1. What is the difference between ‘map()’ and ‘forEach’ methods?

   Map:-

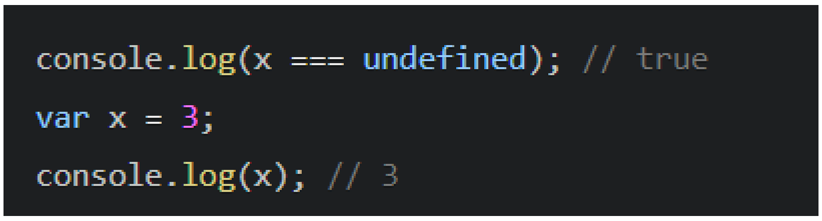
- The map method creates a new array every time with populated results of calling a provided function on every element in the calling array.
- If you want to change, alternative or use data you can pick a map because it will return a new array.
- Return value new array will be created based on your callback function
- Original array is not modified.
- NewArray is not created after end of a method call
- Map method is **CHAINABLE**.

   ForEach:-

- The for each method executes with the provided function once of each array element.
- If you don’t want a returning array you can use forEach or even for loop.
- The return value is undefined.
- Original array is not modified.
- NewArray is not created after the end of a method call
- forEach method is **NOT CHAINABLE.**

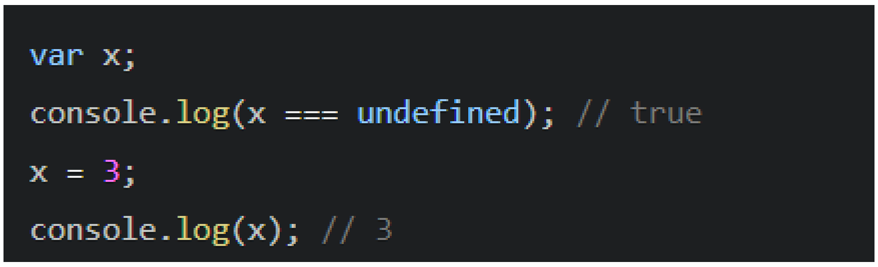
2. What is Hoisting in Javascript?

Hoisting is a Javascript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that no matter where the functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local. Note that hosting only moves the declaration, not the initialization.



```
console.log(x === undefined); // true
var x = 3;
console.log(x); // 3
```

The above code snippet can be visualized in the following way.



```
var x;
console.log(x === undefined); // true
x = 3;
console.log(x); // 3
```

3. What is Set in JavaScript?

Set is another data structure in JavaScript which is similar to Array but the values are unique. It is a collection of elements where each element is stored as a value without any keys.

```
const roadmap = new Set();
roadmap.add('JavaScript');
roadmap.add('JavaScript');

roadmap.add('dynamic');
roadmap.add(1995);

console.log(roadmap.size); // 3, because the value 'JavaScript' is already
console.log(roadmap.has('JavaScript')); // true

roadmap.delete('JavaScript');
console.log(roadmap.has('JavaScript')); // false
console.log(roadmap.size); // 2
```

4. What is the difference between Map and WeakMap in JavaScript?

- The Map object holds key-value pairs and remembers the original insertion order of the keys. Whereas, the WeakMap object is a collection of key/value pairs in which the keys are weakly referenced.
- You can use any data type as a key or value in a Map whereas in WeakMap you can only use objects as keys.
- The WeakMap is not iterable whereas Map is.
- In WekaMap it holds the weak reference to the original object which means if there are no other references to an object stored in the WeakMap, those objects can be garbage collected.

5. What is Javascript?

JavaScript (often abbreviated as JS) is a high-level, versatile, and widely-used programming language primarily known for its role in web development. It enables interactive and dynamic behavior on websites.

6. Difference between ‘Promise.all()’ and ‘Promise.allSettled()’?

- Promise.all() rejects immediately if any of the promises reject whereas Promise.allSettled() waits for all of the promises to settle (either resolve or reject) and then return the result.

Initialize

```
const promise1 = Promise.resolve('Promise 1 resolved');
const promise2 = Promise.reject('Promise 2 rejected');
```

Using ‘Promise.all()’

```
Promise.all([promise1, promise2])
  .then((values) => {
    console.log(values);
  })
  .catch((error) => {
    console.log('An error occurred in Promise.all():', error);
  });

// Output:
// An error occurred in Promise.all(): Promise 2 rejected
```

In the above code, the Promise.all() rejects immediately when any of the promise2 rejects.

Using Promise.allSettled()

```
Promise.allSettled([promise1, promise2]).then((results) => {
  results.forEach((result, index) => {
    if (result.status === 'fulfilled') {
      console.log(
        `Promise ${index + 1} was fulfilled with value:`,
        result.value
      );
    } else {
      console.log(
        `Promise ${index + 1} was rejected with reason:`,
        result.reason
      );
    }
  });
});

// Output:
// Promise 1 was fulfilled with value: Promise 1 resolved
// Promise 2 was rejected with reason: Promise 2 rejected
```

In the above code, the Promise.allSettled() waits for all the promises to settle (either resolve or reject) and then returns the result.

7. How to handle event bubbling in JavaScript?

- Event bubbling is a concept in the Document Object Model (DOM) that describes the way in which events propagate or “bubble up” through the hierarchy of nested elements in DOM.
- When an event, such as mouse click, occurs on a DOM element, the event will be handled by the element first, then its parent element, and so on, until the event reaches the root element. This behavior is called event bubbling.

```
const parent = document.querySelector('.parent');
const child = document.querySelector('.child');

// Scenario of clicking on the child element
parent.addEventListener('click', () => {
  console.log('Handled Last');
});

child.addEventListener('click', () => {
  console.log('Handled First');
});
```

In the above example, when you click on the child element, the event will be handled by the child element first, then its parent element, and so on, to the root element unless you stop the propagation (event.stopPropagation()) of the event.

8. What is event capturing?

Event capturing is a mechanism that propagates from the parent to child element. Event capturing is the first phase of event propagation. In this phase, the event is captured by the outermost element propagated to the inner elements. It is also known as trickling. It is the opposite of event bubbling.

[Bubbling vs Capturing in JavaScript | JavaScript Events Tutorial - YouTube]

9. Explain ‘alert()’, ‘prompt()’, and ‘confirm()’ methods in JavaScript?

alert()

The alert() method displays an alert box with a specified message and an OK button

```
alert('Hello World!');
```

Prompt ()

The prompt() method displays a dialog box that prompts the visitor for input. A prompt box is often used if you want the user to input a value before entering a page. The prompt() method returns the input value if the user clicks OK. If the user clicks Cancel, the method returns null.

```
const name = prompt('What is your name?');
console.log(name);
```

Confirm ()

The confirm() method displays a dialog box with a specified message, along with an OK and a Cancel button. This is often used to confirm or verify something from the user.

```
const result = confirm('Are you sure?');
console.log(result); // true/false
```

10. What are the Scopes in JavaScript?

A scope is a set of variables, objects and functions that you have access to. There are three types of scopes in JavaScript. Which are Global Scope, Function Scope (Local Scope) and Block Scope.

11. What is lexical scope?

- Lexical scoping defines the scope of a variable by the position of that variable declared in the source code. According to the lexical scoping, the scope can be nested and the inner function can access the variable declared in its outer scope.
- Lexical scope is the definition area of an expression. In other words, an item's lexical scope is the place in which the item got created. Another name for lexical scope is static scope.

12. How can you find unique values in an array?

Using 'Set'

```
const roadmaps = ['JavaScript', 'React', 'Node.js', 'Node.js', 'JavaScript']
const uniqueRoadmaps = [...new Set(roadmaps)];
console.log(uniqueRoadmaps); // ['JavaScript', 'React', 'Node.js']
```

Using 'Filter'

```
const roadmaps = ['JavaScript', 'React', 'Node.js', 'Node.js', 'JavaScript']
const uniqueRoadmaps = roadmaps.filter(
  (roadmap, index) => roadmaps.indexOf(roadmap) === index
);
console.log(uniqueRoadmaps); // ['JavaScript', 'React', 'Node.js']
```

Using ‘reduce()’

```
const roadmaps = ['JavaScript', 'React', 'Node.js', 'Node.js', 'JavaScript'];
const uniqueRoadmaps = roadmaps.reduce((unique, roadmap) => {
  return unique.includes(roadmap) ? unique : [...unique, roadmap];
}, []);
console.log(uniqueRoadmaps); // ['JavaScript', 'React', 'Node.js']
```

Using ‘forEach()’

```
const roadmaps = ['JavaScript', 'React', 'Node.js', 'Node.js', 'JavaScript']
const uniqueRoadmaps = [];
roadmaps.forEach((roadmap) => {
  if (!uniqueRoadmaps.includes(roadmap)) {
    uniqueRoadmaps.push(roadmap);
  }
});
console.log(uniqueRoadmaps); // ['JavaScript', 'React', 'Node.js']
```

Using ‘for … of’

