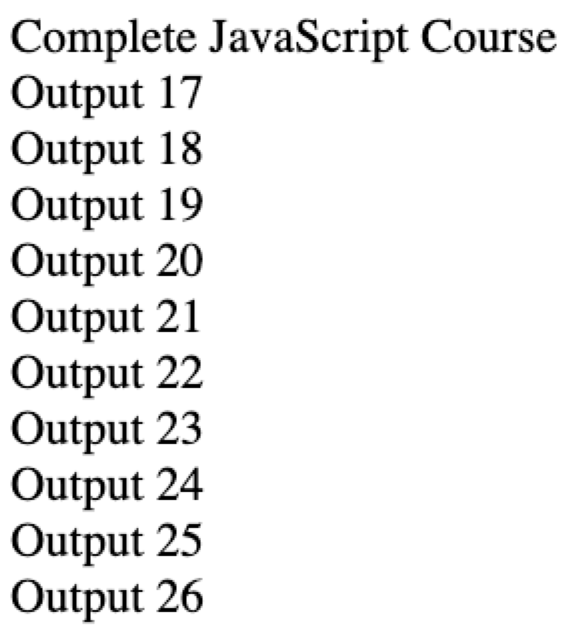
JavaScript Closure Advanced

In this example, the makeAdder function takes in a single argument x and returns an inner function that takes in a second argument y. The inner function has access to the x variable declared in the parent scope and uses it to add x and y together and return the result.

We can see here that the outer function makeAdder has been executed twice and it returns two different inner functions which are assigned to different variables add5 and add10 and these inner functions are able to remember their respective parent scope values of x.

https://youtu.be/8EgbirmLt0g

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  }
}

let add5 = makeAdder(5);
console.log(add5(3)); // outputs 8
console.log(add5(4)); // outputs 9

let add10 = makeAdder(10);
console.log(add10(5)); // outputs 15
console.log(add10(6)); // outputs 16
```



```
Complete JavaScript Course
Output 17
Output 18
Output 19
Output 20
Output 21
Output 22
Output 23
Output 24
Output 25
Output 26
```

```
const output = document.querySelector('#output');

function adder(val){
   return function(val2){
       return val + val2;
   }
}

let a1 = adder(15);
console.log(a1(2));

for(let x=0;x<10;x++){
  output.innerHTML += `<div>Output ${(a1(2+x))}</div>`;
}
```

JavaScript Image Gallery and Dynamic Image Gallery using page classes or create page elements on the fly with code

https://youtu.be/nsGGMAYnLbs

Here is an example of a JavaScript image gallery maker that creates a simple image gallery with prev/next buttons to navigate through the images:

```
<div id="gallery">
  <img src="image1.jpg" id="current-image">
  <button id="prev-button">Prev</button>
  <button id="next-button">Next</button>
</div>

<script>
  var images = ["image1.jpg", "image2.jpg",
"image3.jpg", "image4.jpg"];
  var currentIndex = 0;
  var gallery = document.getElementById("gallery");
  var currentImage =
document.getElementById("current-image");
  var prevButton =
document.getElementById("prev-button");
  var nextButton =
document.getElementById("next-button");

  prevButton.addEventListener("click", function() {
    currentIndex--;
    if (currentIndex < 0) {
      currentIndex = images.length - 1;
    }
    currentImage.src = images[currentIndex];
  });

