www.techglobalschool.com

www.techglobalschool.com

JavaScript Mock-2 Questions and Answers

How does JavaScript differ from compiled languages, and what unique features characterize JavaScript as an interpreted language?

- JavaScript is an interpreted language, which means it's executed line-by-line, like reading a script.
- Compiled languages are first translated into machine code before execution.
- This gives JavaScript the advantage of being more flexible and easier to use.

Can you explain what NodeJS is and why it's important for JavaScript?

- NodeJS is a runtime environment for JavaScript that allows it to run on servers and outside of web browsers.
- It's crucial for server-side scripting, building APIs, and creating scalable network applications.

What are the differences between 'var', 'let', and 'const'?

- They are all used for variable declaration.
- **var** keyword is used in all JavaScript code from 1995 to 2015.
- **let** and **const** keywords were added to JavaScript in 2015.
- **var** is function-scoped, which means it doesn't respect block scope.
- **var** is hoisted while **let** and **const** cannot be hoisted.
- **let** and **const** are block-scoped and allow better control over variable scope.
- **let** variables can be reassigned, while **const** variables are constant once assigned.
- **const** is especially useful for declaring values that shouldn't change during execution.

List and briefly describe the basic data types in JavaScript.

- There are 7 primitives: **String**, **Number**, **Bigint**, **Boolean**, **Undefined**, **Null**, **Symbol**
- There are also Reference-Object data types: **arrays**, **objects**, **collections**, etc.
- **String**: A String is a sequence of characters like "John Doe".Strings are written with quotes. You can use single or double quotes but mostly double quotes.
- **Numbers**: All JS numbers are stored as decimal numbers (floating point).Numbers can be written with, or without decimals.
- **BigInt**: All JS numbers are stored in a a 64-bit floating-point format. JavaScript BigInt is a new datatype (2020) that can be used to store integer values that are too big to be represented by a normal JavaScript Number.
- **Boolean**: Boolean can only have two values: true or false. It is used often with conditional statements.
- **Undefined**: In JS, a variable without a value, has the value undefined. The type is also undefined.

- **Array**: JS arrays are used to store multiple data and written with square brackets []. Array items are separated by commas. Array indexes are zero-based.
- **Object**: JS objects are used to store complex data with key-value pairs and written with curly braces {}. Object properties are written as name:value pairs, separated by commas.

What is the difference between 'null' and 'undefined' in JavaScript?

- **null** represents an intentional absence of any object value, while **undefined** indicates that a variable has been declared but hasn't been assigned a value.
- Type of **null** is object while type of **undefined** is undefined.
- **null** is converted to zero while performing primitive operations while **undefined** is converted to NaN.

What are the differences between '=' and '==' and '==='?

- **=** is the assignment operator, used to assign a value to a variable.
- **==** is the equality operator, which compares values for equality with type coercion.
- **===** is the strict equality operator, which compares values for equality without type coercion.
  NOTE: **type coercion** is the process of converting a value from one data type to another, usually during runtime, to facilitate an operation or comparison. JavaScript performs **type coercion** in various situations, particularly when using operators like == (equality) and + (addition).
- Using **===** is recommended for accurate comparisons, as it considers both value and type.

What are the 'template literals' in JavaScript?

- Template literals or template strings are string literals allowing embedded expressions.
- These are enclosed by the back-tick (`) character instead of double or single quotes.
- In ES6, this feature enables using dynamic expressions and offer improved readability and flexibility when constructing complex strings.

What is array in JavaScript and why do we need it?

- An array in JavaScript is a special data structure that allows you to store and organize multiple values (elements) in a single variable.
- It's like having a collection of items in a list.
- Arrays are essential because they provide a way to work with groups of related data efficiently.
- You can access, manipulate, and iterate through array elements, making it easier to perform tasks like storing a list of names, numbers, or any other data.
- The typeof operator in JavaScript returns "object" for arrays.

What are the differences between array slice and splice methods?

- **array.slice(start, end)**: This method creates a new array by extracting a portion of the original array from the start index (inclusive) to the end index (exclusive). The original array remains unchanged.
- **array.splice(start, deleteCount, item1, item2, ...)**: The splice method changes the original array. It removes elements starting from the start index, as specified by deleteCount. You can also add new elements at that position if you provide item1, item2, and so on. It returns an array containing the removed elements.

What are the differences between array map, filter and reduce methods?

- The **map()** method creates a new array by applying a provided function to each element of the original array.
- It returns a new array of the same length as the original array, with each element transformed by the provided function.

```
const numbers = [1, 2, 3];
const doubled = numbers.map(x => x * 2); // [2, 4, 6]
```

- The **filter()** method creates a new array containing elements that meet a specified condition.
- It returns a new array with elements for which the provided function returns true

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(x => x % 2 === 0); // [2, 4]
```

- The **reduce()** method applies a provided function to accumulate values of an array, resulting in a single value.
- It returns a single accumulated value, such as a sum, product, or any other reduced result.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0); //
10
```

- **map()** and **filter()** both return new arrays based on the original array, while **reduce()** returns a single accumulated value.
- **map()** transforms each element based on the provided function, **filter()** selects elements that meet a condition, and **reduce()** accumulates values.

What are the differences between array some and every methods?