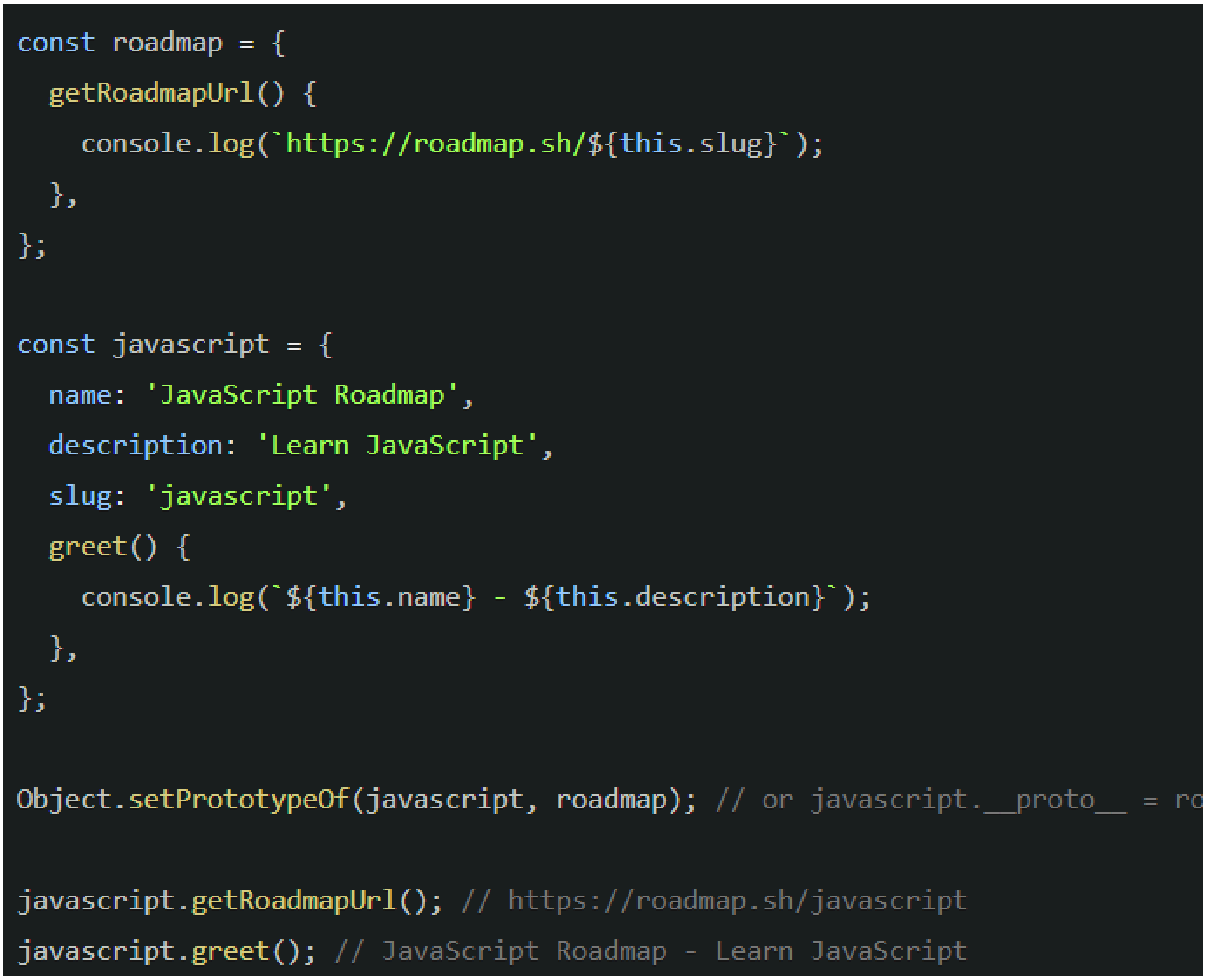
```
const roadmaps = ['JavaScript', 'React', 'Node.js', 'Node.js', 'JavaScript']
const uniqueRoadmaps = [];
for (const roadmap of roadmaps) {
  if (!uniqueRoadmaps.includes(roadmap)) {
    uniqueRoadmaps.push(roadmap);
  }
}
console.log(uniqueRoadmaps); // ['JavaScript', 'React', 'Node.js']
```

13. What is the purpose of the ‘async/await’ in JavaScript?

The async/await, introduced in ES2017, provides a more readable and cleaner way to handle asynchronous operations compared to callbacks and promises. An async function always returns a promise, and within such a function, you can use await to pause execution until a promise settles.

14. What is Prototype Chain in JavaScript?

- The prototype chain in JavaScript refers to the chain of objects linked by their prototypes. When a property or method is accessed on an object, JavaScript first checks the object itself. If it doesn't find it there, it looks up the property or method in the object's prototype. This process continues, moving up the claim from one prototype to the next, until the property or method is found or the end of the chain is reached (typically the prototype of the base object, which is null). The prototype chain is fundamental to JavaScript's prototypal inheritance model, allowing objects to inherit properties and methods from other objects.



```
const roadmap = {
  getRoadmapUrl() {
    console.log(`https://roadmap.sh/${this.slug}`);
  },
};

const javascript = {
  name: 'JavaScript Roadmap',
  description: 'Learn JavaScript',
  slug: 'javascript',
  greet() {
    console.log(`${this.name} - ${this.description}`);
  },
};

Object.setPrototypeOf(javascript, roadmap); // or javascript.__proto__ = ro

javascript.getRoadmapUrl(); // https://roadmap.sh/javascript
javascript.greet(); // JavaScript Roadmap - Learn JavaScript
```

In the above example, the JavaScript object inherits the getRoadmapUrl() method from the roadmap object. This is because the javascript object's prototype is set to the roadmap object using the Object.setPrototypeOf() method. In the javascript object, the getRoadmapUrl() method is not found, so JavaScript looks up the prototype chain and finds the getRoadmapUrl() method in the roadmap object.

15. What is the difference between *var*, *let*, *const* in JavaScript?

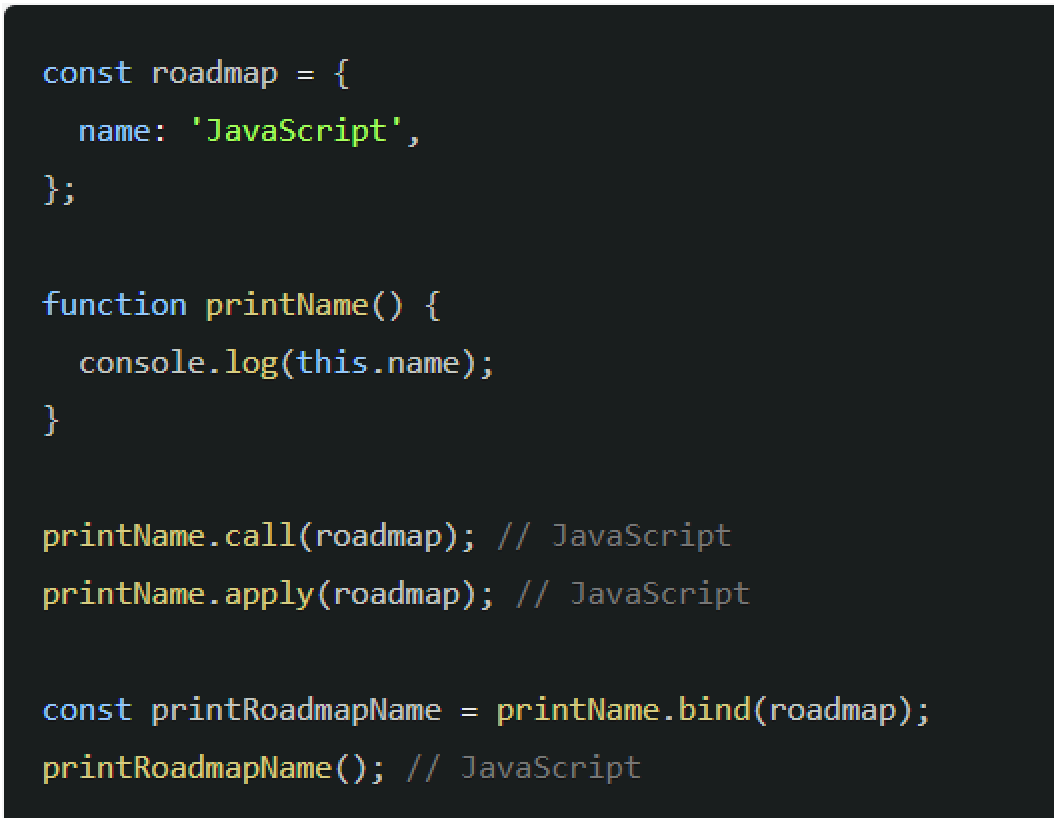
In Javascript , *var* is function-scoped and was traditionally used to declare variables. Let and Const are block-scoped. The key difference between let and const is that let allows for reassignment while const creates read-only references.

16. What is Type Casting?

Type conversion (or typecasting) means transfer of data from one data type to another. Implicit conversion happens when the compiler(for compiled languages) or runtime (for script languages like JavaScript) automatically converts data types.

17. What are Explicit binding in JavaScript?

Explicit binding is a way to explicitly state what the *this* keyword is going to be bound to using *call*, *apply* or *bind* methods of a function.



```
const roadmap = {
  name: 'JavaScript',
};

function printName() {
  console.log(this.name);
}

printName.call(roadmap); // JavaScript
printName.apply(roadmap); // JavaScript

const printRoadmapName = printName.bind(roadmap);
printRoadmapName(); // JavaScript
```

In the above example, the *this* keyword inside the *printName()* function is explicitly bound to the *roadmap* object using *call*, *apply* or *bind* methods.

18. What are the different ways to declare a variable in JavaScript?

There are three ways to declare a variable in JavaScript *var*, *let*, and *const*.

19. How to use the `do … while` loop in JavaScript?

The do … while statement creates a loop that executes a block of code once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
let i = 0;

do {
  console.log(i);
  i++;
} while (i < 3);

// 0
// 1
// 2
```

20. How to run a piece of code only once after a specific time?

To run a piece of code after a certain time, you can use the *setTimeout* function in JavaScript. It accepts a function and a time interval in milliseconds. It returns a unique id which you can use to clear the timeout using *clearTimeout* function.

```
const timeoutId = setTimeout(() => {
  console.log('Hello World');
}, 1000);

// Output:
// Hello World
```

In the above code, the *setTImeout* function runs the callback function after 1000 milliseconds (1 second) and prints (1 second) and prints *Hello World* to the console. It returns a unique id which you can use to clear the timeout using *clearTimeout* function.

```
clearTimeout(timeoutId);
```

21. What is IIFE in JavaScript?

The IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function () {
  console.log('Hello Roadmap!');
})();
```

The IIFE is frequently used to create a new scope to avoid variable hoisting from within blocks.

```
(function () {
  var roadmap = 'JavaScript';
  console.log(roadmap);
})();

console.log(roadmap); // ReferenceError: name is not defined
```

22. What is the ‘preventDefault()’ method in JavaScript?

The event.preventDefault() method is used to prevent the default action of an event. For example, when you click on a link, the default action is to navigate to the link’s URL. But, if you want to prevent the navigation, you can use the *event.preventDefault()* method.

```
const link = document.querySelector('a');

link.addEventListener('click', (event) => {
  event.preventDefault();
  console.log('Clicked on link!');
});
```

23. Is JavaScript a compiled or interpreted language?

JavaScript is an interpreted language. This means that the JavaScript code is not compiled before it is executed. Instead, the JavaScript engine interprets the code at runtime.