  nextButton.addEventListener("click", function() {
    currentIndex++;
    if (currentIndex >= images.length) {
      currentIndex = 0;
    }
    currentImage.src = images[currentIndex];
  });
</script>
```

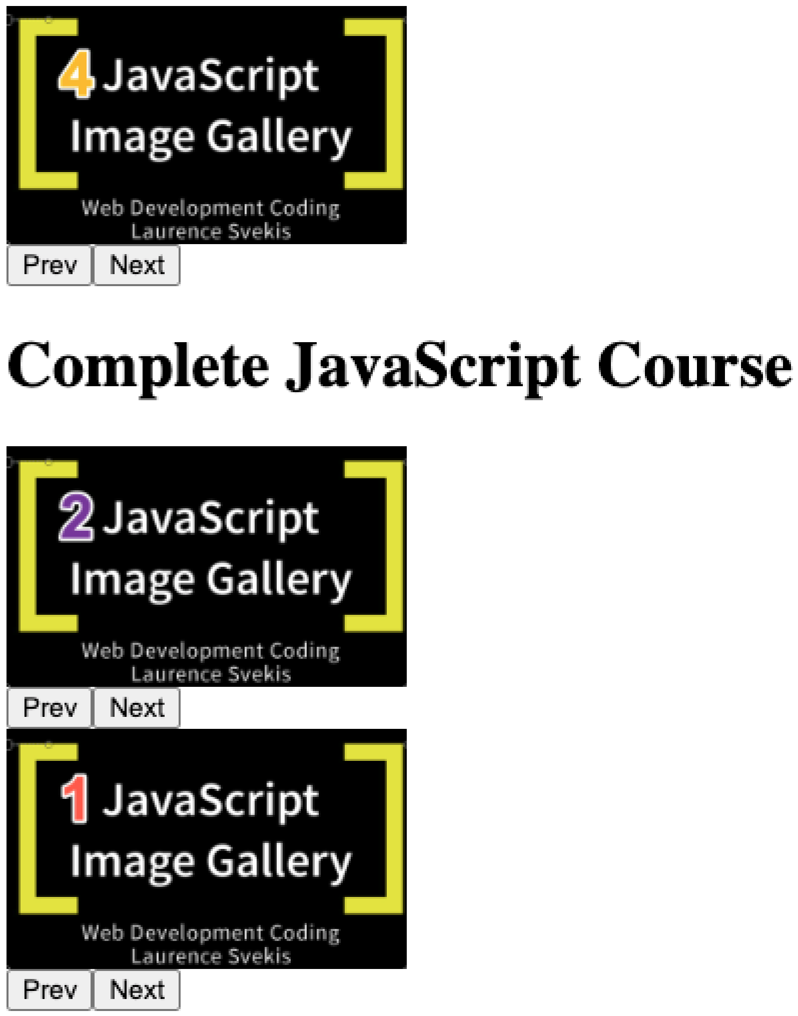
This example uses JavaScript to select the elements from the HTML, and add event listeners to the prev/next buttons to navigate through the images in the images array when clicked. The currentIndex variable keeps track of the current image being displayed, and the currentImage.src property is updated to show the next/prev image in the array when the buttons are clicked.

The above code is an example of a JavaScript image gallery maker that creates a simple image gallery with prev/next buttons. The code uses JavaScript to select the necessary elements from the HTML, such as the gallery container, current image, and prev/next buttons. It then adds event listeners to the prev/next buttons, so that when they are clicked, the current image being displayed is updated to the next/prev image in the images array. The currentIndex variable keeps track of the current image being displayed, and it is updated each time the prev/next buttons are clicked. When the current index reaches the end of the images array, it resets to the first image, thus creating an infinite loop.

Dynamic Image Gallery

How to Create an Image Gallery and Dynamic Image Gallery with JavaScript Code

The image gallery can also be used within a function to create multiple image galleries all working independently. Either creating them on the fly within the code or selecting existing elements with the class name and generating images within those elements.



Complete JavaScript Course

```
const output = document.querySelector('.output');
const images =
['one.jpg','two.jpg','three.jpg','four.jpg'];
/*
```

```
for(let x=0;x<12;x++){
   const el = document.createElement('div');
   output.append(el);
   cGallery(el);
}
*/
const eles = document.querySelectorAll('.gal');
eles.forEach(el => {
   cGallery(el);
})

function cGallery(parentEle){
   let curIndex = 0;
   const gallery = document.createElement('div');
   const curImage = document.createElement('img');
   curImage.setAttribute('src','one.jpg');
   const btn1 = document.createElement('button');
   btn1.textContent = 'Prev';
   const btn2 = document.createElement('button');
   btn2.textContent = 'Next';
   parentEle.append(gallery);
   gallery.append(curImage);
   gallery.append(btn1);
   gallery.append(btn2);
```

```
    btn1.addEventListener('click',()=>{
        curIndex--;
        if(curIndex<0){
            curIndex = images.length-1;
        }
        console.log(images[curIndex]);
        curImage.src = images[curIndex];
    })
    btn2.addEventListener('click',()=>{
        curIndex++;
        if(curIndex >= images.length){
            curIndex = 0;
        }
        console.log(images[curIndex]);
        curImage.src = images[curIndex];
    })
}
```

HTTP request in Javascript Get JSON data with xhr method and fetch methods

HTTP request in Javascript?

There are several ways to make an HTTP request in JavaScript, including using the XMLHttpRequest object or the fetch() function.

Here is an example of making an HTTP GET request using XMLHttpRequest:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://example.com");
xhr.send();
```