- The **some()** method checks if at least one element in the array satisfies a specified condition.
- It returns true if at least one element satisfies the condition; otherwise, it returns false.

```
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(x => x % 2 === 0); // true
```

- The **every()** method checks if all elements in the array satisfy a specified condition.
- It returns true if all elements satisfy the condition; otherwise, it returns false.

```
const numbers = [2, 4, 6, 8];
const allEvenNumbers = numbers.every(x => x % 2 === 0); //
true
```

How do you sort elements in an array in JavaScript?

- You can sort elements in an array using the **array.sort()** method.
- By default, it sorts elements as strings, which may not give the expected result for numbers.
- To sort numbers correctly, you can provide a comparison function as an argument. For example:

```
const numbers = [5, 1, 3, 2, 4];
numbers.sort((a, b) => a - b); // This sorts the array in
ascending order
```

What are loops in JavaScript?

- Loops in JavaScript are control structures that allow you to execute a block of code repeatedly.
- They are essential for automating repetitive tasks or iterating through arrays, objects, or other data structures.
- JavaScript offers several types of loops:
    - **for loop:** Executes a block of code a specified number of times.
    - **while loop**: Repeats a block of code as long as a condition is true.
    - **do...while loop**: Similar to a while loop, but it always runs the block of code at least once.
    - **for...in loop**: Iterates over the properties of an object.
    - **for...of loop**: Iterates over the values of an iterable object, like an array.
- **break**; is used to terminate the loop.
- **continue**; is used to skip only current iteration if a specified condition occurs and continues with the next iteration in the loop.

What are function expressions?

- Function expressions are a way to define functions in JavaScript by assigning them to variables or using them within expressions.
- Unlike function declarations, which are hoisted to the top of their containing scope and can be called before they appear in the code, function expressions are not hoisted.
- This means that you can only call a function expression after it has been defined.
- Function expressions can also be used as arguments to other functions, stored in arrays or objects, and passed around like any other value in JavaScript.
- They are particularly useful when you need to create functions dynamically or pass them as callbacks to other functions.

```
const greet = function(name) {
  return `Hello, ${name}!`;
};
console.log(greet("John")); // "Hello, John!"
```

What is lambda or arrow functions in JavaScript?

- Lambda functions, also known as arrow functions, are a concise way to define functions in JavaScript.
- They provide a shorter syntax compared to traditional function expressions.
- Arrow functions are particularly useful for defining anonymous functions and for functions with simple, single-line bodies.
- We don't need the function keyword, the return keyword, and the curly brackets.
- Using const is safer and preferred, because a function expression is always constant value.
- We can only omit the return keyword and the curly brackets if the function body has a single statement.
- We can skip the parentheses if we have only one parameter.

```
// Traditional function expression
const add = function(x, y) {
  return x + y;
};

// Arrow function
const add = (x, y) => x + y;
```

What is callback function and why do we need it?

- **A callback function** is a function that is passed as an argument to another function and is executed inside that function.
- Callbacks are commonly used in JavaScript.
- Callbacks are essential because they allow you to write code that doesn't block the execution of other code.
- This is crucial for creating responsive and non-blocking applications, especially in web development.
- Callbacks enable you to define what should happen after a certain operation completes.

What is callback hell?

- **Callback hell**, also known as the "Pyramid of Doom," refers to a situation where multiple nested callback functions make the code structure complex and hard to read.
- It occurs when you have a series of asynchronous operations that depend on each other, leading to deeply nested callback functions.
- **Callback hell** can make code difficult to maintain and debug.
- To address this issue, developers often use techniques like promises, async/await, or modularization to improve code readability and structure.

```
asyncFunction1(() => {
  asyncFunction2(() => {
    asyncFunction3(() => {
      // ... and so on
    });
  });
});
```

What are enhanced object literals in JavaScript?

Enhanced object literals are a set of enhancements introduced in ECMAScript 2015 (ES6) that make it easier and more concise to define objects in JavaScript.

```
//ES6
let x = 10, y = 20;
obj = { x, y };
console.log(obj); // {x: 10, y:20}

//ES5
var x = 10, y = 20;
obj = { x: x, y: y };
console.log(obj); // {x: 10, y:20}
```

What is the currying function in JavaScript?

- Currying is a functional programming technique where a function that takes multiple arguments is transformed into a series of functions, each taking one argument.
- It allows you to partially apply a function by providing one argument at a time.
- Currying is useful for creating reusable functions and for building more specialized functions from general ones.
- It's a powerful technique in functional programming.

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

const add5 = add(5);
const result = add5(3); // Result is 8
```

What are the spread operator and rest parameter and differences between them?

- **Spread Operator (...)**: The spread operator allows you to expand an iterable (e.g., an array) into individual elements.
- You can use it in various contexts, such as when creating a new array, passing function arguments, or merging objects.

```
const arr = [1, 2, 3];
const newArr = [...arr, 4, 5]; // Creates a new array [1, 2, 3, 4, 5]
```

- **Rest Parameter (...paramName)**: The rest parameter is used in function declarations to collect multiple arguments into a single array.
- It allows you to handle an arbitrary number of arguments conveniently.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
```

- The spread operator is used to split elements (e.g., from an array), while the rest parameter is used to collect elements into an array (e.g., function arguments).
- The spread operator is typically used in array or object literals, whereas the rest parameter is used in function parameters.

What is the difference between Shallow and Deep copy?

- **Shallow Copy**: A shallow copy of an object or array creates a new instance of the top-level object or array, but it doesn't duplicate nested objects or arrays. Instead, it references them. Modifying nested objects will affect both the original and the copied object.
- **Deep Copy**: A deep copy, on the other hand, creates a completely independent duplicate of the original object or array, including all nested objects or arrays. Changes made to the copy won't affect the original.