24. What are JavaScript promises?

A Promise in JavaScript represents a value that may not be available yet but will be at some time. Promises provide a way to handle asynchronous operations, offering methods like *.then()* and *.catch()* to register callbacks for success and failure.

25. How does Event Loop work in JavaScript?

The Event loop has two main components. The Call stack and the Callback queue.

Call Stack

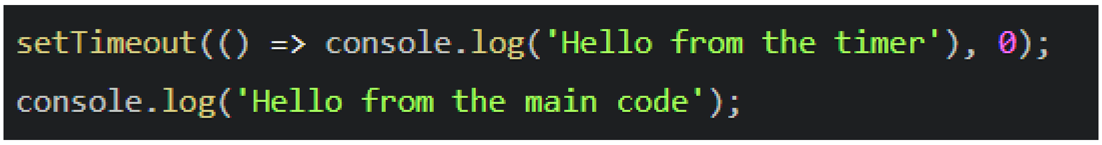
The call stack is a data structure that stores the tasks that need to be executed. It is a LIFO (Last In, First Out) data structure, which means that the last task added to the Call Stack will be the first one to be executed.

Callback Queue

The Call back queue is a data structure that stores the tasks that have been completed and are ready to be executed. It is a FIFO (First In, First Out) data structure, which means that the first task that was added to the Callback queue will be the first one to be executed.

Event Loop's Workflow

- Executes tasks from the Call Stack.
- For an asynchronous task, such as a timer, it runs in the background. JavaScript proceeds to the next task without waiting.
- When the asynchronous task concludes, its callback function is added to the Callback Queue.
- If the Call Stack is empty and there are tasks in the Callback Queue, the Event Loop transfers the first task from the Queue to the Call Stack for execution.



```javascript
setTimeout(() => console.log('Hello from the timer'), 0);
console.log('Hello from the main code');
```

- setTimeout is processed, and because it's asynchronous, its callback is placed in the Callback Queue.
- The next line, 'console.log("Hello from the main code")', is logged immediately.
- Although the timer duration is 0 milliseconds, its callback has to wait until the Call Stack is empty. After the main code logs, the callback is moved from the Callback Queue to the Call Stack and executed.
- The result is "Hello from the main code" being logged before "Hello from the timer"

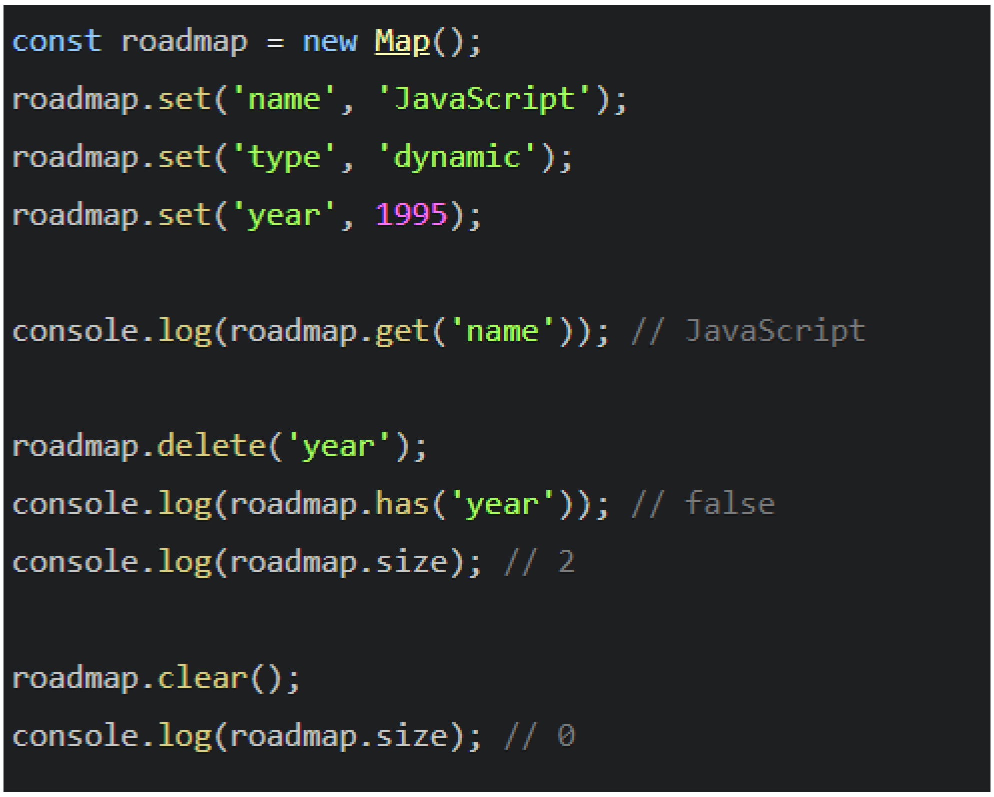
Event loop is just a guardian who keeps good communication with the call stack and callback queue. It checks if the call back is free, then lets you know the call back queue. Then the call back queue passes the callback function to the call stack to be executed. When all the call back functions are executed, the call stack is out and global context is free.

The event loop is one the most important aspects to understand in JavaScript. It is the mechanism that allows JavaScript to perform non-blocking operations. It is the reason why we can use asynchronous code in JavaScript. The Event Loop is a loop that constantly checks if there are any tasks that need to be executed. If there are, it will execute them. If there are no tasks to execute, it will wait for new tasks to arrive.

26. What is a Map in JavaScript?

Map is another data structure in JavaScript which is similar to *Object* but the key can be of any type. It is a collection of elements where each element is stored as a Key value pair. It is also known as a Hash table or a dictionary.

The key can be of any type but the value can be of any type. The key is unique and immutable, whereas the value can be mutable or immutable.

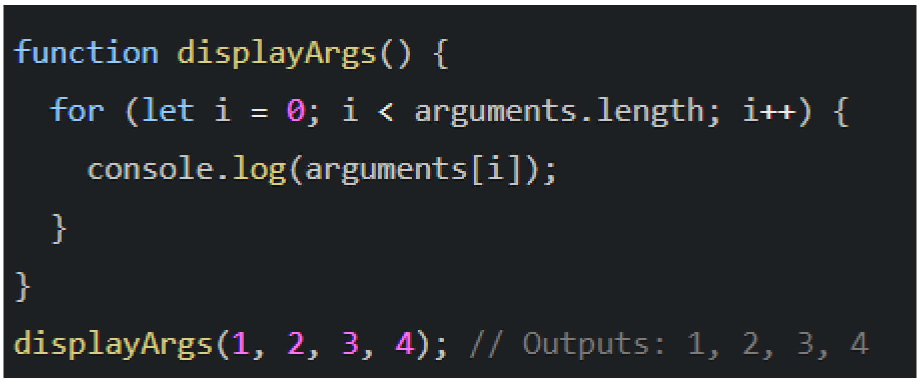


```
const roadmap = new Map();
roadmap.set('name', 'JavaScript');
roadmap.set('type', 'dynamic');
roadmap.set('year', 1995);

console.log(roadmap.get('name')); // JavaScript

roadmap.delete('year');
console.log(roadmap.has('year')); // false
console.log(roadmap.size); // 2

roadmap.clear();
console.log(roadmap.size); // 0
```

27. How to accept a variable number of arguments in a JavaScript function?

In JavaScript, you can accept a variable number of arguments in a function using the arguments object or the rest parameter (. . .).
Using the `arguments` object:

The arguments is an array-like object that holds all of the passed arguments. They are only available inside the function body.



```
function displayArgs() {
  for (let i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}
displayArgs(1, 2, 3, 4); // Outputs: 1, 2, 3, 4
```

Using the rest parameter

The rest parameter allows you to represent an indefinite number of arguments as an array.

```
function displayArgs(...args) {
  args.forEach((arg) => console.log(arg));
}
displayArgs(1, 2, 3, 4); // Outputs: 1, 2, 3, 4
```

The rest parameter (..args in the example) is generally more modern and flexible, and it provides an actual array, unlike the array-like arguments object.

28. How to create an Infinite Loop in JavaScript?

You can use the *while* or *for* loop to create an infinite loop.
While Loop

To create an infinite loop with the *while* loop, we can use the *true* keyword as the condition.

```
while (true) {
  // do something
}
```

For Loop

To create an infinite loop with the for loop, we can use the true keyword as the condition.

```
for (let i = 0; true; i++) {
  // do something
}
```

29. What is the difference between `null` and `undefined` ?

- The null is an assignment value. It can be assigned to a variable as a representation of no value.
- But the undefined is a primitive value that represents the absence of a value, or a variable that has not been assigned a value.

30. What are Heap and Stack in JavaScript?

The heap and stack in JavaScript Engine are two different data structures that store data in different ways.
Stack

The Stack is a small, organized region of memory. It is where primitive values, function calls, and local variables are stored. It follows a “Last In, First Out(LIFO)” order, meaning that the last item added to the stack is the first one to be removed. Each function invocation creates a new stack frame, which contains the function’s local variables, return address, and other contextual data.

Heap

The Heap is a large, mostly unstructured region of memory. It is where *objects*, *arrays*, and *functions* are stored.   Variables from the Stack (e.g., in functions) point to locations in the Heap for these dynamically allocated structures.

When you declare a primitive type (like a number or boolean), it's usually managed in the stack. But when you create an object, array, or function, it's stored in the heap, and the stack will hold a reference to that location in the heap.

```
name = 'JavaScript'; // Stored on the stack
roadmap = { name: 'JS' }; // `roadmap` reference on the stack, actual object
```

In the code above, the primitive value JavaScript for variable name is directly stored on the stack. For the object assigned to the roadmap, its actual data resides in the heap, and the reference to this data (a memory address pointer) is held on the stack.

