Learn JavaScript Code Examples Coding Tips

Basics of JavaScript Code

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.

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.