What is nullish coalescing operator (??)?

- **The nullish coalescing operator (??)** is used to provide a default value when a variable is **null** or **undefined**.
- It's helpful for handling cases where you want to assign a default value only when the variable is explicitly **null** or **undefined**, but not for other falsy values like 0, false, or an empty string.
- The key difference from the logical OR (||) operator is that ?? only falls back to the default value when the variable is null or undefined, not for other falsy values.
- The nullish coalescing operator is particularly useful when dealing with optional properties in objects or when setting default values for function parameters.

```
console.log(null ?? true); // true
console.log(false ?? true); // false
console.log(undefined ?? true); // true
```

What is this keyword and why do we use it in JavaScript?

- **this** keyword in JavaScript refers to the current object or context in which the code is executing.
- It allows us to access and manipulate the properties and methods of the object that the function is a part of.
- **this** is a fundamental concept in JavaScript because it helps us work with different objects and their data without hardcoding specific values.

```
// Using this in a object method
const instructor = {
    fName: 'John',
    lName: 'Doe',
    field: 'Science',
    fullName() {
        return `${this.fName} ${this.lName}`;
    }
};

console.log(instructor.fullName()); // John Doe
```

What is super keyword and why do we use it in JavaScript?

- **super** keyword is used in JavaScript within classes to call methods and access properties of a parent class (or superclass).
- It's typically used when creating a subclass (a class that inherits from another class) to invoke the constructor and methods of the parent class.
- It helps in reusing and extending the behavior of the parent class in the child class.

What is immutable in JavaScript? Explain in detail.

- Immutability in JavaScript refers to the concept of making data or objects unchangeable after they are created.
- Immutable data cannot be modified, which can lead to more predictable and reliable code. Instead of changing the original data, you create new instances with the desired modifications.
- This is often used with data structures like arrays and objects to prevent unintended side effects.

What is scope in JavaScript? Explain in detail.

- Scope determines the accessibility (visibility) of variables, objects, and functions.
- JavaScript has 4 types of scope:

1. Function Scope - (Local Scope)

- Variables declared within a JavaScript function, become LOCAL to the function.
- They can't be used outside the function.
- Variables declared with var, let and const are similar when declared inside a function considering the scope.

2. Block Scope

- JavaScript had only Global Scope and Function Scope before ES6 (2015).
- Variables declared with the var keyword can NOT have block scope.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide Block Scope in JavaScript.
- Variables declared inside a { } block with let or const cannot be accessed from outside the block.

3. Global Scope

- Variables declared globally (outside any function) have Global Scope.
- Global variables can be accessed from anywhere in the program.
- Variables declared with var, let and const are quite similar when declared outside a block.

4. Lexical Scope

- Used in case of nested functions and defines the accessibility of variables for inner and outer functions.
- In simple terms, lexical scope means that a variable or function can only be accessed within the scope in which it is defined.
- The scope of a variable or function is determined by its position in the source code.
- **NOTE**: Before ES6 (2015), JavaScript had only Global Scope and Function Scope.

What is JSON why we need it and what are its common operations?

- **JSON** stands for **JavaScript Object Notation** and is a lightweight data-interchange format.
- It is primarily used for data exchange between a server and a web application, but it has become a universal data format that can be used in various programming languages.
- **JSON** is text-based and easy for humans to read and write, and machines can parse it and generate it.

Why do we need JSON?

- **Data Interchange:** JSON is used for transmitting data between a server and a web application. It provides a standardized format for data exchange, making it easier for different systems to communicate.
- **Human-Readable:** JSON is easy to read, write, and understand, which is useful for developers when debugging or inspecting data.

Common JSON Operations:

- **Serialization**: Converting a data structure (e.g., an object or an array) into a JSON string is known as serialization. This is done using the **JSON.stringify()** method.

```
const data = { name: "John", age: 30 };
const jsonData = JSON.stringify(data);
```

- **Deserialization**: The reverse process, where a JSON string is converted back into a data structure, is called deserialization. This is accomplished using the JSON.parse() method.

```
const jsonString = '{"name":"John","age":30}';
const data = JSON.parse(jsonString);
```

- **Accessing Data**: Once data is in JSON format, you can access specific values using dot notation or bracket notation, just like you would with JavaScript objects.
- **Modifying Data:** You can also modify JSON data by updating its values, adding, or removing properties, and nesting objects or arrays.

What is serialization and deserialization?

- **Serialization**: Converting a data structure (e.g., an object or an array) into a JSON string is known as serialization. This is done using the **JSON.stringify()** method.

```
const data = { name: "John", age: 30 };
const jsonData = JSON.stringify(data);
```

- **Deserialization**: The reverse process, where a JSON string is converted back into a data structure, is called deserialization. This is accomplished using the JSON.parse() method.

```
const jsonString = '{"name":"John","age":30}';
const data = JSON.parse(jsonString);
```

What is hoisting in JavaScript?

- Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed.
- It may seem like variables are declared where you write them, but in reality, they are processed before the code runs.
- However, it's important to note that only the declarations are hoisted, not the initializations. So, while the variable declaration is moved to the top, the value assignment remains in place.

```
console.log(x); // undefined
var x = 5;
// "x" is hoisted, and the code is interpreted as below:
var x;
console.log(x); // undefined
x = 5;
```

What is IIFE(Immediately Invoked Function Expression)?