31. Difference between `appendChild()` and `insertBefore()`?

You can add a new element to the DOM using *appendChild* OR *insertBefore* method.

appendChild

The appendChild method adds a new element as the last child of the specified parent element.

```
const roadmapWrapper = document.querySelector('.roadmap-wrapper');

const roadmap = document.createElement('div');
roadmap.id = 'javascript-roadmap';

roadmapWrapper.appendChild(roadmapTitle);
```

In the above example, the roadmap element is added as the last child of the roadmapWrapper element.

insertBefore

The insertBefore method adds a new element before the specified child element.

```
const roadmapWrapper = document.querySelector('.roadmap-wrapper');

const roadmap = document.createElement('div');
roadmap.id = 'javascript-roadmap';

const roadmapTitle = document.querySelector('#roadmap-title');
roadmapWrapper.insertBefore(roadmap, roadmapTitle);
```

In the above example, the *roadmap* element is added before the *roadmapTitle* element.

32. How to remove an Element from DOM?

To remove a DOM element, you can use the remove or removeChild method of the Node interface.

```
const roadmapWrapper = document.querySelector('.roadmap-wrapper');
const roadmapTitle = document.querySelector('#roadmap-title');

roadmapWrapper.removeChild(roadmapTitle);
roadmapWrapper.remove();
```

33. How to scroll to the top of the page using JavaScript?

In order to scroll to the top of the page, we can use the *scrollTo* method.

```
window.scrollTo(0, 0);
```

34. **Does the arrow function have their own `this`?**

No, arrow functions do not have their own *this*. Instead, they inherit the *this* of the enclosing lexical scope.

35. What is the difference between the arrow function and regular function?
The main differences are,

- Syntax
- Arguments binding
- Use of this keyword
- Using a new keyword
- No duplicate named parameters

Syntax :- A developer can get the same results as regular functions by writing a few lines of code using arrow functions

Arguments binding:- Arrow functions do not have arguments binding.

Use of this keyword:- Unlike Regular functions, arrow functions does not have their own “this” keyword. The value of this inside an arrow function remains the same throughout the lifecycle of the function and is always bound to the value of this in the closest non-arrow parent function.

Using a new keyword:- Regular functions created using function declarations or expressions are constructible and callable. Regular functions are constructible; they can be called using the new keyword. However the arrow functions are only callable and not constructible , ie, arrow functions can never be used as constructor functions.

No duplicate named Parameters:- Arrow functions can never have duplicate named parameters, whether in strict or non-strict mode.
36. How to use `reduce()` method?

You can use the *reduce()* method to reduce an array to a single value. The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

```
array.reduce((accumulator, currentValue) => {
  // ...
}, initialValue);
```

Example, you can use the *reduce()* method to sum all the numbers in an array.

```
const numbers = [1, 2, 3, 4, 5, 6];

const sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0);

console.log(numbers); // [1, 2, 3, 4, 5, 6]
console.log(sum); // 21
```

37. Is Java and JavaScript the same?

No, Java and JavaScript are distinct languages. Their similarity in name is coincidental, much like car and carpet. Java is often used for backend and mobile apps, while JavaScript powers web interactivity and backend.

38. What is the spread operator in JavaScript?

The spread operator in JavaScript is represented by three dots (...). It allows the elements of an array or properties of an object to be expanded or “spread” into individual elements or properties. This can be useful in various contexts, such as when passing elements as function arguments, cloning arrays and objects, or merging arrays and objects.

```
const roadmaps = ['JavaScript', 'React', 'Node.js'];
const bestPractices = ['AWS', 'API Security'];

const resources = [...roadmaps, ...bestPractices];
console.log(resources); // ['JavaScript', 'React', 'Node.js', 'AWS', 'API Se
```

```
const roadmap = {
  name: 'JavaScript',
  type: 'dynamic',
};

const roadmapClone = { ...roadmap }; // shallow copy
console.log(roadmapClone); // { name: 'JavaScript', type: 'dynamic' }
```

39. How to debug JavaScript code?

Debugging JavaScript code can be achieved through various methods and tools. Here's a basic guide.

Console Logging
You can use console.log(), console.warn(), console.error(), etc to print values, variables, or messages to the browser's developer console.

```
console.log('Value of x:', x);
```

Browser Developer Tools

Most modern browsers come equipped with developer tools. You can access these tools pressing *F12* or right-clicking on the web page and selecting *Inspect* or *Inspect Element.*

- Sources Tab - Allows you to see the loaded scripts, set breakpoints, and step through the code
- Console Tab - Displays console outputs and allows for interactive JavaScript execution.
- Network Tab - Helps in checking network requests and responses.

Setting Breakpoints

In the sources tab of the browser's developer tools, you can click on a line number to set a breakpoint. The code execution will pause at this line, allowing you to inspect variables, the call stack, and continue step-by-step.

Debugger Statement

Inserting the debugger; statement in your code will act as a breakpoint when the browser developer tools are open. Execution will pause at the debugger; line.

```
function myFunction() {
  debugger; // Execution will pause here when dev tools are open
  // ... rest of the code
}
```

Call Stack and Scope

In the developer tools, when pause on a breakpoint or debugger; statement, you can inspect the call stack to see the sequence of function calls. The scope panel will show you the values of local and global variables.

Remember, debugging is an iterative process. It often involves setting breakpoints, checking variables, adjusting code, and re-running to ensure correctness.

40. What is the difference between `==` and `===` ?

The == equity operator converts the operands if they are not of the same type, then applies strict comparison. The === strict equality operator only considers values equal that have the same type.

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

41. What is ternary operator in JavaScript?

The ternary operator is a conditional operator that takes three operands. It is frequently used as a shortcut for the if statement.

```
console.log(condition ? true : false);
```

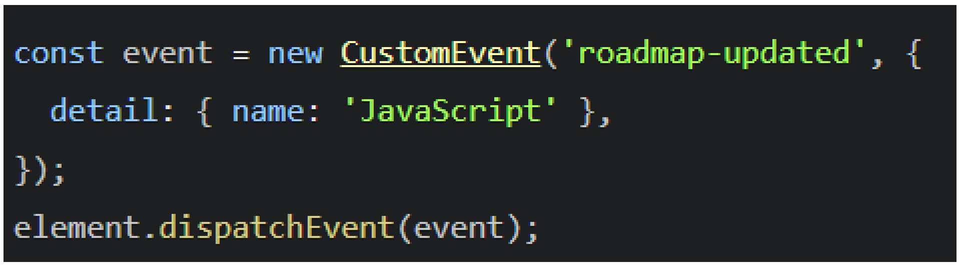
42. What is DOM?

The document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects.

43. How to implement your own custom Event in JavaScript?

You can use the CustomEvent constructor to create a custom event. The CustomEvent constructor accepts two arguments: the event name and an optional object that specifies the event options. And you can use the dispatchEvent method to dispatch the custom event on the target element/document.

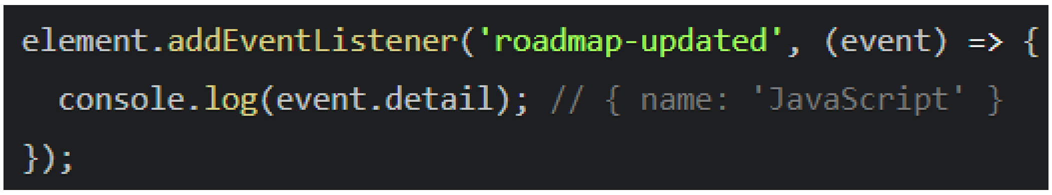
Creating Custom Events



```
const event = new CustomEvent('roadmap-updated', {
  detail: { name: 'JavaScript' },
});
element.dispatchEvent(event);
```

Listening Custom Events

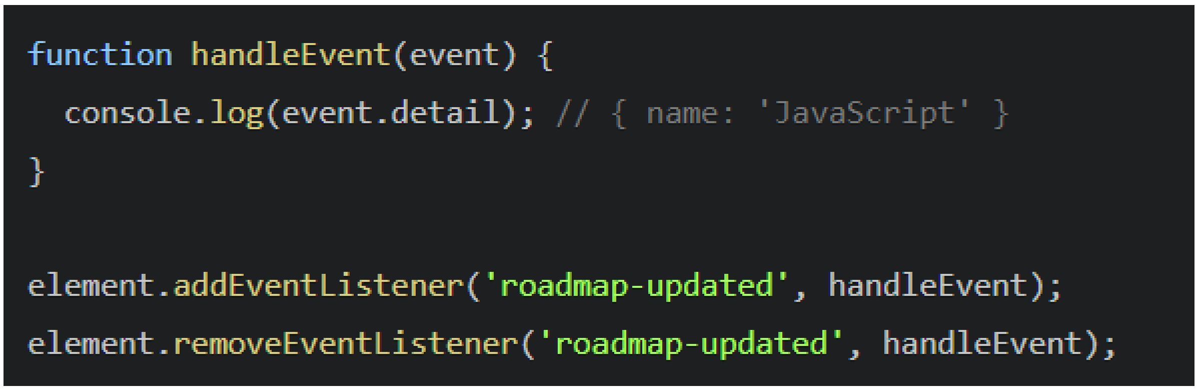
You can listen for custom events using the addEventListener method. The addEventListener method accepts the event and a callback function that is called when the event is dispatched.



```
element.addEventListener('roadmap-updated', (event) => {
  console.log(event.detail); // { name: 'JavaScript' }
});
```

Removing Event Listeners

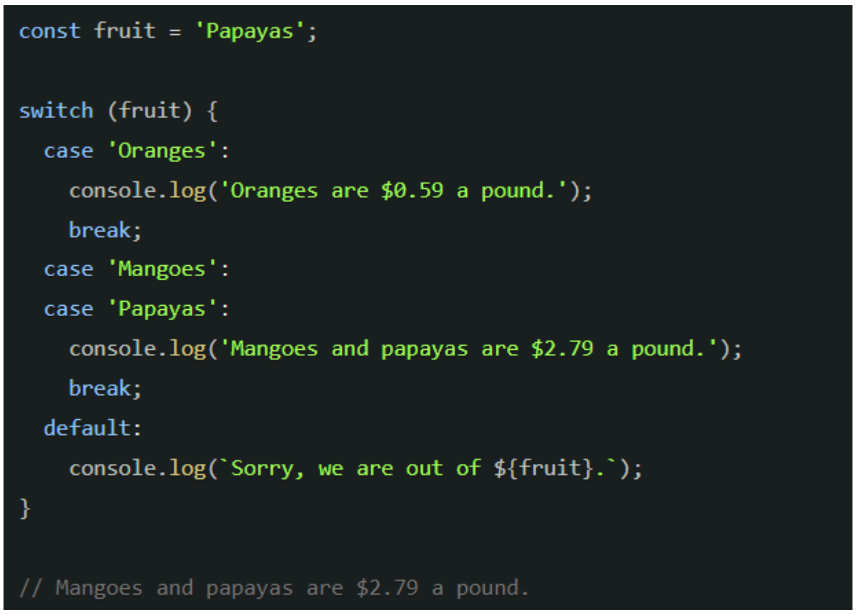
You can remove event listeners using the removeEventListener method. The removeEventListener method accepts the event name and the callback function that was used to add the event listener.



```
function handleEvent(event) {
  console.log(event.detail); // { name: 'JavaScript' }
}

element.addEventListener('roadmap-updated', handleEvent);
element.removeEventListener('roadmap-updated', handleEvent);
```