Here's an example of making an HTTP GET request to a JSON endpoint using the XMLHttpRequest object and parsing the response as JSON:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://example.com/data.json");
xhr.onload = function() {
    if (xhr.status === 200) {
        var data = JSON.parse(xhr.responseText);
        console.log(data);
    } else {
        console.error(xhr.statusText);
    }
};
xhr.onerror = function() {
    console.error(xhr.statusText);
};
xhr.send();
```

In this example, a new XMLHttpRequest object is created, and then opened with the "GET" method and the specified JSON endpoint URL. The onload event is used to handle the response, and onerror event is used to handle any error. The xhr.status is checked and if it's 200, it indicates that the request is successful, then we parse the response as JSON and

log the data to the console. If the xhr.status is not 200, it means there's an error and it logs the error message in the onerror function. If there's any network error, the onerror function is triggered, and it logs the error message.
Finally the request is sent using the xhr.send() method.
Please note that you should always check the response status code and handle it accordingly. Also, XMLHttpRequest is an old API, and **fetch() is more modern and recommended**

The fetch() method is a modern JavaScript method that allows you to make HTTP requests, similar to the XMLHttpRequest object. The fetch() method returns a promise that resolves to the response of the request, which can be a Response or Error object.

When you call the fetch() method, you pass in the URL of the endpoint you want to make the request to. You can also pass in an options object as the second parameter, which allows you to configure the request, such as setting the HTTP method, headers, and body.

The fetch() method returns a promise that resolves to the response of the request. Once you have the response, you can use the .json(), .text(), .blob() methods, etc to access the data of the response.

Here's an example of how you can use the fetch() method:

```
fetch("https://example.com/data.json")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

In this example, the fetch() method is used to make a GET request to a JSON endpoint. The .then() method is used to handle the response, which is passed as a parameter to the first callback function. The response.json() method is used to parse the response as JSON and the result is passed to the second callback function. Finally, the data is logged to the console. If there's any error during the request, it will be caught and logged by the catch function.

The fetch() method is a more modern and recommended way to make HTTP requests in JavaScript, it's more concise and easy to use, and it's supported in most modern browsers.

And here is an example of making an HTTP GET request using fetch():

```
fetch("https://example.com")
  .then(response => response.text())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

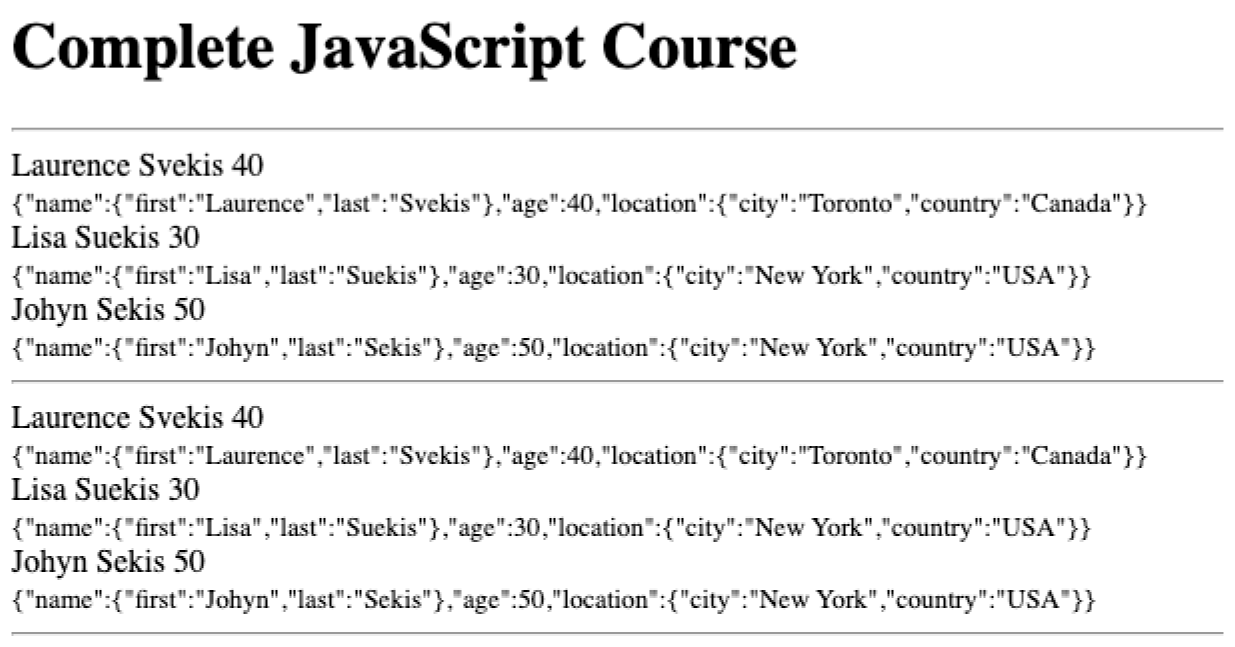
The first example uses the XMLHttpRequest object to create a new request, open it with the "GET" method and the specified URL, and then send it. The response to the request can then be handled using the onload or onerror events.

The second example uses the fetch() function to make the same GET request to the specified URL, and then uses the .then() method to handle the response, which is passed as a parameter to the first callback function. The response is transformed to text and then logged in the second callback function. If there's any error during the request, it will be caught and logged by the catch function.

Here's an example of making an HTTP GET request to a JSON endpoint using the fetch() function and parsing the response as JSON:

```
fetch("https://example.com/data.json")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

The fetch() function is used to make the GET request to the specified JSON endpoint. The response.json() method is then used to parse the response as JSON and the result is passed to the first callback function. In the second callback function, the data is logged. If there's any error during the request, it will be caught and logged by the catch function.


```
Complete JavaScript Course

Laurence Svekis 40
{"name":{"first":"Laurence","last":"Svekis"},"age":40,"location":{"city":"Toronto","country":"Canada"}}
Lisa Suekis 30
{"name":{"first":"Lisa","last":"Suekis"},"age":30,"location":{"city":"New York","country":"USA"}}
Johyn Sekis 50
{"name":{"first":"Johyn","last":"Sekis"},"age":50,"location":{"city":"New York","country":"USA"}}

Laurence Svekis 40
{"name":{"first":"Laurence","last":"Svekis"},"age":40,"location":{"city":"Toronto","country":"Canada"}}
Lisa Suekis 30
{"name":{"first":"Lisa","last":"Suekis"},"age":30,"location":{"city":"New York","country":"USA"}}
Johyn Sekis 50
{"name":{"first":"Johyn","last":"Sekis"},"age":50,"location":{"city":"New York","country":"USA"}}
```

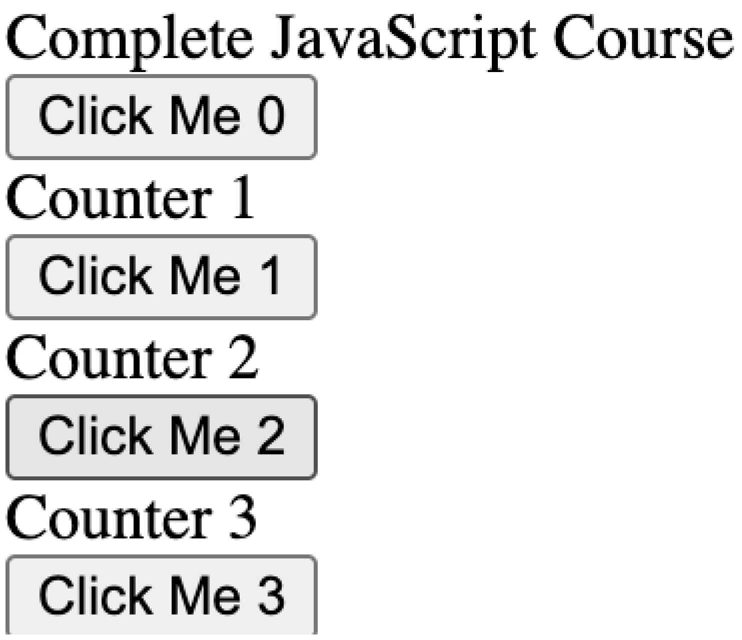
```
const output = document.querySelector('.output');
const url =
'https://www.discoveryvip.com/shared/person1000.json';
const xhr = new XMLHttpRequest();
xhr.open('GET',url);
xhr.onload = function(){
   if(xhr.status === 200){
       const data = JSON.parse(xhr.responseText);
       maker(data);
   }else{
       console.error(xhr.statusText);
   }
}
xhr.onerror = function(){
   console.error(xhr.statusText);
}
xhr.send();
output.innerHTML += '<hr>';

fetch(url)
   .then(res => res.json())
   .then(data =>   maker(data))
   .catch(error => console.log(error));



function maker(data){
   data.forEach(ele =>{
       output.innerHTML += `
       <div>${ele.name.first} ${ele.name.last}
${ele.age}</div>
       <small>${JSON.stringify(ele)}</small>`;
   })
   output.innerHTML += '<hr>';
}
```