- **IIFE** stands for **Immediately Invoked Function Expression.**
- It is a common JavaScript design pattern used to create a function expression and execute it immediately after its declaration.
- **IIFE** is typically wrapped in parentheses to distinguish it as an expression and to prevent its variables from polluting the global scope.

```
(function() {
    console.log('Hello World'); // Hello World
})();
```

What are closures in JavaScript?

- A closure is a function that has access to its own scope, the outer (enclosing) function's scope, and the global scope.
- This allows the inner function to access variables from its containing functions even after they have completed execution.
- Closures are commonly used for data encapsulation, creating private variables, and implementing callback functions.

```
function outerFunction() {
  let outerVar = "I'm from outer function";

  function innerFunction() {
    console.log(outerVar);
  }

  return innerFunction;
}

const myClosure = outerFunction();
myClosure(); // I'm from outer function
```

What are bind, call and apply methods? Why do we use them?

- **bind():** is used to create a new function that, when invoked, has its **this** value set to a specific object, and it allows you to pre-specify arguments.
- This is often used to **bind** a function to a particular context.

```
const greet = function(name) {
  console.log(`Hello, ${name}!`);
};

const greetJohn = greet.bind(null, "John");
greetJohn(); // Hello, John!
```

- **call():** is used to invoke a function with a specific **this** value and arguments passed individually.

```
const person = {
  name: "Alice"
};

function introduce(age) {
  console.log(`I'm ${this.name} and I'm ${age} years old.`);
}
introduce.call(person, 25); // I'm Alice and I'm 25 years old.
```

- **apply():** Similar to call(), the apply() method is used to invoke a function with a specific **this** value and arguments passed as an array.

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

function sum() {
  return numbers.reduce((total, num) => total + num, 0);
}

const result = sum.apply(null, numbers);
console.log(result); // 15
```

What is AJAX?

- **AJAX** stands for **Asynchronous JavaScript and XML**, and is a set of web development techniques used to create asynchronous web applications.
- It allows web pages to send and receive data from a server without the need to refresh the entire page.
- While the acronym mentions **XML**, in practice, data formats like JSON are often used instead.
- **AJAX** is a key technology for building dynamic and interactive web applications.

What are the different ways to deal with Asynchronous Code?

- **Asynchronous JavaScript** refers to the ability of JavaScript to execute code that doesn't block the main thread of execution.
- This allows for non-blocking operations like network requests, file reading/writing, timers, and user interactions, where the program doesn't have to wait for one operation to complete before moving on to the next one.
- Asynchronous operations are crucial for building responsive and efficient web applications.
- To handle asynchronous JavaScript, we use one or more of the following techniques;

1. Callbacks:

- Callbacks are functions that you pass as arguments to another function and are executed once the asynchronous operation is complete.
- Callbacks have been a traditional way to handle asynchronous code in JavaScript.

2.Promise:

- Promises are a more modern and structured way to handle asynchronous code.
- They represent a value that might be available now, in the future, or never.
- Promises provide methods like .then() to attach success and error handlers to execute when the asynchronous operation completes or fails.

3. Async/Await:

- Async functions provide a way to write asynchronous code that looks more like synchronous code, making it easier to read and reason about.
- We can use the async keyword before a function declaration and await inside that function to pause execution until a promise is resolved.

4. Event Handlers:

- Event-driven programming is common in web development.
- We can attach event handlers to elements and respond to events like clicks, inputs, or HTTP requests.
- **NOTE**: Promises and async/await have become the preferred ways to handle asynchronous code due to their readability and maintainability, especially in complex applications.
- Additionally, many modern web APIs and libraries are designed around promises and async/await, making them a natural choice for working with asynchronous JavaScript.

What is promise, why wee ned it and what are the states of promise?

- A promise in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.
- Promises are used to handle asynchronous operations in a more structured and readable way, making it easier to manage complex workflows.

Promises have three states:

- **Pending**: Initial state when the promise is neither fulfilled nor rejected.
- **Fulfilled (Resolved)**: The state when the asynchronous operation is successful, and the promise is assigned a value.
- **Rejected**: The state when the asynchronous operation encounters an error or fails, and the promise is given a reason for failure.

What is an async function in JavaScript?

- An **async function** in JavaScript is a special type of function that allows you to write asynchronous code in a more synchronous and readable manner.
- It makes working with promises more intuitive and simplifies error handling using **try/catch**.
- Inside an **async function**, we can use the **await** keyword to pause the function's execution until a promise is resolved, making it look like the code is executing in a synchronous manner.

What is call stack in JavaScript?

- It is a mechanism for the JS interpreter to keep track of its place in a script that calls multiple functions.
- It is simply keeping the track of what function is currently being run and what functions are called from within that function.
- It works based on **LIFO (last-in-first-out)** principle.

What is an event loop in JavaScript?

- The event loop in JavaScript is a crucial part of handling asynchronous code execution.
- It's responsible for managing the execution of callbacks and events, such as those associated with timers, user interactions, and I/O operations.
- The event loop ensures that code execution remains non-blocking, allowing the program to remain responsive.
- The event loop continually checks the call stack for functions to execute and the message queue for pending events or callbacks to process.
- It moves items from the queue to the stack when the stack is empty, ensuring that asynchronous tasks are executed in the order they were received.

What are modules and why do we need them?

- Modules in JavaScript are a way to encapsulate code into separate, reusable, and maintainable files.
- They help organize code by breaking it into smaller, focused units, each with its own scope.
- Modules improve code maintainability, reusability, and reduce the risk of naming conflicts in large applications.
- Modules are crucial for structuring complex applications, allowing developers to work on specific parts of a project independently, and facilitating code sharing and collaboration.

Module Systems in JavaScript:

- **CommonJS:** CommonJS is a module system used in server-side JavaScript (e.g., Node.js). It uses require to import modules and module.exports to export them.