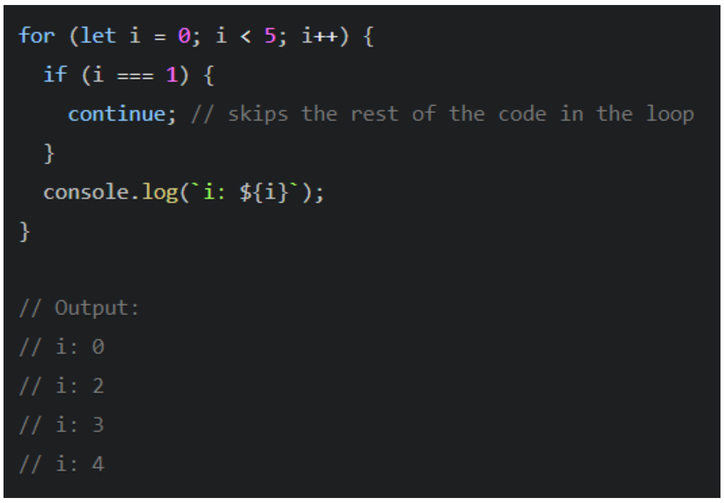
44. Switch case statement in JavaScript?

The switch statement evaluates an expression, matching the expression’s value to a *case* clause, and executes statements associated with that *case*, as well as statements in cases that follow the matching case.

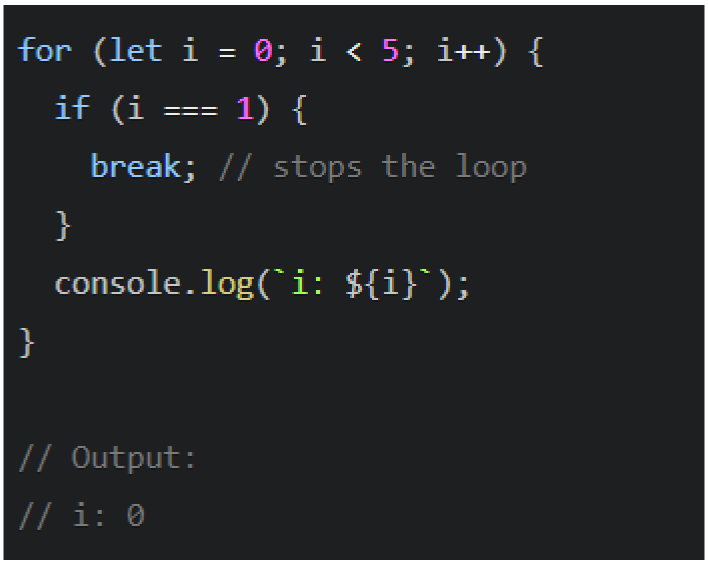


```javascript
const fruit = 'Papayas';

switch (fruit) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Mangoes':
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
    break;
  default:
    console.log(`Sorry, we are out of ${fruit}.`);
}

// Mangoes and papayas are $2.79 a pound.
```

45. Uses of `break` and `continue` statements in JavaScript?
You can use break and continue in loops to alter the flow of the loop. Break will stop the loop from continuing, and continue will skip the current iteration and continue the loop.



```javascript
for (let i = 0; i < 5; i++) {
  if (i === 1) {
    continue; // skips the rest of the code in the loop
  }
  console.log(`i: ${i}`);
}

// Output:
// i: 0
// i: 2
// i: 3
// i: 4
```



```javascript
for (let i = 0; i < 5; i++) {
  if (i === 1) {
    break; // stops the loop
  }
  console.log(`i: ${i}`);
}

// Output:
// i: 0
```

46. Does the `forEach()` method return a new array?

No, the `forEach` method does not return a new array. It simply calls a provided function on each element in the array.

```
const roadmaps = ['JavaScript', 'React', 'Node.js'];


roadmaps.forEach((roadmap) => {
  console.log(roadmap);
});
```

47. What is an increment operator in JavaScript?

As the name says, the increment operator increases the value of a variable by 1. There are two types of increment operators: *pre-increment* and *post-increment*.

Pre-Increment

The pre-increment operator increases the value of a variable by 1 and then returns the value. For example,

```
let x = 1;
console.log(++x); // 2
console.log(x); // 2
```

Post-Increment

The post-increment operator returns the value of a variable and then increases the value by 1. For Example,

```
let x = 1;
console.log(x++); // 1
console.log(x); // 2
```

48. How to get viewport dimensions in JavaScript?

You can use window.innerWidth and window.innerHeight to get the viewport dimensions.

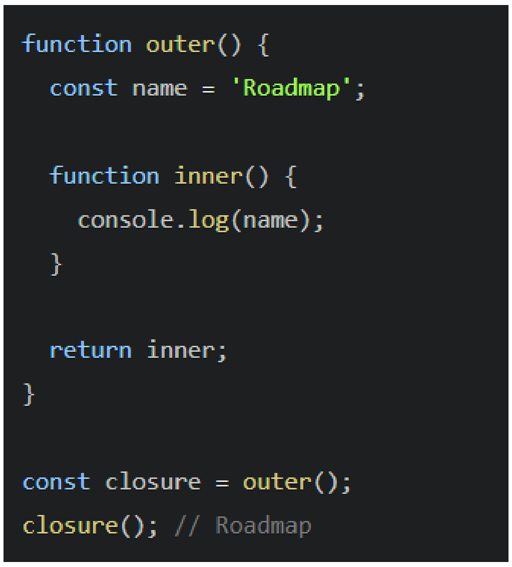
49. How to use the `finally` block in Promise?

The finally block will be executed when the promise is resolved or rejected.

```
promise
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.log(error.message);
  })
  .finally(() => {
    console.log('Finally Promise has settled');
  });
```

50. What is a closure in JavaScript?

A closure is a function that has access to its outer function scope even after the outer function has returned. This means a closure can remember and access variables and arguments of its outer function even after the function has finished.



```
function outer() {
  const name = 'Roadmap';

  function inner() {
    console.log(name);
  }

  return inner;
}

const closure = outer();
closure(); // Roadmap
```

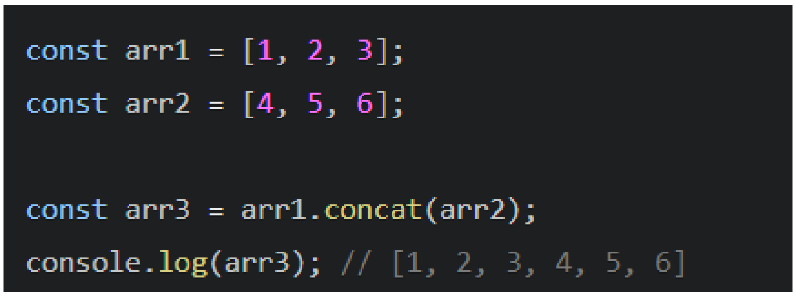
In the above example, the inner function has access to the name variable of the outer function even after the outer function has returned. Therefore, the inner function forms a closure.

51. Can you merge multiple arrays in JavaScript?

Yes, you can merge multiple arrays into one array using the concat() method, or the spread operator …

concat()

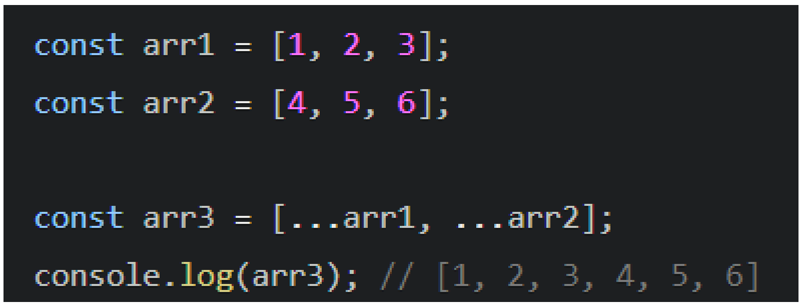
The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.



```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

const arr3 = arr1.concat(arr2);
console.log(arr3); // [1, 2, 3, 4, 5, 6]
```

Spread Operator

The Spread operator … used to expand an iterable object into the list of arguments.



```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

const arr3 = [...arr1, ...arr2];
console.log(arr3); // [1, 2, 3, 4, 5, 6]
```

52. How to select DOM elements using `querySelector()` and `querySelectorAll()` ?

For selecting elements in the DOM, the querySelector and querySelectorAll methods are the most commonly used. They are both methods of the document object, and they both accepts a CSS selector as an argument.

querySelector

The querySelector method returns the first element that matches the specified selector. If no matches are found, it returns null.

```
const roadmapWrapper = document.querySelector('.roadmap-wrapper');
const roadmapTitle = document.querySelector('#roadmap-title');
```

querySelectorAll

The querySelectorAll method returns a NodeList of all the elements that match the specified selector. If no matches are found, it returns an empty NodeList.

```
const roadmapItems = document.querySelectorAll('.roadmap-item');
```

53. How to create a new Element in DOM?

To create a new DOM element, you can use the document.createElement method. It accepts a tag name as an argument and returns a new element with specified tag name. You can set attributes to the element.

```
const div = document.createElement('div');

div.id = 'roadmap-wrapper';
div.setAttribute('data-id', 'javascript');
console.log(div); // <div id="roadmap-wrapper" data-id="javascript"></div>
```

54. How to make an Object immutable in JavaScript?

To make an object immutable, you can use the Object.freeze() method. It prevents the modification of existing property values and prevents the addition of new properties.

```
const roadmap = {
  name: 'JavaScript',
};

Object.freeze(roadmap);

roadmap.name = 'JavaScript Roadmap'; // throws an error in strict mode
console.log(roadmap.name); // JavaScript
```

55. Difference between `defer` and `async` attributes in JavaScript?

The main difference between defer and async is the order of execution.