JSON Code

```
[
{
  "name": {
    "first": "Laurence",
    "last": "Svekis"
  },
  "age": 40,
  "location": {
    "city": "Toronto",
    "country": "Canada"
  }
},
{
  "name": {
    "first": "Lisa",
    "last": "Suekis"
  },
  "age": 30,
  "location": {
    "city": "New York",
    "country": "USA"
  }
},
{
  "name": {
      "first": "Johyn",
      "last": "Sekis"
    },
    "age": 50,
    "location": {
      "city": "New York",
      "country": "USA"
    }
  }
]
```

How to add Fade Out and Fade in to page elements pure JavaScript

Learn how to apply fade in and fade out effects to HTML page elements with pure JavaScript code. Select and create new page elements dynamically with code, add event listeners and have the page elements fade in and fade out once the event is triggered. Adding Fade Effects to new and existing Page Elements



Complete JavaScript Course

```
const output = document.querySelector('#output');
for(let x=0;x<5;x++){
   const el = document.createElement('div');
   output.append(el);

   const btn = document.createElement('button');
   btn.textContent = `Click Me ${x}`;
   el.append(btn);

   const div = document.createElement('div');
   div.style.transition = 'opacity 1500ms';
   div.style.opacity = '1';
   div.textContent = `Counter ${x+1}`;
   el.append(div);
   btn.addEventListener('click',()=>{
       if(div.style.opacity === '1'){
           div.style.opacity = '0';
       }else{
           div.style.opacity = '1';
       }
   })
}

const fademe = document.querySelectorAll('.fader');
fademe.forEach((ele)=>{
   ele.style.transition = 'opacity 500ms';
   ele.style.opacity = '1';
   ele.addEventListener('click',(e)=>{
       ele.style.opacity = '0';
   })
})
   <div id="output">Complete JavaScript Course </div>
   <div class="fader">One</div>
   <div class="fader">Two</div>
   <div class="fader">Three</div>
   <div class="fader">Four</div>
   <script src="app1.js"></script>
```

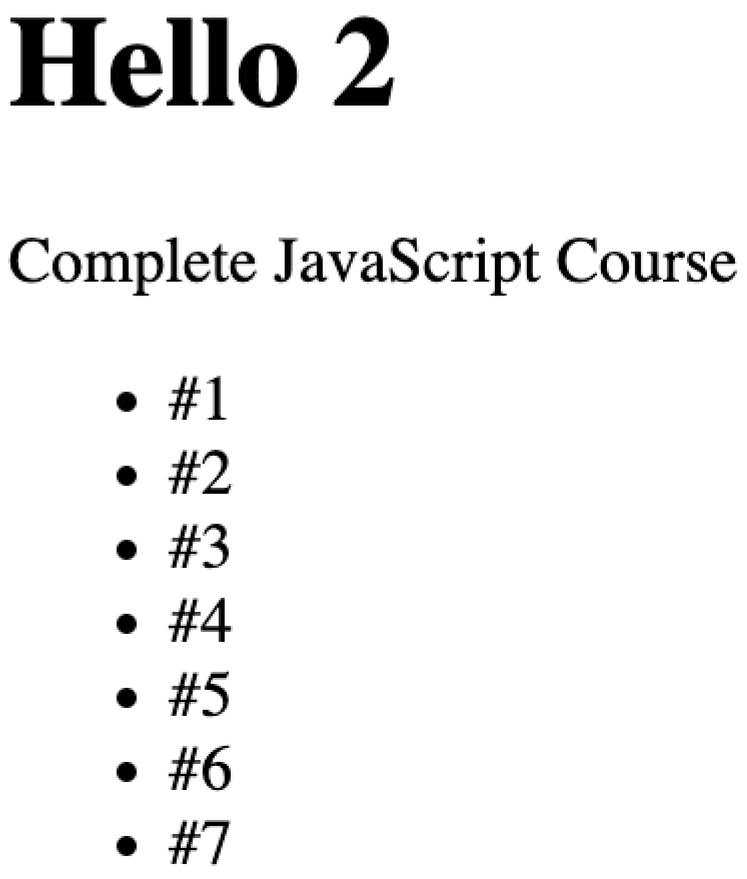
How to create page HTML elements with JavaScript code append prepend before after pure JavaScript