```
// ModuleA.js
module.exports = { message: "Hello from Module A" };

// main.js
const moduleA = require('./ModuleA');
console.log(moduleA.message);
```

- **ES6 Modules:** ES6 introduced the **import** and **export** syntax for working with modules in JavaScript.

Export:

- To make parts of a module accessible outside of the module, we use the export keyword.
- We can export variables, functions, or classes.

Import:

- To use the exported code from a module in another module, we use the import statement.

```
// ModuleA.js
export const message = "Hello from Module A";

// main.js
import { message } from './ModuleA.js';
console.log(message);
```

How does prototypal inheritance work in JavaScript?

- In JavaScript, objects can inherit properties and methods from other objects through a mechanism called **prototypal inheritance.**
- Each object has an internal reference to another object called its **prototype**.
- When you access a property or method on an object, JavaScript first looks for it on the object itself. If it doesn't find it, it searches up the prototype chain until it finds the property or reaches the end of the chain.

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.sayHello = function() {
  console.log(`Hello, I'm ${this.name}`);
};

const dog = new Animal("Fluffy");
dog.sayHello(); // Output: "Hello, I'm Fluffy"
```

What is a prototype chain in JavaScript?

- The **prototype chain** is the series of objects linked through their prototypes. When you access a property or method on an object, JavaScript traverses this chain to find the property or method.
- An object's prototype is the object it inherits from, and that inherited object can have its own prototype, forming a chain of objects. The chain continues until the end of the chain is reached (typically the Object.prototype, which is the root of the chain).

What is a constructor, why do need it and how do we use it?

- **What is a Constructor**: A constructor function is a regular function used to create and initialize objects in JavaScript. It's called with the new keyword to create instances of objects.
- **Why We Need It**: Constructors allow you to create multiple objects with the same structure and behavior. They help encapsulate object creation and property assignment logic.
- **How to Use It**:

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

const john = new Person("John", 30);
```

What are classes in ES6, how do you extend and why do we need it?

- **JS Classes**: In ES6 (ECMAScript 2015), JavaScript introduced a more structured way to define constructor functions using the class keyword.
- Classes provide syntactical sugar over constructor functions, making object-oriented programming in JavaScript more readable and familiar to developers from other languages.
- **Extending Classes**: You can extend a class using the extends keyword to create a subclass.
- Subclasses inherit properties and methods from their parent class.
- **Why We Need Classes**: Classes provide a more structured and organized way to define and work with objects, making code more modular and easier to understand, especially for larger applications.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Dog extends Animal {
  bark() {
    console.log(`${this.name} says woof!`);
  }
}
```

What are static properties and methods in JavaScript?

- **Static Properties** are properties that belong to a class itself rather than to instances of the class.
- They are accessed using the class name, not an instance of the class.
- **Static Methods** are methods that are called on the class itself, not on instances.
- They are often used for utility functions that don't require access to instance-specific data.

```
class MathUtils {
  static add(x, y) {
    return x + y;
  }
}

const sum = MathUtils.add(5, 3); // Static method usage
```

Double or Triple the Word

Write a function named as **doubleOrTripleWord()** which takes a **string** word as an argument and returns the given word back tripled if the string length is even or doubled if the string length is odd when invoked.

Examples:
doubleOrTripleWord("Tech") -> "TechTechTech"
doubleOrTripleWord("Apple") -> "AppleApple"
doubleOrTripleWord("") -> ""
doubleOrTripleWord(" ") -> " "
doubleOrTripleWord("1") -> "11"
doubleOrTripleWord("22") -> "222222"

First and Last Word

Write a function named as **firstlastWord()** which takes a **string** word as an argument and returns the first and last words from the given string when invoked.
NOTE: Return empty string if the given string does not have any word.

Examples:
firstLastWord("Hello World") -> "HelloWorld"
firstLastWord("I like JavaScript") -> "IJavaScript"
firstLastWord("Hello") -> "HelloHello"
firstLastWord("") -> ""
firstLastWord(" ") -> ""

Has Vowel

Write a function named **hasVowel()** which takes a **string** argument and returns **true** if the **string** has a vowel, returns **false** if the **string** doesn't contain any vowel letter.
NOTE: Vowels are = a, e, o, u, i.
NOTE: Ignore upper/lower cases.

Examples:

```
hasVowel("")                -> false
hasVowel("Javascript")      -> true
hasVowel("Tech Global")     -> true
hasVowel("1234")            -> false
hasVowel("ABC")             -> true
```

Start Vowel

Write a function named as **startVowel()** which takes a **string** word as an argument and returns **true** if given **string** starts with a vowel, and **false** otherwise when invoked.
NOTE: Vowel letters: a, e, i o, u, A, E, I, O, U

Examples:

```
startVowel("Hello")     -> false
startVowel("Apple")     -> true
startVowel("orange")    -> true
startVowel("")          -> false
startVowel("    ")      -> false
startVowel("123")       -> false
```

Average of Edges

Write a function named **averageOfEdges()** which takes three **number** arguments and will return average of min and max numbers.