Defer Attribute

A <script> element with a defer attribute, it will continue to load the HTML page and render it while the script is being downloaded. The script is executed after the HTML page has been completely parsed. Defer scripts maintain their order in the document.

```
<script defer src="script1.js"></script>
<script defer src="script2.js"></script>
```

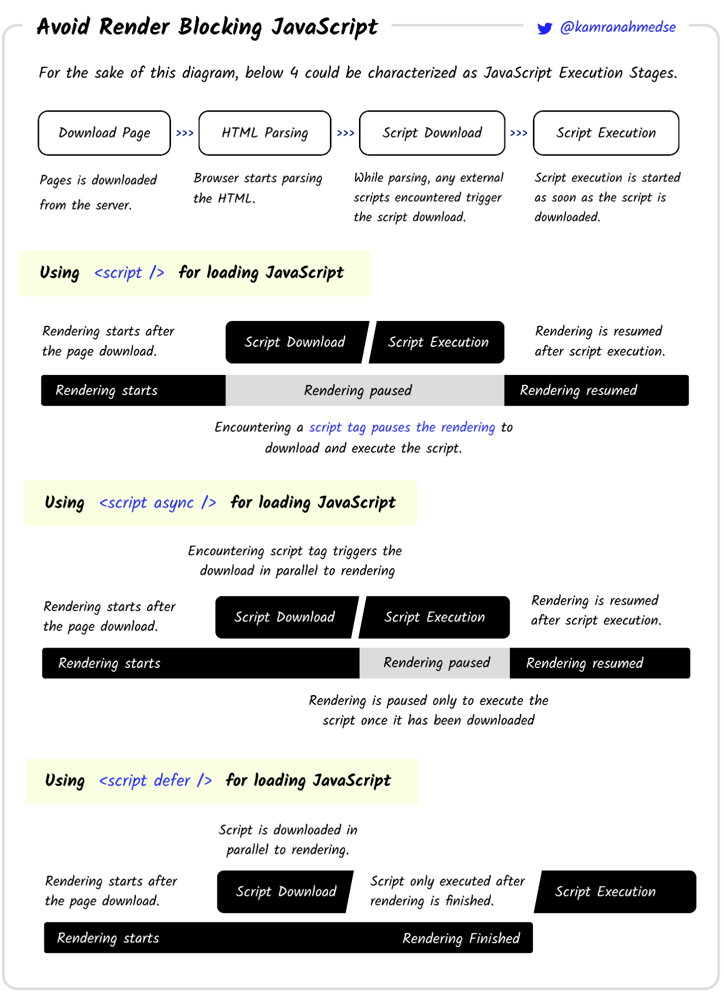
In the example above, script1.js will be executed before script2.js. The browser will download both scripts in parallel, but the script1.js will be executed after the HTML page has been parsed and script2.js will be executed after the script1.js has been executed.

Async attribute

On the other hand, A <script> element with an async attribute, it will pause the HTML parser and execute the script immediately after it has been downloaded. The HTML parsing will resume after the script has been executed.

```
<script async src="script1.js"></script>
<script async src="script2.js"></script>
```

In the example above, the browser will download both scripts in parallel, and execute them as soon as they are downloaded. The order of execution is not guaranteed.

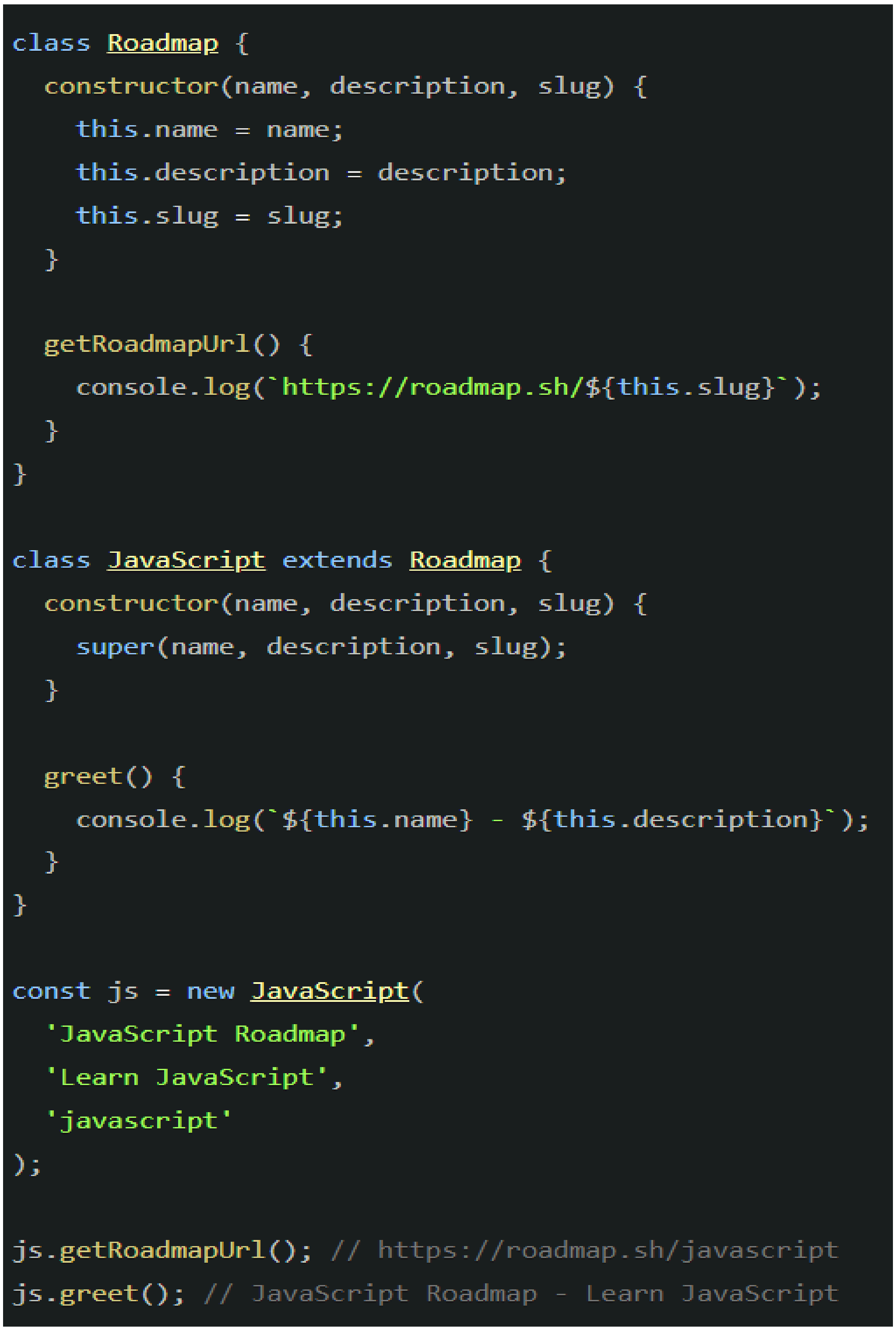

Avoid Render Blocking JavaScript
@kamranahmedse
For the sake of this diagram, below 4 could be characterized as JavaScript Execution Stages.
Download Page
HTML Parsing
Script Download
Script Execution
Pages is downloaded from the server.
Browser starts parsing the HTML.
While parsing, any external scripts encountered trigger the script download.
Script execution is started as soon as the script is downloaded.
Using <script /> for loading JavaScript
Rendering starts after the page download.
Script Download
Script Execution
Rendering is resumed after script execution.
Rendering starts
Rendering paused
Rendering resumed
Encountering a script tag pauses the rendering to download and execute the script.
Using <script async /> for loading JavaScript
Encountering script tag triggers the download in parallel to rendering
Rendering starts after the page download.
Script Download
Script Execution
Rendering is resumed after script execution.
Rendering starts
Rendering paused
Rendering resumed
Rendering is paused only to execute the script once it has been downloaded
Using <script defer /> for loading JavaScript
Script is downloaded in parallel to rendering.
Rendering starts after the page download.
Script Download
Script only executed after rendering is finished.
Script Execution
Rendering starts
Rendering Finished

56. What is the difference between `map()` and `reduce()` methods?

The map() method creates a new array with the results of calling a provided function on every element in the calling array. Whereas, the reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

57. What is Inheritance in JavaScript?

Inheritance is a way to create a new class from an existing class. The new Class inherits all the properties and methods from the existing class. The new class is called the child class and the existing class is called the parent class.



```javascript
class Roadmap {
  constructor(name, description, slug) {
    this.name = name;
    this.description = description;
    this.slug = slug;
  }

  getRoadmapUrl() {
    console.log(`https://roadmap.sh/${this.slug}`);
  }
}

class JavaScript extends Roadmap {
  constructor(name, description, slug) {
    super(name, description, slug);
  }

  greet() {
    console.log(`${this.name} - ${this.description}`);
  }
}

const js = new JavaScript(
  'JavaScript Roadmap',
  'Learn JavaScript',
  'javascript'
);

js.getRoadmapUrl(); // https://roadmap.sh/javascript
js.greet(); // JavaScript Roadmap - Learn JavaScript
```

58. Is it possible to run JavaScript outside the browser?

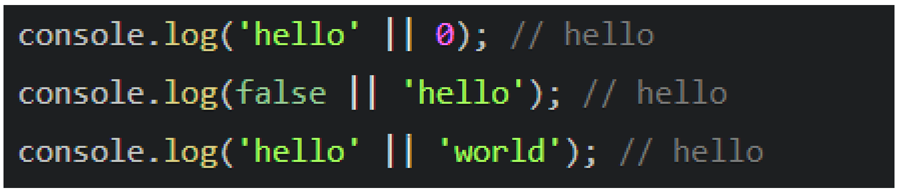
Yes, it is possible to run JavaScript outside the browser. There are several ways to run JavaScript outside the browser. You can use Node.js, Deno, Bun or any other JavaScript runtime environment.