How to append and add new page elements with JavaScript

How to append and add new page elements with JavaScript using append, appendChild, prepend, before and after methods to dynamically add and reposition page elements

Create Page elements with Code

How to append and add new page elements with JavaScript


Hello 2

Complete JavaScript Course

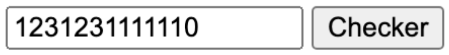
- #1
- #2
- #3
- #4
- #5
- #6
- #7

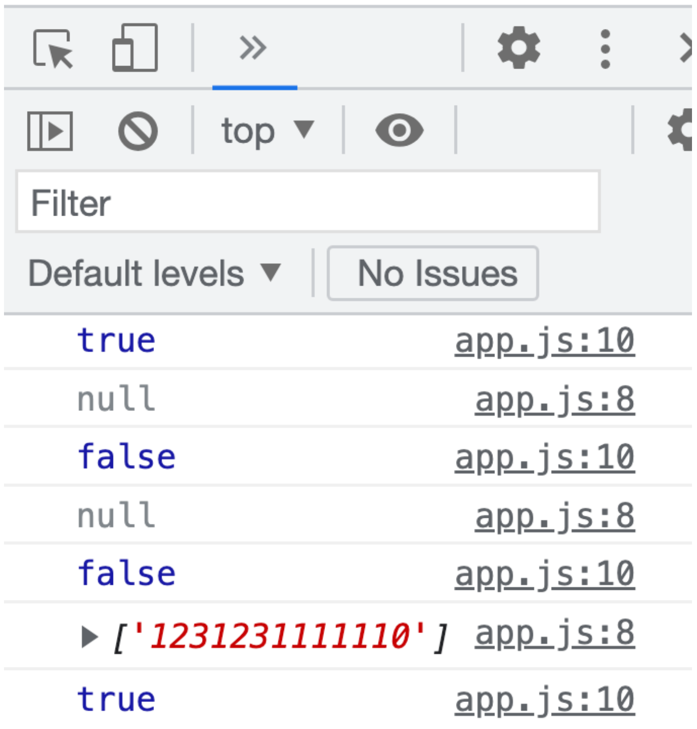
```
const output = document.querySelector('#output');
const pageBody = document.body;
const el1 = document.createElement('h1');
el1.textContent = 'Hello World 1';
console.log(el1);
pageBody.append(el1);
output.append(el1);
const res1 = output.appendChild(el1);
console.log(res1);

res1.textContent = 'Hello 1';
el1.textContent = 'Hello 2';
output.before(el1);
output.after(el1);
output.prepend(el1);
const ul = document.createElement('ul');
output.append(ul);
for(let i=0;i<10;i++){
   const li1 = document.createElement('li');
   li1.textContent = `#${i+1}`;
   ul.append(li1);
}
```

Regex Checking for Numbers in the input field

Check for values that match a Regex pattern in the input field. Push a button and apply the match checker to return the results in the console.

*/^[0-9]**$/g = Only numbers in the string
 /[0-9]+/g = Will return numbers in the result ignore non digits 0-9
/[\D]/g = Every Character other than digits
 /\d/g = Digits separated





```
true                app.js:10
null                 app.js:8
false               app.js:10
null                 app.js:8
false               app.js:10
▶ ['1231231111110']  app.js:8
true                app.js:10
```

```
<!DOCTYPE html>
<html>
<head>
 <title>JavaScript Course</title>
</head>
<body>
 <div>
   <input type="text" id="nums">
   <button id="btn">Checker</button>
 </div>
 <script src="app.js"></script>
</body>
</html>
```

```
const nums = document.querySelector('#nums');
const btn = document.querySelector('#btn');

btn.onclick = ()=>{
   const inputValue = nums.value;
   const patt = /^[0-3]*$/g;
   const results = inputValue.match(patt);
   console.log(results);
   const valNum = results != null;
   console.log(valNum);
}
```