Examples:

```
averageOfEdges(0, 0, 0)   -> 0
averageOfEdges(0, 0, 6)   -> 3
averageOfEdges(-2, -2, 10)   -> 4
averageOfEdges(-3, 15, -3)   -> 6
averageOfEdges(10, 13, 20)   -> 15
```

Swap First and Last Characters

Write a function named **replaceFirstLast()** which takes a **string** argument and returns a new **string** with the first and last characters replaced.
NOTE: If the length is less than 2, return an empty string.
NOTE: Ignore extra spaces before and after the string.

Examples:

```
replaceFirstLast("")                -> ""
replaceFirstLast("Hello")           -> "oellH"
replaceFirstLast("Tech Global")     -> "lech GlobaT"
replaceFirstLast("A")               -> ""
replaceFirstLast("    A      ")     -> ""
```

Swap First and Last Four Characters

Write a function named as **swap4()** which takes a **string** word as an argument and returns the string back with its first and last 4 characters swapped when invoked.
NOTE: Return empty string if the given string does not have 8 or more characters.

Examples:

```
swap4("abc")            -> ""
swap4("JavaScript")     -> "riptScJava"
swap4("TechGlobal")     -> "obalGlTech"
swap4("")               -> ""
swap4("          ")     -> ""
swap4("Apple")          -> ""
```

Swap First and Last Words

Write a function named as **swapFirstLastWord()** which takes a **string** word as an argument and returns the string back with its first and last words swapped when invoked.
NOTE: Return empty string if the given string does not have 2 or more words.

Examples:

```
swapFirstLastWord("Hello World")          -> "World Hello"
swapFirstLastWord("I like JavaScript")    -> "JavaScript like I"
swapFirstLastWord("foo bar foo bar")      -> "bar bar foo foo"
swapFirstLastWord("")                     -> ""
swapFirstLastWord("  ")                   -> ""
swapFirstLastWord("Hello")                -> ""
swapFirstLastWord("Hello   ")             -> ""
```

Count Positive Numbers

Write a function named **countPos()** which takes an **array** of numbers as an argument and returns how many elements are positive when invoked.

Examples:
countPos([-45, 0, 0, 34, 5, 67]) -> 3
countPos([-23, -4, 0, 2, 5, 90, 123]) -> 4
countPos([0, -1, -2, -3]) -> 0

Find Even Numbers

Write a function named as **getEvens()** which takes 2 number arguments and returns all the even numbers as an array stored from smallest even number to greatest even number when invoked.

NOTE: Make your code dynamic that works for any numbers and return empty array if there are no even numbers in the range of given 2 numbers.

Assume you will not be given negative numbers.

Examples:
getEvens(2, 7) -> [2, 4, 6]
getEvens(17, 5) -> [6, 8, 10, 12, 14, 16]
getEvens(4, 4) -> [4]
getEvens(3, 3) -> [ ]

Find Numbers Divisible By 5

Write a function named as getMultipleOf5() which takes 2 number arguments and returns all the numbers divisible by 5 as an array stored from first found match to last found match when invoked.

NOTE: Make your code dynamic that works for any numbers and return empty array if there are no numbers divisible by 5 in the range of given 2 numbers.

Assume you will not be given negative numbers.
Examples:
getMultipleOf5(3, 17) -> [5, 10, 15]
getMultipleOf5(23, 5) -> [20, 15, 10, 5]
getMultipleOf5(5, 5) -> [5]
getMultipleOf5(2, 4) -> [ ]

Count Negative Numbers

Write a function named **countNeg()** which takes an **array** of numbers as an argument and returns how many elements are negative when invoked.

Examples:
countNeg([-45, 0, 0, 34, 5, 67]) -> 1
countNeg([-23, -4, 0, 2, 5, 90, 123]) -> 2
countNeg([0, -1, -2, -3]) -> 3

Count A

Write a function named **countA()** which takes a **string** argument and returns how many A or a there are in the given **string** when invoked.
NOTE: Ignore case sensitivity.

Examples:
countA("TechGlobal is a QA bootcamp") -> 4
countA("QA stands for Quality Assurance") -> 5
countA("Cypress") -> 0

Count Words

Write a function named **countWords()** which takes a **string** argument and returns the total count of words in the given **string** when invoked.
NOTE: Be careful about the extra whitespaces before and after the string.

Examples:
countWords(" Javascript is fun ") -> 3
countWords("Cypress is an UI automation tool. ") -> 6
countWords("1 2 3 4") -> 4

Factorial

Write a function named as **factorial()** which takes a **number** as an argument and returns the factorial of the number when invoked.
NOTE: Mathematically, the factorial of a non-negative integer n is defined as:
n! = n × (n-1) × (n-2) × ... × 2 × 1
Assume you will not be given a negative number.

Examples:
factorial(5) -> 120
factorial(4) -> 24
factorial(0) -> 1
factorial(1) -> 1

Count 3 or Less

Write a function named as **count3OrLess()** which takes a **string** word as an argument and returns the count of the words that has 3 characters or less when invoked.