59. What are the Logical Operators in JavaScript?

There are four logical operators in JavaScript. || (OR), && (AND), !(NOT), and ?? (Nullish Coalescing). They can be used with boolean values, or with non-boolean values.

OR (||)

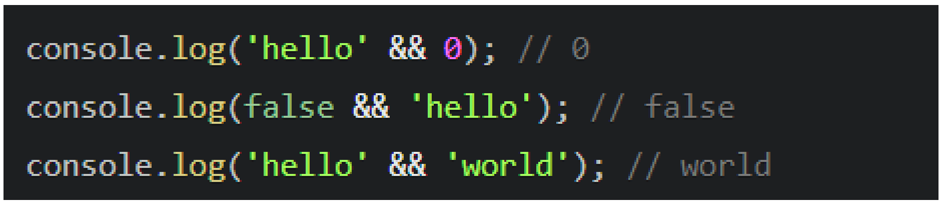
The OR operator (||) returns the first truthy value, or the last value if none are truthy.



```
console.log('hello' || 0); // hello
console.log(false || 'hello'); // hello
console.log('hello' || 'world'); // hello
```

AND (&&)

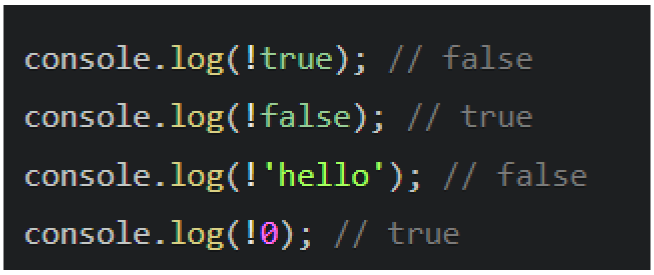
The AND (&&) operator aka logical conjunction returns the first falsy value, or the last value if none are falsy.



```
console.log('hello' && 0); // 0
console.log(false && 'hello'); // false
console.log('hello' && 'world'); // world
```

NOT (!)

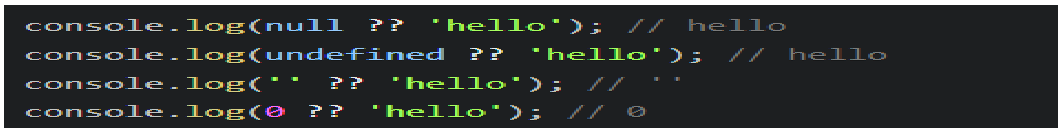
It simply inverts the boolean value of its operand



```
console.log(!true); // false
console.log(!false); // true
console.log(!'hello'); // false
console.log(!0); // true
```

Nullish Coalescing (??)

The Nullish Coalescing Operator (??) returns the right operand if the left one is null or undefined, otherwise, it returns the left operand. It’s useful for setting default values without considering falsy values like 0 or ‘’ as absent.



```
console.log(null ?? 'hello'); // hello
console.log(undefined ?? 'hello'); // hello
console.log('' ?? 'hello'); // ''
console.log(0 ?? 'hello'); // 0
```

60. What are Labeled Statements in JavaScript?

JavaScript labeled statements are used to prefix a label to an identifier. It can be used with break and continue statements to control the flow more precisely.

```
loop1: for (let i = 0; i < 5; i++) {
  if (i === 1) {
    continue loop1; // skips the rest of the code in the loop1
  }
  console.log(`i: ${i}`);
}
// Output:
// i: 0
// i: 2
// i: 3
// i: 4
```

61. Are references copied in JavaScript?

No references are not copied in JavaScript. When you assign an object to a variable, the variable will contain a reference to the object. If you assign the variable to another variable, the second variable will also contain a reference to the object. If you change the object of one of the variables, the change will be visible using the other variable.

62. What is Comma Operator in JavaScript?

The Comma Operator , evaluates each of its operands (from left to right) and returns the value of the last operand.

```
let x = 1;
x = (x++, x);

console.log(x); // 2
```

63. How to define multiline strings in JavaScript?

In order to define multiline strings in JavaScript, you need to use template literals. Template literals are enclosed by the backtick (``) character instead of double or single quotes. Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (`${expression}`).

64. How to measure dimensions of an Element?

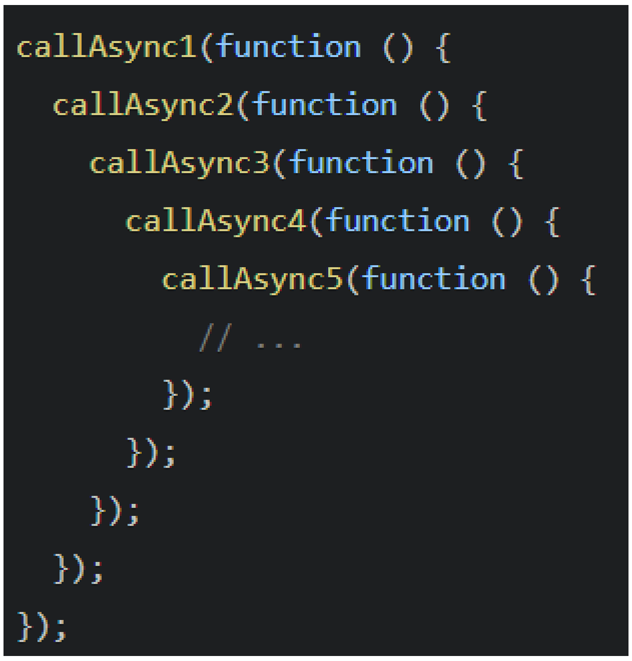
You can use the getBoundingClientRect method to get the dimensions of an element.

```
const roadmapWrapper = document.querySelector('.roadmap-wrapper');
const dimensions = roadmapWrapper.getBoundingClientRect();

console.log(dimensions); // DOMRect { x: 8, y: 8, width: 784, height: 784,
```

65. What is callback hell in JavaScript?

Callback hell, often referred to as Pyramid of Doom, describes a situation in JavaScript where multiple nested callback become difficult to manage, leading to unreadable and unmaintainable code. It often arises when performing multiple asynchronous operations that depend on the completion of previous operations. The code starts to take on a pyramidal shape due to the nesting.

Example of callback hell

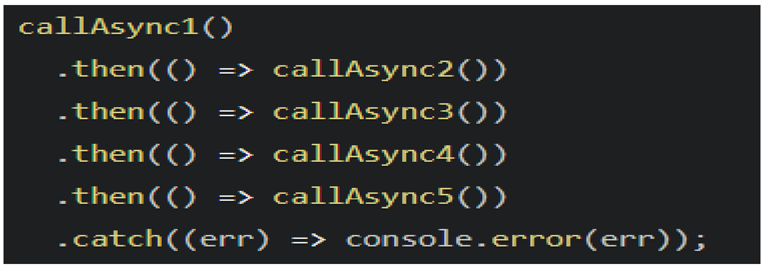


```
callAsync1(function () {
  callAsync2(function () {
    callAsync3(function () {
      callAsync4(function () {
        callAsync5(function () {
          // ...
        });
      });
    });
  });
});
```

Strategies to avoid callback hell

Developers can address or avoid callback hell by using strategies like modularizing the code into named functions, using asynchronous control flow libraries, or leveraging modern JavaScript features like Promises and async/await to write more linear, readable asynchronous code.

Promise Chaining



```
callAsync1()
  .then(() => callAsync2())
  .then(() => callAsync3())
  .then(() => callAsync4())
  .then(() => callAsync5())
  .catch((err) => console.error(err));
```

Async/Await

```
async function asyncCall() {
  try {
    await callAsync1();
    await callAsync2();
    await callAsync3();
    await callAsync4();
    await callAsync5();
  } catch (err) {
    console.error(err);
  }
}
```

66. Does the `map()` method mutate the original array?

No, the map() method does not mutate the original array. It returns a new array with the results of calling a provided function on every element in the calling array.

```
const roadmaps = ['JavaScript', 'React', 'Node.js'];

const renamedRoadmaps = roadmaps.map((roadmap) => {
  return `${roadmap} Roadmap`;
});

console.log(roadmaps); // ['JavaScript', 'React', 'Node.js']
console.log(renamedRoadmaps); // ['JavaScript Roadmap', 'React Roadmap',
```

67. Is it possible to run 2 lines of code at the same time in JavaScript?

No, it is not possible to run 2 lines of code at the same time in JavaScript. JavaScript is a single-threaded language, which means that it can only execute one line of code at a time. However, it is possible to run 2 lines of code at the same time using asynchronous code.

68. How to enable strict mode in JavaScript?

To enable strict mode in JavaScript, you need to add the following line at the top of the file or function `use strict` ;

69. How to handle errors in async/await?

In order to handle errors in async/await, we can use the try/catch statement.

Rejecting a promise

```
const promise = new Promise((resolve, reject) => {
  reject(new Error('Something went wrong'));
});
```

Try/catch statement

```
async function main() {
  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error.message);
  }
}
```

The catch block will be executed when the promise is rejected or when an error is thrown inside the try block.

70. How to parse JSON in JavaScript?

In order to parse JSON, you can use the JSON.parse() method. It parses a JSON string and returns the JavaScript equivalent.

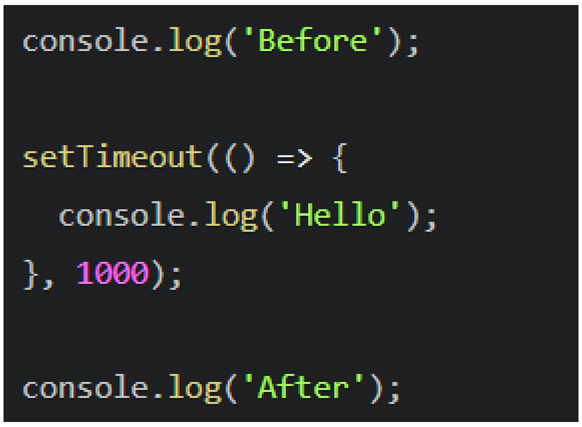
```
const json = '{"name":"JavaScript","year":1995}';
const roadmap = JSON.parse(json);

console.log(roadmap.name); // JavaScript
console.log(roadmap.year); // 1995
```

71. Asynchronous vs Synchronous code?

The difference between Asynchronous and Synchronous code is that Asynchronous code does not block the execution of the program while Synchronous code does.

Asynchronous Code

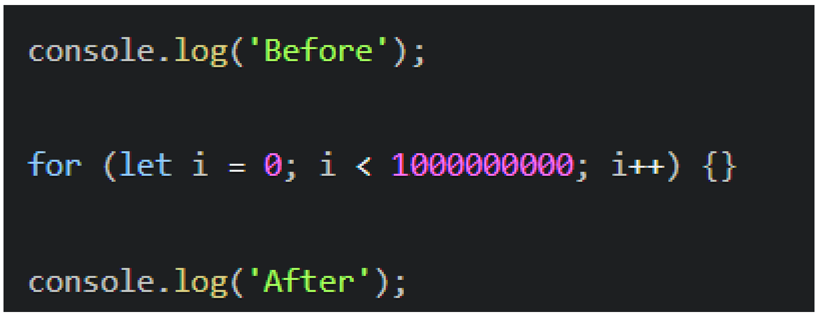
Asynchronous code is executed in the background and it does not block the execution of the program. It is usually used to perform tasks that take a long time to complete, such as network requests.



```
console.log('Before');

setTimeout(() => {
  console.log('Hello');
}, 1000);

console.log('After');
```

Synchronous Code

Synchronous code is executed in sequence and it blocks the execution of the program until it is completed. If a task takes a long time to complete, everything else waits,

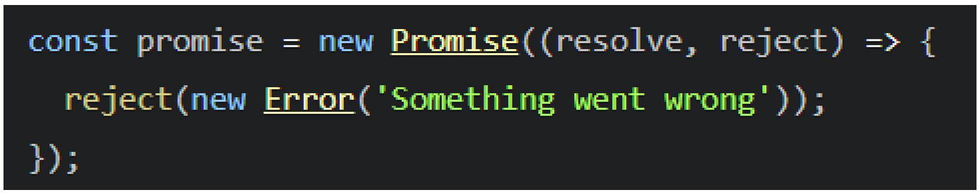


```
console.log('Before');

for (let i = 0; i < 1000000000; i++) {}

console.log('After');
```

72. How to handle errors in Promises?

In order to handle errors in promises, we can use the catch method or the second argument of the then method.

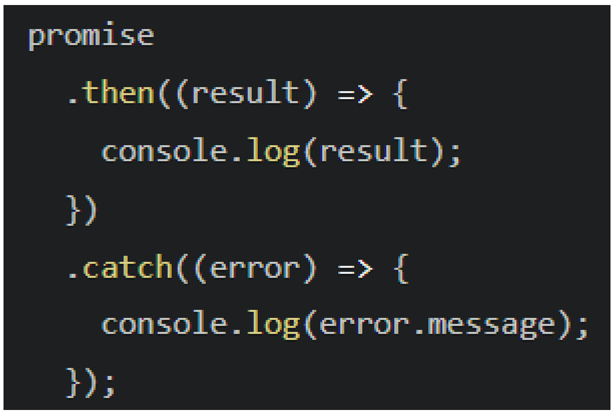
Rejecting a promise



```
const promise = new Promise((resolve, reject) => {
  reject(new Error('Something went wrong'));
});
```

Catch Method

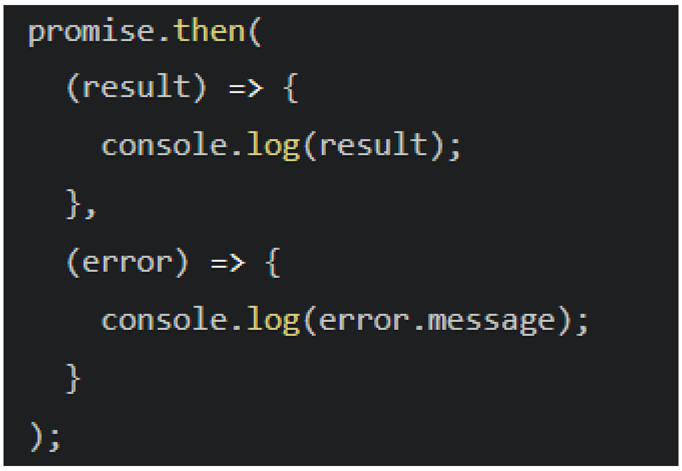
In this method, we can pass a callback function that will be called when the promise is rejected.



```
promise
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.log(error.message);
  });
```

<u>Second argument of the then method</u>

In this method, we can pass two callback functions as arguments. The first one will be called when the promise is resolved and the second one will be called when the promise is rejected.



```
promise.then(
  (result) => {
    console.log(result);
  },
  (error) => {
    console.log(error.message);
  }
);
```

73. Garbage collection in JavaScript?

The JavaScript engine uses automatic garbage collection. JavaScript automatically manages memory by freeing up space used by objects no longer needed. This algorithm is called Mark and Sweep, which is performed periodically by the JavaScript engine.

74. *What is event delegation?*

In simple terms, event delegation is just a way to add one event listener to listen for multiple child elements on the dom. Instead of having to attach a listener to each element and running the risk of slowing down your application or breaking something when you need to add to your code, event delegation prevents this.

75. What is the difference between functional programming and OOP?

<u>Functional Programming:-</u>

- Functional programming emphasizes on evaluation of functions.
- Functional programming uses immutable data

- Functional programming does follow declarative programming model
- Parallel programming supported by functional programming
- Functional programming, the statements can be executed in any order
- In functional programming, recursion is used for iterative data
- The basic elements of functional programming are variables and functions.
- Functional programming is used only when there are few things with more operations.

OOP:-

- Object oriented programming based on concept of objects
- Object oriented uses the mutable data
- Object oriented programming does follow imperative programming model
- Object oriented programming not support parallel programming
- In OOP's the statements should be executed in particular order
- In OOP's loops are used for iterative data
- The basic elements of object oriented programming are objects and methods

Object oriented programming is used when there are many things with few operations.

76. What is the difference between Shallow Copy and Deep Copy?

Shallow Copy:-

- Shallow copy method creates a copy where the source and copied variable reference remains the same. This means that modifications made in one place would affect both places.
- Common methods like Array.concat(), Array.from(), Object.assign(), etc creates a shallow copy.

Deep Copy:-

- Deep copy method creates a copy where the source and copied variable reference are completely different. This means that modifications made in one place would only affect the variable where we are making a change.
- Spread(...) operator creates a deep copy when there is no nesting.
- One of the ways to create a deep copy is by using JSON.parse() and JSON.stringify()

77. What is the difference between Slice and Splice and Split?

Split:-

Split a function that splits the given string into an array of substrings. The split method doesn't change the original string array and will return the new array.

Slice:-

- Slice method doesn't change the original array. It is a method of both arrays and strings. Slice method can take two arguments. First arguments is required, second argument is optional
- First argument that represent where to start selection
- Second argument that represent where to end selection

- It does not mutate the original array.
- Make a shallow copy of the original array.
- Returns the copied array.

<u>Splice:-</u>

- Splice method changes the original array. It can remove elements, replace existing elements, or add new elements to an array. It can take 3+ arguments.
- First argument is indexed and required.
- Second argument is optional and represents the number of items to be removed.
- Third argument is optional and represents the number of items to be added. Arguments can be increased.
- Mutates the original array

78. What is the difference between Object seal and Object freeze?

Object.freeze()

The Object.freeze() method freezes an object. This is actually the most strict way of making objects immutable. It does the following things.

- Prevents adding new properties
- Doesn't allow remove existing properties
- Doesn't allow to change immediate properties of object
- Attributes of child objects can be modified.

Object.seal()

With sealing, the situation is a little bit different, The Object.seal() method seals an object, preventing new properties from being added to the object. It does the following things.

- Allows changing existing properties of an object.
- Prevents adding new properties.
- Don't allow removing existing properties.

79. Is Arrow function hoisted?

Hosting is the default behavior of moving all the declarations at the top of the scope before code execution.Basically it gives us an advantage that no matter where functions and variables are declared, they moved to the top of their scope regardless of whether their scope is global or local. Like all other functions in Javascript, the arrow function is not hoisting the main reason that you cannot call them before initialization.

80. What is the difference between for of and for in?

- A for… of loop works for arrays(and similar iterables) and loops through the values.
- A for… in loop works for objects and it loops through the object's keys.
- In Javascript, the for… in loop is used to iterate over the enumerable properties of an object, while the for .. of loop is used to iterate over the values of an iterable object.

	for … in	for … of

Applies to	Enumerable Properties	Iterable Collections
Use with Objects?	Yes	No
Use with Arrays?	Yes, but not advised	Yes
Use with Strings?	Yes, but not advised	Yes

81. What is a callback?

- A javascript callback is a function which is to be executed after another function has finished execution.
- A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.
- Call back functions are functions that are passed as a parameter to another function
- Callbacks ensure that the function will not run before the task is completed but will run right after the task is completed. It develops asynchronous Javascript code and keeps them safe from errors. In Javascript, the way to create a callback function is to pass it as a parameter to another function then call it back after the task is completed.

Ways to handle callbacks

- Use of Promise - A promise can be generated for each call back. When the callback is successful, the promise is resolved, and if the callback fails, the promise is rejected.
- Use of Async-Await - Asynchronous functions are executed sequentially with use of await, execution stops until the promise is resolved and function execution is successful.

82. What is the call, bind and apply method in Javascript?

Call:- call is a special function in javascript, using the call method we can borrow the functions. Which is used to invoke the function directly using the reference of this variable.

Apply:- The only difference between the call and apply method is the way of passing the arguments to the function. Instead of passing the arguments individually in the call method, here we can pass an array of arguments.

Bind:- It looks exactly the same as the call method. Like the call method instead of calling the method directly, the Bind method binds the methods with a copy of the object and returns a new function. Using the bind method we can invoke a function later.

83. Using the object constructor in Javascript is not recommended why?

Using the OBJECT constructor in Javascript is discouraged because it can lead to unexpected behavior and is less efficient than using object literals. The Object constructor can be misused, and its behavior can vary based on the number and types of arguments passed to it, which may lead to unintended results. Using object literals ({}) is generally considered more straightforward and clearer.