Examples:

```
count3OrLess("Hello")                -> 0
count3OrLess("Hi John")              -> 1
count3OrLess("JavaScript is fun")    -> 2
count3OrLess("My name is John Doe")  -> 3
count3OrLess("")                     -> 0
```

Remove Extra Spaces

Write a function named as **removeExtraSpaces()** which takes a **string** word as an argument and returns the string back with all extra spaces removed when invoked.

Examples:

```
removeExtraSpaces("Hello")                          -> "Hello"
removeExtraSpaces("     Hello   World  ")           -> "Hello World"
removeExtraSpaces("     JavaScript is      fun")    -> "JavaScript is fun”
removeExtraSpaces("")                               -> ""
```

Middle Number

Write a function named **middleInt()** which takes three **number** arguments and return the middle number.

Examples:

```
middleInt(1, 2, 2)      -> 2
middleInt(5, 5, 8)      -> 5
middleInt(5, 3, 5)      -> 5
middleInt(1, 1, 1)      -> 1
middleInt(-1, 25, 10)   -> 10
```

First Duplicate Element

Write a function named as **firstDuplicate()** which takes an array argument and returns the first duplicated number in the array when invoked.

NOTE: Make your code dynamic that works for any array and return -1 if there are no duplicates in the array. For two elements to be considered as duplicated, value and data types of the elements must be same.

Examples:

```
firstDuplicate([ 3, 7, 10, 0, 3, 10 ])          -> 3
firstDuplicate([ 5, 7, 7, 0, 5, 10 ])           -> 5
firstDuplicate([ 5, '5', 3, 7, 4 ])             -> -1
firstDuplicate([ 123, 'abc', '123', 3, 'abc' ]) -> 'abc'
firstDuplicate([ 1, 2, 3])                      -> -1
firstDuplicate([ 'foo', 'abc', '123', 'bar' ])  -> -1
```

Find All Duplicate Elements

Write a function named as **getDuplicates()** which takes an array argument and returns all the duplicated elements in the array when invoked.

NOTE: Make your code dynamic that works for any array and return empty array if there are no duplicates in the array. For two elements to be considered as duplicated, value and data types of the elements must be same.

Examples:

```
getDuplicates([ 0, -4, -7, 0, 5, 10, 45, -7, 0 ])          -> [ 0, -7 ]
getDuplicates([ 1, 2, 5, 0, 7 ])                           -> [ ]
getDuplicates(['A', 'foo', '12' , 12, 'bar', 'a', 'a', 'foo' ])  -> [ 'foo', 'a' ]
getDuplicates([ 'foo', '12' , 12, 'bar', 'a' ])            -> [ ]
```

Count Vowels

Write a function named as **countVowels()** which takes a **string** word as an argument and returns the count of the vowel letters when invoked.
NOTE: Vowel letters are A,E, O, U, I, a, e, o, u, i

Examples:

```
countVowels("Hello")              -> 2
countVowels("JavaScript is fun")  -> 5
countVowels("")                   -> 0
```

Reverse String Words

Write a function named as **reverseStringWords()** which takes a **string** as an argument and returns string back with each word separately reversed when invoked.

NOTE: Make your code dynamic that works for any string. Make sure you consider extra spaces before and after words in the given string.

Examples:

```
reverseStringWords("Hello World")          -> "olleH dlroW"
reverseStringWords("I like JavaScript")    -> "I ekil tpircSavaJ"
reverseStringWords("Hello")                -> "olleH"
reverseStringWords("")                     -> ""
reverseStringWords(" ")                    -> ""
```

Count Consonants

Write a function named as **countConsonants()** which takes a **string** word as an argument and returns the count of the consonant letters when invoked.
NOTE: A letter that is not vowel is considered as a consonant letter.

Examples:

```
countConsonants("Hello")                   -> 3
countConsonants("Hello World")             -> 8
countConsonants("JavaScript is fun")       -> 12
countConsonants("")                        -> 0
```

Count Multiple Words

Write a function named as **countMultipleWords()** which takes an **array** as an argument and returns the count of the elements that has multiple words when invoked.
NOTE: Be careful about the extra whitespaces before and after the array element.

Examples:

```
countMultipleWords([ "foo", "", "        ", "foo bar", "   foo" ])          -> 1
countMultipleWords([ "foo", "bar", "foobar", "          foobar   " ])      -> 0
countMultipleWords([ "f o o", "b a r", "foo bar", "    foo bar   " ])      -> 4
countMultipleWords([ ])                                                    -> 0
```

FizzBuzz

Write a function named as **fizzBuzz()** which takes 2 number arguments and returns a string composed with below requirements when invoked.

- You need to find all the numbers within the range of given 2 numbers (both inclusive) and store them in a string from smallest to greatest number with a ' | ' separator for each number.
- You will need to convert numbers divisible by 3 to 'Fizz'
- You will need to convert numbers divisible by 5 to 'Buzz'
- You will need to convert numbers divisible by both 3 and 5 to 'FizzBuzz'
- The rest will stay the same.

NOTE: Make your code dynamic that works for any numbers.

Assume you will not be given negative numbers.

Examples:
fizzBuzz(13, 18) -> "13 | 14 | FizzBuzz | 16 | 17 | Fizz"
fizzBuzz(12, 5) -> "Buzz | Fizz | 7 | 8 | Fizz | Buzz | 11 | Fizz"
fizzBuzz(5, 5) -> "Buzz"
fizzBuzz(9, 6) -> "Fizz | 7 | 8 | Fizz"

Palindrome

Write a function named as **isPalindrome()** which takes a **string** word as an argument and returns **true** if the word is palindrome or returns **false** otherwise when invoked.
NOTE: Palindrome: It is a word that is read the same backward as forward
Examples: kayak, civic, madam
NOTE: your function should ignore case sensitivity

Examples:
isPalindrome("Hello") -> false
isPalindrome("Kayak") -> true
isPalindrome("civic") -> true
isPalindrome("abba") -> true
isPalindrome("ab a") -> false
isPalindrome("123454321") -> true
isPalindrome("A") -> true
isPalindrome("") -> true

Prime Number

Write a function named as **isPrime()** which takes a **number** as an argument and returns **true** if the number is prime or returns **false** otherwise when invoked.

NOTE: Mathematically, Prime number is a number that can be divided only by itself and 1. It cannot be divided by any other number. The smallest prime number is 2 and 2 is the only even prime number.

Examples: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31…
NOTE: The smallest prime number is 2 and there is no negative prime numbers.

Examples:
isPrime(5) -> true
isPrime(2) -> true
isPrime(29) -> true
isPrime(-5) -> false
isPrime(0) -> false
isPrime(1) -> false

Add Two Arrays

Write a function named **add()** which takes two **array** of numbers as argument and returns a new **array** with sum of given arrays elements.
NOTE: Be careful about the array sizes as they could be different.

Examples:
add([3, 0, 0, 7, 5, 10], [6, 3, 2]) -> [9, 3, 2, 7, 5, 10]
add([10, 3, 6, 3, 2], [6, 8, 3, 0, 0, 7, 5, 10, 34]) -> [16, 11, 9, 3, 2, 7, 5, 10, 34]
add([-5, 6, -3, 11], [5, -6, 3, -11]) -> [0, 0, 0, 0]

No Elements With A

Write a function named **noA()** which takes an **array** of **strings** as argument and will return a new **array** with all elements starting with "A" or "a" replaced with "###".

Examples:
noA(["javascript", "hello", "123", "xyz"]) -> ["javascript", "hello", "123", "xyz"]
noA(["apple", "123", "ABC", "javascript"]) -> ["###", "123", "###", "javascript"]
noA(["apple", "abc", "ABC", "Alex", "A"]) -> ["###", "###", "###", "###", "###"]

No Elements Divisible By 3 and 5

Write a function named **no3and5()** which takes an **array** of integer **numbers** as argument and will return a new **array** with elements replaced by conditions below.
If element can be divided by 5, replace it with 99
If element can be divided by 3, replace it with 100
If element can be divided by both 3 and 5, replace it with 101

Examples:
no3and5([7, 4, 11, 23, 17]) -> [7, 4, 11, 23, 17]
no3and5([3, 4, 5, 6]) -> [100, 4, 99, 100]
no3and5([10, 11, 12, 13, 14, 15]) -> [99, 11, 100, 13, 14, 101]

No Elements Equals 13

Write a function named **no13()** which takes an **array** of numbers as argument and return a new **array** with all 13s replaced with 0s.

Examples:
no13([1, 2, 3 ,4]) -> [1, 2, 3 ,4]
no13([13, 2, 3]) -> [0, 2, 3]
no13([13, 13, 13 , 13, 13]) -> [0, 0, 0, 0, 0]
no13([]) -> []

Remove Duplicates

Write a function named **removeDuplicates()** which takes an **array** argument and returns a new **array** with all the duplicates removed.

Examples:
removeDuplicates([10, 20, 35, 20, 35, 60, 70, 60]) -> [10, 20, 35, 60, 70]
removeDuplicates([1, 2, 5, 2, 3]) -> [1, 2, 5, 3]
removeDuplicates([0, -1, -2, -2, -1]) -> [0, -1, -2]
removeDuplicates(["abc", "xyz", "123", "ab", "abc", "ABC"]) -> ["abc", "xyz", "123", "ab", "ABC"]
removeDuplicates(["1", "2", "3", "2", "3"]) -> ["1", "2", "3"]

No Digits

Write a function named **noDigit()** which takes a **string** argument and returns a new **string** with all digits removed from the original string.

Examples:

```
noDigit("")                      -> ""
noDigit("Javascript")            -> "Javascript"
noDigit("123Hello")              -> "Hello"
noDigit("123Hello World149")     -> "Hello World"
noDigit("123Tech456Global149")   -> "TechGlobal"
```

No Vowel

Write a function named **noVowel()** which takes a **string** argument and returns a new **string** with all vowels removed from the original string.

Examples:

```
noVowel("TechGlobal")   -> "TchGlbl"
noVowel("AEOxyz")   -> "xyz"
noVowel("Javascript")   -> "Jvscrpt"
noVowel("")   -> ""
noVowel("125$")   -> "125$"
```

Sum Of Digits

Write a function named **sumOfDigits()** which takes a **string** argument and returns sum of all digits from the original **string**.

Examples:

```
sumOfDigits("Javascript")          -> 0
sumOfDigits("John's age is 29")    -> 11
sumOfDigits("$125.0")              -> 8
sumOfDigits("")                    -> 0
```

Array Factorial

Write a function named **arrFactorial()** which takes an **array** of numbers as argument and return the **array** with every number replaced with their factorials.

Examples:

arrFactorial([1, 2, 3 ,4]) -> [1, 2, 6, 24]
arrFactorial([0, 5]) -> [1,120]
arrFactorial([5 , 0, 6]) -> [120, 1, 720]
arrFactorial([]) -> []